Proceedings of the GCC Developers Summit

May 25–27, 2003
Ottawa, Ontario
Canada

Contents

Optimizing for Space: Measurements and Possibilities for Improvement

Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács
Research Group on Artificial Intelligence
University of Szeged
Aradi vértanúk tere 1., H-6720 Szeged, Hungary, +36 62 544126
`{beszedes,gertom,gyimi,alga,lac}@cc.u-szeged.hu`, `http://gcc.rgai.hu/`

Abstract

GCC's optimization for space seems to have been often neglected, in favor of performance tuning. With this work we aim at determining the weakpoints of GCC concerning its optimization capability for space. We compare (1) GCC with two non-free ARM cross-compiler toolchains, (2) how GCC evolved from release 3.2.2 to version 3.3, and (3) two runtime libraries for the Linux kernel. All tests were performed using the C front end and for the ARM target both as standalone and as Linux executables. The test suite is comprised of applications from well-known benchmark suites such as SPEC and Mediabench. An optimal combination of compiler (and linker) options with respect to minimal code size is elaborated as well. We conclude that GCC 3.3 steadily improves with respect to version 3.2.2 and that it is only about 11% behind a high-performance non-free compiler. At the same time, we were able to document a number of issues that deserve further investigation in order to improve code generation for space.

1 Introduction

GCC is increasingly used as a cross-compiler to produce programs for embedded systems. Although performance in terms of speed is also important, in many cases the amount of consumed resources (memory, energy, etc.) plays an even greater role in the case of devices with limited resources. So, when GCC is used to build these software, the code produced should be as small as possible. Indeed, GCC is able to optimize for space but, alas, it seems that this objective was often neglected when designing and implementing various code generation and optimization algorithms [1, 5]. We may conclude the same when we consider the fact that beside the vital regression testing methods and the results of several benchmark suites available on GCC web pages [9, 8, 3], no word is spoken about benchmarking *code size*. In fact, were unable to find any related publication at all which deals with the assessment of compilers' capabilities for space optimization.

With this work we attempted to determine the weakpoints of GCC concerning its optimization capability for space. We present the results of our assessments where we compared:

- GCC for standalone executable with two non-free ARM cross-compiler toolchains,

- How GCC evolved from release 3.2.2 to version 3.3, and
- Two runtime libraries for GNU/Linux, glibc [2] and μClibc [7].

All tests were performed using the C front end and for the ARM target (both for standalone and Linux executables) as this combination is one of the most frequently used nowadays for embedded applications. A testbed was utilized with applications from various well known benchmark suites.

We did our best to discover the optimal combination of compiler (and linker) options with respect to minimal code size; we elaborate on the relevant ones for GCC and propose a set of options to extend the default settings for code size. With this option set an improvement of nearly 5% was achieved.

In the investigation we included both the object sizes produced by the compiler and the linked executable sizes to see what effect the runtime libraries had on the overall linked code size. Comparing only object sizes, one non-free compiler is about 11% better than GCC, but in the case of executables this ratio rises to 32%.

We investigated the generated code by GCC more thoroughly and finally we document several issues that deserve further investigation in order to improve code generation for space. These include the lack of interprocedural optimizations, the required unit at a time compilation, more intelligent handling of `-Os`, etc.

In Section 2 we describe our measurement environment and methodology. Section 3 deals with GCC's different compiler options and there also we give our proposal for the best combination. Sections 4 and 5 present the actual results for standalone executables and Linux libraries, respectively. Finally, in Section 6 we summarize our conclusions and give our view on the possibilities for improving GCC.

2 Measurement Environment

For all three objectives of our investigation presented in the previous section, we have set up a common measurement environment. It consists of a collection of test programs that are suitable for compiling and measuring code size for all compilers and configurations under investigation. The environment is able to perform these measurements and present the data in a simple form ready for further processing. In addition, it also facilitates the execution of the executable programs.

2.1 Compiler Toolchains

In each experiment we employed C as the source language and the chosen target architecture was ARM (32-bit ARM instruction set). Two types of target code were used: standalone programs (that run on the hardware without an operating system) and Linux target for the ARM architecture (for GCC compiler `arm-elf` and `arm-linux` machines, respectively). The following toolchains were used for the measurements:

- GCC 3.2.2 version with newlib version 1.10.0 [6] for standalone target (with binutils version 2.13)
- GCC 3.3 prerelease snapshot (2003-04-14) with the same newlib and binutils
- GCC version 3.2.2 with glibc version 2.2.5 [2] for Linux target
- GCC 3.3 prerelease snapshot (2003-04-14) with glibc version 2.2.5

- GCC version 3.2.2 with μClibc version 0.9.15 [7]

- Two non-free compilers for ARM architecture configured as standalone targets. These will be denoted by *Compiler 1* and *Compiler 2* in the following discussions. The former uses `elf` output format, while the latter produces `coff` files.

The switches that control optimization for space were turned on for all toolchains. In addition, several further options (both compiler and linker) that enable or disable certain code optimization and/or generation algorithms were also set that resulted the most compact code size. The combination of these extra options was determined by trial and error, and for GCC toolchains we elaborate on these in Section 3.

For each GCC toolchain the runtime libraries were compiled using the same options as for the test programs. (Neither of the two non-free compilers libraries were prepared in such way.) The use of such libraries has an effect where the executables are compared, because the overall code size incorporates library code as well.

2.2 Testbed

The testbed used in the experiments consists of two parts: small example programs and real applications from several well-known benchmark suites (GNU applications, SPEC CPU2000 [10], MediaBench [4]). In the following table some information is given about the sizes of the test programs:

Test project	*files*	*lines*	*bytes*	*exec.*
`bzip2`	1	4,250	121,279	1
`catdvi`	6	770	24,332	1
`flex`	21	19,571	530,312	1
`g721`	8	1,725	46,980	2
`gsm`	29	5,982	182,809	1
`jpeg`	84	34,181	1,150,110	6
`mcf`	25	2,414	53,310	1
`mpeg2enc`	22	7,608	217,864	1
`osdemo`	147	68,434	1,925,141	1
`parser`	18	11,391	356,526	1
`sed`	20	12,393	365,886	1
`P3szogr`	1	48	1,568	1
`_3szog`	1	48	1,419	1
`abc`	1	17	443	1
`arg`	1	25	390	1
`datum`	1	48	870	1
`eltelt`	1	32	939	1
`endian`	1	18	258	1
`geometry`	1	435	11,869	1
`lnkoszt`	1	52	1,121	1
`minimax`	1	52	1,444	1
`static`	1	35	460	1
`szinusz`	1	52	1,372	1

The first column shows the number of files that constitute the test project, the second one gives the total number of program lines, and the third column gives the size of the source code in bytes. In the last column the number of executables that are built from the test project is shown.

All test programs were compiled to produce the object files and the given executable programs were prepared by linking. These objects and the linked executables for each of the toolchain under investigation were used for measurement.

In the following for each measurement the small programs (the last 12) are treated jointly and are denoted by "small."

2.3 Measurement Method

The way to measure the size of the generated code (i.e. its compactness) is not always trivial. As obvious, we chose to investigate the final binary machine code (instead of, for example, the assembly code).

Objects and executables. The granularity of the code was a further aspect: should we measure the function sizes individually, the object code for a complete compilation unit, or investigate the size of the linked executable? In this paper we present the results for the latter two because in certain environments both can be interesting. When we compare the object sizes the effectiveness of the compiler proper is actually compared,[1] while in the second case the whole compiler toolchain is assessed including the compiler, the linker and libraries as well. This is because the size of a linked program depends on the size of the libraries and also how they are processed by the linker. Hence, in this paper we mostly rely on comparing objects which is more informative with regard to a compiler's optimization capability for space.

In order to get the best possible results when measuring executables, we also built the libraries of GCC toolchains with the same flags as the test sources. With the libraries of the two non-free compilers we were not able to do the same.

Standalone and Linux programs. Another dimension of the categorization we investigated was both kinds of targets: standalone executables (i.e. for without an operating system) and executables built for a specific operating system (in our case GNU/Linux). Although the same compiler is used with the same settings, the resulted binaries generally contain several notable differences: a few in the case of objects and a significant difference with executables. These are mostly due to different executable production and to the fact that different runtime libraries are used for the two cases (i.e. in the case of GCC, newlib and glibc).

One would expect that with objects there should be no difference at all. However, some minor impact of the library is still noticeable. The library headers should contain the same standard prototypes (e.g. standard functions), but the difference comes from the different implementation of some features. For example, some standard names can be implemented using macros and function calls as well.

Clearly, then, measuring the size of the executables incorporates a much greater impact of the library code. It is apparently measurable on standalone executables. However, the situation becomes more complicated when we investigate executables built for Linux. The reason for this is that Linux executables do not embed the library code, but they maintain only references to the so-called shared objects, which are linked at runtime. (Even if static linking is used some functionality will still be implemented in the operating system rather than the executable.) We present some results for Linux executables in Section 5.

Sections. Another problem was deciding which parts of the generated files we should take into account (obviously the size of the binary file is not relevant because of various headers, etc.). The generated program code consists of many parts; instructions, data and so on, which are generally separate in a binary file (in the *sections*). However, in many cases these parts can be intermixed (e.g. executable code can contain embedded data). In addition, several other sections are generally also put into the binary file, which are of no interest with respect to the size of the code. These include the debug sections, symbol tables, etc.

The different types of object files (`elf` and `coff`) can have different kinds of sections and, what is more, the different compilers may

[1] Note, that the library implementation still has a minimal impact on the object sizes because of the library headers, which are also translated by the compiler. Consider for example, that macros can be used to implement function-like behavior.

use various strategies for laying out code and data into sections. More specifically, different compilers may split some code into several sections, or put other things together in one section. For example, `elf` files contain one (or more) initialized read-write data section(s), while `coff` files contain program code that will initialize the data at runtime. So no common handling could be used and the combination of the sections to be incorporated in the measurements needed to be determined separately for each toolchain.

In each case we summarized the size of only those sections that contains generated code that is directly used by the program. These are the sections that contain executable code and constant- or initialized read-write program data. Note, however, that executable code and constant data cannot always be clearly separated (there are constant data items which are "hidden" in the executable code) so we handle them together during the comparison.

We experimented with two kinds of section combinations: (1) the size of sections containing program code or constant data (referred to as "read-only sections") and (2) the size of sections that contain any kind of program data, which also includes read-write data (referred to as "all sections"). We decided to follow the second approach because it seemed to be the most reasonable because of the above-mentioned various types of handling of initialized read-write data.

Measurement tools. When assessing both the object and executable sizes the `elf` and `coff` files needed to be investigated. To this end different methods for extracting the section sizes were employed because of the different binary formats. The program `size` (part of *binutils*) is a suitable tool for extracting the size of the mentioned sections from `elf` files. We were unaware of any similar tool for `coff` files. The program `coffdump` extracts the sizes of the sections from `coff` files, but not in a summarized form. Fortunately, all `coff` files contain almost the same sections and have the same names. We examined what kind of data was contained in the sections, and counted the required sizes by hand. (Fortunately, only one of the non-free compilers uses this format, with all other toolchains including GCC we were able to extract code sizes automatically.)

Execution. The measurement environment is capable for executing the built programs using a simulator for standalone programs and an ARM-based hardware device with Linux system for Linux binaries. We ran the programs and checked their outputs for validating the compiler toolchain with components of different versions, and for verifying the correctness of various compiler option combinations. Throughout our measurements only those configurations were used that produced correct and running programs.

3 Compiler and Linker Options

With each toolchain investigated we sought to find the best possible combination of options with respect to code size. In general, compilers provide a special optimization option that instructs them to optimize for space rather than for speed. With GCC, this option is the switch called `-Os`.

3.1 Best Options for Space in GCC

Commonly, `-Os` is used internally in GCC to enable or disable certain optimization algorithms, but generally any part of the compiler proper can depend on this option and perform differently when space is the concern. However, there are a number of other compiler options (mostly related to optimization) which have a notable effect on the size of the generated code. By experimenting with these options we found that `-Os` alone does not produce the minimal code for our testbed. Hence we determined the combination of options on top of `-Os`, which proved to be the best on our testbed.[2]

The following table summarizes the final choice of options, which we used in all our trials (except where mentioned otherwise). (Note, that some of these are implicitly enabled or disabled by `-Os`,[3] therefore we supply the options later in the command-line so that they will be overridden.)

Compiler Option	*3.2*	*3.3*
`-Os`	yes	yes
`-mno-apcs-frame`	yes	yes
`-fomit-frame-pointer`	yes	yes
`-ffunction-sections`	yes	yes
`-fdata-sections`	yes	yes
`-fno-force-mem`	yes	yes
`-fno-force-addr`	yes	yes
`-fno-inline-functions`	yes	yes
`-fnew-ra`	no	yes
`-fbranch-probabilities`	yes	yes
`-finline-limit=1`	yes	yes
`-fno-schedule-insns`	yes	yes
`-fno-optimize-sibling-calls`	yes	yes
`-fno-if-conversion`	no	yes
`-fno-thread-jumps`	yes	yes
`-fno-hosted`	yes	yes

Some options were not available in GCC 3.2 releases, evidently they were left out in the cases when this release was measured. We will use the notation *opt-1* for the best options for 3.2.2 and *opt-2* for the best options for 3.3.

The option `-mno-apcs-frame` is specific to the ARM target. We also used another ARM-specific option `-mno-thumb-interwork` to tell the compiler that we were generating for just 32-bit ARM instruction set.

Two interesting options are `-ffunction-sections` and `-fdata-sections` which generate only one function/data per section and this helps the linker to omit the unused functions/data from the executable. Generally speaking, they do not influence the object sizes, but the executables may become smaller.

Another notable option is `-fno-inline-functions` which disables the automatic inlining of GCC. In general, automatic inlining performs very badly with respect to code size and it could be made more intelligent.

The linker also has a number of options that were worth experimenting with. We determined the following combination which produced an overall smaller code than the default:

[2]One option belongs to this set if it produces an overall gain with respect to the default `-Os`, so it may happen that in some cases it performs worse. It may also happen that one option combined with another one degrades the overall result, but of course, we could not try every combination of the options available.

[3]This is the list taken from the GCC 3.3 sources:

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fbranch-probabilities -fcaller-saves
-fcprop-registers -fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdefer-pop
-fdelete-null-pointer-checks
-fexpensive-optimizations -fforce-mem
-fgcse -fif-conversion
-fif-conversion2 -floop-optimize
-fno-merge-constants
-fno-reorder-blocks
-foptimize-sibling-calls -fpeephole2
-fregmove -freorder-blocks
-freorder-functions
-frerun-cse-after-loop
-frerun-loop-opt -fstrength-reduce
-fstrict-aliasing -fthread-jumps
```

options that depend on a define:

```
-fdelayed-branch -fomit-frame-pointer
-fschedule-insns
-fschedule-insns-after-reload.
```

Linker Option
`-O 2`
`-gc-sections`
`-relax`
`-no-whole-archive`

The options listed above produced, on our testbed, an overall improvement in code size of 4.78% with respect to using only `-Os`. Figure 1 shows the results separately for each program. To obtain the relevant data we used the GCC 3.3 snapshot with only `-Os` turned on and compared it to the same compiler with additional options from the table above (average object sizes of test projects in standalone target). The total sizes of the test projects is given with the project's name in bytes.

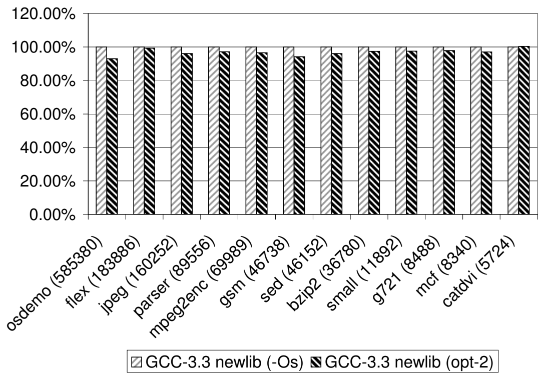

Figure 1: The effect of additional compiler options

We can see from the above plot that every test program has benefited from these options, especially the bigger ones (except `flex`, which is probably due to the fact that it contains uncommonly large amount of data).

In Section 5 we present some data which shows that a marked improvement in library size can also be achieved using this options set.

Due to the above results we propose to add these to the default operation of `-Os` in future releases of GCC (at least for the ARM target).

3.2 Other Optimization Options

There are high number of optimization options (starting in `-f`) in GCC that can be given on command-line (170+). Most of them have a binary state and so a corresponding `-fno-`*`XXX`* is also normally present. We examined all available options in GCC 3.3 but of course, we could not try all of the possible combinations, so we followed a simple approach in that an option (both the enabling and disabling versions) was added to the list of good options if it brought improvement over the default `-Os`. An individual option was tried separately from the others rather than by cumulating them. The final result is given in the previous section.

Many of the investigated options had some problems or did not yield improvements and hence they were ignored. In the following we categorize these options rather than listing them all (they can be found in the GCC manual). Those options that are not mentioned here did not improve the code (the correctness of the output was not verified either).

Combined use. The following options separately produced certain improvements, but their combined effect was not better on average: `-ffast-math` `-ffreestanding` `-fno-builtin` `-fno-inline` `-fno-sched-interblock` `-fno-sched-spec` `-fsched-spec-load` `-fvolatile-static`.

Parameterized options. For this work we were not able to include the investigation of those options that accept some parameters (i.e. not a binary). This parameter is generally a number but in some cases it can be a string. We only investigated `-finline-limit=`*`number`* which showed a minor improvement. The following options were left with their default settings: `-falign-functions=`*`number`* `-falign-labels=`*`number`* `-falign-loops=`*`number`* `-falign-jumps=`*`number`* `-fcall-used-`*`number`* `-fcall-saved-`*`number`* `-fdiagnostics-show-location=`*`string`* `-ffixed-`*`number`* `-fmessage-length=`*`number`* `-fsched-verbose=`*`number`* `-fstack-limit-register=`*`number`* `-fstack-limit-symbol=`*`string`* `-ftls-model=`*`number`*.

Invalid generated code. The options listed here always produced smaller code, but these codes could not be correctly executed on GCC 3.3: `-fshort-double` `-fsingle-precision-constant` `-funsafe-math-optimizations`. These should be investigated for possible bugs in GCC.

Irrelevant option. Some options are either not implemented in GCC 3.3 or they did produce some extremely small code. These are the following: `-fallow-single-precision` `-fcall-saved` `-fcall-used` `-fconstant-string-class` `-fdiagnostics-show-location` `-fdump-tree` `-ffixed` `-finline-functions` `-finline-limit` `-finstrument-functions` `-fleading-underscore` `-fmessage-length` `-fno-allow-single-precision` `-fno-call-saved` `-fno-call-used` `-fno-constant-string-class` `-fno-diagnostics-show-location` `-fno-dump-class-hierarchy` `-fno-dump-translation-unit` `-fno-dump-tree` `-fno-fixed` `-fno-function-sections` `-fno-inline-limit` `-fno-message-length` `-fno-pretend-float` `-fno-sched-verbose` `-fno-stack-limit-register` `-fno-stack-limit-symbol` `-fno-tabstop` `-fno-template-depth` `-fpreprocessed` `-fpretend-float` `-fprofile` `-fprofile-arcs` `-fsched-verbose` `-fshort-enums` `-fssa` `-fstack-limit` `-fsyntax-only` `-ftabstop` `-ftemplate-depth`.

4 Compiler and Toolchain Comparisons

In this section we present the results of a comparison of the sizes of objects and executables of GCC configured for a standalone target with two non-free compilers. The two compilers shall remain anonymous, which will be referred to as *Compiler 1* and *Compiler 2*. In both cases the best configuration of compiler options was used for code size. In the diagrams *opt-1* denotes the best options for GCC 3.2.2 and *opt-2* the best options for 3.3.

A comparison of objects is more informative with regard to a compiler's optimization capability for space, because in this case no pregenerated code of libraries or startup routines are included.

All sizes comprise of the program section sizes (as described in Section 2.3), and we present these in a relative form: with respect to GCC 3.3 snapshot with our option-set (elaborated in Section 3).

4.1 Compiler Results on Objects

In Figure 2 the average achievement of the C compilers is shown in terms of object size. The values are computed as the sum of the sizes of all objects of the test programs, and are shown as relative to GCC.

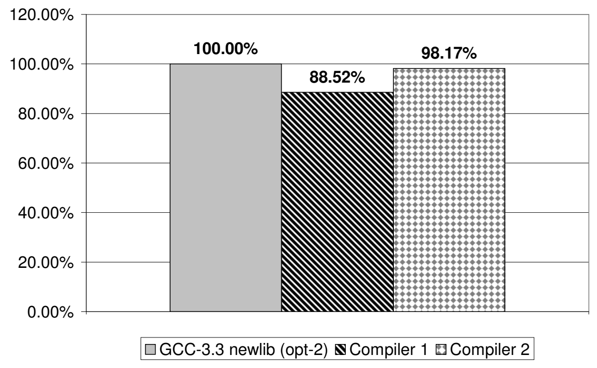

Figure 2: Average compiler results for objects

As can be seen, *Compiler 1* provides the best results and *Compiler 2* is still better than GCC. The gain in size achieved by *Compiler 1* is 11.48% and 1.83% by *Compiler 2* relative to the size of the objects compiled with GCC.

The same measurement is shown in more detail in Figure 3. It shows the effect of the C compilers separately for the different test programs. The sizes of the objects are summarized per test project (which is shown in parentheses after the project name at the bottom of the diagram in bytes).

The optimization capabilities of the compilers seems to be similar for each test project: *Compiler 1* produces the smallest code; the sizes of the result of *Compiler 2* are between the sizes of the output of *Compiler 1* and GCC.

4.2 Toolchain Results on Executables

We also investigated the difference in the generated code size of the executable files using the same environment and options as for the

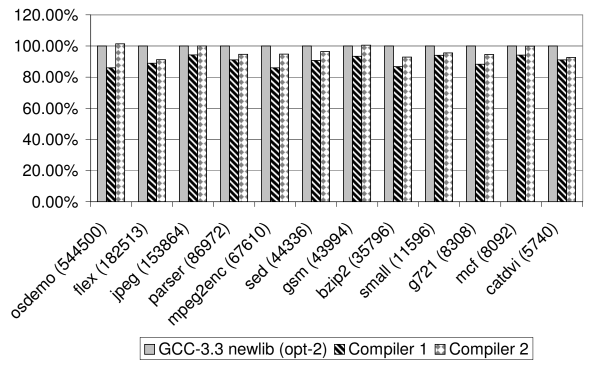

Figure 3: Individual compiler results for objects

objects. We performed this comparison for standalone executable images, which means that apart from the application objects, the library code and the effectiveness of the linker is also incorporated in these number.

In Figure 4 the average result of executable sizes is shown. We computed the average values in the same way as for the objects, so they are simple sums of the program section sizes in the executables. Relative values are shown as well with respect to GCC.

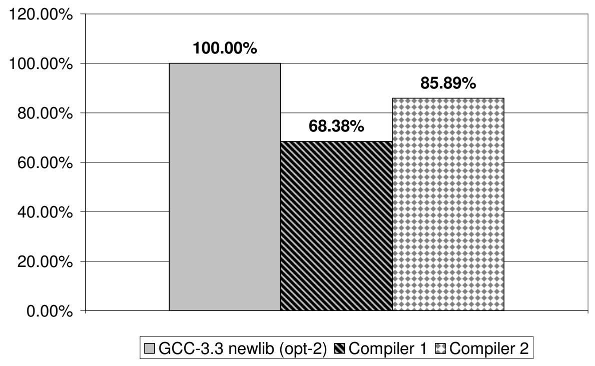

Figure 4: Average toolchain results for executables

We can observe that the ranking of the toolchains regarding code size in this comparison has not changed with respect to investigating only the compilers. The differences are, at the same time, more significant than in the case of objects comparison (about twice as much).

Apparently, the reason for this is twofold: the tools use different implementations of standard C runtime libraries and the linkers may also behave differently. It is an open question whether the difference in the libraries causes a bigger difference or it is the linker that is responsible (e.g. by performing different optimizations at link time). Whatever the case, the comparison of the executables is not as a good measure of the toolchains as a comparison of the objects is a measure of the compilers, because the implementation of the libraries is also an important factor, which is included in the result.

In Figure 5 the same measurement is shown in more detail individually for the various executables. The sizes of the executables are summarized per test project (which is shown in parentheses after the executable name at the bottom of the diagram in bytes).

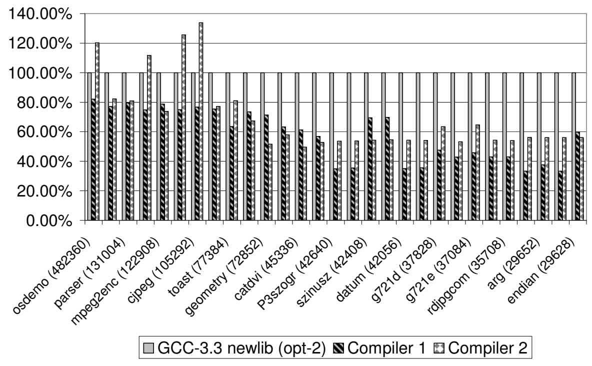

Figure 5: Individual toolchain results for executables

As can be seen, the ranking of the three toolchains does not always show the same order as in the average case, but we can see that *Compiler 1* is still in all but one cases much better than GCC. *Compiler 2* produced both the worse and the best results: there are cases when this tool had the largest code, but there are also cases where it seems to be the best tool.

5 Results for Linux Libraries

Apart from using as a cross-compiler generating standalone executable images, GCC is also widely used to generate programs for GNU/Linux. Hence we thought that it would be a good idea to investigate the sizes of the generated objects and executables in this case as well. In these experiments we used a GCC compiler configured for the `arm-linux-elf` target with the same environment and compiler options as for the standalone target (the only exception being that we needed to omit the `-ffunction-sections` option of GCC because it caused some problems when executing the programs on a Linux system). In this case we employed the commonly used GNU library *glibc* [2].

The Linux executables are not comparable with a standalone configuration (namely, with the GCC `arm-unknown-elf` target or with the two non-free compilers). This is because Linux uses shared objects that are linked at runtime to the executable (see Section 2.3). Nevertheless, objects should be comparable. Our results showed that the objects for Linux target have a smaller code size than objects for standalone target (by 8.35% with GCC 3.2.2 on our testbed). By examining the compiled objects we found that the size differences were primarily due to the different implementation of the library headers.

5.1 glibc vs. μClibc

Alas we could not find any other compiler toolchain (either free or non-free) that was able to generate for Linux target. Only the μClibc toolchain [7] could serve as a comparison basis. However it also uses the GCC compiler, so it really compares two implementations of the standard C runtime libraries.

We performed all measurements on the testbed and investigated the sizes of the objects and executables as well. We used GCC version 3.2.2 because the later versions (3.3 snapshots and the active development 3.4) are not supported by μClibc. With glibc- and μClibc-based toolchains we used the same compiler options that we found to be best for size with the standalone target (as described in Section 3). It is interesting to note that compiling the libraries using our combination of options brought a significant improvement in library size with respect to the default settings: 3.22% for glibc and 2.04% for μClibc (computed for shared object binaries and not for static libraries).

An interesting observation was that the μClibc toolchain generally produces a slightly larger code size (1 or 2% at most) than GCC with glibc. We do not present the actual results here. Rather it is more interesting to look at the difference in the sizes of the actual libraries.

We measured the total code section sizes for all the generated library files. On average the μClibc library was smaller by 80.58% (1.94MB vs. 0.38MB) for the shared object binaries, and was smaller by 59.49% (1.59MB vs. 0.64MB) for static libraries counting simply the sum of all sections in all of the library files.

6 Conclusion: Improvements and Limitations

Assessing a compiler's effectiveness in optimizing for space poses a number of difficulties. Based on our measurement results presented in previous sections, we can say that the most reliable way is to compare the section sizes containing program code and data in objects rather than executables. This is because the implementation of the libraries is also an important factor: all tools work with their own implementation, and this difference is also included in the result.

We managed to narrow the gap between a high-performance non-free compiler and GCC 3.3 using our own set of compiler options from 15.71% to 11.48% measured on objects for a standalone target. However, this number is nearly double when we consider executables. This suggests that not only GCC needs improvement, but the associated libraries as well (in this case newlib).

Things get more complicated if we wish to compare toolchains configured for Linux target and not for standalone. This is because Linux uses shared objects that are linked at runtime. In this case the only reasonable thing is to measure the size of the corresponding libraries. For example, we found that the total size of μClibc,—an alternative library to glibc—is far less than glibc (only one fifth).

6.1 Improvement of Prerelease 3.3

In the previous sections we presented the results of measurements with the latest snapshot of GCC 3.3 version. We performed the same experiments with version 3.2.2 as well (which is the last official release at the time of writing) and found that prerelease 3.3 has improved slightly in terms of optimizing for space. In this section we summarize the results of our measurements of what are the exact improvements.

The average difference between object sizes generated by GCC 3.2.2 and the 3.3 snapshot configured for standalone (with newlib) is only 0.31%. With both configurations we used the best compiler options, where some options are new to 3.3 and therefore not present in measurements with 3.2.2 (see Section 3). Figure 6 shows the same separately for each program of the testbed.

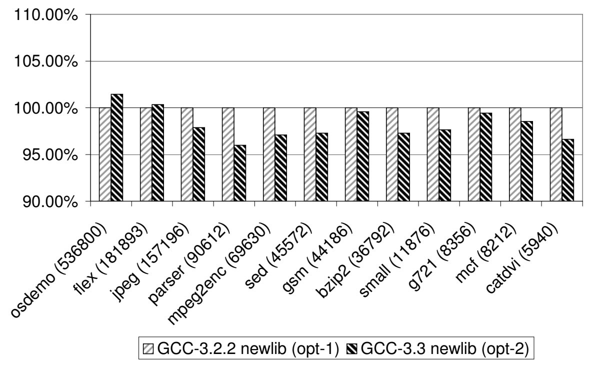

Figure 6: Improvement of GCC 3.3

Overall, no extraordinary improvement can be seen from this diagram and, in fact, the biggest program even shows that the older GCC generates smaller code. The difference is slightly larger in the case of executables; (it is 1.86% on average measured under the same conditions as for objects), which can also be attributed to the library code which is incorporated into the executable.

We also investigated the amount of improvement that can be achieved with Linux libraries. We prepared the glibc binaries using GCC 3.2.2 and 3.3 snapshot using the best options and found that with the new version the library was 0.95% smaller, which is similar to what we got for object sizes above. Figure 7 shows this improvement for each library component.

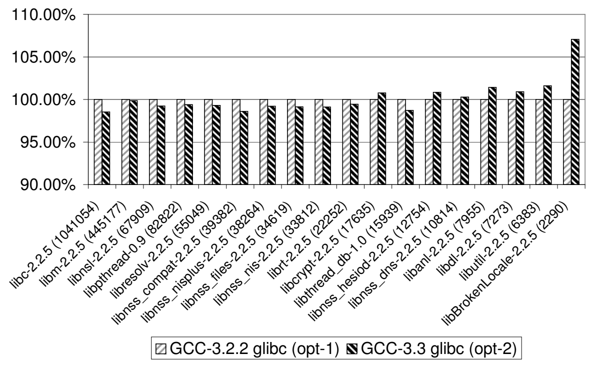

Figure 7: Improvement of GCC 3.3 measured on glibc

We made some investigations to found out what enhancements in GCC 3.3 caused this improvement in code size. There are a number of minor issues that could probably account for this, like some smaller optimizer improvements and target specific optimizations. However, we think that the major factor was the introduction of the new register allocation algorithm. In fact, by disabling `-fnew-ra` in GCC 3.3, the difference of 0.31% between 3.2.2 and 3.3 using the best options disappears and GCC 3.3 becomes to produce larger code by 0.29% on average!

6.2 Remaining Problems

By looking at the generated code in more depth, we managed to identify several weakpoints of GCC that could be improved in order to generate a more compact code. Another group of issues addresses GCC's limitations that are due to its architecture and logic of compilation. Some of them may not be solved or at least with very high effort. In the following we summarize the main issues for providing some starting point to future improvements.

Unit at a time compilation. GCC generally translates one function at a time and therefore it misses the opportunity of performing such optimizations that rely on seeing all functions of a compilation unit at the same time. With version 3.4 there was recently added the possibility for unit at a time compilation, but its utilization in optimization has not yet been fully achieved. If this feature is fully implemented in GCC, it would enable, for example, the sharing of global variables, the elimination of unused static functions, and the sharing of common data among functions (when the function-per-section option is not used).

More intelligent `-Os`. Generally, when `-Os` is turned on it means `-O2` with some additional optimization algorithms being implicitly enabled. In addition, any part of GCC can check for the state of this option. However, the semantics of this option could be further improved. First, a more careful selection of algorithms that need to be enabled could be implemented, similar to those proposed in Section 3. This could be further enhanced using the possibility for target-specific configuration of this switch. Furthermore, if `-Os` could act as an orthogonal option to other levels of optimization, it would offer for an even more flexible configuration.

Interprocedural optimizations. Due to the above-mentioned missing unit at a time compilation, no interprocedural optimization algorithms could be used. A number of existing algorithms could be extended to interprocedural operation, which would undoubtedly produce significant improvement, e.g. interprocedural dead-code elimination and redundant code elimination [1, 5]. Even some evidently redundant code constructs are currently generated by GCC. Consider, for example, the following code and notice that the call to function `foo` will be superfluously generated:

```
int a,b;
int foo(int x) { return x; }
void bar() {
  a = 1;
  b = foo(a);
}
```

Minor optimization issues. Here we list several minor issues that are related to some optimization algorithm (or are possibly specific for ARM target).

- The organization of loops is sometimes too complicated with redundant condition checking at higher optimization levels.
- The organization of the generated code for the `switch` statement can be made more optimal, especially when jump tables are used.
- RTL code generation from trees can be made more optimal than that for the current naïve preorder walk.
- Automatic function inlining does not seem to take into account when code size is the objective rather than speed. In this case only those functions should be inlined, which produce smaller code than calling the function.
- In ARM target, multiple variable load and save instruction are generated only for simple cases.

Library issues. Although the inadequacies of library implementations are not the subject of this article, we would like to remind the reader of the fact that the library headers indeed have some impact on the size of the generated code, which we elaborate in Section 2.3. Another interesting observation of ours was that a lot of space could be saved if some operators could be implemented by a library function call. For example, if integer division and modulo operators (`/` and `%`) would have a corresponding library function then for targets where these operations are not part of the instruction set, a simple call would be generated instead of the inline implementation of the division. Naturally, this would require that all library implementations provide such builtin functions for certain commonly-used operators.

6.3 Conclusion

We have seen that GCC is getting better and better with regard to code size. The latest version 3.3 (using an optimal combination of options) is only 11.48% worse than a high-performance non-free compiler. In Figure 8 we summarize the results of our measurements.

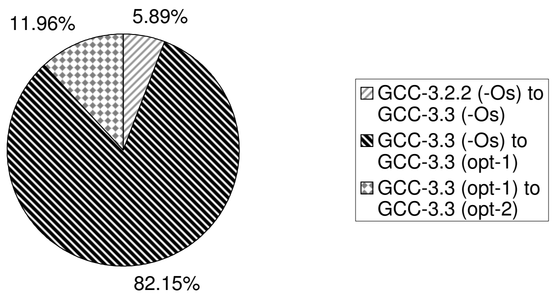

Figure 8: Summary of improvements

In this diagram we can observe (1) how much improvement version 3.3 brings with `-Os` only (0.3%), (2) the effect of a combination of options that we suggest over `-Os` measured on GCC 3.3 (4.15%) and (3) the effect of some new algorithms in GCC 3.3 (0.61%). These three constitute the total difference of 5.06% between GCC 3.2.2 with `-Os` and GCC 3.3 with *opt-2*.

Nevertheless there still are a number of issues—which we summarized in the previous section—that could make GCC's capabilities of optimization for space even better and this way shift its mainly academic use nowdays towards industry environments to become a serious competition to non-free commercial compilers.

7 Availability

The present document and related information including complete measurement data are available at

`http://gcc.rgai.hu/docs.php`

The homepage `http://gcc.rgai.hu/` aims to collect and maintain references to official GCC pages in connection with the ARM port.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Pub Co, 1985.

[2] Homepage of glibc. `http://www.gnu.org/software/libc/`

[3] Charles Leggett's benchmarks. `http://annwm.lbl.gov/bench/`

[4] Homepage of MediaBench. `http://www.cs.ucla.edu/~leec/mediabench/`

[5] S.S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[6] Homepage of newlib. `http://sources.redhat.com/newlib/`

[7] Homepage of μClibc. `http://www.uclibc.org/`

[8] SPEC 2000 tests by Andreas Jaeger. `http://www.suse.de/~aj/SPEC/`

[9] SPEC 95 tests by Diego Novillo. `http://people.redhat.com/dnovillo/spec95/`

[10] Standard Performance Evaluation Corporation – spec. `http://www.spec.org/`

Gcc Compile Server

Per Bothner
Apple Computer
`pbothner@apple.com`

The way the `gcc` user-level program invokes other programs (such as `cc1`, `as`, and `ld`) to compile programs has changed little over the years. Except for the recent integration of the C pre-processor `cpp` with the compiler proper `cc1`, it works very much like the original Bell Labs K+R C compiler: The `gcc` driver runs a fresh `cc1`/`cc1plus`/... executable for each source C/C++/... program that needs to be compiled, reading a single input source file, and writing a single assembler output file.

This model has (at least) two big disadvantages:

- Compiling or re-compiling many files is slow. Most obviously there is the the overhead of repeatedly creating a fresh executable. Even more significant is that each included header file has to be re-read from scratch for each main file. This is a big problem especially for C++, and has lead to work-arounds like pre-compiled header files.
- When compiling a source file the compiler has no knowledge of what is in other source files. This limits the opportunities for "cross-module" (or "whole-program") optimization, such as inter-module inlining.

The compile server project improves on these problems as follows:

- The can be more than one input files to a compilation, and they are compiled together to a single output file. It can create tree representation for all the input files, and delay code generation and optimizations such as inlining until it has read all the input files.
- The compiler can be invoked in *server mode*, in which case it enters a loop, waiting for compilation requests. Each request specifies the name of one or more input files to compile, and the name of a requested output assembler file. When the compiler is done with one file, it does some cleaning up, and then waits for the next compilation request.

We will primarily discuss the latter server mode, but multiple-file-compilation is relevant to this discussion because both mechanisms require changing the logic and control flow in the compiler proper.

The compile server compiles multiple source files, without any extra `fork`ing or `exec`ing. This provides some speedup, and so does having to only once initialize tables and built-in declarations. However, the substantial speedup comes from processing each header file only once. The current work concentrates on front-ends that make use of `cpplib`, i.e. the C family of languages. The goal is to achieve performance comparable or better than with pre-compiled headers, but without having to create or manage PCHs. You are also a lot more flexible in terms of order of reading header files. Specifically, the goal is to avoid re-parsing the same header files many time, by re-using the tree nodes over multiple compilations. Similar ideas can benefit other languages (such as Java) that import declarations from external modules (or classes).

This paper describes highly experimental work-in-progress. The current prototype handles C tolerably well, and handles some non-trivial C++ packages.

The compile server (as currently implemented) uses the same working directory and command line flags (such as `-I` and `-D`) for all compilation requests.

1 Invoking the compiler

The `gcc` user-level driver takes a command line with some number of flags, one or more input file names, and optionally an output file name. It uses a fairly complex set of rules to select which other applications it needs to run. One of these is the compiler "proper", which for C is the `cc1` program. The driver executes `cc1` once for each C input file name, creating an assembler file each time. The driver may then invoke the assembler once for each assembly file, creating relocatable binary files, which may then be linked together forming an executable or a shared library.

One part of this project is to change the `gcc` driver so that when it is asked to compile multiple C source files it just call `cc1` once, passing all the input files names to `cc1`. The latter also had to be changed so it could handle multiple input file names, compile them all, and create a single output file. This potentially speeds up compilation time, but more importantly it enables cross-module optimizations such as inter-module inlining.

Handling multiple input files is valuable, but doesn't help much with interactive development, where there are typically many frequent debug-edit-compile cycles. It would speed things up if the compiler could remember state from previous compiles between compilations. Another issue concerns existing `Makefile` scripts, which often use a separate `gcc` command for each source files. Therefore we need an actual *server*, which sits around in the background waiting for compilation requests. We want to use the existing `gcc` command-line interface so we don't have to re-write existing `Makefile`s, except that an environment variable or a single flag will request that `gcc` use a compile server. Then you can just do:

```
make CC='gcc --server'
```

2 Server protocol

The server uses Unix domain sockets to communicate with its clients. Using TCP/IP would be more general, and would be needed for a project where compilations are distributed to different machines. However, there are a number of existing projects and products that do distributed builds, and that is not the focus or goal of this project, so far. (Distributed compilation based on the compile server code may be an interesting future project.)

Unix domain sockets are more efficient that TCP/IP sockets, and are a good match for a non-distributed compile server. Domain sockets are bound to file names in the local file system. Each compilation uses the current working directory to resolve file names, so it makes sense for the server to bind itself to a socket in the current directory. (A future version might be able to change working directories for different compilations.) For the `cc1` compiler, the server listens on a socket bound to `./.cc1-server`.

The server is started by adding `-fserver` to the `cc1`/`cc1plus`/... command line. All options are otherwise as normal, except you leave out the names of the input and output files. The server does a `listen`, and enters a loop using `accept` to wait for connections. For each connection, it enters another loop, waiting for *server commands*. Each command is a single line, starting with a *command letter*. Following the command is a sequence of zero or more quoted string arguments. The quote character can be any byte: using a single quote `"'"` is human readable, but the `gcc` driver uses nul bytes `'\000'` since they cannot appear in arguments. Following the arguments is a newline character that terminates the command.

The following server commands are or will be supported:

F ("flags") Set or reset the command-line flags. (This is not implemented at the time of writing.) It is followed by zero or more nul-terminated flag values, terminated by a newline. Do not use this to set input or output file names. For example:

```
F\0-I/usr/include\0\0-DDEBUG=1\0\0-O2\0\n
```

T ("timeout") Followed by an integer in milliseconds. Sets the time-out duration. If no requests come in during that time, the server exits. If the timeout is 0, the server exits immediately.

I ("invalidate") Followed by a list of nul-terminated filenames. Any cached data for the named files are invalidated. Can be used by an IDE when an include file has been edited. (The server can also `stat` the files, but it may be more efficient to avoid that.)

S ("source") Followed by a file name argument, which is the name of an input source file to compile. There can by multiple `S` commands in a row, in which case all of the input files are compiled, producing a single output file.

O ("output") Followed by a file name argument, which is the name of the output assembler file. The file names from previous `S` commands (since the last `O` command) are all compiled to produce the named output assembler file.

The server currently writes out diagnostics to its standard error, but it should instead send them back to the client using the socket, so the client can write out diagnostics on its standard error.

The client is either the `gcc` command, or some IDE. It could also be an enhanced `make`. It calls `socket`, and then attempts to `connect` to `"./.gcc-server"`. If there is no server running, it starts up a server, and tries again. (This part has not yet been implemented.)

If the `gcc` command is asked to compile multiple source files, it only opens a connection to the server once, and only sends a single `F` command. If a `-o` option is specified (and `-E` is not specified) then (as an optimization) we can have `gcc` do a multi-file compile, specifying a single `O` output file but multiple `S` input files.

3 Initialization

Initializing the compiler is relatively straightforward when compiling a single file. But a server needs three levels of initialization:

1. Initialization that only needs to be done once. For example creating the builtin type nodes, and declaring `__builtin` functions.

2. Initialization that needs to be done for each compilation request (i.e. for each output file). For example opening the assembler output file, and initializing various data structures used by the compiler back-end.

3. Initialization that needs to be done for each input file. For example making available any macros defined with `-D` command-line flags - even if a previous source file `#undef`'d it. Also clearing out any top-level declarations left over from previous source files.

The historical code base has a number of assumptions and dependencies that are no longer appropriate with the compile server. We interface between the language-independent `toplev.c` and the language front-ends uses callback functions that needed some changing: The call-backs and functions that do one-time-only initialization use the word `initially`, while the `init` is used for initialization that is done once per compilation request. For example the modified file `c-common.c` contains both `c_common_initially` and `c_common_init`.

In general, we want to do as much as possible in `initially` functions rather than `init` functions. The obvious reason is to avoid redoing work needlessly, but there is a more important reason: The goal of the compile server is to save and re-use trees across compilations. These will make use of various builtin trees, such as `integer_type_node`. If these builtins get re-defined, then any trees that make use of them will become invalid.

The CPP functions make use of a `cpp_reader` structure that maintains the state of the pre-processor. The global `parse_in` is initialized to a `cpp_reader` instance allocated at `initially`-time. This is important, because the `cpp_reader` maintains a lot of state, including a cache of header file contents, that we want to preserve across compilations. In fact, a simple compile server that only preserves the contents of header files is one option for a less ambitious compile server.

4 Caching text vs tokens vs trees

The fundamental design question for a compile server is what state to save between compilations. Three options come to mind:

- Preserving header file text is easy to implement, especially as `cpplib` already has a cache that does this. We just need to tweak things a little bit. This is especially useful if the OS is slow in handling file lookup, doesn't handle memory-mapped files, or doesn't do a good job buffering files. Otherwise, the benefit should be minor. However, it is a modest change which should be easy to implement.

- We can also preserve the raw token streams in the header files, before macro-expansion. This allows macros that expand differently for different compilations. However, we'd have to use some new data structure for preserving the tokens, and then feed them back to `cpplib`. Any performance advantage over preserving text is likely to be modest, and unlikely to justify the rather radical changes to `cpplib` that would seem to be required.

- Preserving the post-macro-expansion token stream seems more promising. Saving and later replaying the token stream coming out of `cpplib` doesn't appear to be very difficult, and would save the time used for re-reading and re-lexing header files, though it would not save the time spent on parsing and semantic analysis. On the other hand consistency checks and dealing with some of the ugly parts of the languages and the compiler are simpler.

- The best performance gain comes from saving and re-using the tree nodes after parsing and name lookup. This assumes that (normally) a header file consists mainly of declarations (including macro definitions), and the "meaning" of these declarations does not change across compilations. That "meaning" may depend on declarations in other header files, but the "meaning" of those declarations is also constant. (The C++ language specification enshrines something similar in the one-definition rule.)

  Thus if we parse a declaration in a header file, the result normally is a decl node or a macro. Re-parsing the same header file will result in an equivalent decl node or macro. So instead of re-parsing and creating new nodes, we can just skip parsing and re-use the old one.

  A difficultly in re-using trees is determining when it is actually safe and correct to do so, and when we have to re-parse the header file. Another complication is that the compiler modifies and merges trees after-the-fact in various ways. We will discuss these issues below.

  The current prototype takes this approach.

5 Granularity of re-parsing

We say that a header file or portion of one is *parsed* when actual characters are lexed, parsed, and semantic actions performed. A file or portion of one is *re-parsed* when the same text is parsed a second or subsequent time, either because the same file is included multiple times without guards, or because we're processing a new main file. The goal of the compile server is to minimize re-parsing text. Instead, we want to *re-use* a file or portion of a file, which means we want to achieve the semantic effects of parsing (typically creating and adding declarations into the global scope), without actually scanning or parsing the text. We say that we *process* a file or a portion of one to mean either parsing or re-using it.

We will later discuss how we can determine when it is ok to re-use (a part of) a file, and we have to re-parse it, but first let us consider granularity of re-parsing: When we need to re-parse, how much should we re-parse? The following approaches seem possible:

1. Re-read the entire header file. This is conceptually simple, since deciding whether to re-use or re-parse is decided when we see an `#include`. This avoids any complications about managing and seeking to a position within a file. However, this is not a major benefit, given that `cpplib` already caches entire header files, and seeking within a buffer is trivial. The problem with this approach is that handling conditionals within a header file is difficult. We have to decide at the beginning of the file whether any of it is invalid, and whether any conditional compilation directives may "go the other way" compared to when we originally parsed the file. This is doable, but non-trivial. Also, this approach may be excessively conservative, in that we have to invalidate too much.

2. Re-read a header file fragment between any pre-processor directives. Each header file is cached in a buffer. (This is not new with the compile server.) When a header file is re-used, we read from the saved buffer. Pre-processor directives (including conditionals) are handled in the

normal way, by reading from the buffer. However, if the fragment following a directive (or the beginning of file) is valid, we we just restore its declarations, and skip ahead to the next directive (or end of file). This approach has the big advantage that we can use the existing code for evaluating and processing directives. It does have the disadvantage that we have to re-parse and re-evaluate directives, but simplicity and consistency probably is more valuable. There is also a simplification because fragments (unlike header files) don't nest.

This is what is currently implemented.

3. Re-read a header file fragment between conditional compilation directives. It is a refinement of the previous option, except that `#define` (and `#undef`) are treated as part of a fragment, rather than delimiting fragments. A big advantage is that we can re-use macro definitions, without having to re-parse them.

   I think this may be the best approach, but I haven't explored it yet.

4. Re-read just an individual declaration. The problem with this is that we need to maintain some amount of state with each fragment, and the cost goes up if we make the fragments too small: There are usually lots of declarations. The advantage of smaller fragments is that there is less to re-parse when a declaration becomes invalid, which reduces the chance of other declarations becoming invalid. However, we expect that this will not compensate for the extra overheads, so we have not investigated this option.

Using fragments as the unit of re-parsing lets us handle cases like this easily, where we can re-use `D1`, even if we later find out we have to re-parse `D2`:

```
#if M1
D1;
#endif
#if M2
D2
#endif
```

Header guards (as shown below to protected against multiple inclusion) are no problem when using fragments. The processing of the `#include` and the header guard doesn't change when the compile server uses fragments - the only difference is how it handles the body of the header file.

```
#ifndef __H
#define __H
...
#endif
```

6 Entering and exiting fragments

The pre-processer uses callsbacks `enter_fragment` and `exit_fragment` to let the language-front-end know about the start and end of fragments. These are bounds to the functions `cb_enter_fragment` and `cb_exit_fragment` in `c-common.c`.

The preprocessor maintains a cache of header files, including their text. Each header file also gets a `struct cpp_fragment` chain. A new `cpp_fragment` is created whenever `cpplib` starts processing a fragment and there isn't already a `cpp_fragment` for that location. This is done at the start of a header file, and after each preprocessor directive. (We will probably change the code so that `#define` and `#undef` do not delimit fragments.) If the language-specific callback returns non-NULL, then the fragment has to be (re-)parsed normally. The preprocessor remember the returned pointer, and it is passed back on subsequent enter_fragment calls for the same cpp_fragment.

If cb_enter_fragment returns NULL, it means the fragment can be re-used. The preprocesor skips ahead to the end of the fragment, ignoring anything skipped. The cb_enter_fragment will have performed any semantic actions for the fragment, such restoring declarations into the top-level scope.

At the end of a fragment, cpplib calls the exit_fragment callback, which performs any language-specific actions needed if this fragment is a candidate for future re-use. Note that exit_fragment is not called if enter_fragment returned NULL.

7 Dependencies

Before we can re-use a saved fragment, we need to determine if the declarations it *depends on* have changed, When a declaration is parsed, identifiers appearing in it (such as parameter type names) are resolved using other declarations, macros, and other dependencies. So conceptually for each declaration we must remember the set of other declarations and "things" that it depends on. This is the former's *depends-on-set*. A pre-condition for re-using a declaration when compiling a new file is that any declarations it depends on also have been re-used in the new compilation: A depended-on declaration must have been processed, or else it will not be defined, and it must not have been re-parsed, in case that defined the declarations to something new.

Consider a header file h1.h containing:

```
#if M1
typedef int word;
#else
typedef long word;
#endif
/* Define flags,
   which depends on word. */
extern word flags;
```

Assume the first time we #include h1.h, M1 is true, so word and flags are defined. Assume M1 is false the next time we #include h1.h, so we get the other definition of word. Thus the saved definition of flags, which depended on the old definition of word, needs to be invalidated, and we have to re-parse the fragment defining flags.

We can use a conservative approximation of the depends-on set. For example, we can for each header file remember the set of other header files it uses, where a header file uses some other header file if any declaration defined in the former header file uses any declaration in the latter header file. We can also remember dependencies at the level of header file fragments. This is the issue of the granularity of remembered dependencies (which is related to but distinct from the granularity of re-parsing). It actually has two parts: Is a depends-on-set a set of declarations, fragments, or files? How many depends-on-sets do we maintain: One for each declaration, for each fragment, or for each file?

Assuming the granularity of re-parsing is a fragment, then there is no point in maintaining a depends-on-set for each declaration. Instead we maintain a depends-on-set for each fragment, which is the union of the depends-on-sets of the declarations *provided* by the fragment.

In the current implementation the elements of a depends-on-set are fragments: I.e. a fragment has a set of other fragments that provide declarations it depends on. This is an optimization, since there is no point in separately remembering more than one declaration from the same fragment (they will all be valid or all invalid).

(However, there is a case for making the depends-on-set elements be declarations rather than fragments, because we then don't have to map from a declaration to the fragment that provided it. The current implementation adds a field to each declaration that points to the fragment that declared it, and this is wasteful. (We can also get the fragments by mapping back from the declarations line number, but this is slower, even if we change to using the line-map structures.) However, we still need an efficient way to determine if a declaration has been reused. We can do that by looking at the declaration's name, and verifying that the name's global binding is the declaration.)

7.1 Implementation details

For efficiency, a depends-on-set is represented as a vector (currently a `TREE_VEC`, but it could be a raw C array). This is more compact than using a list, but has the complication that we don't know how big an array to allocate. To avoid excess re-allocation, we use a global array `current_fragment_deps_stack` (that we grow if needed) and a global counter `current_fragment_deps_end`, This is used for the depends-on-set of the current fragment. When we get to the end of the fragment in `cb_exit_fragment`, we allocate the fragment's depends-on-set (in the field `uses_fragments`), whose size we now know, filling it from `current_fragment_deps_stack`, and then re-setting `current_fragment_deps_end` to 0.

We need to avoid adding the same fragment multiple times to the current depend-on-set. We do that by setting a bit in the fragment when we save it in `current_fragment_deps_stack`. If the bit is set, we don't need to add it. The bit is cleared when in `cb_exit_fragment` we copy the stack into the fragment's `uses_fragments` field.

A global counter `c_timestamp` is incremented on various occasions, and used as a "clock" for various timestamps. Each fragment has two timestamps: `read_timestamp` is set when the fragment is (re-)parsed, while `include_timestamp` is set whenever the fragment is processed (parsed or re-used). Both are set on `cb_enter_fragment`. We also have a global `main_timestamp` set whenever we starting compiling a new main file. For a fragment `f` to be valid (a candidate for re-use), we require that `f.include_timestamp < main_timestamp`, otherwise the fragment has already been processed in this compilation, and re-processing it is probably an error we want to catch. We also require that for each fragment `u` in `uses_fragments` (the depends-on-set) that all of the following are true:

> `u->include_timestamp >= main_timestamp` (i.e. `u` has been processed in this compilation);
>
> that `u.read_timestamp != 0` (it has been parsed at some point!);
>
> and that `u.read_timestamp <= f.read_timestamp` (the most recent time `u` was parsed was before the most recent time that `f` was parsed—i.e., that `u` hasn't been re-parsed since we last used it).

7.2 Depending on the lack of a definition

One subtle complication concerns *negative dependencies*: Some code may work one way if an identifier has no binding and a different way if it has a binding.

One example (from Geoff Keating): Suppose the tag `struct x` is undefined when this is first seen:

```
// in something.h
extern int do_something (struct x *);
```

This is legal C, but the parameter type is a "local" (and useless) type, different from any global `struct x`. Next, suppose `struct x` has been declared (a forward declaration is enough) the next time this fragment is processed. In that case the parameter type of `do_something` is the global `struct x`, and so the meaning of `do_something` has changed. However, the dependency checking discussed about will not catch this, since the first time `something.h` was included there was nothing for it depend on. This particular problem will cause a warning to be written out the first time, and we can at the same time invalidate the current fragment (disabling future re-use).

However, there may be more complex problems involving negative dependencies, for example involving C++ function overloading.

8 Macro dependencies

The meaning of a fragment may also depend on the definition of macros. Consider the following:

```
char buffer[BUFSIZ];
```

If the macro `BUFSIZ` changes, then the the type of `buffer` is different, so the containing fragment would have to be invalidated.

The implementation does not yet check for macro re-definitions.

Assuming we change the implementation so that macro definitions are part of fragments, and we still store dependencies in terms of fragments depending on other fragments, then we have the basics of what we need. All that would need to be added is that when a macro is used, we note that the current fragment depends on the fragment containing the macro definition.

Which fragments get processed will also depend on macros, but since conditional compilation directives are always re-evaluated, this is not a problem.

8.1 Depending on lack of a macro bindings

We also have the issue of negative dependencies for macros: A fragment will use an identifier, and if later that identifier is bound to a macro, then the fragment will be invalid. Consider a header file `a.h`:

```
extern int i, j;
```

and a header file `b.h`:

```
inline int foo() { return i; }
```

Suppose `file1.c` does this:

```
#include "a.h"
#include "b.h"
```

and `file2.c` does this:

```
#include "a.h"
#define int size_t
#define i j
#include "b.h"
```

In `file1.c` the fragment `b.h` depends on `a.h`, since it used `i`. But the meaning of fragment `b.h` in `file2.c` is very different.

The obvious solution is for every fragment to maintain a set of identifiers that the fragments depends on not being bound to macros, and to check this list on fragment re-use. However, this is quite expensive, as fragments will often use many non-macro identifiers. Below, is a less expensive (unimplemented) solution.

8.2 Checking lack of macro bindings

Here is one solution, that is inexpensive in the common case. For each identifier we add two bits:

```
unsigned used_as_nonmacro : 1;
unsigned also_used_as_macro : 1;
```

When an identifier is referenced, and there is no macro definition for it (i.e. `#define strcmp strcmp` doesn't count), then we set the `used_as_nonmacro` bit. This is permanent—we never reset it.

If an identifier with the `used_as_nonmacro` bit gets `#define`d as a macro, then we also set the `also_used_as_macro` bit (which is also permanent). We also invalidate all fragments. We can do this by setting this global (or field in `cpp_reader`):

```
int first_valid_fragment_timestamp;
```

to `c_timestamp`. This forces all fragments to be re-read the next time they are needed.

If an identifier is referenced, and it has the `also_used_as_macro` bit set, then we add it to a list belonging to the current fragment. Then the next time the fragment is needed, to check validity we check the macro state of identifier on that list.

This implementation has the advantage that the common case is cheap, not requiring any extra state except two bits per identifier. (We also need space for a list header in each fragment, but it may be possible to share with some other list.). However, the rare cases get handled without excessive cost.

9 Saving and restoring bindings

While a fragment is being parsed, each language front-end is responsible for remembering the bindings (declarations etc) that are being created, so they can be restored if the fragment is re-used. The code for this is relatively independent of the rest of the compile server code, so it can be written without understanding the details of the server.

Each binding that needs to be remembered is added to the global `fragment_bindings_stack`, which is (currently) a `TREE_VEC`. How much of the stack is currently used is given by the global `fragment_bindings_end`. There are helper functions `note_fragment_binding_1`, `note_fragment_binding_2`, and `note_fragment_binding_3` to add trees to the stack. What is added is up to the front-end; we'll give examples later. At the end of the fragment, `cb_exit_fragment` will allocate a `TREE_VEC` whose length is `fragment_bindings_end`, assign that to the fragments `bindings` field, and copy that many elements from `fragment_bindings_stack`.

If a fragment is re-used, then `cb_enter_fragment` will call the language-specific function `restore_from_fragment`. This is responsible for going through the `bindings` array and restoring the bindings.

The C language front-end currently does the following:

- `pushdecl` calls `note_fragment_`

binding_1, passing it the declaration that is pushdecl's argument.

- pushtag calls note_fragment_binding_1, passing it the TREE_LIST that is used to link the type into the tag scope. This is called when the tag is declared, including forward declarations.
- finish_struct and finish_enum both call note_fragment_binding_3, passing it the struct/union/enum type, the field list or enum values list, and the type size. This is called when a struct/union/enum tag type is defined.

To restore the bindings when re-using a fragment, the function restore_from_fragment in c-decl.c just loops through the bindings TREE_VEC.

- If the element is a declaration, it set the IDENTIFIER_GLOBAL_VALUE of the declaration's name to point to the declaration, and chains it into the names list of the current_binding_level.
- If the element is a TREE_LIST, we know it was created by pushtag. So we chain it into the tags list of the current_binding_level. We also null out the TYPE_FIELD and TYPE_SIZE fields of the tag type, so don't get complaints if there is a later start_struct. This restores a tag type declaration.
- If the element is a type node, then it must have been created by finish_struct or finish_enum, and must be followed by a fields and a size node. Set the TYPE_FIELDS and the TYPE_SIZE fields of the type to those values. This restores a tag type definition.

10 Modification-in-place of trees

As the compilation proceeds, the compiler sometimes modifies existing declarations. This causes some difficulties. Some examples:

- When the C or C++ front-end sees a declaration with the same name as a previous declaration in the same scope, it calls the function duplicate_decls to compare the old and new declarations. This happens most frequently when the old declaration is a forward or tentative declaration. If the declarations match, duplicate_decls may merge the information from the new declaration into the old declaration, and then discard the new declaration. If the old declaration was in a header file that the compile server re-uses, then it will incorrectly also contain the information from the new declaration.
- In C++ functions may be overloaded. When a new function declaration overloads an older function declaration, the latter is converted to a special overload declaration. When a header file containing that declaration is re-used, we may inadvertently also get overloaded functions that aren't supposed to be visible. This may effect overload resolution, or cause future incorrect error messages.
- A header file may contain a tentative structure declaration (such as struct T), and a different header file may contain a definition of the struct with all the fields. We need to be careful that re-using the former does not re-use the latter. Worse, some C programs may re-use the same structure tag for incompatible types. (This is poor style and rare, but we should at least detect it.)

Most of these merging operations are in practice harmless, or at least will very rarely cause problems, though they may cause some errors to not be properly detected. Sometimes the merging operations can be handled by special code, or it may be possible to "clean up" the compiler to avoid them. However, there are so many places in the compiler that modify older tree nodes that we need a general framework for dealing with them. Such a framework is an *undo buffer*.

Whenever the compiler destructively modifies a tree node that "belongs" to some "other" header fragment, then it needs to append to a global undo buffer enough information to undo the modification. Before starting to compile a new main file, the compiler runs through the undo buffer in inverse order, undoing the remembered modifications. This allows fragment re-use to push the associated declarations without contamination from other fragments.

Implementation of the undo buffer has just started, so I don't know how will it will work in practice, or how much undo information is likely to be needed.

11 Some complications

Various unusual cases cause complications.

11.1 Nested #define inside declarations

On GNU/Linux `<bits/siginfo.h>` contains:

```
enum
{
  SI_ASYNCNL = -6,
# define SI_ASYNCNL    SI_ASYNCNL
  SI_SIGIO,
# define SI_SIGIO      SI_SIGIO
...
  SI_KERNEL = 0x80
#define SI_KERNEL      SI_KERNEL
};
```

This causes a problem if `#define` is the end of a fragment, since then we get a bunch of fragments that are not self-contained. If for some reason some but not all of these fragments get invalidated and have to be re-parsed, then the parser will get very confused!

This particular case is not a problem if we implement the model that `#define` is part of a fragment, rather than delimiting one, as I think we should. Another and more general solution if to invalidate a fragment if it starts or ends not at top level: I.e. nested inside some other declaration or scope. We discuss this next.

11.2 Conditional compilation inside declarations

Many systems (including GNU/Linux and Darwin) have code like the following (in `<netinet/ip.h>`):

```
struct timestamp
  {
    u_int8_t len;
    u_int8_t ptr;
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int flags:4;
    unsigned int overflow:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int overflow:4;
    unsigned int flags:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int32_t data[9];
  };
```

This particular case should not be a problem in practice, since the value of `__BYTE_ORDER` is presumably not going to change. However, it is possible that the first or last fragment might becomes invalidated for some reason, causing the non-conditional parts to get re-parsed. In that case, we need to make sure that the conditional parts also get invalidated and re-parsed. (The converse could also be true, though I don't see how that could happen.)

A general solution uses a `currently_nested` variable. It is incremented when starting a declaration (such as an enum, class, template, or inline function), and decremented when exiting the declaration. If `currently_nested` is positive when either `cb_enter_fragment` or `cb_exit_fragment` is called, then the fragment is invalidated, disabling future re-use.

This should be safe, but not ideal, as `struct timestamp` would be needlessly invalidated. It would be better (though unimplemented) to treat all the fragments that contain a part of `struct timestamp` as a single unit. A *fragment group* is a minimal sequence of fragments in the same header file such that if `currently_nested` is true at the end of one fragments then it and the following fragment are both in the group. A fragment "follows" another if it is the next fragment processed during a single processing of its file. For simplicity, we require that there be no macro definitions or undefinitions within the fragment group. When we parse the fragment group, we remember all the conditionals. We treat the fragment group as a single fragment with a single constructed compound conditional. When we process the group the next time, we evaluate this compound conditional at the start of the group. If it matches, we use the fragments declarations like a normal re-use. If it does not match, we re-parse the fragments as multiple normal fragments.

11.3 Other non-nesting

One common example of non-nesting:

```
#ifdef __cplusplus
extern "{"
#endif
```

This causes the following to nested syntactically. However, we don't want it to cause following fragments to be invalidated!

C++ namespaces.may have similar issues.

11.4 Types defined in multiple locations

The C standard requires that both `<stdio.h>` and `<stdlib.h>` define `size_t`. The trick is to do this without a duplicate definition if both are included. One common solution (used on Darwin and other *BSD system) is to define `size_t` in both headers, but use guards:

```
#ifndef __size_t_defined
#define __size_t_defined
typedef __SIZE_TYPE size_t;
#endif
```

Now suppose `a.c` has

```
#include <stdio.h>
#include <stdlib.h>
```

and `b.c` has:

```
#include <stdlib.h>
#include <stdio.h>
```

In this case the dependencies might prevent us from re-using the cached definition of `size_t`. Worse, definitions that depend on `size_t` also have to be invalidated. Note that this is not a problem of the correctness of the compile server, only its performance.

C++ has a "one-definition rule" that requires that each type declaration etc only a single definition: If different compilation units see different definitions, they must be token-by-token the same. In practice this usually means they are in the same header file, but as in the `size_t` example, that is not strictly required. However, if there are multiple definitions, they will have inconsistent source lines. If you ask an IDE for `size_t`'s definition, it will not be able to give a unique answer. This suggests that a good rule of design is the "extended one-definition rule": There should only be a single definition, at a single location in a unique header file.

The `size_t` definitions violate this extended rule. Therefore, I think the "correct" solution is to fix the headers to not do this. (We can use `fixincludes` to avoid having to change the installed headers, of course.) A simple solution is to create a header `bits/size_t.h`:

```
#ifndef _SIZE_T_H
#define _SIZE_T_H
typedef __SIZE_TYPE size_t;
#endif
```

and then have both `stdio.h` and `stdlib.h` do `#include <bits/size_t.h>`.

There are other solutions possible, but this seems the cleanest and simplest. On GNU/Linux systems using glibc, we have:

```
# define __need_size_t
# define __need_NULL
# include <stddef.h>
```

The magic `__need_size_t_` asks `stddef.h` to define `size_t` and nothing else. This satisfies the "extended one-definition rule", and I don't know any reason why it should cause problems for the compile server. It is a rather complex mechanism, though.

12 Results and conclusions

The compile server has been used to compile sets of related C files (some Apple Carbon files) and C++ (parts of the Octave mathematical library). The preliminary results have been impressive, with speeds-ups of 3x or more. However, there are a number of constructs that are not handled correctly, some planned features (such as the undo buffer) have not been implemented yet, and for some constructs it is not clear what the right solution is. So any detailed performance numbers would be premature and misleading.

Work continues on the compile server, since we at Apple believe it has great long-term potential. The latest patches are available by emailing `<per@bothner.com>`.

Thanks to the members and management of the Apple compiler group (including Ted Goldstein, Ron Price, Mike Stump, and Geoff Keating) for discussions and support of this project.

Fortran 95 support in GCC

Paul Brook
paul@nowt.org

Abstract

This paper details the current status of Fortran 95 language support in GCC, with reference to the future targets and goals of the g95 project. Some of the problems encountered and design decisions made in the process of interfacing with the GCC backend code generator will also be discussed.

1 The Evolution of Fortran

Fortran is a programming language primarily designed for performing computationaly intensive mathematical tasks. Indeed the name itself is derived from the words FORmula TRANslation.

Common uses include Finite Element and Computational Fluid Dynamics codes. Authors of Fortran programs are often not professional software developers. It is commonly used in academic research situations where the primary goal is the analysis and solution of the problem, rather than the development of the software itself.

Fortran was originally implemented by IBM as an alternative to assembly language for programming its 704 systems. The development of the language started in 1954, with a manual published in 1956 (there are rumors that the first customer got a preview compiler without manual in December 1955). The first ISO Fortran Standard was released in 1966. Since then, the standard has undergone four major revisions. These are typically named by the year they were released.

Possibly the most significant changes were introduced in the Fortran 90 standard. Many new features were introduced, with the aim of ensuring the language remained viable for use on modern computing systems.

Fortran 90 introduces powerfull array handling facilities. It allows operations to be performed on whole arrays or sections of arrays in a single expression. From the compiler writer's view this is the most complex feature of the language from, as these must be converted into a collection of scalar operations. It also provides opportunities for the compiler to apply more advanced optimization strategies.

The concept of derived types (analagous to C struct types) was also introduced. While many Fortran vendors had previously provided ways to access and manage dynamically allocated storage areas these were only standardized in the Fortran 90 standard.

As well as these additions to the functional capabilities of language, several other syntactical additions were made. These include modules to aid code modularity and reuse, explicit procedure prototypes, block based flow control constructs and the removal of restrictions on the source form imposed by the use of punch paper cards (so-called Hollerith cards).

Fortran 95 contains mostly minor changes relative to Fortran 90, and removes some of the features that were deprecated with the advent of Fortran 90. However the majority of Fortran 77 code is still legal under Fortran 95 rules.

2 The g95 project

The existing GNU Fortran compiler is widely respected, and a very competent compiler. However this is limited to Fortran 77 code. Even the author of g77 didn't believe that one could make a full Fortran 95 compiler based on the existing g77 code. Writing a new frontend from scratch means g95 is not restricted by design decisions made in g77, and is more easily able to take advantage of new technologies introduced into the common GCC middle- and back-ends.

Thus Andy Vaught created the GNU Fortan 95 project. Initial work concentrated on parsing and correctly resolving Fortran 95 source code.

Only in June 2002, when the parser and resolver were mostly complete, did work begin on the code generation pass and interfacing to the rest of GCC. For this reason g95 is able to correctly parse and verify almost all Fortran code, however it is only able to generate executable code for some of it.

Work is currently concentrated on implementing the few remaining constructs, and completion of the IO and runtime libraries.

Steven Bosscher and I created a fork from the original g95 code in January 2003. This is done in an attempt to achieve closer integration between GCC and g95, and to promote a more open development environment.

3 The Parser and Resolver

Fortran grammar predates most modern parsing techniques. It does not distinguish between keywords and identifiers, and in some cases the meaning of an identifier can only be determined from the way it is used. In other cases the same line of code can have different meanings depending on the context in which it is encountered. It is possibly to write automatically generated parsers for fortran. However these are qute complicated as there is not a clean seperation between lexical, syntactic and semantics analysis. G95 uses a hand crafted pattern matching parser which often operated in a recursive manner.

The majority of error checking and name resolution is done in this first pass. During this process a tree structure is contructed to represent the code. Each statement is represented by a node. These are linked together in lists to form code blocks. These are referenced by flow control statements. For example an IF statement node contains pointers to an expression node for the condition, and expression nodes for the true and ELSE blocks.

Constant folding and simplification of intrinsic functions is also performed while building this tree.

This tree is then traversed in a second pass to perform type checking, insert implicit type conversions where necessary, and to resolve overloaded functions. We also resolve calls to intrinsic function calls to the corresponding runtime library function.

After these two passes, the code tree is fully resolved, and any errors will already have been rejected. The completed tree is passed to the code generation interface one program unit at a time. A program unit is a module, top level subroutine or function, or PROGRAM block.

The first two passes are now almost complete, with legal code being parsed correctly. Most illegal code is detected and rejected, however there are still some constraints which are not enforced.

4 Interfacing to GCC

G95 uses the GCC middle end and back ends to perform code generation and optimization. It is currently targeted at the tree-ssa branch of GCC. This uses a language independant, tree based intermediate representation of the code. This is very similar to the tree produced by the parser, except it can only represents scalar operations.

The GCC tree-ssa branch also provides a cleaner seperation between the language specific fontends and the common backend. Previous versions were still quite closely tied to the C frontend.

The translation of scalar code is mostly straighforward. After some initial setup this is simply a matter of transcribing the tree from one data format to the other. This is done by recursively walking the code tree, building the equivalent GCC tree as this is done.

The main complication is that some expressions require additional code to be associated with them. The solution is to use a state structure when translating expressions. This state structure contains the expression itself, and two code blocks. The pre block contains setup code which must be executed before the expression is evaluated. The post block contains code to clean up after the value is no longer needed.

For the majority of scalar operations both the pre and post blocks will be empty. However Fortran allows more complex operations which may require additional code. One example of this is passing the concatenation of two strings as the actual argument of a function. The pre block will contain code to allocate temporary string storage and perform the concatenation. The expression itself will consist of the function call with the temporary as the actual argument. The post block will contain code to free the temporary storage.

The same state structure is also used to hold information needed for the scalarization of array expressions.

5 Arrays

Modern computer systems employ a one dimensionsal memory space. Higher dimensioned arrays are transformed into this space by multiplying the index by the stride, or spacing, between consecutive elements of the corresponding dimension. These values are summed to obtain the offset of the element relative to the origin of the array. In g95 two pointers are used to manipulate array data. A pointer to the first element of data is required for memory management when allocating and freeing the array data. To access the array a biased base pointer is used. This pointer points to the location of element zero of the array. In this way the array can be accessed without needing to involve the lower bound of the array. It may be the case that element zero of the array does not exist. This does not matter, as it is only used as a base point for the offsets; no non-existing element of the array is ever referenced.

For fully contiguous arrays, where elements of the array are stored in consecutive memory locations, the stride of a dimension is equal to the size of all lower dimensions. This often speeds up access to the array as these values may be known at compile time.

The array descriptors used to pass actual arguments (what C calls “parameters”) consist of a pointer to the first element of the array, the upper and lower bounds and the stride of each dimension. Array pointer variables are handled using the same structure. Array sections are accomodated by calculating the origin and strides to match the section, avoiding the need to make temporary copies of the data.

6 Scalarization

Array expressions introduce significantly complications. The first problem is that of scalarization. The Fortran language allows expressions involving operations on sections of arrays or whole arrays. In practical terms an operation on a whole array is simply a special case of an array section where the bounds of the section are the bounds of the array.

In order to evaluate array expressions it is neccessary to break them down into a set of scalar operations. This is done by generating loops, and using the implicit loop variables as indices into the array sections. The evaluation of array expressions involves several stages and two passes of the expression tree.

First the expression tree is traversed to identify which terms are scalar, and which are arrays. During this pass a list of subexpressions is constructed. Operators whose operands are all scalar result in a single scalar value. These subexpressions will be evaluated outside the scalarization loop, so the operands do not require individual processing. If an operator involves has an array valued result, its operands must be considered by the scalarizer.

The next task is to evaluate the bounds of the implicit loops. The array terms in the expression are examined, and one of these is used to determine the bounds of the scalarization loop. Constant bounds are picked by preference as this gives most potential possibilities for optimization. All the terms in an array expression must have the same shape, so the number of elements in each dimension can be determined from a single term.

For each array term an offset and stride relative to the implicit loop are evaluated. It is not neccessary to evaluate the upper bound of all the array sections, except for runtime error checking purposes.

The main body of the scalarization loop is generated using the same routines as are used for scalar expressions. The translation of the expression is performed in the same order as the initial walking, so only the next term in the list needs to be examined during the translation pass.

Operators which have not been marked as specific subexpressions are translated in the normal way after their operands have been processed. When a scalar subexpression is reached, the precalculated value is substituted.

When array expressions are reached, the implicit loop variables are used to index into the array to get a single scalar value. The offset and scaling factor calculated earlier are used to translate from the loop indices to individual array indices.

A naive implementation of this algoritm would require calculation of the offsets for all array indices on every access. However we traverse higher dimension array sections one dimension at a time. Within the inner scalarization loop the offset due to outer dimensions will be constant. We take advantage of this by calculating this offset before entering the inner scalarization loops.

7 Data Dependencies

The Fortran 95 standard specifies that all values on the right hand side of an assignment statement must be evaluated before any assignments take place. This is known as the "load-before-store" principle. In many cases this restriction has no impact as the source terms of the expression and the target variable are not related. However more care must be taken where both the source and target contain the same elements.

Where the source and target elements are not identically matched, the order in which the assignments are performed may effect the result. In some cases these data dependencies may be resolved by ensuring the assignments are performed in the correct order. In other cases an array temporary is required.

The behaviour of g95 in this area is currently quite simplistic. If any unmatched data dependencies are detected, or the expression is too complex to determine the exact dependencies, an array temporary will be used for the whole assignment. In this case two sets of scalarization loops are generated. The first evaluates the source expressions, and stores the result in a temporary array. The second copies the contents of the temporary array to the target array.

There are many optimization techniques that can be applied in order to reduce the size of the temporary required, and to improve memory access patterns within scalarized assignments. G95 currently only contains a partial implementation of the simpler of these.

8 Intrinsic Functions

Fortran includes many intrinsic functions for performing common mathematical and array operations, as well as operations on data which are impossible to implement using the Fortran language itself. Intrinsic functions and subroutines are implemented with a combination of inline code and runtime library calls.

Where inline code is required the expression state structure is used to hold the code to be execured in order to evaluate the expression.

Most of the required library functions have been implemented. However only the generic versions of there have been written. There is still significant scope for optimized versions to take advantage of simpler cases, processor specific features and more advanced algorithms.

9 IO Library

The IO library is currently one of the least complete parts of g95. Most of the infrastructure for the IO library is in place, as is parsing of format strings. However there is still a significant quantity of work required before this is completed. Formatted IO of integers is possible, however IO of real values is still limited.

10 Incomplete Features

The WHERE and FORALL constructs only work for simple cases where no data dependencies exist.

The WHERE construct performs masked array assignments. These are similar to normal array assignments except a third array expression is used as a mask. Only the assignments where the coresponding element of the mask array is true are preformed.

The FORALL construct allows assignments to be performed for all permutations of a set of loop variables. This is equivalent to enclosing the assignment in multiple DO loops except that "load-before-store" semantics apply to the entire set of assignments. An array expression may be used to mask these assignments. The situation is further complicated by the ability to nest additional FORALL and WHERE constucts inside a FORALL block.

Arrays of character strings are not implemented. Some combinations of derived types and character strings are also incomplete.

Large array constructors used as variable initializers are not implemented. These typically contain large implicit DO loops. The simplest solution is to expand these loops at compile time as we do will small constructors. However this process would consume an unreasonably large amount of CPU time and memory.

The solution is to initialize these variables at runtime.

11 Extensions

There are several extensions to the Fortran 95 standard which we would like to see included in g95. The first seven of these will included in the upcoming Fortran 200x standard.

1. Floating point exception handling
2. Allocatable arrays as structure components, dummy arguments, and function results.
3. Interoperability with the C programming language.
4. Parametrized data types.
5. Derived type I/O.
6. Asynchronous I/O.
7. Procedure variables.
8. OpenMP—provides multi-platform shared-memory parallel programming.
9. Cray pointers—provides functionality similar to C pointers.

12 Calling Conventions

The default behavior of g95 is to pass all actual arguments by reference. In many cases this is neccessary as procedures may be called via implicit interfaces. In this case the worst case calling convention must be assumed.

In some cases, eg. elemental procedures or procedures with assumed shape arguments, an explicit intarface must always be used. For these procedures optimizations such as passing INTENT(IN) parameters by value are possible. Although these optimizations are not currently preformed to simplify debugging, they are likley to be implemented in future revisions.

By default all array arguments are passed using an array descriptor. The advantage of this is that it allows discontiguous array section to be passed without requiring an array temporary. The disadvantage of is that such code will not be binary compatible with Fortran 77 code compiled by g77 or other Fortran compilers. To accomodate this, a compile time option is available to force g95 to use a g77 compatible calling convention. Procedures which use features which were not available in Fortran 77 (eg. POINTER arguments or assumed shape arrays) are still passed using the default calling convention.

While passing discontiguous arrays may reduce the overhead of a procedure call, it introduces a penalty every time the parameter is accessed. This is acceptable if only a small proportion of the passed data is accessed. However if the passed array is heavily used it is beneficial to copy the array data into a contiguous array temporary and access it from there. If the array is INTENT(OUT) or INTENT(INOUT) it may also be neccessary to copy the modified data back to the original array.

The default behavior is to automatically add code to the start of a procedure to test for discontiguous arrays and repack them, as this matches the behaviour of most other Fortran compilers. Users are able to inhibit this behaviour when the cost of repacking the array is likley to exceed the increased cost of accessing the array. For cases where the shape of the array is not known at compile time the data is not repacked when the first dimension is contiguous, as this is unlikley to provide any performance gain.

13 Release dates

The tree-ssa branch of GCC is currently slated for mainline integration in GCC 3.5. The current release date for this, and hence the earliest realistic release date for g95, is late 2004.

G95 only generated its first piece of executable code in June 2002, and significant progress has been made since then. It is hoped that by Q4 2003 g95 will be functionaly complete and standards compliant.

We believe that all the major obstacles to inclusion in the GCC source tree have now been overcome. Inclusion in a non-release branch of GCC is expected in the very near future. It is expected that a seperate parallel development tree will still be maintained for the convenience of developers.

14 Acknowledgments

The g95 project was founded by Andy Vaught, without whom g95 would not exist. He also wrote a large portion of the code, braving the more esoteric aspects of fortran grammar and semantics.

Thanks should also be given to Steven Bosscher, Arnaud Desitter and everyone else who has contributed code, patches, ideas or even just support to the project. Also thanks to g77 maintainer Toon Moene for his assistance and support.

A New Loop Optimizer for GCC

Zdeněk Dvořák
SuSE Labs
`dvorakz@suse.cz, http://atrey.karlin.mff.cuni.cz/˜rakdver/`

Abstract

One of the most important compiler passes is a loop optimization. The GCC's current loop optimizer is outdated and its performance, robustness and extendibility are unsatisfactory. A goal of the project is to replace it with a new better one. In this paper we discuss the design decisions – the choice of used data structures and algorithms, usage and updating of auxiliary information,. . . Then we describe the current state with emphasis on still unsolved problems and outline the possibilities for further continuation of the project, including replacing the remaining parts of the old optimizer and introducing new low-level (RTL based) and high-level (AST based) optimizations.

Introduction

It is generally known that most of the time of programs is spent in a small portion of code ([HP]). Those small but critical areas usually consist of loops, therefore it makes sense to expect the optimizations that directly target loops to have a great effect on program performance. Indeed optimizations to improve the efficiency of scheduling, decrease a loop overhead, optimize memory access patterns and exploit a knowledge about a structure of loops in various other ways were devised; see [BGS] for a survey. Certainly no seriously meant compiler may ignore this. GCC contains a loop optimizer that supports the following optimizations:

- Loop invariant motion that moves invariant computations out of loops.
- Strength reduction, induction variable elimination and various other manipulations with induction variables like fitting into machine addressing modes.
- Doloop optimization, i.e. usage of low overhead loop instructions if a target machine provides them.
- Prefetching of arrays used inside loops to reduce cache miss penalties.
- Unrolling of loops to reduce loop overheads, improve the efficiency of scheduling and increase sequentiality of a code.

We refer to this loop optimizer as the old one in the rest of the paper.

The importance of loop optimizations has been recognized for a long time and the old loop optimizer was added to GCC very early (a copyright notice in the `loop.c` file dates it to 1987). The lack of knowledge about the optimization as well as the lack of computing power lead to several design choices that were unfortunate and today cause the optimizer to be much less powerful than it could be. They also cause other problems concerning its robustness, extendibility and restrictions imposed on the other optimizers. This lead us to decide to replace it by a new one by rewriting some parts, adapting some parts for a new infrastructure and extending it by new important optimizations. We refer to the goal of our efforts as the new loop optimizer in the rest of this paper.

The paper is structured as follows: In the section 1 we investigate the structure of the old loop optimizer and problems with it. In the section 2 we discuss goals of the project to replace it and the high-level design choices of the new loop optimizer. Then we continue by providing the detailed description of the current state of the new loop optimizer, including the changes made in the loop analysis. In the following section 3 we describe used data structures and algorithms to update them. In the section 4 we summarize a status of the project, provide some benchmark results and state our future goals.

1 The Old Loop Optimizer

The loop optimizer was added to GCC very early. Due to the lack of a computing power (and partially also the lack of knowledge) in those times, it has several features that are quite unusual for modern compilers.

Firstly the loop discovery is based on notes passed from the front-end. This approach is very fast, but the considered loops are therefore required to form a contiguous interval in the insn chain and to fit into one of a few special shapes (of course covering all of the most important cases). The loops created by non-loop constructs (gotos, tail recursion, . . .) are not detected at all. Optimization passes before the loop optimizer are required to preserve the shape of loops and the placement of loop notes. Most of them fortunately do not modify control flow graph, but those few that do are complicated and restricted by this need.

Additionally sometimes this information is not updated correctly, therefore it must be verified in the loop optimizer itself and the offending loops are ignored. This makes us miss some more optimization opportunities.

The second problem is the handling of jumps inside loops. The global (not specific to a single pass) control flow graph was introduced into GCC very lately (2000), and the loop optimizer works over the insn chain only. Consequently the effects of branches are estimated mostly by simple heuristics and results of loop invariant and induction variable analyses tend to be overly conservative.

As a side issue, the unroller does not update control flow graph, forcing us to rebuild it. This prevents us to gather a profiling feedback before the loop optimizer, as this information is stored in control flow graph. Therefore we cannot use it in the loop optimizer itself and in the previous passes (most notably GCSE and loop header duplication).

The unroller uses its own routines to copy the insn stream, creating an unnecessary code duplication with the other parts of the compiler.

Any single of these problems could probably be addressed separately by modifying the relevant code. Considering them together it seems to be easier to write most of the optimizer again from scratch. Some parts can just be adapted for a new infrastructure (the decision heuristics and execution parts of the invariant motion and induction variable optimizations, the whole doloop optimization pass), but the greatest part has too deeply inbuilt expectations about a loop shape with respect to the insn chain to be usable. We discuss the plans concerning this rewrite in more detail in the following sections.

The source of other complications is the low level of RTL. During the translation to RTL, some of the information about possibility to overflow and types of the registers is lost and we are forced to either rediscover it through nontrivial analysis, use conservative heuristics, produce a suboptimal code containing unnecessary overflow checks or produce a possibly incorrect code. None of these options is particularly good. It would also make dependency analysis quite complicated – it is not present in GCC yet, and the optimizations that require it (the loop reorganization, the loop fusion, ...) are missing. While the current project is mostly RTL based, it will be necessary to address these issues in near future. There are already some efforts for moving the relevant parts of the loop optimizer to the AST level in progress; for more information see section 4.

2 Overview of The New Loop Optimizer

There are several basic principles we have decided to follow:

- The passes that form the loop optimizer should be completely independent on each other. They must preserve the common data structures and it should be possible to run them any number of times and in any order (although of course not all orders are equally effective). This approach is completely different from the old loop optimizer one – there the optimizers called each other in non-transparent manner and most of them had assumptions about information gathered by the other ones. While this approach may be slightly more efficient and perhaps simpler at some places with respect to keeping the information up to date during transformations, we prefer our approach due to its cleanness, extendibility and robustness. We have also initially made some parts of the optimizer quite simplistic, and this approach enables us to replace them later by more involved solutions without greater problems.

- We have decided to generally reuse as much of the existing code as possible and eventually extend it for our purposes, rather than creating our own variations of the existing code. Most importantly we used the `cfglayout.c` code for duplicating basic blocks (this should replace two instances of a similar code, one in `unroll.c` and the other one in `jump.c`) and of course the existing `cfgloop.c` code for a loop analysis (after significant changes described below). We are also currently using the code from `simplify-rtx.c` when computing a number of iterations of a loop. In this case we were unfortunately forced to start working on an alternative RTL simplification code for this purpose. The reason is that the goal of `simplify-rtx.c` is in some sense opposite to what we would need. While we need to simplify the RTL expressions into a simple canonical shape, `simplify-rtx.c` code tries to transform it so that it is efficiently computable. Some of the manipulations it does for this purpose (expressing multiplication through shifts) make it unsuitable for our needs, and some conversion we need to do (using distributive law on products of sums) make the resulting code possibly much less efficient than the original one. The two approaches do not seem to fit together very well.

- As much of the information as possible should be kept up to date at any given time. This concerns mostly complicated operations over loops (unrolling, unswitching, ...), where we express them as a composition of simpler operations that preserve the consistent state rather than making them at once and updating the structures afterwards. This makes the code a bit slower, but much easier to understand and debug (many bugs were caught early due to a possibility to check a consistency after every step).

The optimizer itself consists of the initialization, several optimization passes and the finalization. The finalization part is trivial, just releasing the allocated structures. In the following paragraphs we examine the remaining phases in a greater detail.

The initialization and finalization parts are placed in `loop-init.c`. During the initialization, we calculate the following information (that is kept up to date till the finalization):

- A dominator relation is computed. The dominators are used to define and find natural loops and we use them during loop transformations for several purposes, most importantly during the simple loop analysis to determine expressions (conditions) that are executed (tested) in every iteration of the loop. Also we need them to be able to update the loop structure when parts of the code are removed. The decision to keep the dominator relation always up to date turned out to be somewhat disputable. Having them ready at all times is convenient and makes the parts where they are used quite simple, but updating them is relatively non-trivial and quite costly. Most of their usages would be simple to replace without using them at a little extra cost, but their usage during the removal of a code seems to be crucial.
- Natural loops are found. The natural loop is defined as a part of a control flow graph that is dominated by the loop's header block and backreachable from one of the edges entering the header, called the latch

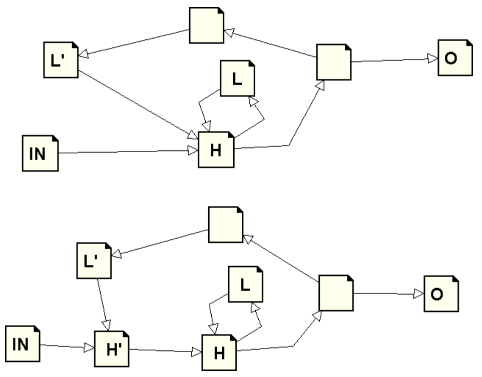

Figure 1: Creating nested loops from loops with shared header.

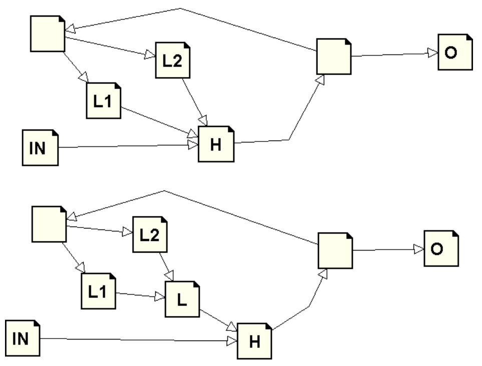

Figure 2: Merging loops with a shared header.

edge. Note that this definition makes it possible for several loops to share the same header block. We do not want to have to handle them specially, so we split the loop header in this case. There are two ways to split the header (figures 1 and 2) – one of them merges the loops together, while the other one creates the nested loops. It is impossible to recognize which of these cases matches the reality just from a control flow graph, and even looking at the source code does not help too much (this kind of loops is often created by continue statements, and it is hard to recognize what behavior describes this situation better). If we have a profile feedback available, we use it to determine whether one of the latch edges is much more frequent than the other ones, i.e. if it behaves like an inner loop, and create the inner loop in this case (this is sometimes called the commando loop optimization). Otherwise we just merge the loops.

- `cfg_layout_initialize` is called to bring the instruction chain into a shape that is more suitable for the transformations. This function removes the unconditional jumps from the instruction stream (the information about them is already included in the control flow graph) and makes it possible to reorganize and manipulate basic blocks in much easier manner.

- Loops are canonicalized so that they have simple preheaders and latches. By this we mean that:

  – Every loop has just a single entry edge and the source of this entry edge has exactly one successor.

  – The source of latch edge has exactly one successor.

  This makes moving a code out of the loop easier, as there is exactly one place where it must be put to (the preheader) and we can put it there without a fear that it would be executed if we do not enter the loop. It also removes the singular case of a loop that consists of just one block. A quite important fact is that the loop latch must now belong directly to the loop (i.e. it cannot belong to any subloop) and the preheader belongs directly to the immediate superloop of the loop (it could belong to a sibling loop if it had more than one successor).

- The irreducible regions are marked. A region of a control flow graph is considered irreducible if it is strongly connected and has more than one entry block (i.e. it contains a depth first search back edge, but the destination block of this edge does not dominate its source, so the region fails to be a natural loop). The irreducible regions are quite infrequent (it is impossible to create them in structured languages without use of a goto statement or a help of the compiler), but we must be able to handle them somehow. In the new loop optimizer they are mostly ignored, just taking them into account during various analyses. The information about them is quite easy to keep up to date unless we affect their structure significantly. This may occur in very rare cases during the unswitching or the complete unrolling. In some of these cases we have resigned on updating the information and rather recompute them from scratch – it is quite fast (just a depth first search over a control flow graph) and much less error prone than to attempt to handle the case that we would not be able to test properly (it is almost impossible to construct a suitable testcase).

The optimization passes are placed in separate files. The currently available optimization passes are:

- Loop unswitching (in `loop-unswitch.c`) – if there is a condition inside a loop that is invariant, we may create a duplicate of the loop, put a copy of the condition in front of the loop and its duplicate that chooses the appropriate loop and optimize the loop bodies using the knowledge of a result of this condition. There are a few points worth the attention. The first is a code growth – if there is a loop with k unswitchable conditions, we end up with 2^k duplicates of the loop. This is not really a problem in practice – the opportunities for unswitching are rare. Also in most of the cases when we have more than one unswitchable condition per loop the values tested in them are identical and they are therefore eliminated already during the first unswitching. (Just for sure the number of unswitchings per loop is limited). The other is testing for invariantness of the condition. As the new loop optimizer is placed after GCSE (and also the old loop optimizer's invariant motion), it is sufficient to just test that the arguments of the condition are not modified anywhere inside the loop.

- Loop unrolling and loop peeling (placed in `loop-unroll.c`). While it would correspond more to our philosophy to have this pass split into several ones, the code and computation sharing between them is so large that it would be impractical. Anyway they are still completely independent and they could be split with a little effort. We perform the following optimizations:

  – Elimination of loops that do not roll at all – this is somewhat exceptional, as this does not increase code size (in fact it decreases it). For this reason we perform this transformation even for non-innermost loops, unlike the other ones.

  – Complete unrolling of loops that iterate a small constant number of times (a loop is eliminated in this case too, but at the cost of a code size growth).

  – Unrolling loops with a constant number of iterations—we may peel a few iterations of the loop and thus ensure that the loop may only exit in a specified copy, therefore enabling us to remove now useless exit tests. For most of the loops we leave the exit in the last copy of the loop body—the exit is usually placed at the end of loop body, and all copies may be merged into a single block in this case. In the rare cases when this is not true we leave the exit in the first copy—in this case it is a bit easier to handle loops of a form `for (i=a; i < a+100; i++)`, where the number of iterations may be either 100 or 0 (in the case of an overflow).

  – Unrolling loops for that the number of iterations may be determined in runtime – the situation is similar here, except that the number of iterations to perform before entering the unrolled loop body must be determined in runtime. The number of iterations to be performed is chosen through a switch statement-like code.

  According to some sources ([DJ]), in both of these cases it is preferable to place the extra iterations after the loop instead due to a better alignment of data (this might also be important if we were doing autovectorisation). This can only be done if the loop has just a single exit and modifications of the loop are more complicated. Also handling of overflows and other degenerate cases becomes much harder. It could however be done for constant time iterating loops with a little effort.

- Unrolling of all remaining loops – this transformation is a bit controversial. The gains tend not to be large (scheduling may be improved and rarely some computations from two consecutive iterations may be combined together), and sometimes we even lose efficiency (due to negative effects of a code growth to instruction caches and an increased number of branches to branch prediction). We only do this if specifically asked to, and even then only if the loop consists of just a single basic block.
- Loop peeling – the situation is similar (additionally we hope that the information about initial values of registers can be used to optimize the few first iterations specially). We gain most for loops that do not iterate too much (optimally we should not even enter the loop). To verify this, we use a profile feedback and therefore perform this transformation only if it is present.

As was already mentioned, we perform these transforms on innermost loops only. This is not a principal restriction (the passes are written so that they handle subloops), but the ratio of a code size growth to a performance gain is bad then, and also duplicated subloops would be more difficult for branch prediction in processors.

The old loop unroller also performs the induction variable splitting to remove long dependency chains created by unrolling that negatively impact scheduling and other optimization passes. We instead leave this work to the webizer pass that is much more general.

There are three basic problems to solve. Firstly there is the code growth. All of the unrolling-type transformations naturally increase a code size. While the greater number of unrollings generally increases effect of the optimization, it also increases a pressure on code caches. It is therefore important to limit the code growth. There are adjustable thresholds that limit the size of resulting loops as well as the maximal number of unrollings. We also use a profile feedback to optimize only relevant parts and try to limit transformations for that gains are questionable in cases when we believe that they might spoil the code instead (for example the loop peeling is not performed without a profile feedback that would suggest that the loop does not roll too much).

A more appropriate solution might be a loop rerolling pass run after scheduling that would revert the effects of a loop unrolling in case we were not able to get any benefits from it.

Secondly we need the analysis to determine a number of the loop's iterations. Currently we use a simplistic analysis that for each exit from the loop that dominates the latch (i.e. is executed in every iteration) checks whether the exit condition is suitable – i.e. if it is comparison where one of the operands is invariant inside the loop and the other one is set at exactly one instruction that is executed exactly once per loop iteration. For such condition we then check whether the variable is increased by constant and attempt to find its initial variable in an extended preheader of the loop (i.e. basic blocks that necessarily had to be executed before entering the loop). Using the simplification machinery from `simplify-rtx.c` we then determine the number of iterations. This turns out to be sufficient in most cases, but things like multiple increases of the induction variable prevents us from detecting the variable. Also often the initial value of the variable is assigned to it earlier, preventing us from recognizing the loop as iterating a constant number of times. Induction variables that iterate in a mode that is narrower than their natural mode are not handled, which causes problems on some of the 64 bit architectures where int type is represented this way. We are currently working on the full induction variables analysis that solves all of these problems.

Thirdly we must decide how much we want to unroll the loop. Currently we take into account just a code growth, thus we unroll the bigger loops less times. For constant times iterating loops we also attempt to adjust the number of unrollings so that the total size of the code is minimal. In other cases we use the heuristic that says that it is good to unroll number of times that is a power of two (because of better alignments and other factors). See the section 4 for discussion of the possible extensions of this scheme and the estimation of gains obtainable by using some better methods.

- Doloop optimization – this pass is just an adaptation of the old loop optimizer's doloop pass that was written by Michael Hayes. The structure of the pass was quite clear and there were no major problems with the transfer. This adaptation of the pass is still only present on rtlopt-branch, due to a bad interaction with the new loop optimizer. This is caused by a overly simplistic induction variable analysis used and should be solved by the improved induction variable analysis that is currently being written.

3 The Data Structures

In this section, we discuss the structure to represent the loops as well as other auxiliary data structures used in the new loop optimizer. We also describe the algorithms used to update them.

We consider a loop A a *subloop* of a loop B if a set of basic blocks inside the loop A is a strict subset of a set of basic blocks inside the loop B. Because we have eliminated the loops with shared headers, the Hasse diagram of a partial ordering of loops by the subset relation is a forest. To make some of the algorithms more consistent, we add an artificial root loop consisting of the whole function body (with an entry block as a header and an exit block as a latch). We maintain this loop tree explicitly. For each of the nodes of the tree we remember the corresponding loop's header and latch. But we do not remember the set of basic blocks that belong to it – if we need to enumerate the whole loop body, we use a simple backward depth first search from its latch, stopping at its header.

To be able to test for the membership of a basic block to the loop we maintain the information about the innermost loop that each basic block belongs to. To speed up the testing for a not necessarily immediate membership to a loop (i.e. including membership to any subloop of the loop), we also maintain the depth in the loop tree and an array of parents for each node in the loop tree. Maintaining the arrays of parents enables us to respond to these queries in a constant time, but makes speed of all updates proportional to the depth of the tree. This works well in the practice, as the tree is usually quite shallow and the structure of a loop tree does not change very often.

Updating the loop tree is straightforward during the control flow graph transformations we use. Most of the optimizations do not change the structure at all. The exception is the loop unrolling and peeling type transformations if some subloops are duplicated (it cannot really occur just now because we optimize only innermost loops, but the code can handle this situation for case we changed this decision) or the unrolled loop is removed, but all of these cases are easy to handle. Note that some of them may create new loops if irreducible regions are present. We ignore these newly created loops (still having them marked as irreducible) – this is conservatively correct and this situation is so rare that it does not deserve any other special handling.

As described in the previous section, there are two further pieces of information we keep up-to-date – the dominators of basic blocks and the information about irreducible regions.

We represent the dominators as the in-branching of immediate dominators. We represent this in-branching using ET-trees. This structure was chosen due to its flexibility – it enables us to perform all relevant operations asymptotically fast (updates and queries for dominance in logarithmic time, finding the nearest common dominator of a set of blocks and enumerating all blocks that are immediately dominated by a given block in a time proportional to the size of the relevant set times a polylogarithmic time). The multiplicative constants however turned out to be quite high and we are considering replacing the structure by some perhaps less theoretically nice but more practical (e.g. a depth first search numbering with holes).

Updating of the dominators in general is not easy. During the transformations we perform we are usually able to handle it by using the fact that they are of a special kind (respecting the loop structure that itself reflects the structure of dominators). In a small portion of cases when we are not able to do it (or the rules to determine how the dominators change would be too complicated) we use a simple iterative approach (similar to [PM]) to update the dominators in the (usually) small set of basic blocks where they could be affected. As already mentioned before, we also consider not keeping the dominators at all and solving the cases when they are currently used without them.

The irreducible regions are determined as strongly connected components of a slightly altered control flow graph. For each loop we create a fake node. Entry edges of the loops are redirected to these nodes, exit edges are redirected to lead from them – this ensures that the parts of the irreducible regions that pass through some subloop are taken into account only in the outer loop. We remember this information through flag placed on the edges that are a part of those strongly connected components. This is sufficient to update the information effectively during the most of the control flow graph transformations. The only difficult case is when a loop that is a part of an irreducible area is removed. We would have to propagate the information about irreducibility through the remnant of its body then. While it could be done, it would be quite difficult to handle all problems (subloops, other irreducible regions). Instead, we simply remark all irreducible regions using the algorithm described above (this situation is quite rare and the algorithm is sufficiently fast anyway).

Benchmarks	Base Ref Time	Base Run Time	Estimated Base Ratio	Peak Ref Time	Peak Run Time	Estimated Peak Ratio
164.gzip	1400	306	458	1400	291	480*
175.vpr	1400	452	310	1400	452	310*
176.gcc	1100	306	360	1100	299	368*
181.mcf	1800	821	219	1800	815	221*
186.crafty	1000	174	574	1000	174	575*
197.parser	1800	534	337	1800	534	337*
252.eon	1300	201	648	1300	199	652*
253.perlbmk	1800	338	533	1800	335	538*
254.gap	1100	280	393	1100	277	398*
255.vortex	1900	414	459	1900	410	464*
256.bzip2	1500	431	348	1500	428	351*
300.twolf	3000	902	333	3000	878	342*
Est. SPECint_base2000			398			
Est. SPECint2000						403

Base flags: -O2 -march=athlon -malign-double -fold-unroll-loops
Peak flags: -O2 -march=athlon -malign-double -funroll-loops

Figure 3: SPECint2000 results for rtlopt-branch on Athlon, 1.7 GHz

Benchmarks	Base Ref Time	Base Run Time	Estimated Base Ratio	Peak Ref Time	Peak Run Time	Estimated Peak Ratio
164.gzip	1400	621	225	1400	605	232
175.vpr	1400	857	163	1400	854	164
176.gcc	1100	624	176	1100	618	178
181.mcf	1800	1354	133	1800	1361	132
186.crafty	1000	285	350	1000	275	364
197.parser	1800	930	194	1800	932	193
252.eon	1300	321	405	1300	331	393
253.perlbmk	1800	538	335	1800	556	324
254.gap	1100	426	258	1100	420	262
255.vortex	1900	817	233	1900	810	235
256.bzip2	1500	770	195	1500	774	194
300.twolf	3000	1709	176	3000	1699	177
Est. SPECint_base2000			224			
Est. SPECint2000						225

Base flags: -O2 -march=athlon -fold-unroll-loops
Peak flags: -O2 -march=athlon -funroll-loops

Figure 4: SPECint2000 results for mainline on Duron, 800 MHz

4 The Current State And Further Plans

Everything described above in the paper (except for the doloop optimizer adaptation) is already merged in the GCC mainline and will be in GCC 3.4. The new loop unroller in connection with webizer and other improvements present on rtlopt-branch outperforms the old one on i686 and even without the webizer the results are comparable (see figures 3 and 4 for results on SPECint2000 testsuite). Its simple procedure to count the number of iterations beats the old loop optimizer's one (it detects 52 loops as iterating a constant number of times on the gap benchmark compilation as opposed to 39 loops the old loop optimizer did). The total number of loops detected is a bit surprisingly almost the same – 3292 by the old loop optimizer, 3298 by the new one – writers of GCC have apparently done very good job in keeping the front-end information about the loops accurate.

We are still quite far from our final goal – fully replacing and removing the old loop optimizer. What remains is to replace or adapt induction variable optimizations (the invariant motion can be solved by GCSE instead) and to solve the problems described below.

While the results from i686 look quite promising, the new loop optimizer has problems on the other architectures. Some of the 64-bit architectures must represent 32-bit integers as subregs of 64-bit registers. The simplistic analysis to determine a number of iterations of the loops is not yet able to handle this case, so the unroller is useless here. This should be solved by introducing the new induction analysis that is needed to replace the induction variable optimization parts anyway.

On some other architectures quite important performance regressions were reported. They might be partially caused by absence of the webizer pass in mainline. We are currently investigating other reasons.

The interesting problem with the new loop unroller is determining whether and how much we should unroll or peel a given loop. There are several possible criterion:

- To decide whether to optimize at all, we use a profile feedback. Not optimizing in cold areas reduces the code growth a lot. To decide whether to peel or to unroll, we try to estimate the number of iterations of a loop using the feedback and to peel a sufficient number of iterations from a loop so that the loop is not entered at all most of the times. We also measure histograms of first few iterations of the loops and use it to determine this more precisely on rtlopt-branch, but the effects are not significant.

- The effects on instruction cache seem to be quite important. There are some works describing how to take them into account ([HBK]), but they would require a global program analysis and it seems questionable whether they would be useful at all. For now we cannot do anything but to attempt to limit the code size growth.

  Similarly duplication of loops whose bodies contain many branches may also affect the performance negatively, as the created jumps increase the pressure on the CPU's branch prediction mechanisms. Sometimes these jumps may also may behave less predictably than the original ones.

- From a scheduling point of view, it would make sense to prefer unrolling loops that contain instructions with long latencies. It might also be useful to take a register allocation into account, attempting to minimize the number of registers needed for computing simple recurrences.

Currently we use only a very simple heuristics to take some of the effects mentioned above into account. To estimate the possible gains of using better methods, we wrote a code that attempts to determine the best possible number of unrollings for each of the loops. It adds a code for each of the loops that measures the total time spent inside it. Then for i between 1 and some upper bound, we unroll all loops i times and gather the profiling data. Finally we choose the best of these times for every loop as the right number of iterations to unroll. This is far from optimal (the added profiling code changes the performance characteristics of a compiled program a lot and the optimal numbers are also dependent on how other loops are unrolled, so measuring them when all are unrolled the same number of times is not completely right), still we achieved about 2% speedup on SPEC2000 this way on i686.

Adapting the rest of old loop optimizer seems to be quite straightforward now. New induction variable analysis pass is just being tested on rtlopt-branch, the next step is either to use it to produce induction variable descriptions suitable for the old induction variable optimization pass, or (more likely) to write a new one, heavily reusing the parts of the old one.

There are additional loop optimizations that should be added to GCC, including

- loop reorganization that makes accesses to arrays more sequential by swapping an order of nested loops if possible.
- loop fusion that joins adjacent loops that iterate the same number of times (perhaps after a small adjustment), to reduce an overhead of loop creating instructions.
- loop splitting that inversely splits the loops into several smaller ones if we know that we are able to optimize them better this way.
- autovectorisation, i.e. usage of SIMD instructions on arrays processed in loops.

All of those (and several other less important) optimizations require a dependency analysis to determine whether it is indeed possible to reorganize computations as needed. It would be pretty painful to determine this on the RTL level, as information about types of variables is almost lost here (partially recoverable only through a complicated analysis) and so is some of the information about overflows. Also the loop reorganization needed would be quite complicated on the RTL level. This makes them more suitable for the AST level. We hope to be able to start a work on them in a few months.

Other optimizations should be better done on AST level from similar reasons, including a part of induction variable optimizations that does not use a machine specific information (like a knowledge of addressing modes etc.) and possibly unrolling and unswitching. There are already some efforts for moving the relevant parts of the loop optimizer to the AST level in progress (Pop Sébastian have recently altered the loop recognition code to work both on RTL and AST levels).

Acknowledgments

The project is based on the GCC code written by hundreds of volunteers. The most of contributions to the loop analysis code we have built upon were by Daniel Berlin, Michael Hayes and Michael Matz. Part of the new loop optimizer was written during the “Infrastruktura pro profilem řízené optimalizace v GCC” (“Infrastructure for profile driven optimizations in GCC”) software project at Charles University, Prague ([DHNZ], [DHNZ-doc]) together with Josef Zlomek, Pavel Nejedlý and Jan Hubička under leadership of David Bednárek and later continued with support of Suse Labs. I would also like to thank Richard Henderson for providing a useful feedback during the merging of the new loop optimizer to the mainline.

References

[BGS] David F. Bacon, Susan L. Graham and Oliver J. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys 26 (1994) p. 345–420.

[DJ] Jack W. Davidson and Sanjay Jinturkar, *An Aggressive Approach to Loop Unrolling*, Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville, June 1995.

[DHNZ] *The "Infrastruktura pro profilem řízené optimalizace v GCC" project specification*, `http://ksvi.mff.cuni.cz/~holan/SWP/zadani/gccopt.txt`

[DHNZ-doc] *The "Infrastruktura pro profilem řízené optimalizace v GCC" project documentation*, `http://atrey.karlin.mff.cuni.cz/~rakdver/projekt/`

[HBK] K. Heydemann, F. Bodin, P. Knijnenburg, *Global Trade-off between Code Size and Performance for Loop Unrolling on VLIW Architectures*, Publication Interne 1390, IRISA, Institut de Recherche en Informatique et Syst'emes Al'eatoires, March 2001.

[HP] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc, San Matea, CA, 1990.

[PM] Paul W. Purdom, Jr. , Edward F. Moore, *Immediate predominators in a directed graph*, Communications of the ACM, v.15 n.8, p.777-778, Aug. 1972

Mudflap: Pointer Use Checking for C/C++

Frank Ch. Eigler
Red Hat
`fche@redhat.com`

Abstract

Mudflap is a pointer use checking technology based on compile-time instrumentation. It transparently adds protective code to a variety of potentially unsafe C/C++ constructs that detect actual erroneous uses at run time. The class of errors detected includes the most common and annoying types: NULL pointer dereferencing, running off the ends of buffers and strings, leaking memory. Mudflap has heuristics that allow some degree of checking even if only a subset of a program's object modules are instrumented.

1 Motivation

C, and to a lesser extent C++, are sometimes jovially referred to as a "portable assembly language." This means that they are portable across platforms, but are low level enough to comfortably deal with hardware and raw bits in memory. This makes them particularly suited for writing systems software such as operating systems, databases, network servers, and data/language processors. These types of software are notorious for pointer-based data structures and algorithms, which C/C++ make easy to express. However, the runtime model of C/C++ does not include any checking of pointer use, so errors can easily creep in.

Several kinds of pointer use errors are widely known by every C/C++ programmer. Accessing freed objects, going past buffer boundaries, dereferencing `NULL` or other bad pointers, can each result in a spectrum of effects, from nothing, through random glitches and outright crashes, to security breaches. Such bugs can induce hard-to-debug delayed failures. Many recent security vulnerabilities of operating systems result from simple stack-smashing *buffer overrun* errors, where pointers go beyond their bounds to corrupt memory, under the influence of malevolent input.

There exist several technologies for catching pointer use errors. They have distinct approaches and capability/performance tradeoffs. For example, from a debugging point of view, it is better to catch the memory corruption at the moment it occurs, because context will be fresh and available. On the other hand, for security protection of a deployed program, it may be enough to catch an error just in time to prevent a breach, which might be much later.

A large class of pointer use errors relates to heap allocation. Writing past the end of a heap object, or accessing a pointer after a `free` can sometimes be detected with nothing more than a library that replaces the standard library's heap functions (`malloc`, `free`, etc.). The Electric Fence[1] package, for example, can manage heap objects that carefully abut inac-

[1]`ftp://ftp.perens.com/pub/ElectricFence/`

cessible virtual memory pages. A buffer overrun there causes an instant segmentation fault. Some libraries provide a protected padding area around buffers. This padding is filled with code that can be periodically checked for changes, so program errors can be detected at a coarser granularity.

The bounded-pointers GCC extension[2] addresses pointer errors by replacing simple pointers with a three-word struct that also contains the legal bounds for that pointer. This changes the system ABI, making it necessary to recompile the entire application. The bounds are computed upon assignment from the address-of operator, and constructed for system calls within an instrumented version of the standard library. Each use of the pointer is quickly checked against its own bounds, and the application aborts upon a violation. Because there is no database of live objects, an instrumented program can offer no extra information to help debug the problem.

The gcc-checker extension[3] addresses pointer errors by mapping all pointer manipulation operations, and all variable lifetime scopes, to calls into a runtime library. In this scheme, the instrumentation is heavy-weight, but the pointer representation in memory remains ABI-compatible. It may be possible to detect the moment a pointer becomes invalid (say, through a bad assignment or increment), before it is ever used to access memory.

The StackGuard GCC extension[4] addresses stack smashing attacks via buffer overruns. It does this by instrumenting a function to rearrange its stack frame, so that arrays are placed away from vulnerable values like return addresses. Further guard padding is added around arrays and is checked before a function returns. This is light-weight and reasonably effective, but provides no general protection for pointer errors or debugging assistance.

The valgrind package[5] is a simulation-based tool for detecting a broad class of pointer use errors. It contains a virtual machine that tracks processor operations, including memory loads and stores and even register arithmetic, to check operations for validity. While it works on unmodified executables, this simulation process is quite slow.

The Purify package[6] is a well-known proprietary package for detecting memory errors. Purify works by batch instrumentation of object files, reverse-engineering patterns in object code that represent compiled pointer operations, and replacing them with a mixture of inline code and calls into a runtime library.

2 How Mudflap Works

Mudflap works by inserting a pass into GCC's normal processing sequence. It comes after a language frontend, and before the optimizers, RTL expanders, and backend. It takes a restricted form of GCC *trees*, which are similar to abstract syntax parse trees, as input. It looks for tree nesting patterns that correspond to the potentially unsafe source-level pointer operations. These constructs are replaced with expressions that normally evaluate to the same value, but include parts that refer to libmudflap, the mudflap runtime. The compiler also adds instrumentation code associated with some variable declarations.

The purpose of this instrumentation is to assert a validity predicate at the use (dereferencing) of a pointer. The predicate is simply whether

[2] `http://gcc.gnu.org/projects/bp/main.html`

[3] `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`

[4] `http://immunix.org/stackguard.html`

[5] `http://developer.kde.org/~sewardj/`

[6] `http://www.rational.com/products/purify_unix/`

or not the memory region being referenced is recognized by the runtime as legal. If not, a violation is detected.

2.1 Object database

As a prerequisite for evaluating memory accesses, the runtime needs to maintain a database of valid memory objects. This database includes several bits of information about each object, which may be retained for some time even after it is deallocated.

- address range
- name, declaration source file, line number
- storage type (stack/heap/static/other)
- access statistics
- allocation timestamp and stack backtrace
- deallocation timestamp and stack backtrace

In order to update and search the object database, libmudflap exports a number of functions to be called by the inserted instrumentation. These basic ones include one to assert that a given access is valid, and a pair to add/remove a memory object to/from the database. These functions are passed pointer and size pairs plus some parameters to classify or decorate accesses and objects. For example, for a stack-based variable that may have pointer-based accesses would have a "register" call at the point of entry into its scope, and a corresponding "unregister" call when control leaves the scope. For heap-based variable, these calls would be performed within hooked allocation and deallocation primitives. For static variables, register calls are done early during program startup, and the effort of an unregister is not wasted during shutdown.

The database happens to be stored as a binary tree, naturally ordered based on the addresses of live objects. Its internal nodes are periodically rotated in order to move nodes for popular objects nearer to the root. There is a separate fixed-size array listing recently deallocated objects, used only during the performance-insensitive processing of violation messages.

During program startup, some selected objects are inserted into the database as special *no-access* regions. These represent address ranges that are certainly out-of-bounds for all instrumented programs, like the `NULL` area, and some libmudflap internal variables. Violations are always signaled when such objects are accessed by instrumented code.

2.2 Lookup cache

In order to hasten the database lookup, something which needs to be done many times, libmudflap maintains a *lookup cache*. This cache is compact global direct-mapped array, indexed using a simple hash function of the pointer value. Each entry in the cache specifies a memory address range that is currently valid to access. If an inline check of the cache for a given access is successful, the application avoids the call into libmudflap for the full-blown checking routine. Though the code may look complicated, it compiles down to a surprisingly small number of instructions.

```
/* uintptr_t: an integral type for
   pointers */
struct __mf_cache { uintptr_t low,
                    high; };
struct __mf_cache
       __mf_lookup_cache [];
uintptr_t __mf_lc_mask;
unsigned char __mf_lc_shift;

/* an approximation */
inline void
INLINE_CHECK (T* ptr, size_t sz, ...)
{
  uintptr_t low = ptr;
  uintptr_t high = low + sz - 1;
  unsigned idx =
    (low >> __mf_lc_shift) &
    __mf_lc_mask;
  struct __mf_cache *elem =
    & __mf_lookup_cache [idx];
  if (elem->low > low ||
      elem->high < high)
    __mf_check (ptr, sz, ...);
}
```

As with any caching scheme, choosing appropriate parameters (the *mask* and *shift* values) is a challenge. libmudflap has defaults suitable for mixed sizes of objects, which can be overridden by the user. In addition, when the runtime detects excessive cache misses, it adaptively tunes the cache parameters to better fit recent access patterns. For example, if accesses to small individual objects dominate, the current heuristic tends to decrease shift values. That way, more of the lower-order bits of the raw pointers remain to pick distinct cache lines. Section 3.1.4 lists libmudflap options that affect the lookup cache.

2.3 Instrumented expressions

Unsafe pointer expressions are easy to recognize when looking at C/C++ code. Systematically, most `*p`, `p->f`, and `a[n]` expression patterns need to be checked. Because mudflap operates in the middle of the compiler, we cannot look for such patterns in the source. Instead, we are given a representation of an entire function that resembles an abstract syntax tree. Expressions like the above are encoded in a web of nodes of GCC's `tree` type structure.

Mudflap traverses a function tree in program order, looking for certain pointer- or array-related constructs. These tree nodes are modified in place, replacing the simple pointer expressions with a GCC *statement expression*[7] that evaluates to the same value, but includes a call to the inline check routine outlined above. Shown in GCC's extended syntax, the expression `p->f` is changed roughly to `({check (p, ...); p;})->f`.

This in-place modification scheme supports recursion for nested constructs like `ptr->array[i]->field`. Here, two separate checks would be emitted: one for the element `ptr->array[i]`, and another to follow that pointer. The checks are performed in natural program order. Alternately, such nested constructs might be presented to mudflap already decomposed into an equivalent sequence of simpler expressions by the GIMPLE[8] transformations.

Table 1 shows the primitive expression patterns mudflap intercepts, and what address range is checked for each. For indirect accesses into larger compound objects, the checked range typically begins at the address of the outermost compound object, and ends by including the specific field or element being referenced. This way, the checked base value for similar accesses into the same structure or array can be constant, and take more benefit from the lookup cache. Notice that the checked range does *not* extend to include the entire compound object. This is because it is legal to allocate

[7]Statement expressions are a GCC extension that allows a brace-enclosed block to be treated as an expression. The last statement in the block is used as the expression value.

[8]`http://gcc.gnu.org/projects/tree-ssa/`

slightly less memory for a variable-sized structure than the raw `sizeof`, as long as the unallocated elements at the end are never accessed. GCC's own source code does this frequently.

2.4 Instrumented declarations

As discussed above, libmudflap's object database is kept up-to-date partly using instrumentation that tracks the lifetime of interesting memory objects. Some of these objects are variables declared as `auto` or `static` and have their addresses taken (or are indexed-into). For example, in the code segment below, the `array` variable needs to be registered with libmudflap (so the `[i]` indexing can be checked), but only for the duration of its scope (so that the returned pointer is invalid to dereference later).

```
char *foo (unsigned i) {
  char array [10];
  array [i] = 'a';
  return & array [i];
}
```

Tracking the lifetime of variables in a scope is tricky because control can leave a scope in several places. (Grossly, it might even enter in several places using `goto`.) The C++ constructor/destructor mechanism provides the right model for attaching code to object scope boundaries. Luckily, GCC provides the necessary facilities even to trees that come from the C frontend. There are several variants: the `CLEANUP_EXPR` node type, and the more modern `TRY_FINALLY_EXPR`. Both tree types take a block (a statement list) and another statement (a *cleanup*) as arguments. The former is interpreted as a sequence of statements such as any that follow a declaration within a given scope/block. The latter is a statement that should be evaluated whenever the scope is exited, whether that happens by `break`, `return`, or just plain falling off the end.[9]

We use this construct in mudflap by inserting one of these special try/finally tree patterns behind every declaration in need of lifetime instrumentation. The statement-list is the remainder of the original function, past the declaration in question, plus a register call for the declared object. The cleanup statement is an unregister call for the same object. The above function becomes the following, rendering `TRY_FINALLY_EXPR` in a Java-like way:

```
char *foo (unsigned i) {
  char array [10];
  try {
    __mf_register (array, 10, ...);
    array [i] = 'a';
    return & array [i];
  } finally {
    __mf_unregister (array, 10, ...);
  }
}
```

Mudflap also emits instrumentation to track the lifetime of some objects in the global scope: variables declared within file scope, or declared `static` within a function. This is done by intercepting assembler-related functions in `gcc/varasm.c`. It turns out at some literals like strings are like local static variables in this respect, so they too are registered. In each case, a list of declarations is accumulated until the end of the compilation unit. At that point a single dummy *constructor* function is synthesized, containing a long list of `__mf_register` calls. The linker arranges to call this and all other constructor functions early during the program startup.

2.5 Library interoperability

The above mechanisms are sufficient for checking pointer operations that are within an instrumented compilation unit. However, it is

[9]However, abrupt exit from a scope via a `longjmp` is not specifically handled at this time.

Sample declarations:

```
struct k {
  int a; /* offset 0 size 4 */
  char b; /* offset 4 size 1 */
}; /* size 8 */
int *iptr;
struct k *kptr;
char cbuf [];
short smtx [6][4];
int i, j;
```

expression	tree structure	check range	
		base	size
`*iptr`	`INDIRECT_REF(iptr)`	`iptr`	`4`
`*kptr`	`INDIRECT_REF(kptr)`	`kptr`	`8`
`kptr->a`	`COMPONENT_REF(INDIRECT_REF(kptr),a)`	`kptr`	`4`
`kptr->b`	`COMPONENT_REF(INDIRECT_REF(kptr),b)`	`kptr`	`5`
`cbuf[i]`	`ARRAY_REF(cbuf,i)`	`cbuf`	`i+1`
`smtx[i][j]`	`ARRAY_REF(ARRAY_REF(smtx,i),j)`	`smtx`	`8*i+2*j+2`

Table 1: Pointer expressions and their checked address ranges

often not possible to recompile an entire application, including the system libraries, with mudflap instrumentation. This means that several aspects of interoperability need to be addressed.

Most C/C++ programs make use of standard library functions (e.g., `strcpy`) that manipulate buffers given pointers. Typically, these libraries are not instrumented by mudflap, so they trust their arguments and don't perform pointer checking. An erroneous program can pass invalid pointers to these libraries, and bypass mudflap protection. libmudflap contains functions that interpose as a variety of such system library routines (though many more are yet to come). Each interposing function checks given buffer/length arguments, then jumps to the original system library. In this case, interposition is performed by replacing system library function names, via *preprocessor directives* implied by mudflap, with libmudflap names. This way, only instrumented object files are affected. Figure 2.5 shows a sample of this type of wrapper function in libmudflap.

In another scenario, an uninstrumented library may return to an instrumented caller some memory allocated from a shared heap. These memory regions should be registered with libmudflap, so that the instrumented code can be allowed to use them. Intercepting calls like `malloc` using preprocessor macros is not possible, since we are dealing with precompiled objects. We must intercept them at link time. Suitable mechanisms are available: *symbol wrapping* (for static linking with GNU ld) or *symbol interposition* (for shared libraries). libmudflap contains a protection mechanism to handle the case where a reentrant libmudflap⇒system-library⇒libmudflap call chain might occur.

```
void * WRAPPED_memmem (const void *haystack, size_t haystacklen,
                      const void *needle, size_t needlelen)
{
  INLINE_CHECK (haystack, haystacklen, __MF_CHECK_READ, "memmem haystack");
  INLINE_CHECK (needle, needlelen, __MF_CHECK_READ, "memmem needle");
  return memmem (haystack, haystacklen, needle, needlelen);
}

size_t WRAPPED_fread (void *ptr, size_t size, size_t nmemb, FILE *stream)
{
  INLINE_CHECK (ptr, size * nmemb, __MF_CHECK_WRITE, "fread buffer");
  INLINE_CHECK (stream, 1, __MF_CHECK_READ, "fread stream");
  return fread (ptr, size, nmemb, stream);
}
```

Figure 1: Sample libmudflap stdlib function wrappers

In yet another scenario, an uninstrumented library may return to an instrumented caller a value that points to some valid static data in the library. This could include objects as mundane as string literals. In this case, no link-time function interception can work, since these addresses are taken without reference to system functions. In order to tell automatically whether such a pointer is valid or not, libmudflap uses *heuristics*. These heuristics are checked when an access check is initially determined as a violation. They may look at other auxiliary platform-dependent data like the program's segment boundaries, stack pointer, and the like, to make a guess. Heuristics may be individually enabled or disabled at run time. See section 3.1.3 for more details.

2.6 Performance

Mudflap instrumentation and runtime costs extra time and memory. At build time, the compiler needs to process the instrumentation code. When running, it takes time to perform the checks, and memory to represent the object database. The behavior of the application has a strong impact on the run-time slowdown, affecting the lookup cache hit rate, the overall number of checks, and the number of objects tracked in the database, and their rates of change. Table 2 lists some of these. A few selected applications have been built with and without mudflap instrumentation, then run to estimate the slowdowns.[10] Table 2 also lists some applications, their performance factors, and associated slowdowns for a default mudflap build and run.

factor	description (+ polarity)
1	rare pointer manipulation
2	few large arrays
3	few addressed variables in scope
4	number cruncher
5	few tree/graph data structures
6	few objects in working set
7	non-changing access patterns

application	factors in effect		slowdown	
	+	−	build	run
BYTE nbench	3,4	1,2,5-7	3.5	3.5
spec2000 bzip2	2,5	1,3,4,6,7	4	5
spec2000 mcf	1-7		5	1.25

Table 2: Performance factors and overall measured slowdowns

[10] We used an x86 Linux host with ample memory, the same mudflap-capable compiler, and same optimization levels and linking modes.

2.7 Future

Mudflap development is ongoing; we anticipate several improvements. Significant performance benefits may arise from changing the instrumentation code (mainly for pointer checks), and functionality and performance benefits from the runtime.

We currently instrument each occurrence of a pointer dereference, even if that same pointer/size pair has been "recently" checked. Such checks could be eliminated if the compiler could prove that a subsequent check is redundant with respect to an earlier one. Extending from this, it may be possible to aggregate multiple checks based on the same pointer or array - imagine sequences of statements that access `ptr->field1` through `ptr->field5`. The compiler could create a single large check[11] near the beginning of a basic block, and eliminate subsequent checks for the same pointer/array. Some checks could be moved out of loops. In exchange for significantly better performance, such optimizations could detect pointer use errors out of program sequence.

Possible future libmudflap enhancements include support for multithreaded applications, growing the list of hooked functions to include more of the system libraries and system calls, more libmudflap entry points for use in an embedded system without a kernel, a better GDB interface, and general tuning.

3 Using Mudflap

Using mudflap is intended to be easy. One builds a mudflap-protected program by adding an extra compiler option (`-fmudflap`) to objects to be instrumented; one links with the same option, plus perhaps `-static`. One may run such a program by just starting it as usual.

[11]A large check would cover the maximal referenced range, including the last referenced field for a pointer, or the largest index for an array. This may require value range propagation or similar analysis.

In the default configuration, a mudflap-protected program will print detailed violation messages to `stderr`. They are tricky to decode at first. Figure 5 in the Appendix contains a sample message, and its explanation.

3.1 Runtime options

libmudflap observes an environment variable `MUDFLAP_OPTIONS` at program startup, and extracts a list of options. Include the string `-help` in that variable, and libmudflap will print out all the options and their default values. The display at the time of this writing is shown in Figure 5 in the Appendix. The next sections describe the options in groups.

3.1.1 Violation handling

The `-viol-` series of options control what libmudflap should do when it determines a violation has occurred. The `-mode-` series controls whether libmudflap should be active.

`-viol-nop` Do nothing. The program may continue with the erroneous access. This may corrupt its own state, or libmudflap's.

`-viol-abort` Call `abort()`, requesting a core dump and exit.

`-viol-segv` Generate a `SIGSEGV`, which a program may opt to catch.

`-viol-gdb` Create a GNU debugger session on this suspended program. The debugger process may examine program data, but it needs to quit in order for the program to resume.

`-mode-nop` Disable all main libmudflap functions. Since these calls are still tabulated if using `-collect-stats`, but the lookup cache is disabled, this mode is useful to count total number of checked pointer accesses.

`-mode-populate` Act like every libmudflap check succeeds. This mode merely populates the lookup cache but does not actually track any objects. Performance measured with this mode would be a rough upper bound of an instrumented program running an ideal libmudflap implementation.

`-mode-check` Normal checking mode.

`-mode-violate` Trigger a violation for every main libmudflap call. This is a dual of `-mode-populate`, and is perhaps useful as a debugging aid.

3.1.2 Extra checking and tracing

A variety of options add extra checking and tracing.

`-collect-stats` Print a collection of statistics at program shutdown. These statistics include the number of calls to the various main libmudflap functions, and an assessment of lookup cache utilization.

`-print-leaks` At program shutdown, print a list of memory objects on the heap that have not been deallocated.

`-check-initialization` Check that memory objects on the heap have been written to before they are read. Figure 5 explains a violation message due to this check.

`-sigusr1-report` Handle signal `SIGUSR1` by printing the same sort of libmudflap report that will be printed at shutdown. This is useful for monitoring the libmudflap interactions of a long-running program.

`-trace-calls` Print a line of text to `stderr` for each libmudflap function.

`-verbose-trace` Add even more tracing of internal libmudflap events.

`-verbose-violations` Print details of each violation, including nearby recently valid objects.

`-persistent-count=N` Keep the descriptions of N recently valid (but now deallocated) objects around, in case a later violation may occur near them. This is useful to help debug use of buffers after they are freed.

`-abbreviate` Abbreviate repeated detailed printing of the same tracked memory object.

`-backtrace=N` Save or print N levels of stack backtrace information for each allocation, deallocation, and violation.

`-wipe-stack` Clear each tracked stack object when it goes out of scope. This can be useful as a security or debugging measure.

`-wipe-heap` Do the same for heap objects being deallocated.

`-free-queue-length=N` Defer an intercepted `free` for N rounds, to make sure that immediately following `malloc` calls will return new memory. This is good for finding bugs in routines manipulating list- or tree-like structures.

`-crumple-zone=N` Create extra inaccessible regions of N bytes before and after each allocated heap region. This is good for finding buggy assumptions of contiguous memory allocation.

`-internal-checking` Periodically traverse libmudflap internal structures to assert the absence of corruption.

3.1.3 Heuristics

As discussed in Section 2.5, libmudflap contains several heuristics that it may use when it suspects a memory access violation. These heuristics are only useful when running a hybrid program that has some uninstrumented parts. Memory regions suspected valid by heuristics are given the special *guess* storage type in the object database, so they don't interfere with concrete object registrations in the same area.

`-heur-proc-map` On Linux systems, the special file `/proc/self/map` contains a tabular description of all the virtual memory areas mapped into the running process. This heuristic looks for a matching row that may contain the current access. If this heuristic is enabled, then (roughly speaking) libmudflap will permit all accesses that the raw operating system kernel would allow (i.e., not earn a SIGSEGV).

`-heur-start-end` Permit accesses to the statically linked text/data/bss areas of the program.

`-heur-stack-bound` Permit accesses within the current stack area. This is useful if uninstrumented functions pass local variable addresses to instrumented functions they call.

`-heur-argv-environ` This option adds the standard C startup areas that contain the `argv` and `environ` strings to the object database.

3.1.4 Tuning

There are some other parameters available to tune performance-sensitive behaviors of libmudflap. Picking better parameters than default is a trial-and-error process and should be undertaken only if `-collect-stats` suggests unreasonably many cache misses, or the application's working set changes much faster or slower than the defaults accommodate.

`-age-tree=N` For tracking a current *working set* of tracked memory objects in the binary tree, libmudflap associates a *liveness* value with each object. This value is increased whenever the object is used to satisfy a lookup cache miss. This value is decreased every N misses, in order to penalize objects only accessed long ago.

`-lc-mask=N` Set the lookup cache mask value to N. It is best if N is $2^M - 1$ for $0 < M \leq 10$.

`-lc-shift=N` Set the lookup cache shift value to N. N should be just a little smaller than the power-of-2 alignment of the memory objects in the working set.

`-lc-adapt=N` Adapt the mask and shift parameters automatically after N lookup cache misses. The adaptation algorithm uses the working set as identified by tree aging. Set this value to zero if hard-coding them with the above options.

3.2 Introspection

libmudflap provides some additional services to applications or developers trying to debug them. Functions listed in the `mf-runtime.h` header may be called from an application, or interactively from within a debugging session.

`__mf_watch` Given a pointer and a size, libmudflap will specially mark all objects overlapping this range. When accessed in the future, a special violation is signaled. This is similar to a GDB watchpoint.

`__mf_unwatch` Undo the above marking.

`__mf_report` Print a report just like the one possibly shown at program shutdown or upon receipt of `SIGUSR1`.

`__mf_set_options` Parse a given string as if it were supplied at startup in the `MUDFLAP_OPTIONS` environment variable, to update libmudflap runtime options.

4 Acknowledgments

The author thanks Ben Elliston for suggesting the mudflap name, Graydon Hoare for prototyping several parts of libmudflap, Diego Novillo for commiserating about GCC internals (and doing something to improve it), Red Hat (my employer) for funding mudflap's development, and future contributors for contributing in the future.

5 Availability

The source code of GCC with mudflap extensions, and of libmudflap, are available from the author, or by anonymous CVS. See `http://gcc.gnu.org/projects/tree-ssa/` for instructions.

```
mudflap violation 3 (check/read): time=1049824033.102085 ptr=080c0cc8 size=1
```

This is the third violation taken by this program. It was attempting to read a single-byte object with base pointer `0x080c0cc8`. The timestamp can be decoded as 102 ms after `Tue Apr  8 13:47:13 2003` via `ctime`.

```
pc=08063299 location='nbench1.c:3077 (SetCompBit)'
      nbench [0x8063299]
      nbench [0x8062c59]
      nbench(DoHuffman+0x4aa) [0x806124a]
```

The pointer access occurred at the given PC value in the instrumented program, which is associated with the file `nbench1.c` at line 3077, within function `SetCompBit`. (This does not require debugging data.) The following lines provide a few levels of stack backtrace information, including PC values in square brackets, and sometimes module/function names.

```
Nearby object 1: checked region begins 8B into and ends 8B into
```

There was an object near the accessed region, and in fact the access is entirely within the region, referring to its byte #8.

```
mudflap object 080958b0: name='malloc region'
bounds=[080c0cc0,080c2057] size=5016 area=heap check=1r/0w liveness=1
```

This object was created by the `malloc` wrapper on the heap, and has the given bounds, and size. The `check` part indicates that it has been read once (this current access), but never written. The liveness part relates to an assessment of how frequently this object has been accessed recently.

```
alloc time=1049824033.100726 pc=4004e482
      libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
      nbench(AllocateMemory+0x33) [0x806a153]
      nbench(DoHuffman+0xd5) [0x8060e75]
```

The allocation moment of this object is described here, by time and stack backtrace. If this object was also deallocated, there would be a similar `dealloc` clause. Its absence means that this object is still alive, or generally legal to access.

```
Nearby object 2: checked region begins 8B into and ends 8B into
mudflap object 080c2080: name='malloc region'
bounds=[080c0cc0,080c2057] size=5016 area=heap check=306146r/1w liveness=4562
alloc time=1049824022.059740 pc=4004e482
      libmudflap.so.0(__real_malloc+0x142) [0x4004e482]
      nbench(AllocateMemory+0x33) [0x806a153]
      nbench(DoHuffman+0xd5) [0x8060e75]
```

Another nearby object was located by libmudflap. This one too was a `malloc` region, and happened to be placed at the exact same address. It was frequently accessed.

```
dealloc time=1049824027.761129 pc=4004e568
      libmudflap.so.0(__real_free+0x88) [0x4004e568]
      nbench(FreeMemory+0xdd) [0x806a41d]
      nbench(DoHuffman+0x654) [0x80613f4]
      nbench [0x8051496]
```

This object was deallocated at the given time, so this object may not be legally accessed any more.

```
number of nearby objects: 2
```

No more nearby objects have been found.
The conclusion? Some code on line 3077 of `nbench1.c` is reading a heap-allocated block that has not yet been initialized by being written into. This is a situation detected by the `-check-initialization` libmudflap option, referred to in section 3.1.2.

Figure 2: Sample libmudflap violation message, dissected

```
This is a GCC "mudflap" memory-checked binary.
Mudflap is Copyright (C) 2002-2003 Free Software Foundation, Inc.

The mudflap code can be controlled by an environment variable:

$ export MUDFLAP_OPTIONS='<options>'
$ <mudflapped_program>

where <options> is a space-separated list of
any of the following options.  Use '-no-OPTION' to disable options.

-mode-nop              mudflaps do nothing
-mode-populate         mudflaps populate object tree
-mode-check            mudflaps check for memory violations [default]
-mode-violate          mudflaps always cause violations (diagnostic)
-viol-nop              violations do not change program execution [default]
-viol-abort            violations cause a call to abort()
-viol-segv             violations are promoted to SIGSEGV signals
-viol-gdb              violations fork a gdb process attached to current program
-trace-calls           trace calls to mudflap runtime library
-verbose-trace         trace internal events within mudflap runtime library
-collect-stats         collect statistics on mudflap's operation
-sigusr1-report        print report upon SIGUSR1
-internal-checking     perform more expensive internal checking
-age-tree=N            age the object tree after N accesses for working set [13037]
-print-leaks           print any memory leaks at program shutdown
-check-initialization  detect uninitialized object reads
-verbose-violations    print verbose messages when memory violations occur [default]
-abbreviate            abbreviate repetitive listings [default]
-wipe-stack            wipe stack objects at unwind
-wipe-heap             wipe heap objects at free
-heur-proc-map         support /proc/self/map heuristics
-heur-stack-bound      enable a simple upper stack bound heuristic
-heur-start-end        support _start.._end heuristics
-heur-argv-environ     support argv/environ heuristics [default]
-free-queue-length=N   queue N deferred free() calls before performing them [4]
-persistent-count=N    keep a history of N unregistered regions [100]
-crumple-zone=N        surround allocations with crumple zones of N bytes [32]
-lc-mask=N             set lookup cache size mask to N (2**M - 1) [1023]
-lc-shift=N            set lookup cache pointer shift [2]
-lc-adapt=N            adapt mask/shift parameters after N cache misses [1000003]
-backtrace=N           keep an N-level stack trace of each call context [4]
```

Figure 3: List of libmudflap runtime options.

Alias Analysis for Intermediate Code

Sanjiv K. Gupta *Naveen Sharma*
System Software Group
HCL Technologies, Noida, India – 201 301
`{sanjivg,naveens}@noida.hcltech.com`

Abstract

Most existing alias analysis techniques are formulated in terms of high-level language constructs and are unable to cope with pointer arithmetic. For machines that do not have 'base + offset' addressing mode, pointer arithmetic is necessary to compute a pointer to the desired address. Most state of the art compilers such as GCC lack the mechanism to determine aliasing between such computed pointers. Few other existing alias analysis techniques described for executable code can handle pointer arithmetic but require large memory when applied to intermediate languages such as RTL. In this paper, we describe a method of disambiguating the computed pointers within a procedure at the intermediate code level. The method is similar to the techniques described for executable code but requires significantly less amount of memory. We have experimented our method with the GCC RTL and it reduces the code size of array manipulating benchmarks by approximately 4-7% for the machines that do not have 'base + offset' addressing mode.

1 Introduction

Various optimization passes like CSE and instruction scheduling rely on alias analysis to determine the aliasing between two memory references. Compile time alias analysis in compilers such as GCC can successfully determine aliasing between two memory references if they (i) use distinct offsets from the same register; or (ii) one of them points to stack. But such compile time alias analysis often fails to determine aliasing between computed pointers and safely assume that these pointers may alias.

To illustrate the computed pointers and aliasing problem with them, let us consider the following piece of code:

```
void foo (double *in)
{
    in[4] += in[3];
}
```

For machines that have 'base + offset' addressing for `double`, GCC generates RTL like,

```
r172 = [r170, 32]
r173 = [r170, 24]
r174 = r172 + r173
[r170, 32] = r174
```

in such cases the GCC can successfully determine that the memory references `[r170, 32]` and `[r170, 24]` do not alias as they use distinct offsets from the same base register.

On machines that do not have 'base + offset' addressing mode for `double`, the compiler will need to compute the pointers to load and store locations. In these cases, the generated RTL will look like,

```
r170 = r160 + 32
r171 = r160 + 24
r172 = [r170]
r173 = [r171]
r174 = r172 + r173
[r170] = r174
```

GCC fails to determine aliasing between the computed pointers `r170` and `r171`. To be safe, GCC simply assumes that these computed pointers alias with each other. The problem with GCC is that it does not have any mechanism to keep track of what address arithmetic have been performed to obtain the computed pointers `r170` and `r171`.

There are algorithms available to keep track of address arithmetic (see [Debray98]); but they work well only with the executable code since the executable programs have small number of registers (i.e. only hard registers). Time and space requirements of such algorithms increase when we try to use them for intermediate code such as RTL as there may be large number of pseudo registers present in the intermediate code. This paper describes an alias analysis algorithm that can be used with the intermediate code to keep track of the address arithmetic efficiently. The algorithm is influenced from the mod-k residue technique for executable code described in [Debray98].

2 Terminology

The mod-k residue algorithm maps each pseudo with a set of possible address values at each program point. Let us first define some basic terms that are required to discuss the algorithm. The term pseudo means a pseudo register in entire discussion.

2.1 A Program Point

A program point refers to a point between two instructions[Muchnick]. A program point *p* between instructions *I1* and *I2* is denoted as *p(I1, I2)*, where *I1* immediately preceds *p* and *I2* immediately follows *p*. Since compilers always keep a chain of instructions available all the time, the preceding instruction *I1* is all which is required to identify a program point *p*.

Any solution that attempts to disambiguate two computed pointers should be able to tell the possible address values represented by each pointer pseudo at each program point. For example, for the pointer pseudos `r1` and `r2` at given program points *p1* and *p2*, the solution must be able to tell the possible address values represented by `r1` at *p1*, and `r2` at *p2*.

2.2 mod-k Residues Set

For compactly storing an address value we consider only some fixed number, say *m*, of the lower bits of the value. This means an abstract address value *val* is represented by its mod-k residue *val mod k*. $(k = 2^m)$. The set of all abstract address values can then be represented by the mod-k residues set $Z = (0, 1, 2, ..., k-1)$. Since $(x \; mod \; k) \neq ((x+\delta) \; mod \; k) \; \forall \; 0 < \delta < k$, the representation can distinguish between addresses involving distinct "small" displacements δ (i.e. less than *k*) from a base register.

The choice of the value *k* is critical for efficiency of the technique. The value *k* determines the size of mod-k residues set; the choice should be made in such a way that it makes storing and manipulating mod-k residues sets low cost operations. Often, the natural word size of the host machine is a good choice. This way we can store a mod-k residue set as a bit vector in a single machine word. Operations such as adding a constant *c* to each member of the set can be simply obtained by rotating the bit-vector by *c* bits. For example, the mod-k residues set (4, 12) can be represented by a machine word whose 4^{th} and 12^{th} bits are ON and rest of the bits are OFF.

In our implementation experiment with GCC on host machine x86, we choose the value of *k* as sizeof(int) so that a mod-k residues set can be stored efficiently in an integer.

2.3 Address Descriptors

The mod-k residues sets by themselves are not adequate for cases where we are not able to predict the actual value of a pseudo `r` at a program point. To deal with this problem we extend mod-k residues set to 'Address Descriptors'. An address descriptor is a pair *{I, Z}*, where *I* is an instruction and *Z* is a mod-k residues set. Given an address descriptor *A*(`r`) = *{I, Z}* for a pseudo `r`, the instruction *I* is the defining instruction of `r`, and *Z* denotes the set of mod-k residues relative to whatever value is computed by instruction *I*.

The address descriptor of a pseudo `r` is computed by analyzing its defining instruction as per the rules described in section 3. If we cannot say anything about the value of a pseudo `r` while analyzing its defining instruction *I*, we associate the address descriptor *{I, (0)}* with `r`. A constant *c* yields an address descriptor *{NONE, (c mod k)}*.

For example consider the following instructions:

```
I1: r172 = [r170]
I2: r173 = 5
I3:
```

The address descriptor of pseudo `r172` at program point *p(I1, I2)* will be *{I1, (0)}* as we can not say anything about the value of `r172` after instruction *I1*. The address descriptor for pseudo `r173` at program point *p(I2, I3)* will be *{NONE, (5)}*.

Further we define two special address descriptors, an address descriptor *{ANY, (all)}* alias with everything and the address descriptor *{NONE, (nothing)}* alias with nothing.

3 Effect of Individual Instructions on Address Descriptors: Keeping track of Address Arithmetic

The operations performed by an instruction modifies certain pseudos; the algorithm defines these operations for address descriptors and applies them to modify address descriptors corresponding to those pseudos. In this section we define assignment, addition, and multiplication operations for address descriptors as they are the most frequent operations occurring in address arithmetic.

3.1 Assignment Instructions

Consider an assignment instruction *I* having the following form,

```
I: dest = src
```

where `dest` is a pseudo and `src` could be a pseudo or some integer constant.

The address descriptor of `dest` pseudo is evaluated as following:

a) If `src` is a pseudo and has a valid address descriptor, the address descriptor of `src` becomes the address descriptor of `dest`.

b) If `src` is a pseudo that does not have a valid address descriptor, the address descriptor of `dest` becomes *{I,(0)}*.

c) If `src` is a constant integer *c*, address descriptor of `dest` will be as *{NONE, (c mod k)}*.

3.2 Addition Instructions

Consider an addition instruction *I* having the following form.

```
I: dest = src1 + src2
```

where `dest` and `src1` are pseudos and `src2` can be a pseudo or an integer constant. Let *{I1, Z1}* and *{I2, Z2}* be the address descriptors of `src1` and `src2` respectively. The address descriptor of pseudo `dest` is then evaluated as following:

a) If *I1 = NONE*, the address descriptor of `dest` becomes *{I2, Z}* (the situation where *I2 = NONE* is symmetric). Here $Z = \{((x + y)\ mod\ k)\ \forall\ x \in Z1, y \in Z2\}$.

b) Otherwise, we can not say anything about the result of this operation. So the address descriptor of `dest` after this instruction *I* is taken to be *{I, (0)}*.

3.3 Multiplication Instructions

Consider a multiplication instruction I having the following form,

```
I: dest = src1 * src2
```

where `dest` and `src1` are pseudos and `src2` can be a pseudo or an integer constant. Let *{I1, Z1}* and *{I2, Z2}* be the address descriptors of `src1` and `src2` respectively. The address descriptor of `dest` is then evaluated as following:

a) If *I1 = NONE*, the address descriptor of `dest` becomes *{I2, Z}* (the situation where *I2 = NONE* is symmetric). Here $Z = \{((x * y)\ mod\ k)\ \forall\ x \in Z1, y \in Z2\}$.

b) Otherwise, we can not say anything about the result of this operation. So the address descriptor of `dest` after this instruction *I* is taken to be *{I, (0)}*.

Though semantics for other operations on address descriptors can be defined but above integral operations suffice in most cases to handle the pointer arithmetic. The address descriptor of the destination pseudo `r` of an unhandled-instruction[1] *I* is taken as *{I,(0)}*.

4 The Algorithm

The algorithm maps each pseudo with its possible values (i.e. an address descriptor) at each program point. Since storing an address descriptor for each pseudo at each program point will require excessive memory, we compute the address descriptors of pseudos defined in a basic block and store them only at the end of the basic block. Using this saved information, the address descriptor of a pseudo at a particular program point within a basic block can be obtained by recomputing the address descriptors of the basic block upto that program point. This recomputing does not take much time as basic blocks happen to be small in most cases.

4.1 Computing Address Descriptors

The instructions in a basic block are analyzed as described in section 3 to compute the address descriptors of pseudos defined in that basic block. The input address descriptors of the basic block are determined as described in the subsection 4.2. The address descriptors computed in the basic block are then saved at the end of that basic block. This saved list of address descriptors at the end of a basic block is called *OUT_LIST* of that basic block.

Storing the address descriptors for all pseudos defined in a basic block in the *OUT_LIST* will

[1]instruction for which the corresponding address descriptor operation is not defined

require very large memory (intermediate code may contain large number of pseudos). Since most of the defined pseudos are local to a basic block, they do not contribute to the input of their successors. To reduce the memory requirements, address descriptors for such pseudos need not be saved in the *OUT_LIST*. The algorithm first identifies all those pseudos that are being used across basic blocks. We call such pseudos as "shared pseudos". The address descriptors for "shared pseudos" only are saved at the end of all basic blocks. This saves lot of memory since there exists usually a small number of "shared pseudos" in intermediate code. If a procedure with *N* basic blocks have *R* shared pseudos, the memory required for storing the address descriptors would be *RN(k+w)* bits, where *w* is the machine word size in bits.

4.2 Propagating Address Descriptors across Basic Blocks

CFG is used to propagate these descriptors across basic blocks. A *union* operation is used to "merge" the information coming along the incoming edges at vertices in the CFG. An input list of address descriptors (we call this an *IN_LIST*) for a basic block is formed by doing the *union* of OUT_LISTs of its predecessors. Thus if the address descriptors for a pseudo `r` being propagated along two incoming edges at a vertex in the CFG are *{I1, Z1}* and *{I2, Z2}*, the resulting address descriptor for pseudo `r` is obtained as,

{I, Z1 union Z2} if *I1=I2=I*.
{ANY, (all)} if $I1 \neq I2$.

For example, as shown in Figure 1, consider a basic block `BB3` having two predecessors `BB1` and `BB2`. If the address descriptors of a pseudo `r101` in the *OUT_LISTs* of `BB1` and `BB2` are *{I0, (8)}* and *{I0, (24)}*, the address descriptor of `r101` in the *IN_LIST* of `BB3` will be *{I0, (8, 24)}*.

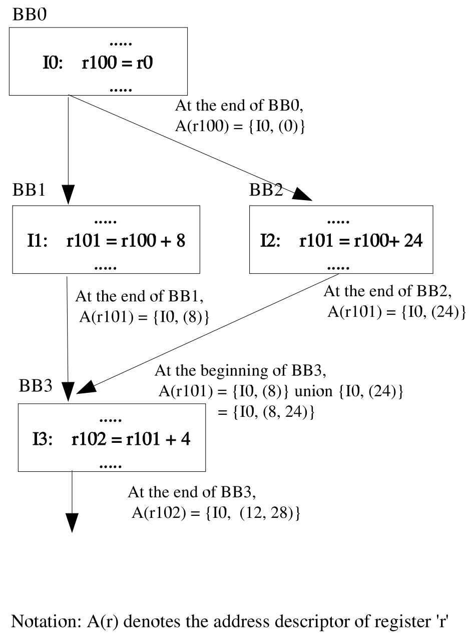
BB0
.....
I0: r100 = r0
.....
At the end of BB0, A(r100) = {I0, (0)}
BB1
.....
I1: r101 = r100 + 8
.....
BB2
.....
I2: r101 = r100+ 24
.....
At the end of BB1, A(r101) = {I0, (8)}
At the end of BB2, A(r101) = {I0, (24)}
At the beginning of BB3, A(r101) = {I0, (8)} union {I0, (24)} = {I0, (8, 24)}
BB3
.....
I3: r102 = r101 + 4
.....
At the end of BB3, A(r102) = {I0, (12, 28)}
Notation: A(r) denotes the address descriptor of register 'r'

Figure 1: Merging of address descriptors

4.3 Building the Fixed alias analysis Information

Multiple iterations over the CFG are done till the address descriptors of all "shared registers" in a procedure become constant, or in other words till the *OUT_LISTs* of all basic blocks become constant. Each iteration computes the *OUT_LIST* of each basic block using the *IN_LIST* of the basic block as input. The *OUT_LIST* computed during the iteration is *union*ed (as described in subsection 4.2) with the saved *OUT_LIST* of the previous iteration and the result is saved as the current *OUT_LIST* of the basic block. Another iteration over CFG is required only if any of the *OUT_LISTs* change in the current iteration. The required information for performing alias analysis is built once we have reached this stage where all the *OUT_LISTs* are fixed. This way we have gathered for all "shared pseudos", all the possible results of operations performed on them by all execution paths.

We can describe this in the following pseudocode,

```
do {
  out_lists_changed = false;
  for each BB in the CFG {
    prepare an IN_LIST of BB
      by doing union of the
      OUT_LISTS of
      predecessors of BB;
    evaluate OUT_LIST_OF_THIS_PASS
      using IN_LIST as input;
    NEW_OUT_LIST = do union of
      OUT_LIST_OF_THIS_PASS
      with the SAVED_OUT_LIST.
    list_changed = false;
    if (NEW_OUT_LIST is not
      equal to SAVED_OUT_LIST) {
      SAVED_OUT_LIST = NEW_OUT_LIST;
      list_changed = true;
    }
    out_lists_changed =
     out_lists_changed | list_changed;
  }
} while (out_lists_changed);
```

To reduce the number of iterations required over CFG, we identify loop counters such as `r = r + const` and populate their address descriptor in a single pass itself. For example, given a loop counter `r` in RTL below,

```
I1: r = 0
...
I7: r = r + 2
```

The address descriptor of pseudo `r` is calculated in the first pass itself as *{NONE, (0,2,4,6,8,10,12,14)}* (for mod-16 alias analysis).

5 Reasoning about alias relationship

Once the required alias information is generated, the aliasing relationship between two computed pointers can be determined in following steps.

Step 1. Given two computed pointers `r1` and `r2`, we retrieve the program points *p1* and *p2* where `r1` and `r2` are dereferenced.

To retrieve the program points for these pointers a hash table is built at the start of the algorithm. For every pointer, this hash table records the instruction in which the pointer is contained. For a pointer, the instruction retreived from the hash table gives the preceding instruction of the program point.

Step 2. Find the basic blocks for the program points *p1* and *p2*; say they are `BB1` and `BB2`.

Step 3. Compute the *IN_LISTs* of `BB1` and `BB2` by doing the union of saved *OUT_LISTs* of their predecessors.

Step 4. Recompute the address descriptors *{I1, (Z1)}* and *{I2, (Z2)}* of the two pointers `r1` and `r2` at the desired program points *p1* and *p2* by traversing within their basic blocks `BB1` and `BB2`.

Step 5. Address descriptors *{I1, (Z1)}* at instruction point *p1*, and *{I2, (Z2)}* at instruction point *p2* denote disjoint addresses if both the following conditions are satisfied.

i) *I1 = I2 = 'I'*.

ii) *Z1 intersection Z2 = NULL*

Condition (i) ensures that both the program points *p1* and *p2* see the same value computed by instruction *I*. Condition (ii) then ensures that relative to this value, the pointer `r1` referred at *p1* is disjoint from the pointer `r2` referred at *p2*.

6 Example

Let us describe the entire algorithm with the help of the following example function in C,

```
void foo (double * a)
{
  int i, j;

  i = 0;
  j = 2;

  if (! a)
    j = j + 4;

  a[i] = a[i + j];
}
```

Figure 2 shows the CFG and RTL generated for SH4 alongwith the address descriptors computed by the algorithm. To determine the aliasing relationship between the computed pointers `r165` and `r162` in basic block BB2, their address descriptors are recomputed using the *IN_LIST* of BB2. Applying the rules of Section 3 on BB2 gives the recomputed address descriptors of `r162` and `r165` as *{I1, (0)}* and *{I1, (16)}*. These address descriptors do not alias since they follow the rules described in step5 of Section 5.

7 Drawbacks

The algorithm is not capable of keeping track of contents of memory. Information about a register is lost if it is saved to memory and then subsequently restored at a later point. Also if a register can have different defining instructions at different predecessors of a CFG vertex, the information is lost while merging them using the *union* operator.

The precision of results obtained also depends on the value of *k*. The algorithm can only

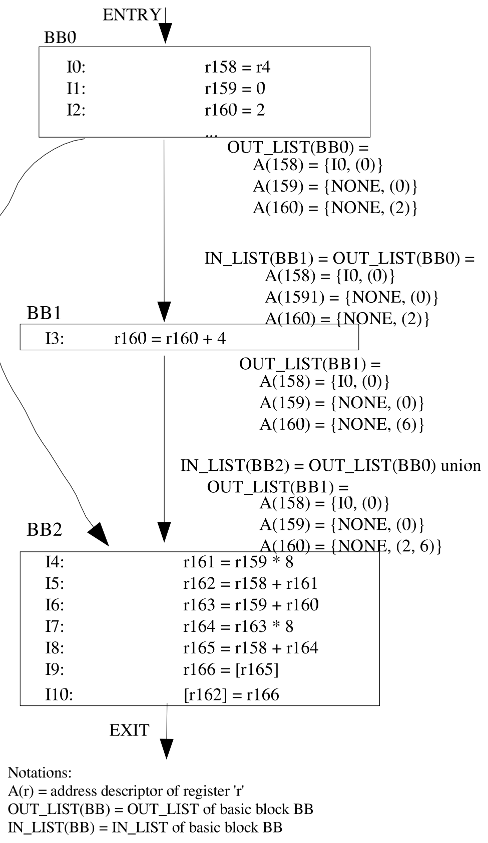

Figure 2: address descriptor based alias analysis

distinguish between the displacements in the range $\{0 < \delta < k\}$. For example, if k=32 then the algorithm will not be able to differentiate between the computed pointers for `&in[9]` and `&in[13]`. Increasing the value of k improves the precision of results obtained but may also increase the execution time of algorithm.

8 Experimentation and Results

We experimented by implementing this algorithm in GCC. Since the compiler was running on an i686 machine, we chose the value of k as 32. We built the cross compiler for ia64-elf target and obtained the data about generated code size. Table 1 given below compares the generated code size for ia64-elf for some of stress-1.17 files with -O2 option. We also observed that our implementation increases the compilation time for programs by about 20%.

file name	**size of .text section (before)**	**size of .text section (after**	**%code size decrease**
dct64.o	9808	9568	2.44
lpc.o	36824	33256	9.68
mdct.o	5936	5488	7.54
polyobj.o	14840	14360	3.23
layer3.o	54760	54344	0.75
tif_lzw.o	24320	24256	0.32
quadrics.o	22000	21840	0.73

Table 1: code size comparison for ia64-elf

9 Acknowledgments

We would like to thank people at gcc@gcc.gnu.org for their invaluable support. We specially thank to Richard Henderson, Diego Novillo, Saumya K. Debray, Daniel Berlin and David Edelsohn for their ideas.

References

[Debray98] S. Debray, R. Muth, and M. Weippert, *Alias Analysis of Executable Code*, In The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 12-24, Orlando, Florida (1998)

[Muchnick] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Inc., Reading, page 303, USA (1997)

[GCC] GCC source code, `http://gcc.gnu.org`

Porting GCC to the AMD64 architecture

Jan Hubička
SuSE ČR, s. r. o.
`jh@suse.cz, http://www.ucw.cz/˜hubicka`

Abstract

In this paper we describe our experience from porting GCC to the AMD64 architecture and the AMD Opteron processor. Our target was a high quality port producing fast code. We discuss decisions taken while designing the Application Binary Interface (ABI) and effect of various code optimizations we implemented. We also present several open issues we would like to solve in the future.

1 AMD64 Instruction Set Overview

The AMD64 architecture [AMD64] is an extension of x86 instruction set to enable 64-bit computing while remaining compatible with existing x86 software. The CPU can operate in 64-bit mode, where semantic of several x86 instructions has been changed. Most notably:

- Single byte encoding of `inc` and `dec` instructions is no longer available. Instead the opcodes are used to encode a new prefix `REX` with four one-bit arguments. First argument is used to overwrite instruction operand size into 64 bits. Other three are used to increase amount of general purpose registers from 8 to 16.
- New 64-bit addressing mode is used by default. Prefix is available to overwrite into 32-bit addressing when needed.
- One of multiple possible encodings of direct addressing has been changed into instruction pointer relative addressing. Instruction pointer relative addressing is now one byte shorter than direct addressing.
- Default operand size remains 32-bit, however stack manipulation instructions, such as `push` and `pop` defaults to 64-bit operand size.
- The immediate operands of instructions has not been extended to 64 bits to keep instruction size smaller, instead they remain 32-bit sign extended. Additionally the `movabs` instruction to load arbitrary 64-bit constant into register and to load/store integer register from/to arbitrary constant 64-bit address is available.
- Several new instructions have been added to allow 64-bit conversions of data types.

Unlike earlier 64-bit architectures GCC has been ported to, some AMD64 features are unique, such as CISC instruction set, generally usable IP relative addressing, partial support for 64-bit immediate operands and more.

2 Application Binary Interface

Since GCC has been one of the first compilers ported to the platform, we had a chance to design the processor specific part of the application binary interface [AMD64-PSABI] from scratch. In this section we discuss the decisions we made and rationale behind them. We also discuss the GCC implementation, as well as problems we encountered while porting the software.

Majority of [AMD64-PSABI] has been designed in the early stages of development with just preliminary implementation of AMD64 support in GCC and no hardware nor simulator available. Thus we had just limited possibilities for experiments and most of our decisions has been verified by measuring of executable files sizes and number of instructions in them.

We never made serious study on how these relate to the performance, but it may be expected that the relation is pretty direct in the cases we were interested in.

2.1 Fundamental Types

We do use 64-bit pointers and `long`. The type `int` is 32-bit. This scheme is known as LP64 model and is used by all 64-bit UNIX ports we are aware of.

64-bit pointers bring expansion of the data-structures and increase memory consumption of the applications. A number of 64-bit UNIX ports also specify a code model with 32-bit pointers, LP32. Because of large maintenance cost of extra model (change of pointer size requires kernel emulation layer and brings further difficulties) and because of support for native 32-bit applications we decided to concentrate on LP64 support first and implement LP32 later only if necessary. See also Section 4.1 for some further discussion.

We considered the `long long` type to be 128-bit, since AMD64 has limited support for 128-bit arithmetics (that comes from extending support for 32-bit arithmetic in 16-bit 8086), however there are many programs that do expect `long long` to be exactly 64-bit, thus we specify optional `__int128` instead. At the moment no library functions to deal with the type are specified so it's usage in C environment may be uncomfortable. This is something we may consider to address in future extension of the ABI document.

The size of `long double` is 128 bits with only first 80 bits used to match native x87 format. The rest is just padding to keep long double values 128-bit aligned so loads and stores are effective. The padding is undefined that may bring problems when one is using `memcmp` to test for equality of two `long double` values.

Additionally we specify `__m64` and `__m128` types for SIMD operations.

All types do have natural alignment. ([i386-ABI] limits the alignment to 32-bit that brings serve performance problems when dealing with `double`, `long double`, `__m64` and `__m128` types on modern CPU.) It is allowed to access misaligned data of all types with the exception of `__m128`, since CPU traps on misaligned 128-bit memory accesses.

GCC Implementation

Our GCC implementation does support all specified types with the exception of `__float128`. At the moment GCC is not ready to support two extended floating point formats having the same size and thus implementing it would require considerable effort.

The 128-bit arithmetics patterns are also not implemented yet so code generated for `__int128` is suboptimal.

Size	Contents	Frame
0–8n	incoming arguments	Previous
8	return address	
0,8	previous `%rbp` value	
0,8	padding	
?	local data	Current
?	register spill area	
0–4	padding	
0–48	register save area	
0,8	padding	
0,8	padding	Allocated
0–8n	outgoing arguments	via `push`

Figure 1: Stack Frame

```
push    %rbp        Save frame pointer
movq    %rsp %rbp   Initialize frame pointer
subq    $48, %rsp   Allocate stack frame
pushq   %rbx        Save non-volatile registers
pushq   %r12        clobbered by function
pushq   %r13
...                 Function body
popq    %r13        Restore registers
popq    %r12
popq    %rbx
leave               Restore %rbp
                    and deallocate stack
ret
```

Figure 2: Function Prologue and Epilogue

Size	Contents	Frame
0–8n	incoming arguments	Previous
8	return address	
0,8	previous `%rbp` value	
0–48	register save area	
0,8	padding	Current
0–96	va-arg registers	
?	local data	
?	register spill area	
0–8	padding	
0–8n	outgoing arguments	

Figure 3: Stack Frame in GCC

2.2 The Stack Frame

Unlike [i386-ABI] we do not enforce any specific organization of stack frames giving compiler maximal freedom to optimize function prologues and epilogues. In order to allow easy spilling of x87 and SSE registers we do specify 128-bit stack alignment at the function call boundary, thus function calls may need to be padded by one extra `push` since AMD64 instruction set naturally aligns stack to 64-bit boundary only.

We additionally specify the red zone of 128 bytes below the stack pointer function can use freely to save data without allocating the stack frame as long as the data are not required to survive function call.

The sample stack frame organization based on extending the usual IA-32 coding practice to 64-bit is shown at Figure 1, the sample prologue code is shown at Figure 2.

GCC Implementation

We found the use of frame pointer and `push`/`pop` instructions to be common bottleneck for the function call performance. The AMD Opteron CPU can execute stores at the rate of two per cycle, while it requires 2 cycles to compute new `%rsp` value in `push` and `pop` operations so the sequence of `push` and `pop` operations executes 4 times slower.

We reorganized the stack frame layout to allow shorter dependency chains in the prologues and epilogues as shown on Figure 3. To save and restore registers we commonly use the sequence of `mov` instructions and we do allocate whole stack frame, including outgoing argument area, using single `sub` opcode as shown in Figure 4. AMD Opteron processor executes the prologue in 2 cycles, while the usual pro-

```
movq   %rbx,-24(%rsp)   Save registers
movq   %r12,-16(%rsp)
movq   %r13,-8(%rsp)
subq   $72, %rsp        Allocate stack frame
...                     Function body
movq   48(%rsp),%rbx    Restore registers
movq   56(%rsp),%r12
movq   64(%rsp),%r13
addq   $72, %rsp        Deallocate stack frame
ret
```

Figure 4: GCC Generated Prologue and Epilogue

logue (Figure 2) requires 9 cycles. Similarly for the epilogues.

Unfortunately the produced code is considerably longer—the size of `push` instruction is 1 byte (2 bytes for extended register), while the size of `mov` is at least 5 bytes. In order to reduce the expenses, GCC does use profile information to use short sequences in the cold function. Additionally it estimates number of instructions executed per one invocation of function and use slow prologues and epilogues when it exceeds given threshold (20 instructions for each saved register).

We found heuristics choosing between fast and short prologues difficult to tune—the prologue/epilogue size is most expensive for small functions where it also should be as fast as possible. As can be seen in the Table 7, the described behavior results in about 1% speedup at the 1.1% code size growth ("prologues using moves" benchmark). Bypassing the heuristics and using moves for all prologues results in additional speedup of 1% and additional 1.1% code size growth ("all prologues using moves" benchmark). The heuristics works better with profile feedback (Table 9). This is something we should revisit in the future.

GCC does always eliminate the frame pointer unless function contain dynamic stack allocation such as `alloca` call. This always result in one extra general purpose register available and fewer instructions executed.

Contrary to the instruction counts, eliminating of frame pointer may result in larger code, because `%rsp` relative addressing encoding is one byte longer than `%rbp` relative one. Thus it may be profitable to not eliminate frame pointer when function do contain many references to the stack frame. Command line option `-fno-omit-frame-pointer` can be used to force use of frame pointer in all functions.

For 64-bit code generation omitting frame pointer results in both smaller and faster code on the average (Tables 7, 8, 9 and 10). In the contrary, for 32-bit code generation it results in code size growth (Tables 11 and 12). This is caused by the fact that increased register file and register argument passing conventions eliminated vast majority of stack frame accesses produced by the 32-bit compiler.

In GCC stack frame layout the register save area and local data are reordered to reduce number of instruction when `push` instructions are used to save registers — the stack frame and outgoing arguments area allocation/deallocation can be done at once using single `sub`/`add` instruction. The disadvantage is that `leave` can not be used to deallocate stack frame in combination with `push` and `pop` instructions. In our benchmarks the new approach brought noticeable speedups for 32-bit code, however it is difficult to repeat the benchmarks since the prologue/epilogue code is dependent on the new stack frame organization and would require some deeper changes to work in the original scheme again.

At the moment GCC is just partly taking advantage of the red zone. We do use red zone for leaf functions having data small enough to fit in it and for saving some temporarily allocated data in instruction generation (so the `sub` and `add` instructions in Figure 4 would be eliminated for leaf functions). For the benefit of kernel programming (signal handlers must take into account the red zone increasing stack size requirements), option `-fno-red-zone` is available to disable usage of red zone entirely.

As can be seen in the Tables 7 and 8, red zone results in slight code size decrease and speedups. The effect depends on how many leaf functions require stack frame. This is uncommon for C programs, but it happens more frequently in template heavy C++ code where function bodies are large due to in-lining (Tables 10 and 9).

We do not use the red zone for spilling registers nor for storing local variables in non-leaf functions as GCC is not able to distinguish between data surviving function calls and data that does not. Extending GCC to support it may be interesting project and may reduce stack usage of programs, however we have no data on how effective the change can be.

To further reduce the expenses, GCC does schedule the prologue and epilogue sequence to overlap with function body. In the future we also plan to implement shrink-wrapping optimization as the expense of saving up to 6 registers may be considerable.

2.3 Stack Unwinding Algorithm

To allow stack unwinding, we do use additional information saved in the same format as specified by DWARF debugging information format [DWARF2]. Instead of `.debug_frame` section specified by DWARF we do use `.eh_frame` section so the data are not stripped.

The DWARF debugging format defines unwinding using the interpreted stack machine describing algorithms to restore individual registers and stack frames. This mechanism is very generic and allows compiler to do pretty much any optimization on stack layout it is interested in. In particular we may eliminate stack frame pointer and schedule prologues and epilogues into the function body.

The disadvantage is the size of produced information and speed of stack unwinding.

GCC Implementation

Implementation in GCC was straightforward as DWARF unwinding was already used for exception handling on all targets except for IA-64. We extended it by support for emitting unwind info accurate at each instruction boundary (by default GCC optimize the unwind table in a way so it is accurate only in the places where exceptions may occur). This behavior is controlled via `-fasynchronous-unwind-tables`.

GCC perform several optimizations on the unwind table size and the tables are additionally shortened by assembler, but still the unwind table accounts for important portion of image file size.

As can be seen in the Table 7 it consumes, at the average, 7.7% of the stripped program binaries size, so use of `-fno-asynchronous-unwind-tables` is recommended for program where unwinding will never be necessary.

The GCC unwind tables are carefully generated to avoid any runtime resolved relocations to be produced, so with the page demand loading tables are never load into memory when they are not used and consume the disc space only.

Main problem are the assembly language functions. At the present programmer is required to manually write DWARF byte-code for any function saving register or having nonempty stack frame in order to make unwinding work. This is difficult and most of assembly language programmers are unfamiliar with DWARF. It appears to be necessary to extend the assembler to support describing of the unwind information using the pseudo-instructions similar to approach used by [IA-64-ABI].

2.4 Register Usage

The decision on split in between volatile (caller saved) and non-volatile (callee saved) register presented quite difficult problem. The AMD64 architecture have only 15 general purpose registers and 8 of them (so called extended registers) require `REX` prefix increasing instruction size. Additionally the registers `%rax`, `%rdx`, `%rcx`, `%rsi` and `%rdi` implicitly used by several IA-32 instructions. We decided to make all of these registers volatile to avoid need to save particular register only because it is required by the operation. This leaves us with only `%rbx`, `%rbp` and the extended registers available for non-volatile registers. Several tests has shown smallest code to be produced with 6 global registers (`%rbx`, `%rbp`, `%r12`–`%r15`).

Originally we intended to use 6 volatile SSE registers, however saving of the registers is difficult: the registers are 128-bit wide and usually only first 64-bits are used to hold value, so saving registers in the caller is more expensive.

We decided to delay the decision until hardware is available and run several benchmarks with different amount of global registers. We also experimented with the idea of saving only lower half of the registers. Our experiments always did lead to both longer and slower code, so in the final version of ABI all SSE registers are volatile.

Finally the x87 registers must be volatile because of their stack organization and the direction flag is defined to be clear.

2.5 Argument Passing Conventions

To pass argument and return values, the registers are used where possible. Registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` are used to pass integer arguments. In particular, register `%rax` is not used because it is often required as special purpose register by IA-32 instructions so it is inappropriate to hold function arguments that are often required to be kept in the register for a long time. Registers `%xmm0`–`%xmm5` are used to pass floating point arguments. x87 registers are never used to pass argument to avoid need to save them in variadic functions.

To return values registers `%rax`, `%rdx`, `%xmm0`, `%xmm1`, `%st0` and `%st1` are used. The usage of `%rax` for return value seems to be considerable win even at the expense of extra `mov` instruction needed for functions returning copy of the first argument and functions returning aggregates in memory via invisible reference.

The aggregates (structures and unions) smaller than 16 bytes are passed in registers. The decision on what register class (SSE, integer or x87) to use to pass/return the aggregate is rather complicated; we do pass each 64-bit part of structure in separate register, with the exception of `__m128` and `long double`.

The argument passing algorithm classifies each field of the structure or union recursively into one of the register classes and then merge the classes that belongs to the same 64-bit part. The merging is done in a way so integer class is preferred when both integer and SSE is used and structure is forced to be passed in memory when difficult to resolve conflicts appears. The aggregate passing specification is probably the most complex part of the ABI and we hope that the benefits will outweight the implementation difficulties. For GCC it requires roughly 250 lines of C code to implement.

Arguments requiring multiple registers are passed in registers only when there are enough available registers to pass argument as a whole in order to simply `va_arg` macro implementation.

Variable sized arguments (available in GCC only as GNU extension) are passed by reference and everything else (including aggregates) is passed by value.

GCC Implementation

It is difficult to obtain precise numbers, but it is clear that the register passing convention is one of the most important changes we made relative to [i386-ABI] improving both performance and code size. The amount of stack manipulation is also greatly reduced resulting in shorter debug information. On the other hand, the most complex part, passing of aggregates, has just minor effect on C code. We believe it will become more important in future for C++ code.

At the moment GCC does generate suboptimal code in number of cases where aggregate is passed in the multiple registers — the aggregate is often offload to memory in order to load it into proper registers. Beside that GCC should implement all nuances of argument passing correctly.

For functions passing arguments in memory, the stack space is allocated in prologue; deallocated in epilogue and plain `mov` operations are used to store arguments. This is in contrast to common practice to use `push` operation for argument passing to reduce code size. Despite that experimental results shows both speedups and code size reductions of the AMD64 binaries when `mov` instructions are used (See `-maccumulate-outgoing-args` in the Table 7, 8, 9, and 10). This is in sharp contrast to IA-32 code generation experience (Tables 11 and 12).

There are multiple reasons for the image size to be reduced. Usage of `push` instructions increases unwind table sizes (about 3% of the binary size). Most of the functions has no stack arguments, however they still do require stack frame to be aligned. This makes GCC to emit number of unnecessary stack adjustments. Last reason seems to be fact that majority of values passed on the stack are large structures where GCC is not using push instructions at all.

2.6 Variable Argument Lists

More complex argument passing conventions require nontrivial implementation variable argument lists. The `va_list` is defined as follows:

```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
} va_list[1];
```

The `overflow_arg_area` points to the end of incoming arguments area. Field `reg_save_area` points to the start of register save area.

Prologue of function then uses 6 integer moves and 6 SSE moves to save argument registers. In order to avoid lazy initialization of SSE unit in the integer only programs, hidden argument in the register `%al` is passed to functions that may use variable argument lists specifying amount of SSE registers actually used to pass arguments.

We decided to use the array containing structure for `va_list` type same way as [PPC-ABI] do to reduce expenses of passing `va_list` to the functions — arrays are passed by reference, while structures by value. This is valid according to the C standard, but brings unexpected behavior in the following function:

```
#include <stdarg.h>
void t (va_list *);
void q (va_list a)
{
  t(&a);
}
```

The function `t` expects address of the first element in the array, while in the second one, the array argument is merely an shortcut for a pointer so it passes pointer to the pointer to the first argument. This unexpected behavior did not trigger in Open Source programs since these already has been cleaned up to work on Power-PC, but has been hit by proprietary software vendors who claimed this to be a compiler bug even when GCC correctly emit an warning message "passing arg 1 of 't' from incompatible pointer type"

GCC Implementation

The register save area is placed on fixed place in stack frame as shown in Figure 3. There is no particular reason for that, but it was slightly easier to implement in GCC.

The computed jump is used in the prologue to save only registers needed. This results in small savings for programs calling variadic function with floating point operands, but makes program calling variadic functions using non-variadic prototypes to crash. Such programs are not standard conforming, but they happen in practice. We noticed the problem for strace and Objective C runtime. We may consider replacing the jump table by single conditional to avoid such crashes.

Second important compatibility problems arises from implicit type promoting. All 64-bit targets supported by SuSE Linux do promote operands to 64-bit values and several packages depend on it. Most notable example is GNOME. While promoting all function operands to 64-bit would be too expensive, we may consider promoting the operands of variadic functions to avoid such compatibility issues.

2.7 Code Models

The 32-bit sign extended immediates and zero extending loads of the immediate allows convenient addressing of only first 2^{31} bytes of the address space. The other areas needs to be addressed via `movabs` instructions or instruction pointer relative addressing. In order to allow efficient code generation for programs that do fit in this limitation (almost all programs today) we define several code models:

small All relocations (code and data) are expected to fit in the first 2^{31} bytes. This is the default model GCC use. This code model can be produced via `-mcmodel=small` command line option.

kernel All relocations (code and data) are expected to fit in the last 2^{31} bytes. This is useful for kernel address space to not overlap with the user address space. This code model can be produced via `-mcmodel=kernel` command line option.

medium Code relocations fit in the first 2^{31} bytes and data relocations are arbitrary. This code model can be produced via

`-mcmodel=medium` command line option. The medium code model has significant code size (about 10%) and noticeable performance (about 2%) penalty (see Tables 7, 8, 9 and 10). These penalties are larger than the authors expectations and probably further improvements to the GCC code generation are possible.

large Code relocations and data relocations are arbitrary. This model is currently not supported by GCC as it would require to replace all direct jumps via indirect jumps. We don't expect this model to be needed in foreseeable future. Large programs can be split into multiple shared libraries.

The position independent code generation can be effectively implemented using the instruction pointer relative addressing. We implemented scheme almost identical to IA-32 position independent code generation practices only replacing the relocations to option global offset table address and index in it by single instruction pointer relative relocation. Similarly the instruction pointer relative addressing is used to access static data.

The resulting code relies on the overall size of the binary to be smaller than 2^{31} bytes. An [AMD64-PSABI] extension will be needed in the case this limitation will become a problem. The performance penalty of `-fpic` is about 6% on AMD64 compared to 20% on IA-32 (see Tables 9, 10, 11 and 12).

3 Implemented Optimizations

In this section we describe target specific optimizations implemented for the first hardware implementation of AMD64 architecture — the AMD Opteron CPU.

The AMD Opteron CPU core has rather complicated structure. The AMD64 instructions are first decoded and translated into micro operations and passed to separate integer and floating point on chip schedulers. Integer instructions are executed in 3 symmetric pipes of overall depth 11 with usual latency of 1 cycle, while floating point instructions are issued into 3 asymmetric pipes (first executing floating point add and similar operations, second having support for long latency instructions and multiple and third executing loads and stores). For more detailed description see also [Opteron].

The processor is designed to perform well on the code compiled for earlier IA-32 implementation and thus has reduced dependency on CPU model specific optimizations. Still several code generation decisions can be optimized as described in detail in [Opteron]. We implemented majority of these and here we describe only those we found most effective.

As can be seen in the Table 11, enabling AMD Opteron tuning via `-march=k8` improves integer program performance by about 10% relative to compiler optimizing for i386. Relative to the compiler optimizing for Pentium-Pro the speedup is only about 1.1%. The optimizations common for Pentium-Pro and Opteron include the scheduling (scheduling for Pentium-Pro still improves Opteron performance), avoiding of memory mismatch stalls, use of new conditional move and `fcomi` instructions and `-maccumulate-outgoing-args`.

For floating point programs the most important optimization is use of SSE instruction set (10%) followed by the instruction scheduling (not visible in the Table 12 because the x87 stack register file does not allow effective scheduling, but noticeable in the Table 10).

3.1 Integer Code Instruction Selection

Majority of IA-32 instructions generated by today compilers are well implemented in the Opteron core so the code generation is straightforward.

In the Tables 7, 8, 9 and 10, "full sized loads and moves" refers to the transformation of 8-bit and 16-bit loads into zero extensions; use of 32-bit reg-reg moves for moving 8-bit and 16-bit values and symmetric change for SSE. The transformation is targeted to avoid hidden dependencies in the on-chip scheduler. The transformation has important effect for SSE code and smaller but measurable effect on code manipulating with 8-bit and 16-bit values.

Second important optimization we implemented is elimination of use `push` and `pop` instructions as mentioned in Section 2.2 and 2.5

Other optimization implemented had just minor effect on overall performance.

3.2 SSE floating point arithmetics

Unlike integer unit, the floating point unit has longer latencies (majority of simple floating point operations takes 3 cycles to execute) and is more sensitive to instruction choice.

The operations on whole SSE registers are usually more expensive than operations on the 64-bit halves. This holds for the move operations also, so it is desirable to always use partial moves when just part of SSE register is occupied (this is common for scalar floating point code). In particular it is desirable to use `movlpd` instead of `movsd` to load double precision values, since `movsd` does clear upper half of the register. `movsd` is the used for register to register moves. This remains the upper half of register undefined that may cause problem when the register is used as a whole for instance for logical operation that has no scalar equivalent. The CPU internally keeps values in different format depending on how they are produced (either single, double precision or integer) when register is in wrong format, serve reformatting penalty occurs.

In order to eliminate reformatting penalties we do reformat the register explicitly before each such operation (fortunately the logical operations are rare in generated code as they are used for conditional moves and fabs/neg expansion only) using `movhlpd`. In the future it may be interesting to implement special pass inserting the conversions only when they are actually needed as most of the `movhlpd` instructions emit are redundant. See "partial SSE register moves" in the Tables 7, 8, 9 and 10 for the comparison of this code generation strategy to the usual one recommended by [Pentium4].

For single precision scalars the situation is different. There is no way conveniently to load single precision data into memory without clearing the upper part of register (`movlps` require 64-bit alignment) and thus we maintain the whole registers in single precision. In particular we do use `movss` to load values and `movaps` for register to register moves.

This scheme brings difficulties with `cvtsi2ss` and similar instructions that do rewrite the lower part only. In this case `xorps` is used first to clear the register. Again the large portion of `xorps` instructions issued this way are redundant because the register is already in specified format. The CPU also special case `cvtsd2ss` instruction where the bytes 4–8 of the register are reformatted to single precision too, however bytes 8–16 remains in the previous format. We risk the reformatting penalty here, since bytes 8–16 are rarely in the double precision format because of the use of partial moves described above. We plan to add an command line option to force issuing of the reformatting here. Also we may reconsider this decision in the case we implement the pass for smart placement of reformatting instructions. See Tables 7, 8, 9 and 10, and benchmark "full sized loads and moves" described in Section 3.1.

3.3 Scheduling

Implementation of instruction scheduling was difficult for several reasons. The AMD Opteron CPU has complicated pipeline expanding each operation into multiple micro operations renaming the register operands and executing them separately in rescheduled order. The available documentation is incomplete and the effect of instruction scheduling on such architectures does not appear to be well studied.

As can be seen in Table 10, instruction scheduling enabled via `-fschedule-insns2` remains one of the most important optimizations we implemented for floating point intensive benchmarks. On the other hand the effect is about 10 times lower than on the in-order Alpha CPU (Table 14).

GCC at the present implements only local basic block scheduling that is almost entirely redundant with the out-of-order abilities of the CPU. We experimentally implemented an limited form of trace scheduling and measured an improvement of additional 1% for the SPECfp. Our expectation is that the more global the GCC scheduler algorithm will be, the less redundancies with out-of-order core will be apparent, so the benefits of global algorithms should be comparable to ones measurable on in-order CPUs.

Our implementation represents just a simplified model of the real architecture. We model the allocations of decoders, floating point unit (fadd, fmul and fstore), the multiplier and load store unit. We omit model of the reorder buffers — the micro instructions are assumed to be issued to the execution pipes immediately in the fixed model. This also allows us to omit model of the integer and address generation units as never more than 3 instructions are issued at once.

Most of the stalls the scheduler can avoid are related to loads and stores. In order to avoid the stall it is necessary to model the instruction latencies and the fact that address operands are needed earlier than the data operands. The scheduler can reorder the computations so the data operands are computed in parallel with loads. GCC scheduler does assume that all the results must be available in order to instruction be issued and thus we reduce the latencies of instructions computing values used as data operands of load-execute instructions by up to 3 cycles (the latency of address generation unit). Even when latencies of majority instructions are shorter than 3 cycles and thus we can not reduce the latency enough to compensate the load unit latency, this model is exact for the in-order simplification of CPU described above as the instruction computing data operands must be output before load-execute instruction itself.

4 Experimental Results

We present benchmarks of majority optimizations discussed. We also present the same benchmarks performed on IA-32 and Alpha system where possible to give an comparison of effectivity of individual optimizations on these architectures. We hope this to be useful to apply earlier published results on compiler optimization (such as [FDO]) to the new platform and give a guide of what optimizations are most important. We also present results with two different optimization levels — the standard optimization (`-O2`) used by the majority of distributions today and aggressive optimization (`-O3 -ftracer -funroll-loops -funit-at-a-time` with profile feedback) we found to give best overall SPEC score.

We did use modified prerelease of GCC 3.3 as used by SuSE Linux 8.2 for AMD64. All the runs were performed on SuSE Linux on dedicated machines, however important amount of random noise remains (especially for benchmarks Mesa, Gzip, Perl and Twolf). Due to time limitations the benchmarks were performed with one iteration only except for the benchmarks in the Table 9 and 10 that were computed with 3 iterations. Because the runs were not done on final hardware and because we didn't satisfy the conditions for reportable runs in all tests, we present relative numbers only.

Each table is divided into two sections — first part includes optimizations enabled by default at given optimization level, while the other part contains optimization that user needs to enable by hand either because they are ineffective, inappropriate for given settings or does not obey the language standards. Each table also contains comparison of two runs with equal settings in the first line to present rough approximation of the noise in the numbers. Both performance and sizes of the stripped binaries are presented. The numbers always represent relative speedup (or code size increase) from the run with the specified feature disabled to the run with specified feature enabled. For instance `-fomit-frame-pointer` run in the table 7 compare performance of `-O2 -fno-omit-frame-pointer` to `-O2 -fomit-frame-pointer`. The benchmark "standard optimization" compares `-O0` to `-O2`.

The Following benchmarks were performed:

aggressive optimization compare performance of unoptimized code (`-O0`) to the aggressive optimization settings described above.

all prologue using move eliminate use of all `push` and `pop` operations in the prologues and epilogues except for cases where single register is saved. See Section 2.2.

-fasynchronous-unwind-tables enable production of DWARF2 unwind information. See Section 2.3.

`-fbranch-probabilities` enable profile feedback based optimizations. We implemented majority of transformations described on [FDO] with the exception of function in-lining and switch statement expansion.

`-fgcse` enable global optimizers including (limited form of) partial redundancy elimination, load motion, constant propagation and copy propagation. GCC does contain loop invariant hoisting and extended basic block based value numbering pass making the global optimizers partly redundant.

`-fguess-branch-probability` enable optimizations driven by static profile estimation. The profile is estimated by methods based on [profile] when profile feedback is not available.

`-finline-functions` enable function in-lining.

`-fold-unroll-loops` enable old loop unroller that actually unrolls some loops on Alpha.

`-fomit-frame-pointer` enable elimination of frame pointer by using stack pointer instead. See Section 2.2.

`-foptimize-sibling-calls` transform call to leaf function into jump.

`-fpeel-loops` enable loop peeling.

`-fpic` produce position independent code. See Section 2.7.

`-freorder-blocks` enable intra-function basic block reordering and duplication based on significantly modified software trace cache algorithm [STC].

`-fschedule-insns2` enable post-register allocation local scheduling. See Section 3.3.

`-fschedule-insns` enable pre-register allocation region scheduling (not available for IA-32 and AMD64).

`-fstrength-reduce` enable strength reduction.

`-fstrict-aliasing` enable ANSI-C type based aliasing.

full sized loads and moves avoids use of instructions initializing just portion of the destination registers. See Section 3.2 and 3.1.

`-ftracer` enable super-block formation using algorithm similar to [FDO]. The super-blocks are unified again after optimizations by cross-jumping pass so this transformation is not used to improve scheduling as commonly described in the literature. It is aimed to improve CSE and other transformation by simplifying the control flow.

`-funit-at-a-time` enable optimizations on whole compilation unit. At the moment GCC perform stronger function inlining (in-lining of small functions called before defined and static functions called once) and use register calling conventions for static functions on IA-32. Only effective for C compiler.

`-funroll-all-loops` enable loop unrolling of all small enough loops in the hot spots.

`-funroll-loops` enable loop unrolling for loops with known induction variable. While working on the paper we noticed that our new implementation has important flaw avoiding loops from being unrolled on Alpha architecture.

`-m64` enable 64-bit code generation (used in comparisons relative to IA-32 code).

`-mfpmath=sse` eliminate use SSE(2) instruction set for scalar floating point calculations.

`-mcmodel` controls code and data segment size limits. See Section 2.7.

`-mred-zone` enable use of 128 bytes below stack pointer for local data. See Section 2.2.

partial SSE moves eliminate use of `movlpd` for double precision loads and `movsd` for register to register moves. See Section 3.2.

prologue using move eliminate use of hot `push` and `pop` operations in the prologues and epilogues. See Section 2.2.

standard optimization compare performance of unoptimized code (`-O0`) to the standard optimization settings (`-O2`).

4.1 Real World Performance

One of the main goals has been to develop system ready for both enterprise and desktop

options	slowdown
	0.00%
`-fstrict-aliasing`	-1.13%
`-fasynchronous-unwind-tables`	-0.38%
`-freorder-blocks`	0.00%
`-fomit-frame-pointer`	0.37%
`-mred-zone`	0.38%
`-mfpmath=sse`	0.75%
`-maccumulate-outgoing-args`	0.75%
`-foptimize-sibling-calls`	0.76%
`-fguess-branch-probabilities`	1.54%
`-fschedule-insns2`	2.33%
`-fgcse`	6.88%
`-ffast-math`	-1.88%
`-ftracer`	0.00%
`-frename-registers`	0.74%
`-funroll-loops`	3.38%
`-fpic`	3.39%
`-funroll-all-loops`	5.32%
`-mcmodel=medium`	2.27%
`-fbranch-probabilities`	142.74%

Table 1: Compilation Time Cost (AMD Opteron)

(workstation) use. While the need of 64-bit addressing space for the enterprise is well understood, the effect on desktop performance is often discussed. The main drawback of 64-bit system, as discussed in section 2.1 is the increased memory footprint of the programs and subsequent slowdown of program startup times critical for today desktop systems.

In this section we present few simple benchmarks of this phenomenon on SuSE Linux 8.2. Both the 32-bit and 64-bit version of the system were installed on the equally sized ReiserFS partitions in the default configuration. The tests were performed in the same order on both systems with reboots in between. Additional packages were installed as needed. We hope this procedure to minimize amount of the noise in the numbers.

The Table 2 compares startup times of several programs. As can be seen, the 64-bit system, perhaps surprisingly, is significantly faster in

test	speedup
bootup time	-0.9%
KDE startup from disk	18.1%
KDE startup from cache	14.6%

Table 2: Desktop Performance Relative to 32-bit System

two of them and comparable in bootup times. The Table 3 compares compilation of the package gimp.

As can be seen on Table 4 the memory consumption grows up by about $\frac{1}{4}$ as expected, but due to relative compactness of CISC AMD64 instruction set, the increase is much smaller than one seen after switching to RISC or VLIW systems.

In fact Tables 5 and 6 shows decrease in the code section sizes.

The major growths can be seen in the section `.eh_frame` that is usually not load into the memory and sections related to the dynamic relocations. According to our benchmarks these are not critical, since dynamic loader is still slightly faster in 64-bit version compared to 32-bit.

Overall, we can recommend use of 64-bit system instead of 32-bit on AMD64 machines intended for desktop use as long as memory consumption increased by 25% is not major limitation (that is hardly the case for computers sold today).

5 Runtime Library Optimizations

We made following optimizations to glibc:

- Assembly optimized math functions
- Assembly optimized `memcpy` and

test	speedup real	user	system
`tar xjf`	17.7%	9.8%	4%
`./configure`	-4.3%	0.7%	-31%
`make`	12.9%	19.8%	-39%

Table 3: Gimp Compilation Times Relative to 32-bit System

Table 4: Memory Resources Consumption

test	32-bit	64-bit	increase
konqueror	14 M	18 M	28%
gimp	8.6 M	9.9 M	15%
mozilla	22 M	27 M	22%

section	32-bit	64-bit	increase
`.text`	56216 K	53419 K	-5%
`.bss`	18169 K	21098 K	16%
`.data`	10239 K	14076 K	37%
`.rodata`	17543 K	19734 K	12%
`.eh_frame`	546 K	8269 K	1414%
`.rela.plt`	358 K	1076 K	200%
`.rela.dyn`	40 K	126 K	215%
total	80435 K	91141 K	13%

Table 5: Size of Common Binaries in `/usr/bin`

section	32-bit	64-bit	increase
`.text`	71967 K	67526 K	-7%
`.bss`	33463 K	11557 K	-72%
`.dynstr`	13608 K	13587 K	-1%
`.rodata`	12119 K	12217 K	0%
`.dynsym`	11424 K	7611 K	66%
`.eh_frame`	6367 K	12730 K	99%
`.data`	6018 K	9695 K	61%
`.rela.dyn`	4382 K	12844 K	193%
`.plt`	3898 K	6499 K	66%
`.rela.plt`	1293 K	3888 K	200%
`.got`	823 K	1654 K	100%
total	171812 K	198111 K	15%

Table 6: Size of Common Shared Libraries

`memset` functions that do use prefetch and streaming moves for large blocks

- We found `malloc` implementation in glibc 2.2 to be bottleneck. `malloc` in glibc 2.3 solves this problem.

6 Conclusion

The performance of 64-bit code produced by GCC is superior to 32-bit for CPU bound integer and numeric programs (even in comparison to the best optimizing 32-bit compilers available).

Most important optimizations include usage of newly available extended registers, register argument passing conventions, use of SSE for scalar floating point computations and relaxed stack frame layout restrictions by using DWARF2 unwind information for stack unwinding. The code section of 64-bit binaries is, on the average, 5% smaller than code section of 32-bit binary.

Most noticeable problem is the growth of data structures caused by 64-bit pointers. This problem is noticeable as regression in mcf, parser and gap SPEC2000 benchmarks as well as about 25% increase in memory overhead of usual desktop applications and 10% increase of executable file sizes.

Despite that the overall system performance seems to be improved even for (nontrivial) benchmarks targeted to measure extra overhead of increased memory bandwidth, such as program startup times (0%–20% speedup), compilation (12%) or SPEC2000 integer benchmark suite (3.3%). Still it can be worthwhile to implement LP32 code model to provide an alternative for memory bound applications.

The aggressive optimizations in argument passing conventions also brought several compatibility problems especially when dealing with variable argument lists. Other common problem is lack of support for DWARF2 in `gas` assembler making use of assembly functions in AMD64 code difficult.

By eliminating the common bottleneck of IA-32 code (such common memory accesses caused by register starve ISA and argument passing conventions), the code became more sensitive to compiler optimizations. Number of optimizations we evaluated are more effective in 64-bit than on 32-bit especially those improving instruction decoding bandwidth (AMD64 code usually consists of more instructions with shorter overall latency), instruction scheduling and those that increase register pressure.

In comparison to DEC Alpha EV56 architecture, AMD Opteron is considerably less sensitive on instruction scheduling and in-lining. The first is caused by out-of-order architecture and the second probably by smaller L1 cache.

7 Acknowledgements

The port of GCC to AMD64 was done by a team of developers and I'd like to acknowledge their contributions. Without the numerous discussions and the joint development the port and therefore this paper would not be possible.

Geert Bosch designed stack unwinding and exception handling ABI. Richard Brunner, Alex Dreyzen and Evandro Menezes provided a lot of help in understanding the AMD Opteron hardware. Zdeněk Dvořák implemented the new loop unrolling pass, improved DWARF2 support, and made a number of improvements to profile-based optimizations framework. Andrew Haley finished the gcj (Java compiler) port started by Bo Thorsen. Richard Henderson reviewed majority of the GCC changes. Jan Hubička implemented the first versions of GCC and Binutils ports, co-edited the ABI document, realized the AMD Opteron specific optimizations and some generic ones (unit at a time mode, profile feedback optimizations framework, tracer). Andreas Jaeger ported glibc, provided SPEC2000 testing framework, co-edited the ABI document, and fixed a number of GCC and Binutils bugs. Jakub Jelínek designed and implemented the thread local storage ABI. Michal Ludvig and Jiří Šmíd realized the GDB port. Michael Matz worked on the new register allocator and fixed plenty of GCC bugs. Mark Mitchell edited the ABI document and set up WWW and CVS for the project. Andreas Schwab and Bo Thorsen fixed a number of problems in the linker and assembler. Josef Zlomek redesigned the basic block reordering pass and fixed a number of bugs in GCC.

Andreas Jaeger and Evandro Menezes also reviewed the paper and helped to clarify it.

References

[AMD64] *AMD x86-64 Architecture Programmer's Manual*, AMD (2003).

[Opteron] *Software Optimization Guide for the AMD Opteron^{TM} Processor*, AMD (2003).

[Pentium4] *IA-32 Intel Architecture Optimization Manual*, Intel (2003).

[AMD64-PSABI] *UNIX System V Application Binary Interface; AMD64 Architecture Processor Supplement, Draft*, (Ed. J. Hubička, A. Jaeger, M. Mitchell), `http://www.x86-64.org`, (2003)

[i386-ABI] *UNIX System V Application Binary Interface; IA-32 Architecture Processor Supplement*, Intel (2000).

[IA-64-ABI] *UNIX System V Application Binary Interface; IA-64 Processor ABI Supplement*, Intel (2000).

[PPC-ABI] *UNIX System V Application Binary Interface; PowerPC Processor ABI Supplement* (1995).

[DWARF2] *DWARF Debugging Information Format, Version 2.0.0* UNIX International, Program Languages SIG (1993).

[FDO] *Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha*, Journal of Instruction-Level Parallelism 3 (2000), p. 1–25.

[STC] A. Ramirez, J.L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, *Software trace cache*, Proc. 13th Intl. Conf. on Supercomputing (1999), p. 119–126.

[profile] Y. Wu and J.R. Larus, *Static branch frequency and program profile analysis*, In Proceedings of the 27th International Symposium on Microarchitecture (1994), p. 1–11.

Table 7: 64-bit SPECint 2000 with Standard Optimization (AMD Opteron)

Table 7: Performance (relative speedups in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	1.32	0.14	-0.45	-0.45	-0.17	0.19	0.41	0.11	0.60	0.28	0.27	-0.54	0.13
standard optimization	105.37	82.29	90.55	12.06	87.14	58.23	451.70	97.05	101.18	75.30	142.14	55.99	93.40
`-fguess-branch probabilities`	4.40	4.45	2.90	0.00	2.73	0.19	5.58	5.96	7.43	21.60	2.56	-1.46	4.10
`-fschedule-insns2`	1.62	1.44	2.40	0.22	0.32	0.78	4.90	1.28	-0.45	4.34	0.41	0.93	1.46
`-fstrict-aliasing`	1.48	4.62	1.93	0.00	3.68	0.58	-2.34	1.75	0.75	4.27	4.79	-2.34	1.19
`-mfpmath=sse`	1.93	3.98	-0.23	0.00	-0.09	-0.39	2.11	0.00	1.81	3.94	0.27	0.80	1.06
prologue using move	-0.74	0.14	0.34	0.00	4.04	0.98	-0.43	1.43	0.30	5.71	-0.28	0.13	0.93
full sized loads and moves	-1.76	-0.29	-0.46	0.88	0.96	-0.20	24.90	-1.52	-0.45	-1.04	0.97	-3.61	0.93
`-fgcse`	1.17	4.28	-1.77	1.35	0.48	1.38	2.33	1.75	-1.48	1.55	1.26	0.13	0.92
`-foptimize sibling-calls`	1.62	0.43	-0.12	0.00	3.33	0.00	2.33	-0.35	1.51	2.44	0.27	0.26	0.92
`-finline-functions`	1.62	0.71	0.22	1.11	0.32	3.08	0.30	-1.04	0.58	-0.99	2.21	0.67	0.65
`-fomit-frame-pointer`	0.29	1.58	0.56	0.67	5.00	1.57	-3.03	3.07	-0.60	0.47	2.41	-3.48	0.39
`-freorder-blocks`	3.61	-0.29	-0.57	0.22	2.31	-0.78	0.72	4.06	0.75	3.45	1.84	-5.31	0.39
`-maccumulate-outgoing-args`	1.92	-0.58	0.78	0.45	0.24	-0.39	1.04	-0.12	-0.60	-1.13	0.13	0.80	0.26
`-mred-zone`	1.47	0.14	1.35	-0.23	1.30	-0.20	-1.73	0.00	-0.30	-0.29	0.55	-0.67	0.13
partial SSE moves	-0.30	5.89	-0.92	0.00	0.07	0.00	-1.17	0.00	0.00	-0.10	-0.14	-3.36	-0.27
aggressive optimization	6.34	4.97	8.81	0.67	1.29	25.43	24.14	12.29	7.51	5.69	5.42	4.65	8.40
`-fbranch-probabilities`	5.95	1.71	7.13	0.22	-0.65	16.76	2.98	3.90	0.14	6.95	0.27	3.73	4.07
`-funroll-all-loops`	4.16	0.42	5.60	0.00	-4.28	0.77	16.42	4.02	1.35	0.57	1.82	1.46	2.50
`-funroll-loops`	3.71	0.28	4.17	0.00	0.08	0.58	15.35	1.61	1.35	-4.78	0.55	3.32	2.23
all prologue using move	-0.60	0.56	2.38	-0.23	-0.40	0.58	3.73	3.19	-0.15	-4.29	0.55	4.68	1.05
`-ffast-math`	1.78	0.28	0.67	0.00	-0.25	-0.20	0.31	-0.81	0.15	2.67	1.12	2.64	0.78
`-frename-registers`	-0.15	0.56	-0.68	0.00	0.08	0.58	1.34	-2.19	-0.76	-1.25	0.97	4.92	0.65
`-funit-at-a-time`	0.89	2.71	0.79	0.45	0.72	0.38	0.00	-0.47	-0.45	0.68	0.69	-0.93	0.39
`-ftracer`	3.12	0.14	1.57	0.00	1.13	-0.20	1.76	0.91	-7.81	-3.83	1.40	2.40	0.13
`-cmodel=medium`	-4.30	-1.00	-0.45	0.00	-10.84	0.00	2.18	-3.57	-5.83	-6.27	-2.23	-0.27	-2.51
`-fpic`	-9.11	-1.72	-1.68	0.89	-18.21	-0.78	-1.36	-16.79	-3.76	-15.16	-6.18	-1.48	-6.20

Table 7: File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
standard optimization	-11.24	-23.04	-23.74	-20.59	-17.13	-13.77	-13.71	-20.00	-36.54	-9.42	-15.83	-39.29	-22.31
`-maccumulate-outgoing-args`	-0.42	-4.02	-3.47	-3.34	-0.35	-3.30	-3.15	-3.29	-4.31	-3.60	5.16	-2.51	-3.25
`-fomit-frame-pointer`	-0.26	1.72	-1.13	-0.20	0.04	-3.76	-1.94	-1.24	-1.07	2.08	-0.08	-0.99	-0.71
`-fstrict-aliasing`	0.00	-0.68	-0.15	0.00	0.00	0.00	0.22	0.00	-0.34	-0.66	0.00	-5.02	-0.40
`-mred-zone`	0.00	-0.11	-0.19	0.00	-0.02	0.00	-0.76	0.59	-0.02	0.00	0.00	-0.04	-0.09
`-fschedule-insns2`	0.00	0.02	-0.15	0.00	0.01	0.00	0.02	0.00	0.00	0.02	0.00	-0.07	-0.05
`-fgcse`	-0.11	0.04	-0.16	0.19	0.03	0.11	0.44	0.68	-0.01	-0.68	0.00	-1.16	-0.05
`-foptimize sibling-calls`	0.00	-0.03	0.08	0.00	-0.02	0.00	-0.76	0.48	-0.16	-0.01	-0.23	-0.10	-0.03
partial SSE moves	0.00	0.00	0.00	0.00	0.00	0.00	0.16	0.00	0.00	0.00	0.00	0.01	0.02
full sized loads and moves	0.00	0.00	0.04	0.00	1.21	0.00	0.00	0.00	0.08	-0.01	0.00	0.11	0.08
`-mfpmath=sse`	0.00	-0.64	-0.15	0.00	0.00	0.00	2.34	-0.01	0.00	0.00	0.00	-1.64	0.13
prologue using move	-0.11	1.06	1.01	0.00	1.26	-0.34	0.91	0.84	1.44	2.55	0.00	0.16	1.14
`-freorder-blocks`	7.06	2.71	4.43	0.00	4.05	3.67	1.07	5.72	3.42	5.60	10.89	4.22	4.19
`-finline-functions`	-0.73	1.15	8.85	-0.20	0.24	28.60	0.12	6.55	3.37	1.99	29.84	0.68	5.49
`-fguess-branch probabilities`	7.00	4.41	5.82	0.00	3.60	3.34	2.64	6.67	5.85	8.74	10.89	3.97	5.66
`-fasynchronous unwind-tables`	7.12	10.28	7.38	6.31	3.76	17.16	4.83	9.26	9.04	7.88	18.14	5.34	7.71
`-fbranch-probabilities`	-4.91	-2.07	-2.20	0.82	0.11	0.02	-2.44	-3.92	-3.74	-4.72	-7.30	-1.80	-2.85
`-funit-at-a-time`	-22.64	-4.95	-1.50	0.00	0.00	0.00	0.00	-0.82	-0.08	-0.01	0.00	-0.10	-1.09
`-ffast-math`	0.00	-0.03	0.00	0.00	0.00	0.00	0.00	-0.68	0.00	-0.02	0.00	0.01	-0.09
`-frename-registers`	0.00	0.26	0.97	0.00	0.28	0.00	1.99	0.68	0.24	0.04	0.00	1.83	0.78
all prologue using move	-0.73	4.14	1.14	-0.96	-0.33	2.18	1.35	0.87	1.60	0.52	-0.77	2.38	1.17

Table continues on next page…

Table 7 Continued—File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
`-ftracer`	0.00	1.27	1.29	0.00	0.13	0.00	2.50	2.01	2.46	1.31	0.00	1.54	1.56
`-funroll-loops`	13.30	7.92	3.18	1.34	4.22	7.11	1.26	2.70	12.57	0.02	9.82	8.70	4.21
`-funroll-all-loops`	13.30	9.53	4.29	24.50	4.71	14.20	1.43	3.38	15.76	0.66	9.82	14.40	5.71
`-fpic`	12.11	6.53	3.62	1.14	21.40	9.38	1.92	6.48	15.53	9.16	7.06	16.66	7.55
`-mcmodel=medium`	13.62	8.10	7.10	0.00	17.57	7.44	6.35	8.29	8.35	6.64	9.90	13.33	8.09
aggressive optimization	-14.42	4.03	21.89	5.12	6.44	44.45	-0.47	8.80	7.38	0.73	40.05	3.93	11.08

Table 8: 64-bit SPECfp 2000 with Standard Optimization (AMD Opteron)

Table 8: Performance (relative speedups in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	-0.28	-0.13	0.00	0.00	0.23	-2.07	0.14	0.00	0.00	0.00	-0.16
standard optimization	102.22	54.49	633.14	220.37	79.20	22.69	90.76	111.08	204.34	192.64	142.52
`-mfpmath=sse`	9.30	0.12	3.31	2.38	11.68	102.55	0.28	8.32	11.53	6.01	12.43
`-fguess-branch-probabilities`	7.62	0.00	6.42	2.78	7.48	0.42	-2.23	-1.27	-0.29	4.72	2.75
partial SSE moves	2.86	0.13	2.95	3.21	3.34	-3.26	0.86	3.11	3.86	3.33	2.12
full sized loads and moves	2.13	0.26	1.35	1.98	6.38	0.69	0.00	2.00	1.45	1.55	1.78
`-fstrict-aliasing`	0.00	0.12	0.00	0.19	2.22	5.22	-2.23	0.90	0.00	5.08	1.44
`-fschedule-insns2`	2.23	0.00	7.72	0.78	0.34	-1.40	-2.50	0.90	4.50	1.01	1.28
`-freorder-blocks`	0.97	0.12	0.18	0.19	13.09	2.28	0.28	0.00	-1.42	0.00	1.28
`-fomit-frame-pointer`	2.51	0.00	4.53	0.38	-0.58	-1.80	-1.13	0.90	-0.29	3.63	0.95
prologue using move	-3.24	0.00	0.00	0.00	3.58	0.69	0.00	-0.14	0.00	0.00	0.15
`-finline-functions`	0.13	0.12	0.00	0.19	1.85	-1.51	1.84	-0.52	0.28	-0.17	0.15
`-foptimize sibling-calls`	0.82	0.12	0.18	0.19	-0.46	-0.97	0.00	0.12	0.00	0.00	0.00
`-mred-zone`	0.00	0.00	0.00	0.38	0.57	0.97	-2.10	-0.26	0.00	0.16	0.00
`-maccumulate-outgoing-args`	0.55	-0.13	0.18	0.00	0.45	-3.46	0.00	0.00	-0.29	0.33	-0.16
`-fgcse`	1.37	0.00	-7.19	-5.15	-0.23	0.69	0.42	-0.64	-4.14	-2.13	-1.71
aggressive optimization	5.57	-0.91	6.60	4.26	4.14	-1.93	7.96	3.58	10.63	-2.34	3.15
`-funroll-all-loops`	2.72	-0.13	1.88	2.32	-1.50	5.58	0.42	3.58	-0.29	1.16	1.58
`-funroll-loops`	2.72	0.00	1.88	2.51	-0.92	2.67	2.13	3.58	-0.29	1.16	1.57
`-ffast-math`	0.81	0.00	0.00	2.13	1.26	-3.16	0.99	4.74	0.57	1.50	0.94
all prologue using move	4.18	0.00	-0.39	0.19	0.23	-0.98	1.86	-0.27	1.14	0.34	0.63
`-fbranch-probabilities`	-3.44	0.12	-0.94	0.38	15.14	-1.40	-0.15	-0.65	0.85	-3.35	0.15
`-funit-at-a-time`	0.13	0.12	-0.19	0.00	3.93	-3.54	0.14	0.12	0.00	-0.17	0.15
`-frename-registers`	-3.54	-0.26	5.66	-0.39	-7.23	-1.11	4.97	3.46	0.86	-0.34	0.15
`-ftracer`	-0.82	0.00	0.00	0.00	-2.87	-2.35	-0.15	0.77	0.86	-0.67	-0.64
`-cmodel=medium`	2.73	-0.26	-0.19	-0.39	-3.69	-0.83	-0.72	-1.03	-14.95	-0.17	-1.90
`-fpic`	0.95	0.00	0.37	-0.97	1.72	-0.29	0.71	-0.13	-20.98	-0.17	-1.90

File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
standard optimization	-25.71	-26.52	-36.03	-60.14	-34.62	-15.82	-33.14	-32.33	-38.32	-30.33	-36.85
`-maccumulate-outgoing-args`	-1.63	-0.71	-1.83	-0.71	-3.40	-2.07	-1.80	-2.77	-1.12	-1.17	-1.89
`-fschedule-insns2`	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.02	-0.43	0.00	-0.21
`-mred-zone`	0.00	0.00	-0.19	-2.31	-0.13	-0.08	-0.14	-0.12	-0.03	-0.12	-0.14
`-fgcse`	0.00	-8.64	-4.00	-10.19	-0.74	1.91	-0.38	0.00	1.70	-3.61	-0.07

Table continues on next page

Table 8 Continued—File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	-0.13	0.07	0.00	-0.05	0.00	0.00	-0.04
`-foptimize sibling-calls`	0.00	0.00	0.00	0.00	-0.24	0.00	0.00	0.04	-0.02	0.68	-0.02
full sized loads and moves	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.75	0.08
`-fomit-frame-pointer`	0.00	0.47	0.75	-1.97	-0.05	0.39	-0.14	0.37	0.12	5.74	0.43
partial SSE moves	0.00	0.23	0.00	0.71	0.79	0.00	0.00	0.24	0.43	0.89	0.53
prologue using move	-0.28	0.00	0.00	0.11	1.78	0.00	0.00	0.26	-0.02	0.70	0.53
`-freorder-blocks`	0.00	0.47	0.00	0.11	2.44	0.00	0.00	2.62	0.86	1.37	1.38
`-mfpmath=sse`	0.00	2.16	0.00	6.26	-1.57	0.00	-0.14	3.19	2.65	4.39	1.60
`-fguess-branch probabilities`	-0.28	1.43	0.00	-0.36	5.10	12.16	10.56	3.04	0.41	1.19	2.09
`-finline-functions`	0.00	0.00	0.00	0.00	5.39	19.96	0.13	0.42	1.29	1.50	2.45
`-fasynchronous-unwind-info`	9.34	3.15	6.75	1.92	10.46	16.55	13.01	6.21	1.25	3.83	4.67
`-fbranch-probabilities`	0.64	0.15	0.76	0.19	-5.23	0.70	0.61	-2.11	-0.28	-0.06	-1.58
`-ffast-math`	0.00	-0.95	0.00	0.58	-0.83	-13.04	-0.27	-5.57	0.86	0.00	-0.35
`-funit-at-a-time`	0.00	0.00	0.00	0.00	-0.07	0.00	0.00	-0.03	0.00	0.00	-0.03
all prologue using move	-0.28	1.40	0.37	1.29	0.78	-1.02	-0.40	2.26	0.61	1.96	0.86
`-ftracer`	0.00	0.00	0.00	0.00	2.37	0.07	0.00	5.45	0.43	3.35	1.51
`-frename-registers`	0.00	0.47	0.00	2.65	1.78	0.00	0.00	2.60	2.58	0.86	2.10
`-funroll-loops`	1.93	24.69	6.32	6.42	7.95	20.05	0.65	11.14	3.02	6.63	5.63
`-funroll-all-loops`	1.93	24.69	7.25	6.42	8.19	20.05	2.35	11.14	3.02	6.63	5.73
`-fpic`	0.45	0.23	0.93	2.24	5.92	9.28	7.71	4.91	8.04	3.75	6.51
`-mcmodel=medium`	0.09	4.93	0.00	7.49	3.53	0.85	1.83	5.45	24.62	6.36	14.32
aggressive optimization	71.81	164.20	125.37	57.30	11.28	97.53	52.54	12.91	26.21	34.10	26.45

Table 9: 64-bit SPECint 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	-0.28	-0.41	0.20	-0.45	0.00	-0.16	0.00	-0.11	0.84	0.00	0.13	0.38	0.12
aggressive optimization	112.35	91.73	103.60	14.72	86.01	97.56	589.65	130.46	111.79	74.46	151.98	56.79	106.81
`-fbranch-probabilities -fguess-branch...`	8.40	2.62	10.71	0.22	3.38	21.72	27.67	27.67	14.24	10.37	4.39	-1.56	9.49
full sized loads and moves	1.00	0.67	-0.53	0.00	0.97	-0.48	56.39	1.79	0.71	0.62	0.13	4.64	4.61
`-fbranch-probabilities`	2.69	0.00	5.62	-0.45	2.62	19.85	-0.92	11.94	4.06	2.29	1.07	0.51	3.77
`-m64`	9.90	0.27	3.39	-22.19	42.29	-2.13	45.66	0.30	-1.25	6.29	8.28	-13.33	3.38
`-funroll-loops`	1.69	0.54	0.41	0.22	0.88	1.41	16.94	7.59	0.56	1.73	0.93	4.62	3.12
`-freorder-blocks`	4.95	1.22	4.51	0.22	3.89	1.89	2.40	13.06	-0.56	-1.42	0.40	1.15	2.48
`-fomit-frame-pointer`	0.13	0.00	2.19	0.44	2.03	1.73	2.31	5.38	-0.28	1.08	1.47	5.05	2.10
`-fstrict-aliasing`	-0.56	4.80	0.82	0.44	1.04	1.89	1.61	2.08	1.72	1.64	5.88	1.15	1.85
`-finline-functions`	-0.42	0.54	1.55	2.02	1.86	5.21	1.01	-0.31	0.42	3.62	3.13	2.75	1.85
`-ftracer`	-0.69	-0.27	0.30	0.00	1.12	0.78	5.20	3.93	0.14	0.27	0.53	4.90	1.60
`-fschedule-insns2`	0.27	2.62	0.41	0.22	4.24	0.46	2.57	1.55	0.99	3.34	1.61	0.64	1.47
`-mred-zone`	-0.42	0.13	0.61	0.66	0.96	0.31	-1.33	1.56	-0.56	7.01	-0.14	3.56	1.22
`-fgcse`	2.70	4.06	1.14	-0.23	3.47	-0.77	-0.51	-0.82	2.29	1.27	0.93	0.25	1.10
`-mfpmath=sse`	-0.28	2.48	-0.52	0.66	1.95	0.78	9.05	0.72	0.14	-2.80	-0.14	1.42	1.10
`-frename-registers`	-0.42	1.22	-1.13	-0.45	4.24	0.46	-1.90	-0.72	-0.97	1.91	1.47	4.81	0.98
`-funit-at-a-time`	-0.56	3.50	-1.23	0.22	1.12	0.93	0.16	-1.42	2.73	3.43	-0.27	2.64	0.98
prologue using move	-0.43	0.54	1.06	0.43	1.06	0.79	-2.75	1.89	3.63	6.29	-0.14	-0.26	0.86
partial SSE moves	-0.29	0.81	0.10	-0.44	0.00	0.63	0.00	0.62	0.00	0.26	-0.40	4.78	0.73
`-foptimize sibling-calls`	0.00	-0.14	0.61	0.22	0.96	0.78	1.96	0.00	-1.93	-1.86	-0.27	3.15	0.60
`-maccumulate-`	-0.28	0.94	-0.11	-0.23	2.53	0.46	1.18	-0.72	2.43	-0.81	0.13	0.63	0.48

Table continues on next page...

Table 9 Continued—Performance (relative speedups in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
`outgoing-args`													
`-fstrength-reduce`	-0.42	0.26	-1.22	0.00	0.64	0.00	-0.59	-1.81	0.42	4.30	-0.14	-0.13	0.00
all prologue using move	-1.13	-0.27	-0.32	-0.22	1.28	0.94	6.46	-0.11	1.54	-1.33	0.39	0.50	0.61
`-ffast-math`	-0.28	0.40	-1.24	-0.23	-1.92	0.00	0.08	0.10	0.56	1.34	-0.27	-3.56	-0.73
`-fpeel-loops`	0.00	0.13	-1.13	0.22	-1.20	-0.62	0.08	-1.34	-1.69	-3.86	-0.40	-0.26	-0.73
`-funroll-all-loops`	0.00	0.13	0.10	0.00	-0.48	-0.16	-0.84	2.04	-2.12	-5.58	0.26	-7.90	-1.70
`-cmodel=medium`	-5.12	-1.21	-2.97	0.44	-10.61	-0.78	-1.09	0.00	0.28	-4.85	-0.67	-7.74	-3.28
`-fpic`	-12.73	-1.89	-2.36	-0.89	-13.88	-6.96	-4.36	-12.79	-2.11	-18.23	-10.03	-8.87	-8.12

File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	-24.01	-19.87	-6.95	-16.43	-11.81	24.89	-14.11	-12.48	-31.74	-8.77	17.87	-36.91	-13.57
`-fbranch-probabilities`	-12.51	-8.07	-5.50	-0.95	-2.64	-2.55	-5.80	-7.77	-14.58	-5.56	-12.11	-10.22	-7.10
`-maccumulate-` `outgoing-args`	-1.79	-1.55	-2.33	-1.44	-0.87	-2.85	-3.31	-1.77	-4.10	-3.78	3.05	-1.85	-2.58
`-fgcse`	0.73	-1.16	-1.95	-0.37	-1.92	-1.27	-0.32	-0.59	-0.38	-0.68	-0.06	-3.33	-1.23
`-fomit-frame-pointer`	-1.38	1.02	-0.81	-0.91	-0.27	-1.20	-1.94	-1.43	-1.10	1.41	-0.06	-1.20	-0.72
`-fstrict-aliasing`	0.12	-1.14	-0.11	-0.73	0.00	0.36	0.36	-0.58	-0.56	-0.66	0.00	-5.14	-0.46
`-mred-zone`	0.00	-0.06	-0.06	0.00	0.00	0.00	-0.34	-0.04	-0.02	0.12	0.00	-0.05	-0.05
`-fschedule-insns2`	-0.07	-0.06	-0.07	-0.19	0.01	0.07	0.00	-0.01	-0.02	0.00	0.00	-0.04	-0.03
`-foptimize` `sibling-calls`	0.06	-0.04	0.10	0.00	0.00	-0.04	-0.45	-0.20	0.13	-0.01	-0.06	-0.05	-0.03
`-fstrength-reduce`	0.24	0.11	-0.01	0.18	0.01	0.03	-0.02	0.00	0.10	0.00	0.00	0.12	0.02
partial SSE moves	0.00	0.27	0.00	0.00	0.00	0.01	0.24	0.00	0.00	0.00	0.00	0.01	0.03
full sized loads and moves	0.18	0.09	0.17	0.00	0.00	0.40	0.01	0.00	0.13	0.00	0.00	0.07	0.10
`-mfpmath=sse`	0.00	-1.35	-0.05	-0.55	-0.14	-0.08	3.34	-0.58	0.00	0.00	0.00	-1.39	0.15
prologue using move	0.00	0.07	0.14	0.00	-0.05	0.40	-0.02	0.45	0.28	0.37	-0.06	0.06	0.20
`-funroll-loops`	1.73	0.98	0.34	3.97	1.51	3.22	0.28	0.04	1.00	0.00	0.00	0.77	0.52
`-freorder-blocks`	0.24	0.11	1.05	-0.55	0.00	-0.04	0.20	0.63	0.36	0.00	0.00	0.21	0.53
`-frename-registers`	1.35	1.18	1.26	0.00	1.47	0.71	2.27	0.67	0.62	0.66	0.00	2.19	1.16
`-ftracer`	0.67	1.36	1.57	2.61	2.02	2.29	0.44	1.30	1.61	2.01	0.00	0.58	1.43
`-fbranch-probabilities` `-fguess-branch...`	6.09	4.09	5.60	5.44	6.03	9.87	-0.21	3.90	3.58	4.49	7.78	3.27	4.40
`-funit-at-a-time`	-14.10	2.25	12.02	0.00	2.04	5.62	0.00	4.14	6.08	2.66	7.60	1.92	5.94
`-m64`	16.48	-2.64	8.02	18.47	-19.00	15.52	0.25	11.38	9.65	-5.69	8.64	-3.44	3.90
`-finline-functions`	8.71	7.94	23.54	2.80	3.51	39.11	-0.09	11.96	9.86	4.17	39.65	2.71	12.98
`-ffast-math`	0.00	-0.02	0.03	0.00	0.00	0.00	0.00	-0.05	0.00	-0.02	0.00	0.01	0.00
`-funroll-all-loops`	0.00	0.23	0.04	2.18	0.00	1.26	0.00	0.57	0.09	0.00	0.00	-2.94	0.03
`-fpic`	16.27	4.69	-6.01	0.18	17.87	-21.91	0.96	1.39	6.50	7.12	-21.77	14.97	0.38
`-fpeel-loops`	1.57	0.39	0.35	1.63	1.98	5.80	0.00	0.57	0.96	0.00	0.00	1.25	0.66
all prologue using move	2.18	2.85	1.30	1.45	0.26	2.63	2.31	1.71	2.95	2.77	-0.72	2.62	1.91
`-mcmodel=medium`	14.15	9.85	7.56	19.12	18.58	7.95	5.97	9.93	9.90	7.91	21.15	12.94	9.01

Table 10: 64-bit SPECfp 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	1.30	0.00	0.89	0.56	-5.34	-0.28	0.00	-0.13	-1.29	1.21	-0.16
aggressive optimization	101.11	53.87	686.79	225.30	101.38	26.80	100.81	123.51	225.00	180.97	149.23
`-m64`	5.00	-0.27	16.25	9.79	28.55	83.54	-1.31	19.17	28.33	20.86	19.34
`-mfpmath=sse`	13.97	0.12	2.40	2.33	7.04	100.28	1.79	16.64	22.22	5.67	13.80
`-fbranch-probabilities` `-fguess-branch...`	-0.83	0.39	10.83	3.96	19.62	2.23	-0.28	6.85	2.24	0.70	3.98
partial SSE moves	1.58	0.13	2.18	1.76	0.70	1.27	-2.51	3.17	6.14	2.54	1.74
`-fstrict-aliasing`	0.13	0.00	0.00	0.00	-0.90	4.49	1.37	5.49	0.00	4.71	1.73
full sized loads and moves	-2.25	0.26	3.31	1.16	4.29	2.40	2.92	0.86	2.25	0.89	1.57
`-fschedule-insns2`	0.13	0.12	13.06	0.57	-9.93	1.53	-0.68	5.49	3.71	1.58	1.41
`-ftracer`	0.27	0.00	-0.19	-0.19	-2.85	0.97	1.79	1.10	0.00	0.34	0.15

Table continues on next page...

Table 10 Continued—Performance (relative speedups in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
`-mred-zone`	-0.95	0.00	-0.19	1.15	-2.32	0.13	1.09	0.00	0.00	0.00	-0.16
prologue using move	-1.53	0.13	-0.18	-0.20	0.91	-0.84	-0.14	0.00	0.00	-0.18	-0.16
`-frename-registers`	0.00	0.00	4.52	-0.76	-12.07	1.83	3.21	1.84	1.39	-1.03	-0.31
`-fbranch-probabilities`	-1.61	0.00	-0.37	-0.57	7.36	-0.83	-0.14	-0.49	0.83	-4.16	-0.32
`-fomit-frame-pointer`	-1.08	0.00	0.54	0.95	-11.17	-0.69	0.68	0.85	0.00	1.94	-0.62
`-finline-functions`	0.00	0.12	-0.19	0.00	-12.12	2.97	1.23	0.36	-0.28	0.00	-0.77
`-maccumulate-outgoing-args`	3.20	-0.13	0.00	-0.19	-9.94	-0.70	0.40	-0.13	-0.28	0.00	-0.78
`-freorder-blocks`	1.08	0.00	-0.19	-0.19	-11.27	1.11	0.13	1.72	0.00	0.00	-0.78
`-funroll-loops`	-2.43	-0.13	0.00	1.34	-11.02	0.83	0.54	3.25	0.00	0.34	-0.78
`-foptimize sibling-calls`	-1.20	0.00	-0.37	0.00	-13.20	0.97	-0.28	-0.49	0.00	0.34	-1.23
`-fstrength-reduce`	-1.85	0.00	-0.37	5.20	-13.15	-0.14	0.95	-0.85	1.39	-2.04	-1.23
`-funit-at-a-time`	-0.96	0.12	-0.19	-0.19	-11.26	0.00	1.09	0.00	0.00	0.00	-1.24
`-fgcse`	-1.46	-0.39	-7.52	-4.36	-12.53	1.26	0.40	-0.13	-1.63	-3.19	-3.02
`-ffast-math`	-2.01	0.00	-0.19	1.13	14.99	-0.70	2.16	1.45	-0.83	2.94	1.86
`-fpeel-loops`	9.94	0.00	-0.19	0.18	0.00	-0.83	-1.22	0.00	0.00	-0.18	0.62
`-funroll-all-loops`	-0.41	0.12	0.00	-0.19	0.00	0.98	-1.49	-0.13	0.00	0.17	-0.16
`-fpic`	5.42	-0.13	0.00	-0.95	14.84	0.55	-1.76	0.00	-20.67	-0.18	-0.63
all prologue using move	-5.90	0.00	-0.89	-0.39	0.20	-0.28	0.54	-0.62	0.00	0.17	-0.78
`-cmodel=medium`	-0.54	-0.13	-0.55	-1.71	9.68	-3.19	-1.76	-3.88	-16.53	-1.22	-2.01

File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
aggressive optimization	-16.48	-15.91	-34.31	-57.92	-33.11	8.36	-29.40	-26.61	-36.44	-25.42	-34.22
`-fbranch-probabilities`	0.55	-8.26	-2.73	-3.79	-12.90	-10.98	-9.59	-7.97	-4.00	-7.95	-7.22
`-maccumulate-outgoing-args`	-1.93	-0.62	-1.78	-0.78	-3.49	-0.97	-0.99	-1.92	-0.80	-1.19	-1.67
`-mred-zone`	0.00	-0.21	-0.37	-2.03	-0.77	-0.13	-0.13	-0.03	-0.01	-0.30	-0.30
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	-0.75	6.80	-10.04	0.00	0.00	-0.18	-0.27
`-fgcse`	0.00	-8.64	-4.00	-10.19	-0.74	1.91	-0.38	0.00	1.70	-3.61	-0.07
`-fschedule-insns2`	0.00	0.00	0.00	0.00	-0.10	0.00	0.00	0.00	0.00	0.00	-0.03
prologue using move	-0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00
`-foptimize sibling-calls`	0.00	0.00	-0.37	0.00	-0.18	0.00	0.00	0.00	0.36	0.10	0.13
full sized loads and moves	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.00	0.34	0.08	0.17
`-freorder-blocks`	0.00	0.00	0.00	0.00	0.03	0.24	0.49	0.00	0.42	-0.09	0.21
`-funit-at-a-time`	0.00	0.00	0.00	0.00	0.11	0.12	4.67	1.85	0.00	0.00	0.23
`-fomit-frame-pointer`	8.70	0.82	0.91	-1.92	-0.51	-0.73	-0.38	0.51	0.40	5.13	0.57
`-fstrength-reduce`	0.00	0.00	0.18	-0.51	0.03	0.00	0.12	0.00	1.20	0.12	0.59
partial SSE moves	0.00	0.20	0.18	0.39	0.77	0.60	0.00	0.00	0.82	0.23	0.65
`-ftracer`	11.68	0.41	-1.26	0.00	0.03	0.36	0.87	5.54	0.00	0.92	0.70
`-funroll-loops`	10.37	14.33	2.03	2.81	0.03	6.59	3.06	2.39	0.35	2.96	1.09
`-fbranch-probabilities -fguess-branch…`	12.12	15.26	2.69	3.25	0.02	19.33	5.59	8.65	0.43	4.67	1.92
`-frename-registers`	8.99	0.82	0.54	2.99	2.38	1.85	1.76	2.69	2.57	1.58	2.53
`-finline-functions`	0.00	0.00	0.00	0.00	5.92	18.22	4.94	2.41	1.27	1.84	2.75
`-mfpmath=sse`	8.70	2.96	2.03	8.08	-0.75	6.59	3.99	5.54	5.28	5.13	3.72
`-m64`	45.40	201.01	156.05	26.51	17.41	39.81	27.06	23.41	28.79	38.22	28.68

Table continues on next page…

Table 10 Continued—File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
`-ffast-math`	0.00	-0.83	0.00	0.94	-0.85	-6.44	-4.84	-8.23	0.40	-0.18	-0.81
`-funroll-all-loops`	0.00	0.00	0.00	0.00	0.00	0.24	0.61	0.00	0.00	0.00	0.01
`-fpeel-loops`	0.00	0.00	0.00	1.39	0.00	0.36	1.36	0.00	0.00	0.12	0.07
all prologue using move	-0.49	8.82	1.79	1.28	2.22	8.41	0.99	2.15	0.36	4.23	1.49
`-fpic`	0.65	-6.38	2.35	1.11	5.32	-3.71	13.13	2.23	6.58	3.47	5.21
`-mcmodel=medium`	0.00	9.45	2.17	7.98	5.43	10.44	11.27	5.24	23.48	6.72	14.49

Table 11: 32-bit SPECint 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	1.06	-0.14	0.42	0.69	0.11	0.00	-0.13	0.20	-0.28	0.85	0.71	3.52	0.75
aggressive optimization	96.74	76.81	73.11	14.74	56.38	83.61	349.45	111.06	98.34	71.82	122.25	67.09	89.12
`-march=i386` to `k8`	5.23	8.41	3.45	0.17	9.02	6.80	82.00	-0.52	0.41	14.78	2.45	8.52	10.08
`-fbranch-probabilities -fguess-branch...`	8.34	2.37	12.33	1.40	4.25	7.49	17.57	14.35	8.99	12.75	6.47	0.87	7.37
`-fbranch-probabilities`	2.94	0.41	10.33	0.17	2.91	5.43	0.61	8.82	2.41	8.26	6.45	0.77	3.89
`-fomit-frame-pointer`	8.64	1.36	0.84	0.17	2.26	6.51	0.73	0.41	4.58	2.66	6.25	3.78	3.26
`-fgcse`	1.99	1.52	-2.27	-0.69	0.57	-4.36	5.14	8.00	2.67	2.93	1.86	2.98	1.77
`-finline-functions`	0.90	1.96	0.00	2.84	2.91	6.62	1.86	0.82	1.41	3.34	1.87	1.78	2.17
`-ftracer`	0.15	1.94	4.58	-0.52	-0.34	-2.23	3.94	9.70	0.13	1.74	3.05	0.77	1.78
`-fschedule-insns2`	2.30	2.22	2.47	-0.35	2.32	0.15	0.12	1.87	-0.69	2.04	1.73	2.70	1.52
`-funit-at-a-time`	-0.60	8.91	3.47	-0.18	2.55	-1.50	0.12	7.50	-1.10	1.83	0.28	-0.67	1.39
`-freorder-blocks`	1.99	0.68	7.88	-0.87	3.52	0.76	-0.37	1.24	-0.83	2.23	2.01	-1.00	1.26
`-funroll-loops`	-0.31	-0.55	0.00	0.34	0.22	-1.79	6.77	0.72	0.69	2.71	1.14	3.53	1.25
`-march=ppro` to `k8`	5.91	-1.89	2.37	0.34	0.45	-4.22	2.63	0.30	1.11	0.38	2.75	2.60	1.13
`-maccumulate-outgoing-args`	0.60	-0.28	0.53	0.00	2.67	-2.08	5.95	2.62	0.27	4.06	1.00	-2.15	0.88
`-frename-registers`	-0.30	1.65	-0.94	-1.04	0.68	-2.67	-1.57	0.00	-0.14	2.74	0.85	5.49	0.75
`-foptimize sibling-calls`	-0.16	0.27	2.24	0.34	-0.34	-1.93	-1.21	-0.11	0.69	1.93	0.56	0.11	0.25
`-fstrict-aliasing`	1.07	-1.37	0.21	1.39	-0.12	0.00	0.12	0.10	0.55	0.09	0.71	0.55	0.25
`-fstrength-reduce`	-0.16	0.54	-0.53	-1.04	0.57	-2.51	0.12	0.00	-1.10	-1.14	0.28	1.10	-0.25
`-funroll-all-loops`	3.10	-0.28	0.31	-0.87	0.11	2.73	0.49	2.98	0.68	1.14	-0.15	1.98	1.00
`-mfpmath=sse`	1.83	2.32	1.28	-1.38	0.11	0.45	0.36	0.51	1.39	0.94	0.85	0.32	0.75
`-ffast-math`	-0.31	1.09	0.63	0.34	-0.46	0.15	0.12	0.72	0.55	0.86	0.42	0.44	0.50
`-fpeel-loops`	2.29	0.00	-0.32	-0.52	0.90	3.17	0.00	0.10	-3.43	-0.29	0.70	-1.20	0.00
`-fpic`	-20.49	-5.64	-17.55	-3.28	-29.60	-28.19	-10.27	-29.75	-23.00	-35.03	-25.65	-17.66	-20.81

File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	-18.85	-6.25	3.51	-21.10	2.34	33.46	-4.21	-6.83	-22.83	-2.91	33.80	-22.33	-4.05
`-fbranch-probabilities`	-14.82	-8.93	-5.82	0.67	-1.96	-3.46	-5.89	-7.95	-14.56	-3.10	-11.81	-10.11	-6.87
`-fgcse`	1.21	-1.15	-1.23	0.00	2.31	-0.93	0.20	0.52	0.21	0.51	-1.59	-1.60	-0.28
`-foptimize sibling-calls`	0.07	0.11	0.09	0.00	0.07	0.00	-1.44	0.05	0.01	-0.03	-1.18	-0.02	-0.14
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
`-fstrength-reduce`	0.21	0.09	0.02	0.00	0.05	-0.29	0.03	0.09	0.12	0.00	0.00	-0.19	0.02
`-fschedule-insns2`	-0.15	0.21	-0.07	0.00	-0.07	0.00	1.63	-0.02	-0.04	-0.01	0.00	0.03	0.15
`-march=ppro` to `k8`	-2.15	1.33	-0.40	0.00	-0.36	0.00	5.56	-0.29	-0.49	0.10	-1.18	0.31	0.40
`-funroll-loops`	3.06	0.81	0.32	0.00	1.16	2.91	0.08	0.21	0.88	0.08	2.31	0.31	0.48
`-frename-registers`	0.49	0.48	0.52	0.00	0.51	0.00	1.42	0.81	0.22	0.10	1.02	0.31	0.55
`-freorder-blocks`	-0.08	-0.06	1.22	0.00	0.50	-0.03	0.17	0.82	0.29	0.10	0.53	0.22	0.62
`-fomit-frame-pointer`	-1.77	2.89	0.39	0.00	-0.14	0.77	4.52	-0.79	0.17	2.38	-2.80	-0.11	0.95
`-ftracer`	0.00	1.33	1.78	0.00	4.56	2.91	0.31	2.07	1.71	2.56	0.29	0.31	1.80
`-fbranch-probabilities -fguess-branch...`	6.98	3.72	6.73	0.67	9.29	9.37	-0.26	4.48	3.81	4.67	6.41	2.35	4.93
`-maccumulate-`	1.29	6.40	6.00	0.00	1.95	2.47	0.38	2.07	4.64	19.88	3.13	4.36	5.87

Table continues on next page...

Table 11 Continued—File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
outgoing-args													
`-funit-at-a-time`	-11.69	6.01	13.64	0.00	2.27	6.00	0.00	4.45	7.07	2.65	6.53	1.86	6.58
`-march=i386` to `k8`	1.43	9.46	9.78	0.00	3.65	6.00	8.00	4.13	6.70	21.21	4.02	8.63	9.24
`-finline-functions`	10.90	8.91	28.84	0.00	3.79	39.55	0.16	13.26	10.95	4.65	50.44	2.30	14.46
`-ffast-math`	0.00	-0.79	0.01	0.00	-0.02	0.00	0.00	-0.13	0.00	-1.23	0.00	-0.06	-0.21
`-funroll-all-loops`	0.00	0.25	0.05	0.00	0.07	2.83	0.00	0.03	0.07	0.03	1.19	0.21	0.15
`-fpeel-loops`	2.19	1.15	0.39	0.00	2.81	6.13	0.00	0.21	0.88	0.02	1.25	1.61	0.72
`-fpic`	12.59	6.19	-4.89	0.00	14.80	-27.60	10.58	4.43	1.15	1.35	-21.21	9.83	0.84
`-mfpmath=sse`	-0.08	1.15	-0.03	0.00	-0.06	0.00	10.10	0.17	0.00	0.00	1.19	-1.80	1.13

Table 12: 32-bit SPECfp 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	0.13	0.00	0.00	-0.21	0.28	2.57	-0.14	0.00	6.00	0.00	0.72
aggressive optimization	77.83	27.22	445.45	148.97	56.22	-30.46	92.25	101.18	122.37	156.08	98.56
`-march=i386` to `k8`	6.02	0.00	2.53	3.17	13.31	1.54	-0.65	1.49	-3.05	2.11	2.41
`-fbranch-probabilities`	3.49	0.39	4.74	4.28	0.72	1.81	-1.42	7.93	-2.16	0.20	1.66
`-fguess-branch`…											
`-fomit-frame-pointer`	-0.14	0.12	3.49	2.25	9.32	1.02	0.38	0.29	0.00	1.03	1.63
`-march=ppro` to `k8`	8.34	0.00	0.00	-0.82	10.41	-1.50	0.26	-0.59	-0.94	-0.62	1.10
`-fstrength-reduce`	10.13	-0.26	1.46	1.03	-8.02	-1.54	0.13	0.89	-0.32	3.64	0.91
`-funroll-loops`	3.93	0.00	0.00	0.61	-7.65	1.81	0.52	4.62	0.95	-0.21	0.36
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	-1.27	-0.13	0.14	0.00	0.00	0.00
`-frename-registers`	0.81	0.12	-0.62	0.00	-5.69	-0.52	1.98	-0.15	0.63	0.62	-0.19
`-funit-at-a-time`	0.13	0.00	0.00	0.00	-5.75	0.25	2.25	0.29	0.00	0.00	-0.19
`-ftracer`	1.65	0.00	0.00	0.00	-6.54	0.51	0.39	2.26	-0.32	-0.82	-0.37
`-finline-functions`	0.00	0.00	0.00	0.00	-7.14	3.70	1.85	-0.15	0.00	-0.21	-0.37
`-maccumulate-` `outgoing-args`	2.20	0.00	0.20	0.20	-6.37	-0.76	-0.40	0.00	0.00	0.41	-0.37
`-foptimize` `sibling-calls`	-0.27	0.00	0.00	0.00	-6.44	2.84	0.00	0.14	-0.32	0.00	-0.37
`-fschedule-insns2`	-0.54	0.13	1.04	2.72	-6.49	-0.26	-1.67	1.34	-6.48	1.04	-0.72
`-freorder-blocks`	0.68	-0.13	0.20	0.00	-4.78	-1.52	-0.13	1.04	-1.55	-0.62	-0.73
`-fbranch-probabilities`	1.78	0.00	-0.21	-2.80	0.00	-2.53	0.26	-1.17	-0.63	-2.23	-0.91
`-fgcse`	2.21	-0.39	0.20	-2.40	-3.99	2.02	-0.13	-0.59	-10.68	0.20	-1.43
`-mfpmath=sse`	2.43	0.25	3.29	-0.21	12.53	97.20	-0.14	1.47	13.20	3.30	10.14
`-ffast-math`	1.21	0.25	0.00	2.04	3.13	-0.26	3.89	0.58	-0.95	3.09	1.44
`-fpeel-loops`	3.78	0.00	0.00	2.25	0.00	0.51	-0.26	0.00	0.00	0.00	0.54
`-funroll-all-loops`	0.00	0.12	0.00	0.00	0.00	-2.54	-0.26	0.14	0.00	0.00	-0.19
`-fpic`	-5.15	0.25	-3.72	3.46	-0.43	-1.31	-10.15	-2.36	-11.64	-1.45	-3.10

File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
aggressive optimization	-3.88	-1.94	-20.88	-25.85	-23.54	14.89	-16.01	-17.99	-17.79	-11.69	-18.60
`-fbranch-probabilities`	0.24	-2.71	0.69	-4.31	-14.27	-7.93	-4.87	-11.72	-4.35	-7.09	-7.78
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
`-march=ppro` to `-march=k8`	0.00	0.53	0.00	-4.45	1.81	0.77	0.10	0.41	-0.60	0.00	0.04
`-funit-at-a-time`	0.00	0.00	0.00	0.00	0.24	0.00	3.26	0.43	0.00	0.00	0.14
`-freorder-blocks`	0.00	0.00	0.00	0.04	0.15	-0.12	0.43	0.31	0.28	0.00	0.20
`-foptimize`	0.00	0.00	0.00	0.00	0.26	0.00	0.00	0.02	0.23	1.56	0.30

Table continues on next page…

Table 12 Continued—File size (relative increase of the size of stripped binaries in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
`sibling-calls`											
`-frename-registers`	0.00	0.26	0.00	0.04	0.25	0.33	0.65	0.02	0.61	0.00	0.38
`-ftracer`	7.98	0.00	0.00	0.00	0.07	0.44	0.76	5.35	0.00	1.44	0.66
`-funroll-loops`	5.15	6.26	0.00	1.15	0.06	7.76	1.21	0.43	0.07	2.48	0.57
`-fgcse`	-1.84	3.89	0.00	-4.45	-0.79	0.11	0.54	0.09	2.85	-3.17	0.76
`-fbranch-probabilities`	10.49	6.75	0.69	1.60	-0.55	11.19	2.58	7.22	0.63	3.16	1.38
`-fguess-branch...`											
`-fschedule-insns2`	0.00	0.81	0.00	1.44	0.63	0.66	1.32	3.68	2.53	2.07	1.90
`-fomit-frame-pointer`	2.10	1.60	0.00	4.64	2.24	0.00	0.54	4.52	1.22	9.28	2.41
`-fstrength-reduce`	0.00	-1.33	0.00	31.50	-0.04	-2.69	-0.22	-0.54	4.37	3.07	3.14
`-finline-functions`	0.00	0.00	0.00	0.00	6.28	13.54	6.74	1.95	1.85	3.97	3.23
`-march=i386` to `-march=k8`	7.17	-4.61	0.00	1.44	6.05	0.55	0.87	-0.68	4.19	8.23	4.35
`-maccumulate-outgoing-args`	7.52	1.91	0.00	0.71	3.53	1.23	1.77	0.43	6.49	9.36	5.03
`-ffast-math`	0.00	-0.81	0.00	0.23	-1.37	-31.41	-31.50	-6.71	-0.07	-0.78	-1.89
`-funroll-all-loops`	0.00	0.00	0.00	0.00	0.00	0.11	0.65	0.00	0.00	0.00	0.01
`-fpeel-loops`	0.77	0.00	0.00	0.42	0.00	0.22	1.19	0.00	0.06	0.00	0.08
`-fpic`	4.90	-6.17	0.00	-25.58	9.63	-3.10	2.72	5.98	7.44	-0.10	5.74
`-mfpmath=sse`	4.04	7.23	0.00	10.72	2.53	7.29	8.28	15.12	8.83	6.81	7.33

Table 13: 64-bit SPECint 2000 with Aggressive Optimization (DEC Alpha EV56/600Mhz)

Performance (relative speedups in percent)													
options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	0.00	-0.66	0.71	0.00	1.63	0.00	0.60	0.00	8.02	5.84	-0.55	4.72	1.96
aggressive optimization	143.98	77.03	73.26	16.94	105.84	141.75	505.83	119.81	128.84	94.27	180.89	71.33	115.27
`-fschedule-insns2`	16.23	10.00	1.51	2.20	11.18	2.75	20.56	8.84	2.87	3.78	15.38	5.63	8.08
`-fschedule-insns`													
`-funit-at-a-time`	1.42	2.63	3.73	3.67	-2.83	28.33	18.57	16.66	0.00	16.42	3.52	5.51	7.63
`-finline-functions`	5.18	2.63	2.18	1.47	14.63	31.62	1.19	8.33	0.00	22.13	4.73	2.70	6.84
`-fbranch-probabilities`	1.45	7.58	6.06	2.22	15.47	27.50	5.03	14.00	2.08	-3.48	0.00	0.65	6.16
`-fguess-branch...`													
`-fbranch-probabilities`	9.30	2.66	6.81	5.97	-4.33	29.66	-1.18	9.93	4.22	17.51	2.31	-0.66	5.44
`-fschedule-insns2`	7.87	6.16	3.75	0.72	7.69	-0.89	7.84	7.38	1.42	3.14	7.89	5.63	5.03
`-fomit-frame-pointer`	0.00	0.00	2.94	0.00	5.34	2.01	5.76	7.69	3.52	3.18	5.26	1.33	2.63
`-freorder-blocks`	0.71	0.00	2.15	0.00	14.10	1.31	-6.94	5.62	3.52	4.48	2.22	2.64	2.63
`-fgcse`	5.42	0.00	1.41	0.72	-1.02	-0.65	14.86	1.19	2.09	2.54	2.82	-0.66	1.94
`-fif-conversion`	2.96	5.47	0.00	2.20	4.97	0.65	13.15	0.00	2.08	3.20	-0.56	1.31	2.61
`-fstrength-reduce`	-3.53	-1.28	1.44	2.18	-3.30	-0.65	22.30	-2.96	2.08	-1.87	2.27	4.08	1.97
`-funroll-loops`	-1.42	0.00	2.18	0.00	22.29	0.00	3.65	-0.60	-1.37	1.87	0.00	-3.88	1.30
`-fstrict-aliasing`	-2.88	4.08	-0.71	0.73	2.13	8.45	-16.97	4.34	4.25	3.16	4.59	0.65	0.65
`-frename-registers`	0.71	0.64	0.71	-0.72	5.40	0.66	5.73	2.40	0.68	-12.50	-1.11	3.44	0.65
`-foptimize`	-2.12	0.00	0.71	-1.42	-14.80	-0.65	2.42	-2.49	0.68	1.86	1.11	-0.65	-1.28
`sibling-calls`													
`-ftracer`	0.00	-4.55	0.00	-2.16	-12.07	-0.65	3.06	-2.95	0.00	1.25	1.11	-7.10	-2.59
`-ffast-math`	-1.44	-3.73	-2.12	2.18	7.65	0.00	-1.78	-0.59	1.37	8.60	-0.55	1.32	1.29
`-funroll-all-loops`	0.70	-0.65	-2.78	0.72	2.59	1.30	5.16	4.40	0.00	-3.04	0.55	-3.25	0.00
`-fpeel-loops`	0.00	3.28	-0.71	-1.43	4.44	1.30	-3.51	-2.36	0.00	0.61	0.00	-1.30	0.00
`-fold-unroll-loops`	0.00	0.64	0.00	0.72	-4.62	1.31	10.71	3.03	-1.37	-2.54	-6.63	0.00	0.00
`-fpic`	0.00	-2.64	0.00	0.73	-13.23	3.63	-4.10	-0.65	-1.40	-3.71	5.48	-2.65	-2.05

File size (relative increase of the size of stripped binaries in percent)													
options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	-38.22	-29.20	-9.28	-42.75	-28.90	5.66	-49.91	-12.38	-36.23	-17.64	-3.00	-39.40	-22.85
`-fbranch-probabilities`	-10.66	-1.50	-2.43	0.79	-0.71	2.11	-4.12	-6.17	0.00	-3.29	-9.80	-5.73	-3.09
`-fomit-frame-pointer`	-10.98	-3.61	-1.53	0.00	-1.19	-3.23	-7.01	-2.35	-2.88	-2.10	-1.09	-3.01	-2.64
`-fgcse`	-0.25	-1.53	-1.07	0.00	-0.87	-1.56	-1.29	-0.48	0.08	0.01	-10.13	0.00	-0.84

Table continues on next page...

Table 13 Continued—File size (relative increase of the size of stripped binaries in percent)

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
`-fstrict-aliasing`	0.03	-1.22	0.00	0.00	-0.07	-0.28	0.26	-0.20	-0.53	-0.26	0.00	-3.01	-0.28
`-freorder-blocks`	-0.04	0.01	0.31	0.00	-0.43	0.01	-1.35	-0.23	-0.27	0.00	0.00	0.00	-0.09
`-foptimize sibling-calls`	0.06	-0.01	0.23	0.00	0.00	0.01	-1.26	-0.04	0.10	0.00	0.00	0.00	-0.02
`-frename-registers`	0.06	-0.09	0.00	0.00	-0.10	0.02	0.08	-0.09	0.01	-0.03	0.00	0.00	-0.02
`-fif-conversion`	-0.10	-0.19	0.28	0.00	0.15	-0.21	-1.31	0.05	0.04	0.00	0.00	0.00	-0.01
`-fstrength-reduce`	0.06	0.33	0.00	0.00	0.23	-0.48	0.05	0.06	0.20	0.01	0.01	0.00	0.04
`-funroll-loops`	0.06	0.00	0.27	0.00	0.12	0.34	0.00	0.05	0.00	0.00	0.00	0.00	0.12
`-ftracer`	0.04	0.63	2.22	0.00	3.66	5.31	0.11	2.09	-0.11	3.25	0.00	3.19	1.99
`-funit-at-a-time`	-20.22	0.29	9.22	0.83	1.09	6.54	-4.12	4.22	0.00	-1.08	0.30	-2.99	3.12
`-fbranch-probabilities -fguess-branch...`	0.46	4.61	5.48	0.79	5.40	6.52	0.20	4.37	0.06	4.34	0.42	3.34	3.90
`-fschedule-insns2`	0.00	4.24	4.73	0.00	3.87	0.00	5.63	3.53	4.29	3.47	0.00	3.41	4.06
`-fschedule-insns2 -fschedule-insns`	0.00	4.42	5.01	0.00	3.87	0.00	7.14	4.76	5.25	4.69	0.00	3.41	4.76
`-finline-functions`	0.47	8.20	23.93	0.79	3.89	43.62	-4.17	14.35	0.00	2.22	52.11	-2.89	11.68
`-ffast-math`	-0.31	-0.09	-0.01	-0.40	-0.06	-0.12	-0.04	-0.01	-0.07	-0.04	-0.12	-0.03	-0.04
`-funroll-all-loops`	0.99	0.31	0.00	0.00	0.43	2.29	0.00	0.11	0.00	0.02	0.00	0.00	0.13
`-fpeel-loops`	12.32	0.57	0.03	0.00	2.11	6.22	0.00	0.18	0.00	0.04	0.22	0.00	0.49
`-fpic`	-1.53	1.09	0.12	0.39	1.78	5.18	2.52	1.25	1.35	0.21	1.28	0.80	0.92
`-fold-unroll-loops`	12.39	8.85	-1.48	0.00	5.54	5.61	2.90	2.75	13.59	0.00	11.26	9.30	2.83

Table 14: 64-bit SPECfp 2000 with Aggressive Optimization (DEC Alpha EV56/600Mhz)

Performance (relative speedups in percent)

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	avg
	0.00	-0.75	-0.21	0.00	0.93	0.00	0.83	-0.84	-1.76	0.00
`-fschedule-insns2 -fschedule-insns`	14.49	10.74	50.22	17.06	28.57	7.60	17.08	24.61	25.41	21.69
`-fschedule-insns2`	1.93	0.00	0.92	3.25	34.50	7.73	4.67	5.26	0.00	5.78
`-fstrength-reduce`	9.27	0.75	2.71	4.88	2.85	1.19	2.56	0.84	1.81	3.17
`-fbranch-probabilities -fguess-branch...`	3.12	0.00	1.44	1.33	14.21	7.10	3.41	-0.83	0.90	3.14
`-ftracer`	1.85	0.00	1.02	0.20	8.54	1.14	0.82	0.84	6.79	2.36
`-fbranch-probabilities`	1.85	-0.75	-1.83	0.40	5.85	1.65	8.03	1.69	-1.74	1.56
`-funit-at-a-time`	2.48	0.74	-0.21	0.40	7.25	-1.68	10.00	2.56	-3.45	1.56
`-fstrict-aliasing`	0.00	0.00	-0.21	0.00	2.35	-6.56	9.00	0.84	0.90	0.77
`-fomit-frame-pointer`	2.48	-0.75	-0.41	0.00	4.34	-0.58	6.19	0.00	-0.88	0.77
`-fgcse`	0.60	0.00	0.00	-2.18	3.33	6.50	6.14	0.84	-2.61	0.76
`-finline-functions`	1.85	-0.75	-0.31	0.20	7.42	-9.40	2.56	0.84	-2.59	0.00
`-freorder-blocks`	0.00	-0.75	0.00	0.10	4.32	-5.24	6.14	-1.64	0.00	0.00
`-frename-registers`	0.60	-1.49	0.40	0.40	5.85	-1.66	0.82	0.84	-1.79	0.00
`-foptimize sibling-calls`	-0.61	0.00	-1.42	0.10	2.35	-4.66	5.21	0.00	-1.76	0.00
`-fif-conversion`	0.00	0.00	0.20	0.20	0.94	1.10	4.31	-0.84	-0.90	0.00
`-funroll-loops`	0.60	-2.99	-1.01	0.10	1.87	-3.98	0.83	0.00	-0.88	-0.77
`-fold-unroll-loops`	6.66	-0.75	0.20	2.43	-36.75	3.48	-4.96	1.66	3.63	-3.08
`-ffast-math`	-0.60	0.00	0.10	0.30	-0.47	2.90	-5.47	-0.83	-2.59	-0.76
`-fpic`	0.63	0.00	-0.21	0.20	-2.04	2.95	0.00	0.00	0.87	0.00
`-funroll-all-loops`	0.00	-0.75	0.71	0.70	0.00	7.55	-0.82	-4.17	-1.79	0.00
`-fpeel-loops`	3.63	0.00	0.20	6.06	0.00	4.06	-4.14	0.00	0.00	0.76

File size (relative increase of the size of stripped binaries in percent)										
options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	total
`-fbranch-probabilities`	0.37	-0.11	0.20	0.15	-7.43	-6.42	-0.06	-0.92	-2.47	-4.77
`-funit-at-a-time`	0.37	-0.11	0.20	0.15	-7.37	-6.42	0.57	0.03	-2.47	-4.61
`-fomit-frame-pointer`	0.00	-0.53	-1.53	-0.35	-3.45	-7.19	-2.12	-4.38	-1.30	-2.96
`-fgcse`	0.00	-26.92	0.57	-8.87	-1.06	-7.19	0.25	-0.02	-0.74	-1.93
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	-0.31	-7.19	-2.17	-0.10	-0.15	-0.44
`-fif-conversion`	0.00	-0.21	-0.09	-0.08	-0.22	-0.73	0.05	0.31	-0.03	-0.11
`-foptimize sibling-calls`	0.00	0.00	0.00	0.00	-0.04	0.00	-0.06	-0.01	0.00	-0.02
`-freorder-blocks`	0.00	0.10	0.00	0.02	0.28	-0.19	0.11	-0.43	-0.20	0.07
`-funroll-loops`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.00
`-frename-registers`	0.00	0.10	0.16	-0.08	0.13	0.18	0.22	0.11	-0.32	0.05
`-finline-functions`	0.37	0.20	0.28	0.13	-0.74	19.29	8.46	1.21	-1.87	0.09
`-ftracer`	8.22	0.00	0.08	0.02	0.22	0.55	0.22	1.01	1.72	0.79
`-fbranch-probabilities -fguess-branch...`	9.36	0.84	1.36	-1.20	0.00	2.79	0.97	4.33	1.80	1.17
`-fstrength-reduce`	7.50	2.68	3.28	7.17	-0.17	0.00	0.22	0.37	7.24	1.70
`-fschedule-insns2`	3.87	2.47	3.73	6.90	5.47	3.11	4.91	5.97	6.53	5.64
`-fschedule-insns2 -fschedule-insns`	3.78	2.47	4.26	11.04	5.55	3.11	5.66	5.97	6.97	6.25
`-fpic`	-1.98	-0.42	0.24	-2.57	-0.09	2.76	1.10	-1.06	-0.08	-3.83
`-ffast-math`	-0.21	-2.00	-1.05	-0.97	0.27	0.30	-0.60	-0.76	-0.61	-0.17
`-fpeel-loops`	0.00	0.00	0.00	0.90	0.00	7.73	2.60	0.00	0.00	0.30
`-funroll-all-loops`	0.00	0.00	0.81	0.29	0.00	7.73	1.13	0.37	0.27	0.34
`-fold-unroll-loops`	2.71	36.40	15.56	5.15	4.52	23.73	6.75	15.75	8.91	7.79

Porting to 64-bit GNU/Linux Systems

Andreas Jaeger
SuSE Linux AG
aj@suse.de, http://www.suse.de/˜aj

Abstract

More and more 64-bit systems are showing up on the market—and developers are porting their applications to these systems. Most code runs directly without problems—but there is a number of sometimes quite subtile problems that developers have to be aware of for portable programming and porting.

This paper illustrates some problems on porting an application to 64-bit and also shows how use a 64-bit system as development platform for both 32-bit and 64-bit code. It will give hints especially to application and library developers on writing portable code using the GNU Compiler Collection.

1 Introduction

With the introduction of AMD's 64-bit architecture, AMD64, implemented in the AMD Opteron and Athlon64 CPUs, another 64-bit processor family enters the market and users are going to buy and deploy these systems. A new architecture offers new challenges for both system developers (compare [JH]) and application developers.

This paper will give hints especially to application and library developers to write portable code and make use of their 64-bit development machine. While the paper discusses general 64-bit and porting problems specific to other platforms, the AMD64 platform is used as primary example. Other architectures that the author has access to and is familiar with are discussed also. A brief characteristic of these 64-bit Linux platforms[1] is given in table 1.

Differences between platforms and therefore the need to port software can be attributed to at least one of:

Compiler Different compilers have different behavior. This can mostly be avoided with using the same version of the GNU compilers.

Application Binary Interfaces (ABI) An ABI specifies sizes of fundamental types, function calling sequence and the object format. In general the ABI is hidden from the developer by the compiler.

CPU The effect of different CPUs is mainly visible through the ABI. The differences visible to developers include little or big endian, whether the stack grows up or down, or whether the fundamental size is 32-bit or 64-bit.

C Library Different C libraries might not implement the same subset of functions or have architecture dependent versions. The GNU C Library tries to unify this but there are always architecture dependent differences.

[1]The only missing 64-bit platforms that I am aware of are MMIX and SuperH SH 5 but there is no Linux port for them.

Kernel All access to the Linux kernel is done through functions of the C Library. A newer kernel might have additional functionality that the C Library then can provide.

Application developers will mainly have portability problems due to different CPUs and different ABIs and the discussion here will concentrate on these.

The paper is structured as follows: Section 2 mentions why 64-bit programs are advantageous. The following section discusses execution of both 32-bit and 64-bit programs on one system and development on such a system. Section 4 shows how easy porting should be and then goes into all the subtleties and problems that nevertheless arise.

2 Advantages of 64-bit Programs

The main limitation of 32-bit programs that push developers to 64-bit programs is the limited address space. A 32-bit program can only address 4 GB of memory. Under a 32-bit x86 kernel the available address space is at most 2-3 GB (3.5 GB with a special kernel and static linking of an application) since the kernel also needs some of that memory. Nowadays applications need larger and larger address spaces and performance can be greatly improved with large caches which is a benefit especially for databases.

Besides larger address space most recent 64-bit processors introduce additional features over the previous processor generation for improved performance.

As an example the 64-bit AMD Opteron processor has some architectural improvements, like a memory controller integrated into the processor for faster memory access which eliminates high latency memory structure. Programs written in 64-bit mode for AMD Opteron take implicitly advantage of this but also of further new features:

- 8 additional general purpose and 8 additional floating point registers
- RIP addressing (instruction-pointer relative addressing mode) to speed up especially handling of shared libraries[JH].
- A modern Application Binary Interface [AMD64-PSABI].
- A large address space (currently 512 TB per process).

3 64-bit and 32-bit Programs on One System

The CPU architects of the 64-bit architectures AMD64, MIPS64, Sparc64, zSeries and PowerPC64 designed their CPUs in such a way that these 64-bit CPUs can execute 32-bit code natively without any performance penalty. The most sold 64-bit platform is the MIPS architecture but it—due to its usage nowadays mainly in embedded systems—mainly runs in 32-bit mode. Under Linux the 64-bit platforms PowerPC64 and Sparc64 in general only use a 64-bit kernel but have no significant 64-bit application base.

All these architectures nevertheless share the way that their 32-bit support is done. The support of two architectures is commonly called "biarch support" and there's also the general concept of "multi-arch support."

A 64-bit architecture that can execute 32-bit applications natively offers some extra challenges for developers:

- The kernel has to support execution of both 32-bit and 64-bit programs.

Architecture	`uname -m`	Size	Endian	Libpath	Miscellaneous
Alpha	`alpha`	LP64	little	lib	
AMD64	`x86_64`	LP64	little	lib64	executes x86 code natively
IPF	`ia64`	LP64	little	lib	executes x86 code via emulation
MIPS64	`mips64`	LP64	both	lib64	executes MIPS code natively
PowerPC64	`ppc64`	LP64	big	lib64	executes PowerPC code natively
Sparc64	`sparc64`	LP64	big	lib64	executes Sparc code natively
PA-RISC64	`parisc64`	LP64	big	—	only kernel support, no 64-bit user land, executes 32-bit PA-RISC code natively
zSeries (s390x)	`s390x`	LP64	big	lib64	executes s390 code natively

Table 1: 64-bit Linux Platforms

- The system has to be installed in such a way that 32-bit and 64-bit libraries of the same name can exist on one system.
- The tool chain should handle development of both 32-bit and 64-bit programs.

3.1 Kernel Implications

The kernel side is not part of this paper but the requirements for the kernel implementation should be stated:

- Starting of programs for every architecture supported by the ABI, e.g. for both 32-bit and 64-bit.
- System calls for every architecture in a way that is compatible to the corresponding 32-bit platform. For example a program that runs on x86 should run on AMD64 without any changes.

  One problem here is the `ioctl()` system call which allows to pass any kind of data to the kernel including complex data structures. Since the kernel needs to translate these data structures to the same structure for all supported architectures, some `ioctl()`s might only be supported for the primary architecture. This restriction only hits administration programs, like LVM tools.

3.2 Libraries: `lib` and `lib64`

If a system only supports execution of one architecture, all libraries will be installed in paths ending with `/lib` like `/usr/lib` and user-level binaries in paths ending with `/bin`, e.g. `/usr/bin`. But if there's more than one architecture to support, libraries will exist in flavors for each architecture but with the same name, e.g. there's a `libc.so.6` for 32-bit x86 and one for 64-bit code on an AMD64 system. The problem now is where to install these libraries.

Following the example set by the Sparc developers, all the other 64-bit biarch platforms install the 64-bit libraries into paths ending with `/lib64`, e.g. `/usr/X11R6/lib64`. The 64-bit dynamic linker is configured to search these library paths. For 32-bit libraries nothing has been changed.

This setup has the advantage that packages build for the 32-bit platform can be installed without any change at all. For them everything is the same as on the corresponding 32-bit platform, no paths are changed at all. For example the binary x86 RPM package of the Acrobat Reader can be installed directly on AMD64 systems and works without any change at all.

For 64-bit programs a little bit more work is needed since often configure scripts search directly the library paths for certain libraries but then find only the 32-bit library in e.g. `/usr/lib` or makefiles have paths hard-coded. Configure scripts created by GNU `autoconf` offer an option to specify the library install path directly and if you use RPM, you can use for example the following in your spec file:

```
configure --prefix=/usr --libdir=%{_libdir}
```

Also `ldconfig` handles both 32-bit and 64-bit libraries in its configuration (`/etc/ld.so.conf`) and cache files (`/etc/ld.so.cache`). `ldconfig` marks 64-bit libraries in the cache so that the dynamic linker can easily detect 32-bit and 64-bit libraries.

3.3 Development for Different ABIs

GCC can be build as a compiler that supports different ABIs on one platform. Depending on the architecture a number of different ABIs or instruction sets are supported, e.g. for ARM it is possible to generate both ARM and Thumb code. The GNU binutils also support these different ABIs.

The framework is especially useful for a biarch compiler and the 64-bit GNU/Linux platforms AMD64, MIPS, Sparc64 and zSeries (s390x) have support to generate code not only for the 64-bit ABI but also for the corresponding 32-bit (31-bit for zSeries) ABI. The PowerPC64 developers have not yet implemented this in GCC but I expect that they follow the same road.

Note that in the following text only the C compiler (`gcc`) is mentioned. The whole discussion and options are also valid for the other compilers in the GNU Compiler Collection: The C++ compiler (`g++`), the Ada compiler (`gnat`), the Fortran77 compiler (`g77`) and the Java compiler (`gcj`).

3.3.1 The AMD64, Sparc64 and zSeries Way

For AMD64, Sparc64 and zSeries the compiler generates by default 64-bit code. To generate 32-bit code for x86 (on AMD64) or for Sparc (on Sparc64), the compiler switch `-m32` has to be given to GCC. Compilation for 31-bit zSeries on a 64-bit zSeries needs the `-m31` option. The assembler and linker have similar switches that GCC passes to them. The compiler also knows about the default library paths, e.g. `/usr/lib` vs. `/usr/lib64` and invokes the linker with the right options. An example compile session is given in figure 1.

3.3.2 MIPS and its ABIs

MIPS does not only support support 32-bit and 64-bit programs, it also support two different ABIs for 32-bit programs. The three ABIs can be summarized as follows:

Name	Library Path	GCC Switch
o32 (old 32-bit)	`/lib`	`-mabi=o32`
n32 (new 32-bit)	`/lib32`	`-mabi=n32`
n64 (64-bit)	`/lib64`	`-mabi=64`

Note that the Linux Kernel so far supports only the o32 ABI completely, support for the other two is currently been worked on.

3.3.3 Toolchain

GCC knows how to invoke assembler and linker to generate 64-bit or 32-bit code. Therefore in general GCC should be just passed the right option for compilation and linking. In cases where developers really need to call the binary utilities[2] directly for 32-bit code, there's

[2]Calling these directly might also harm since GCC passes extra options to the binary utilities. For example

```
$ gcc hello.c -o hello64
$ gcc -m32 hello.c -o hello32
$ file ./hello32 ./hello64
./hello32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
./hello64: ELF 64-bit LSB executable, AMD x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$ ldd ./hello32 ./hello64
./hello32:
        libc.so.6 => /lib/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
./hello64:
        libc.so.6 => /lib64/libc.so.6 (0x0000002a9566b000)
        /lib64/ld-linux-x86-64.so.2 =>
                       /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)

$ gcc -L /usr/X11R6/lib -L /usr/X11R6/lib64 xhello.c -o xhello64 -lX11
/usr/lib64/gcc-lib/x86_64-suse-linux/3.3/../../../../x86_64-suse-linux/bin/ld:
skipping incompatible /usr/X11R6/lib/libX11.so when searching for -lX11
$ gcc -m32 -L /usr/X11R6/lib -L /usr/X11R6/lib64 xhello.c -o xhello32 -lX11
$ ldd ./xhello64 ./xhello32
./xhello64:
        libX11.so.6 => /usr/X11R6/lib64/libX11.so.6 (0x0000002a9566b000)
        libc.so.6 => /lib64/libc.so.6 (0x0000002a95852000)
        libdl.so.2 => /lib64/libdl.so.2 (0x0000002a95a94000)
        /lib64/ld-linux-x86-64.so.2 =>
                        /lib64/ld-linux-x86-64.so.2 (0x0000002a95556000)
./xhello32:
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x40029000)
        libc.so.6 => /lib/libc.so.6 (0x400f8000)
        libdl.so.2 => /lib/libdl.so.2 (0x4022e000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Figure 1: Example Compile Sessions on AMD64

a short list of these options for the GNU binutils in table 2. The user can inquiry most of these options directly with calling `gcc -v` to print out the commands issued by the compiler.

3.3.4 Caveat: Include Files for Multi-Arch Compilation

The support for different ABIs on one systems has one problem: What happens if different versions of a library are installed that have a different interface? For example, the 64-bit library could be an older version than the 32-bit library and the newer version has changed data types or signatures of functions. Since there is only one include directory for all ABIs (there is no `/usr/include64`!), the system administrator has to take care that installed header files are correct for all ABIs and libraries. In the worst case the include file has to include support for each ABI using preprocessor conditionals. As an example, the GNU C Library has quite a few kernel dependent interfaces that are different between architectures. The include files for e.g. AMD64 therefore have—where

the linker `ld` will not produce correct C++ binaries if not called with the right set of options which GCC does automatically.

Tool	Option for 32-bit code		
	AMD64	Sparc64	zSeries
ar	No option needed		
as	`--32`	`-32`	`-m31`
gcc,g++,…	`-m32`	`-m32`	`-m31`
ld	`-m elf_i386`	`-m elf32_sparc`	`-m elf_s390`
nm	No option needed		
strip	No option needed		

Table 2: Options for 32-bit Code Generation on 64-bit Architectures

necessary—constructs like the following (from <`bits/fcntl.h`>):

```
#include<bits/wordsize.h>
[...]
# if __WORDSIZE == 64
#  define O_LARGEFILE 0
# else
#  define O_LARGEFILE 0100000
# endif
```

3.3.5 Debugging

The GNU Debugger (gdb) is currently getting enhanced to be able to debug a number of different architectures and ABIs. So, in the future, we could have a GDB that debugs all binaries that can run on one architecture, e.g. both 32-bit x86 and 64-bit programs on AMD64 systems. Currently this is not possible and therefore a separate debugger has to be used for every ABI. For example, SuSE Linux on AMD64, has a `gdb` binary to debug AMD64 programs and a `gdb32` binary for x86 programs.

The system tracer `strace` has on some architectures, e.g. AMD64 and Sparc64 already the capability to trace both 32-bit and 64-bit programs. On other systems both a 32-bit and a 64-bit version needs to be put in place with different names.

3.3.6 Changing the Personality

The output of `uname -m` is used by e.g. `configure` to check for which architecture to build. This can cause problems if you build on a 64-bit system for the corresponding 32-bit architecture since then `configure` might decide that this is a cross-compilation instead of a native compilation. For such cases the output of `uname -m`, the so called personality, can be changed with a special system call. The personality is inherited by children from their parents. There exists a user space program to change the personality and it can be used e.g. on AMD64 as:

```
$ uname -m
x86_64
$ linux32 bash
$ uname -m
i686
```

to create a shell with changed personality for further development.

The name of the user space program is different on different architectures, the following list contains those names that we are aware of:

Architecture	Personality Tool
AMD64	`linux32`
PowerPC64	`powerpc32`
Sparc64	`sparc32`
zSeries	`s390`

3.4 Development

So, with the complete toolchain supporting different ABIs, it is now possible to develop both 64-bit and 32-bit programs on one machine. Instead of having two machines heating the room, a developer can use only a 64-bit box as development machine and still produce and test 32-bit code.

To develop 32-bit code on an AMD64 system, the developer has to add the `-m32` option to the compiler flags, no other changes are needed in general.

For the development of native 64-bit AMD64 code on the same machine, the only change might be to change the library path if another library path as `/usr/lib64` is used. It is even safe to give both the 32-bit and the 64-bit path, the linker will find the right library directly (but emit warnings) as shown in figure 1.

4 64-bit Porting: Hints and Pitfalls

Porting to a 64-bit system is not a problem for portable programs. Unfortunately most programs are not really portable and therefore need to be changed to run correctly on another platform.

The porting effort on GNU/Linux platforms is lower than e.g. between Unix and GNU/Linux since all GNU/Linux platforms use the GNU C Library. The C Library tries to use a common implementation and headers for all platforms which eases portability. Using the same C Library cross platforms means:

- Usage of the same functions: The set of functions is the same in general. Only a few functions are architecture specific and those are needed in general to access hardware which is platform specific.
- A different layout of structures: The C Library implements the different processor specific ABIs and therefore structures can have different length and members.

Therefore a program that is written portable, without reference to platform specific features, in general can be easily ported from on platform to the other, e.g. from 32-bit to 64-bit.

Each platform has its own special "features," meaning that some non-portable code works on all platforms except one. Keeping these problems in mind helps writing portable code and eases debugging of non-portable code.

Most of the problems arise in C and therefore this language is used everywhere in this paper. Some of these problems might not arise in C++ since C++ has some stricter rules.

The general problem is that sizes of fundamental types on different platforms, and especially between 32-bit and 64-bit platforms, are different and therefore not all types are interchangeable.

4.1 "Portable" x86/AMD64 Inline Assembler

There are some things that can not be done portably in general. One issue is inline assembler. For processors from the same family, like x86 and AMD64 processors, often assembler code can be shared. But this is not possible between different architectures.

A small example for inline assembler on x86 and AMD64 is the following function:

```
/* ffs -- find first set bit in a
```

```
   word, counted from least
   significant end.  */
int
__ffs (int x)
{
  int cnt, tmp;
  /* Count low bits in X; store in
     %1.*/
  asm ("bsfl %2,%0\n"
       "cmovel %1,%0\n"
   /* If number was zero, return
      -1.*/
   : "=&r" (cnt), "=r" (tmp)
   : "rm" (x), "1" (-1));
  return cnt + 1;
}
```

This would be compiled by GCC for x86 to:

```
mov    $0xffffffff,%eax
mov    %eax,%edx
bsf    0x4(%esp),%ecx
cmove  %edx,%ecx
mov    %ecx,%eax
inc    %eax
ret
```

The assembler for AMD64 looks like this:

```
mov    $0xffffffff,%eax
mov    %eax,%edx
bsf    %edi,%ecx
cmove  %edx,%ecx
mov    %ecx,%eax
inc    %eax
ret
```

This example worked fine since `int` is 32-bit on both x86 and AMD64 and the same instructions can be used. For datatypes `long` this scheme cannot be used since it's 32-bit on x86 and 64-bit on AMD64. The size of `long long` is 64-bit on both architectures but since AMD64 has 64-bit registers code can be written that is more efficient.

Using the inline assembler in that function made it possible for the developer to ignore the different passing conventions in this example. For x86 the parameter `x` is passed on the stack (`0x4(%esp)`) and for AMD64 in the lower 32 bits of register `RDI` (`%edi`).

4.2 Sizes and Alignment of Fundamental Datatypes and Structure Layout

On 64-bit platforms pointers and the type `long` have a size of 64 bits while the type `int` uses 32 bits. This scheme is known as the LP64 model and is used by all 64-bit UNIX ports. A 32-bit platform uses the so-called ILP32 model: `int`, `long` and pointers are 32 bits.

The differences in sizes (in bytes) between the 32-bit x86 and the 64-bit AMD64 are summarized in the following table:

Type	i386	AMD64
`long`	4	8
pointer	4	8
`long double`	12	16

Besides the different sizes of fundamental types, different ABIs specify also different alignments. A `double` variable, for example, is aligned on x86 to 4 bytes but aligned to 8 bytes on AMD64 despite having the same size of 8 bytes. Structures will therefore have a different layout on different platforms. Additionally some members of structures might be in a different order or the newer architecture has additional members that could not have been added to the older one.

It is therefore important not to hard code any sizes and offsets. Instead the C operator `sizeof` has to be used to inquire sizes of both fundamental and complex types. The macro `offsetof` is available to get the offsets of structure members from the beginning of the structure.

4.2.1 `int` vs. `long`

Since the sizes of `int` and `long` are the same on a 32-bit platforms, programmers have often been lazy and used `int` and `long` interchangeably. But this will not work anymore with 64-bit systems where `long` has a larger size than `int`.

A few examples:

- Due to its size a pointer does not fit into a variable of type `int`. It fits on Unix into a `long` variable but the `intptr_t` type from ISO C99 is the better choice.
- Untyped integral constants are of type (unsigned) `int`. This might lead to unexpected truncation, e.g. in the following snippet of non-portable code:

  ```
  long t = 1 << a;
  ```

  On both a 32-bit and a 64-bit system the maximal value for a can be 31, since the type of `1<<a` is `int`. To get a shift done in 64-bit (a `long` calculation), `1L` has to be used.
- The type of identifiers of an enumeration is implementation defined but all constants get the same type. GCC by default gives them type `int`, unless any of the enumeration constants needs a larger type.

4.3 Function Prototypes

If a function is called in C without function prototypes, the return value is `int`—and that's a 32-bit type on all 64-bit Linux platforms. For arguments the integer promotions are performed and arguments of type `float` are promoted to `double`.

Such a missing prototype can easily lead to a segmentation fault. For example if `malloc()` or `memcpy()` are used without a prototype, the resulting binary might break because of:

`malloc()` The return value is a 32-bit entity and therefore only half of the bits of the returned address might be stored in the variable that holds the return value making the pointer invalid.

`memcpy()` The first two arguments are pointers that take the source and target address. If, instead of the 64-bit pointers, only the lower 32 bits are passed to `memcpy()`, the function will access random memory (note this can only happen if the pointer has been assigned to a variable of `int` and that variable is used for passing).

4.4 Variable Argument Lists

The problem with variable argument lists is the same problem as with missing function prototypes: At the call side an argument is passed to a function but the function expects an argument of a different size.

If you pass in a 32-bit value, it is normally passed in 64-bit registers or on the stack as 64-bit value. The question now is what to do with the unused 32 bits? The 32-bit value can be zero-extended so that the unused bits are all zero, it can be sign-extended giving all zeros or all ones, and it can be left unspecified (as on AMD64). If the called function expects now a 64-bit value where it gets a 32-bit value, the function might not work as expected.

The important rules are:

- If you pass 32-bit values, like variables of type `int`, the called function has to take out 32-bit values.
- If the function expects 64-bit values, like `long` or pointers, the caller has to pass

64-bit values. Note that `0` is not the same as a `NULL` pointer since those have different sizes.

Another topic is usage of `va_lists`. You cannot copy variables of this type directly. This works on those platforms that use a pointer to implement `va_lists` but not on others. Use instead the function-like macro `va_copy`.

4.5 Function Pointers

Often programmers assume that all pointers have the same format but this is not guaranteed by the ISO C standard.

On IPF, PA-RISC and PowerPC64 a pointer to a function and a pointer to an object are represented differently. For example on IPF, a function pointer points to a descriptor containing the function address and the value of the GP (global pointer, used with shared libraries) register:

```
struct ia64_fdesc {
  uint64_t func;
  uint64_t gp;
  };
```

The GP register needs to be set with the right value before calling any function.

This means the following should not be done in a portable program:

Compare function pointers Since there can be more than one descriptor for any function, different function pointers for the same function will have different values.

Locate function The function pointer will not point directly to the function, so it cannot be used easily to find the actual code of the function.

Construct function pointer from data address This will fail since the GP register will not be setup correctly.

4.6 Using Bitwidth-Dependent Types Portably

Some applications depend on specific sizes for their datatypes. As has been mentioned before, this cannot be done portably in general. ISO C99 introduced a new header file `stdint.h` that defines datatypes having specified widths and a corresponding set of macros. The following types are also specified:

Exact-width integer types Signed integer types of the form `intN_t` (unsigned: `uintN_t`) with width N are defined in general with widths 8, 16, 32, or 64. A `int32_t` is therefore a signed 32-bit integer.

Minimum-width integer types The types `int_leastN_t` for signed and `uint_leastN_t` for unsigned integers with a width of at least N bits are defined. The widths 8, 16, 32 and 64 are required to be supported.

Fastest Minimum-width integer types The types `int_fastN_t` for signed and `uint_fastN_t` for unsigned integers with width at least N bits are defined as types that are usually the fastest of all integer types having at least this width. Width of 8, 16, 32 and 64 are required to be supported.

Integer types holding pointers The integer types `intptr_t` and `uintptr_t` can hold a pointer, a conversion between pointer and this integer type is always possible.

Greatest-width integer types The integer types `intmax_t` and `uintmax_t` hold any value of any signed/unsigned integer type.

Note that an ISO C99 implementation does not need to implement all of these types. The GNU C Library implements all of them for all platforms.

In addition to these types a number of macros are defined to give the limits of the types.

Inclusion of the header `inttypes.h` defines additional macros for format specifiers both for `printf` and `scanf` for these types, and some conversion functions like `strtoimax`.

An example of the usage of the types and the format specifier for printing is:

```
#include <inttypes.h>
#include <stdio.h>
int
main (void) {
  intmax_t u = INTMAX_MAX;
  printf("The largest signed integer"
         " is: %" PRIdMAX "\n", u);
  return 0;
}
```

4.7 Using `printf` and `scanf`

ISO C99 introduced a few new format specifiers to allow printing and scanning of certain types that might have architecture dependent size. These are `%p` for printing a pointer value and the `%Z` size modifier for arguments of type `size_t`. An example:

```
...
void *p;
printf("p has value %p and "
       "size %Zd\n", p, sizeof(p));
```

4.8 Unsigned and Signed Chars

The ISO C Standard does not define the signedness of the type `char`.[3] A definition like `char foo;` creates an unsigned variable on some platforms but a signed one on others. If you use variables of type `char` as small integers, you should specify whether you need a signed or an unsigned type. Also comparisons with `char` variables should take this into account, the following code snippet will not give the desired outcome if `char` is unsigned:

```
char c;
if (c < 0)
  puts("Non-ascii character");
```

During compilation GCC should generate the warning "`warning: comparison is always false due to limited range of data type`".

Platforms with an unsigned char type are both 32-bit and 64-bit versions of S390 and PowerPC. GCC has the options `-fsigned-char` and `-funsigned-char` to change the signedness of type `char`.

4.9 Evaluation of Floating-Point Arithmetic

A common confusion happens when suddenly algorithms using floating-point arithmetic give different results. The IEEE754 standard defines that the basic operations have to be exact. But nevertheless, results might vary between architectures.

The problem happens with operations of type `float` and `double` since on the popular x86 architecture these operations are evaluated in the x87 FPU in `long double` precision. The compiler might choose to leave intermediate results (with a type of `long double`) in the x87 FPU or convert them back to the target type. Depending when this conversion happens, different rounding errors occur.

A small example to show the differences is:

[3]Note that this is not a 64-bit problem but it is one of those differences you'll notice when porting and is therefore worth mentioning.

```
#include <stdio.h>
int
main (void)
{
  float b, c;

  b = 1 / 3.0f;
  c = b * 3.0f - 1.0f;
  printf ("c: %.20f\n", c);
  return 0;
}
```

Compiling and executing this program on an Linux/AMD64 system gives different results between 32-bit x86 and 64-bit binaries:

```
$ gcc t.c -m32 -o t32
$ gcc t.c -o t64
$ ./t32
c: 0.00000002980232238770
$ ./t64
c: 0.00000000000000000000
```

Note that the example gives the same results if compiled with optimization since without optimization `b` is stored in memory as type `float` but with optimization `b` is left in the FPU.

ISO C99 defines the macro `FLT_EVAL_METHOD` for this in the header <`float.h`>. It is set to:

0 If evaluation is done with the range and precision of the type. This is the value on nearly all Linux systems.

1 If evaluation of expressions of type `float` and `double` is done to the range and precision of `double` and of `long double` to the range and precision of `long double`.

2 If all evaluations is done to the range and precision of type `long double`. This is the value on Linux/x86.

-1 Indeterminable.

This problem with different results due to the evaluation of floating-point arithmetic is not a genuine 64-bit problem but a problem between x86 code and all other platforms and therefore might hit developers porting from x86 to other platforms, e.g. to AMD64.

4.10 Shared Libraries

Most architectures have the constraint that shared libraries need to be compiled as PIC-code using the `-fPIC` switch to GCC. Even for those architectures that allow it, like x86, it is not desirable to do so since a shared library should live once in the memory and get then shared by all applications using it. But non-PIC code cannot be shared.

Architectures that force to use `-fPIC` for shared libraries include AMD64, IPF, and PA-RISC.

4.11 How to Check for 64-bit?

Starting with GCC 3.4, **all** LP64 platforms will define the macros `__LP64__` and `_LP64` that can be used e.g. in preprocessor defines. Earlier GCC releases define this macro only on a few platforms or OSes. For GCC 3.2 and 3.3, the macros are defined on NetBSD, for IPF (every OS), for PA-RISC (every OS) and for AMD64 running Linux (starting with GCC 3.2.3).

In general it is possible to check for 64-bit with the architecture builtins of GCC, e.g. with:

```
#if defined(__alpha__)\
 ||defined(__ia64__)\
 ||defined(__ppc64__)\
 ||defined(__s390x__)\
 ||defined(__x86_64__)
```

but this needs to be enhanced for each new 64-bit platform. The better solution is to write portable code that does not need to check for architecture details.

4.12 Optimized Functions, Macros, and Builtins

The GNU Compiler Collection uses the same optimizations on all platforms but some of them are tuned in different ways and others need help from the architecture specific back-end. One area were this occurs especially are builtin functions.

A function like `strlen` can be implemented in the following ways:

As builtin in GCC The compiler can detect that e.g. the arguments to `strlen` are constant and evaluate the function at compile time. It can also optimize the function to an inline function and do a loop instead of calling the external `strlen` function. This can be disabled with `-fno-builtin` or `-fno-builtin-`*`function`*.

As macro in Glibc The C Library implements a number of functions as macros. The string inline functions can be disabled with a definition of `__NO_STRING_INLINES`, some of them are only enabled if `__USE_STRING_INLINES` is passed. For details check the header `/usr/include/string.h` directly. Inlining of mathematical functions can be disabled by defining `__NO_MATH_INLINES`. Also it is allowed to disable a specific macro like #`undef strlen`.

As function in Glibc ISO C99 forces to implement all required functions as functions. Therefore for example `strlen` will always be in the C Library.

Some developers decide to override the C Library functions and write their own optimized implementation. This works fine for one system consisting of a specific CPU, a specific C Library and GCC version. But going to another architecture, better optimizations might be possible, e.g. reading 8 bytes at once instead of 4 in `strlen`, or current code is penalized, e.g. alignment is mandatory for 8 byte access.

So, instead of writing something just for one program, it should be done in a generic way in GCC or glibc so that all programs can benefit from one optimization.

A number of functions in Glibc are written in hand-optimized assembler for some architectures and where this is not done, a good C implementation is used. On AMD64 the compiler has builtins for the common string functions and also for some mathematical functions and uses them depending on the arguments and enabled compiler optimizations, e.g. `-Os` disabled most builtins since they would increase size.

The pitfalls regarding porting here are that a programs does optimizations that are not valid for a new architecture or does not expect that a function might be implemented as a macro or builtin.

4.13 Useful Compiler Flags

An incomplete list of GCC compiler flags that might be useful for porting code:

-Wall Enables a number of default warnings, should be used for all code

-W Enables additional warnings. Some of them are hard to avoid so this might not be useful for all code.

-Wmissing-prototypes Warn about missing prototypes, this is especially important for

64-bit ports.

5 Conclusion

Despite the different problems we encountered at SuSE while porting to the various 64-bit platforms (first for Alpha, later for IPF, zSeries, AMD64 and PowerPC64), the number of packages with actual problems is getting smaller and smaller since code has less platform specific assumptions and is more portable.

Also development of the toolchain has been improved recently and there is more focus on creating bi-arch toolchains to allow compilation for different ABIs on one system.

I hope that the problems mentioned and explained will help further in writing portable and efficient code.

6 Acknowledgments

Porting to any new architecture means building on the foundations that others have led, learning from their experiences and tackling with others all the subtleties of non-portable programming. I'd like to thank especially my colleagues Andreas Schwab for help in lots of debugging sessions and bug fixing of tools and programs, Stefan Fent and Stefan Reinauer for driving the port of SuSE Linux to AMD64 and thereby encountering many of the problems mentioned in this paper, Jan Hubička and Michael Matz for porting and fixing GCC and the ABI on AMD64—and all of them for their discussions on these issues. Thanks also to Michael Matz and Evandro Menezes for reviewing the paper.

References

[AMD64] *AMD64 Architecture Programmer's Manual*, AMD (2003).

[Opteron] *Software Optimization Guide for the AMD Opteron™ Processor*, AMD (2003).

[AMD64-PSABI] *UNIX System V Application Binary Interface; AMD64 Architecture Processor Supplement, Draft*, (Ed. J. Hubička, A. Jaeger, M. Mitchell), `http://www.x86-64.org`, (2003)

[i386-ABI] *UNIX System V Application Binary Interface; IA-32 Architecture Processor Supplement*, Intel (2000).

[ISOC99] *Programming Languages—C*, ISO/IEC 9899:1999 (1999)

[IEEE754] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE754-1985 (1985).

[JH] *Porting GCC to the AMD64 Architecture*, Jan Hubička, GCC Summit (2003).

Architecture for a Next-Generation GCC

Chris Lattner *Vikram Adve*
University of Illinois at Urbana, Champaign
`{lattner, vadve}@cs.uiuc.edu`
`http://llvm.cs.uiuc.edu`

Abstract

This paper presents a design and implementation of a whole-program interprocedural optimizer built in the GCC framework. Through the introduction of a new language-independent intermediate representation, we extend the current GCC architecture to include a powerful mid-level optimizer and add link-time interprocedural analysis and optimization capabilities. This intermediate representation is an SSA-based, low-level, strongly-typed, representation which is designed to support both efficient global optimizations and high-level analyses. Because most of the program is available at link-time, aggressive "whole-program" optimizations and analyses are possible, improving the time and space requirements of compiled programs. The final proposed organization of GCC retains the important features which make it successful today, requires almost no modification to either the front- or back-ends of GCC, and is completely compatible with user makefiles.

1 Introduction

The GNU Compiler Collection (GCC) [15] is in many ways the centerpiece of the Free Software movement. It supports several source languages and a plethora of back-ends for various targets, providing a unified target for free software. GCC has been successful because of its extreme portability, stability, and because it is able to compile and optimize several popular source languages (C, C^{++}, Java, etc) to each target. Unfortunately, despite the success of the GCC compiler suite as a whole, the optimization infrastructure is still not competitive with commercial compilers.

Over the years, the GCC optimizer has evolved from compiling a statement at a time, to compiling and optimizing entire functions at a time, to the (still very new) support for unit-at-a-time compilation (compiling and optimizing all of the functions in a translation unit together). As the scope for analysis and optimizations increases, the compiler is better able to reduce the time and space requirements for the generated code.

This paper proposes the next logical step for the GCC optimizer: extend it to be able to analyze and optimize *whole programs* at link-time[1], enabling new optimizations and making existing analyses and optimizations more powerful. For example:

- inlining across translation units
- whole-program alias analysis
- interprocedural register allocation
- interprocedural constant propagation
- data layout optimizations
- exception handling space optimizations

[1]This capability would be optional and could be enabled only when the program is compiled at the "`-O4`" level of optimization, for example.

- sorting initializer priorities at link-time

The key challenges to whole-program optimization are to enable powerful transformations while keeping compile times reasonable, and to keep the user-visible development process unchanged (e.g. user makefiles).

The architecture that we propose is based on a new language-independent low-level code representation that preserves important type information from the source code. The use of a low-level, SSA-based representation allows the compiler to perform a variety of optimizations at compile time, off-loading work from the link-time optimizer. However, the link-time optimizer can only perform meaningful optimizations on the program if it has enough high-level information about the program to prove that aggressive optimizations are safe. Because of this, the low-level code representation is typed (using a language-independent constructive type system) and directly exposes information about structure and array accesses to the optimizer.

The link-time optimizer is designed to combine the translation units of a program together and do the final whole-program optimization. After the program is optimized, machine code is generated at link-time for the entire program at once, allowing a variety of interprocedural low-level code optimizations to be performed.

The Low-Level Virtual Machine (LLVM) [10] is an implementation of the architecture and intermediate representation [11] described in this paper, which allows us to be more concrete when describing aspects of the design. This system has served as the host for several research projects [7, 13, 12] which require whole-program information as well as a host for a variety of traditional compiler optimizations.

We hope that the lessons learned by the LLVM project will be useful to the GCC community, and are willing to contribute as much code to the GNU project as there is interest in. We are planning to have our first public release of LLVM, with a liberal license, in the Summer of 2003. However, LLVM will only be discussed when it helps clarify the ideas in the proposed architecture, this paper is intended to be a GCC paper, not an LLVM paper.

This paper is organized as follows: Section 2 describes the proposed high-level architecture in detail, including modifications that would need to be made to the GCC infrastructure. Section 3 describes important aspects of the proposed intermediate representation for the system. Section 4 describes LLVM, our existing implementation of the proposed design. Section 5 describes other work related to the proposed design, and Section 6 wraps up the paper.

2 High-Level Compiler Architecture

The proposed high-level architecture is illustrated in Figure 1. The essential aspect of this design is that it separates the current `cc1` program into two components: a front-end compiler and an optimizing linker. The front-end retains all of the responsibilities of current GCC front-ends (preprocessing, lexical analysis, parsing, semantic analysis, etc..) and should work unmodified in the new system. After each function is parsed and checked for semantic errors it is "expanded" from the "tree" representation to the new language-independent intermediate representation (described in Section 3). Once the entire translation unit has been translated (and if no errors have occurred), a standard set of mid-level optimizations are performed on the translated module. After these optimizations are finished, a "`.o`" file is emitted which contains IR assem-

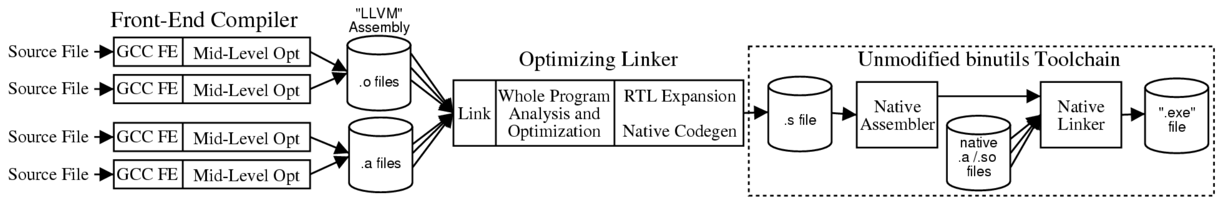

Figure 1: High-Level Compiler Architecture for Whole-Program Optimization

bly code for the representation.

When the optimizing linker is invoked, it reads in all of the translated IR files and any libraries compiled to the intermediate representation. It links these files together into a single-file representation of the program, on which it can run whole-program analyses and optimizations. Finally, once these analyses and transformations are complete, the GCC back-end is invoked to expand the intermediate representation into RTL and use the configured target description to produce a native `.s` file.

After the optimizing linker produces a native `.s` file, the compilation process proceeds through the standard system assembler and linker (to resolve any symbols in libraries that were not available in the IR form), finally producing a native executable.

2.1 Compatibility and Implementation

One of the key features of this design is that it is compatible with the standard "compile and link" models of compilation, and is thus fully compatible with existing makefiles. In order to provide this compatibility, the link phase of the `gcc` compiler driver is extended to invoke the optimizing linker and system assembler (if necessary) during the standard link step of the compile process. In this way, any input files that are in the IR format are automatically linked together and optimized without interfering with the compilation and linking of standard translation units and libraries. If no files in the IR format are present, the entire invocation of the optimizing linker is skipped.

Another important aspect of the design is how the compiler works when whole-program optimization is not enabled. If not enabled, each translation unit is either compiled a function at a time or a unit at a time (depending on the setting of the `-funit-at-a-time` switch), through the mid-level optimizer, RTL expansion, and code generation phases of the compiler. This produces a native `.s` file, which can be processed with the standard system assembler and linker, as before.

For this approach to be feasible, a large amount of code must be shared between the optimizing linker and the compiler front-ends. This can either be accomplished through the use of libraries that are shared between the two (which would contain the existing GCC back-end, and any shared optimizations on the IR), or by making both logical pieces be part of the same binary. In either case, the actual organization of the existing GCC code base would not have to change in any substantial way.

2.2 Architectural Issues Affecting Performance

In addition to providing the desired functionality and compatibility with existing systems, it is crucial that the compiler does not slow down unacceptably — even if whole-program optimization is only enabled at `-O4`. In practical terms, this design addresses the issue by performing as much optimization as possible at compile time.

Any time a source file is changed, it must be recompiled and the application must be relinked. In order to reduce the amount of work that must be done, this design allows most traditional optimizations to be performed in the compiler front-end stage, rather than requiring all optimization to occur at the link stage (as is common for whole-program optimizers). Because most aggressive scalar optimizations are performed at compile-time, they would not need to be rerun at link time, reducing the time for compilation. Of course, the compiler performance issue does not even arise unless the user is modifying the program and recompiling at `-O4`.

Optionally with this design, the compiler could try to minimize the amount of recompilation necessary when a change occurs by keeping track of which interprocedural information is used to modify functions in other translation units, building a dependence graph between the modules [4]. In practice, however, this would make the compiler much more complicated and prone to subtle bugs that are hard to reproduce. We feel that although the cost of recompilation is still fairly substantial in our system (native code must be regenerated for the entire application), that the extra complexity introduced into the compiler must be weighed against the recompilation time penalties, and thus may be impractical.

3 Code Representation

The representation used to analyze and manipulate the program determines what kinds of transformations are possible and when in the compilation process they must be performed to be successful. As mentioned earlier, we propose using a language-independent, low-level, SSA-based, strongly-typed representation as the sole representation used for the mid-level and link-time optimizers. This representation is a first-class assembly language, which includes all of the information necessary to represent the program (and is in fact directly interpretable). Concrete details of the representation used by LLVM are included in Section 3.2.

Using a low-level three-address code representation based on Static Single Assignment [6] form enables the direct application of many well-known and efficient global optimizations. SSA form permits *sparse* optimizations that do not, in general, require bit-vector data-flow analysis to compute results. Using a three-address code representation (as opposed to an tree structured representation) also makes transformations easy to develop and reason about.

Many transformations need information about the high-level behavior of the program to be effective. In order to preserve this information, we propose that the representation maintain a strong (but language-neutral) type system, which captures information about pointer, structure, and array accesses in the program. Working with the LLVM system we find that this type information allows for a variety of high-level analyses and transformations [7, 13, 12] while the nature of the low-level representation makes it very easy to manipulate. Another advantage of type information is that it makes detecting and understanding bugs in transformations much easier.

The goal of the program representation is to enable as many different types of optimizations as possible. Because of this, it is important that the representation be able to represent *all parts* of a program (including global variables, and file scope `asm` statements, for example) in a form that allows transformations to modify it. Another useful feature of the representation is a stable textual format ("assembly language") that can be read and written by the compiler. Given this, it is trivial to write unit tests for transformations and to debug transformations in isolation from the rest of the compiler, and the representation can be directly interpreted for immediate feedback on a transformation.

3.1 Performance Aspects of the Representation

Once the optimizing linker brings together the compiled program into one module, the interprocedural analysis and optimization passes are used to improve the program. Because these passes operate on the entire program at once, however, the efficiency of each analysis or optimization is critical. For this reason, several aspects of the representation are designed to make these transformations as efficient as possible.

In particular, the use of an SSA-based representation allows for efficient, sparse, global optimizations, and can make flow-sensitivity much less important in many analyses (reducing cost substantially). In addition, the three-address code representation has a small memory footprint and simple memory ownership semantics (eliminating the need for it to live on a garbage collected heap). In our experience with LLVM, code optimizers for a sparse representation can be several times faster than optimizations on a dense representation like RTL.

3.2 A Concrete LLVM Example

Figure 2 gives an example of a C function and the corresponding LLVM module it compiles to. The example shows several important aspects of the LLVM representation. In particular, it gives a simple example of the type system, basic instruction flavor, and demonstrates some instructions. More details about the LLVM representation can be found in the LLVM language reference [11].

LLVM uses a simple constructive type system composed of primitive types, structures, arrays, and pointers. Although this is a very simple type system, we believe that it contains the key features necessary for a front-end to lower any high-level type onto it. For example, the LLVM C^{++} front-end lowers classes with inheritance into nested structure types. Types are very important in the LLVM system, and everything that can be used as an operand to an instruction has a type.

Functions in LLVM contain a list of basic blocks, and each basic block contains a list of instructions. LLVM has only 29 instructions, which include standard instructions like `load`, `xor`, `set`*cc*, etc and a `phi` instruction for representing SSA form[2]. Intraprocedural control flow in LLVM is very simple (consisting of conditional branches, unconditional branches, and the `switch` instruction).

Everything in LLVM is explicit: there are no fall-through branches, all address arithmetic is exposed (at the level of structures, pointers, and arrays), and all references to memory use the `load` and `store` instructions. This makes the language more uniform and simple to analyze and transform.

The `getelementptr` instruction in LLVM provides the mechanism for structured address arithmetic[3]. The `getelementptr` instruction is exactly analogous to sequences of array subscript and structure index expressions, returning the address of the last element indexed[4]. For example, the `%tmp.1` instruction in Figure 2(b) first indexes into the 0^{th} element

[2]SSA ϕ-nodes are eliminated during the register allocation phase of native code generation.

[3]LLVM code can also cast a pointer to an integer type, add an arbitrary offset to it, then cast it back to a pointer, if unstructured address arithmetic is necessary.

[4]The example in Figure 2(a) uses the strange syntax '`T[0].x`' instead of using the equivalent '`T->x`' to make the correspondence more clear.

```
typedef struct QuadTree {
  double Data;
  struct QuadTree
     *Children[4];
} QT;

void Sum3rdChildren(QT *T,
           double *Result) {
  double Ret;
  if (T == 0) { Ret = 0;
  } else {
    QT *Child3 =
      T[0].Children[3];
    double V;
    Sum3rdChildren(Child3,
                   &V);
    Ret = V + T[0].Data;
  }
  *Result = Ret;
}
```

(a) Example function

```
%struct.QuadTree = type { double, [4 x %QT*] }
%QT = type %struct.QuadTree

void %Sum3rdChildren(%QT* %T, double* %Result) {
entry: %V = alloca double            ;; %V is type 'double*'
       %tmp.0 = seteq %QT* %T, null  ;; type 'bool'
       br bool %tmp.0, label %endif, label %else

else:  ;;tmp.1 = &T[0].Children[3]  'Children' = Field #1
       %tmp.1 = getelementptr %QT* %T, long 0, ubyte 1, long 3
       %Child3 = load %QT** %tmp.1
       call void %Sum3rdChildren(%QT* %Child3, double* %V)
       %tmp.2 = load double* %V
       %tmp.3 = getelementptr %QT* %T, long 0, ubyte 0
       %tmp.4 = load double* %tmp.3
       %tmp.5 = add double %tmp.2, %tmp.4
       br label %endif

endif: %Ret = phi double [ %tmp.5, %else ], [ 0.0, %entry ]
       store double %Ret, double* %Result
       ret void  ;; Return with no value
}
```

(b) Corresponding LLVM code

Figure 2: C and LLVM code for a function

from the pointer, then into the 1^{st} structure element (the "Children" member), then into the 3^{rd} element of the array. Structured address arithmetic exposes the necessary high-level information about structure and array accesses directly to analyses and transformations which need it.

One important aspect of the LLVM language is that all references to memory happen with `load` and `store` instructions, and that there is no "address-of" operation. In LLVM, all objects which live in memory (global variables, functions, the heap, and the stack) are explicitly allocated and exposed by their address, not their value. In Figure 2, for example, the `V` variable is required to live in memory so that its address may be passed into a recursive invocation of `Sum3rdChildren`. Because it is impossible to take the address of a virtual register, stack memory must be explicitly allocated with the `alloca` instruction[5], and any references to `V` must use `load` and `store` instructions. This dramatically simplified def-use chain construction for virtual registers, which would otherwise require some form of alias-analysis to construct.

A final example illustrating how LLVM simplifies the development of transformations is the operators that it lacks. In particular, LLVM does not have (or need) any unary operators or a copy instruction. Instead of providing the standard negate and bitwise complement unary operators, LLVM represents these with stan-

[5]When the back-end is invoked, all fixed sized `alloca`s in the entry block are treated the same as address-exposed automatic variables.

dard binary operators where one operand is a constant ("`neg x`" = "`sub 0, x`" and "`not x`" = "`xor x, -1`"). This reduces the dependence on a "canonical form" for the representation and simply reduces the number of instructions that need to be handled.

The lack of a copy instruction is possible through the use of SSA form, and because def-use chains are trivially computed and always available. Any time a copy instruction would be inserted (to replace a redundant computation for example) it is sufficient to replace any uses of the destination with uses of the source operand (by following the def-use chains), implicitly performing copy propagation automatically. This simple feature has actually avoided several phase-ordering issues that would otherwise require unnecessary passes over the representation to do copy propagation between other passes.

4 LLVM Compiler Infrastructure

The LLVM Compiler Infrastructure [10] currently consists of approximately 130,000 lines of C^{++} code and a the front-end, which is a patch against the mainline GCC CVS tree. This code largely implements the design presented in this paper, although there are some differences. This section describes these differences, the implementation status of LLVM, some other features of LLVM that make writing transformations simpler, and some insights that we have had while working on LLVM.

4.1 Implementation Status

The LLVM C front-end is based on the mainline GCC CVS repository. It generates code by calling LLVM versions of functions that are equivalent to the RTL-expansion routines (e.g. `llvm_expand_expr`, `llvm_expand_function_start`, `make_decl_llvm`, etc.) during compilation. These routines build up an LLVM version of the translation unit, which is then written to the "`.s`" file all at once (allowing "unit-at-a-time" style transformations to be performed from within GCC in the future).

Instead of modifying the `cc1` binary to interface directly to the LLVM optimizations written in C^{++}, `cc1` directly emits the expanded code without any optimization at all. When the `gcc` compiler driver invokes the "assembler", we actually have it invoke a program called `gccas` which parses the LLVM assembly file, runs a series of LLVM optimizers on it, then emits a compressed bytecode file (the `.o` file). The interface to `gccas` is intentionally designed to be identical to the interface of the standard system `as` tool, to avoid having to make changes to spec files.

When the user (or a makefile) links the program using our `gcc` compiler driver, it invokes our `gccld` tool. This tool reads the `.o` files specified, links in the appropriate bytecode files from any `.a` files, and then runs a series of interprocedural optimizations on the program. At this time, we directly emit an LLVM bytecode file for the entire program, instead of automatically invoking a native code generator.

Once the program has been optimized and is available in a single bytecode file, there are several ways to execute the resultant program. LLVM provides a very slow (but portable) reference interpreter for bytecode files, a Sparc V9 native code generator, a C back-end, and a Just-In-Time (JIT) compiler for the IA32 architecture.

A large number of LLVM optimizations and analyses are available, including passes for:

- Traditional SSA based optimizations: ADCE, GCSE, LICM, PRE, SCCP, induction variable canonicalization, reassociation, value numbering, register promotion, etc. . .

- Control Flow Graph based optimizations and analyses: critical edge elimination, loop canonicalization, various dominator, post-dominator, and control dependence graph related analyses, interval construction, natural loop construction, CFG simplification, path profiling instrumentation, etc. . .

- Interprocedural analyses and transformations: call graph construction, several interprocedural alias analyses, global variable merging, dead global elimination, inlining, Data Structure Analysis [13], automatic pool allocation [12], interprocedural mod/ref, etc. . .

In addition to pure infrastructure, the LLVM system also provides a large test suite. The three main sections of the test suite are the regression tests (which contain thousands of tests for transformations and other tools), feature tests (which demonstrate how instructions and idioms are used in LLVM), and program tests (which compile benchmarks and other programs with the various code generators, ensuring that they produce code whose behavior agrees with a native compiler). The LLVM web site also hosts a variety of documentation describing aspects of the infrastructure.

LLVM is also still under development. In particular, the C^{++} front-end is nearing completion (runtime library support for exception handling is the major missing portion), Sparc V9 support for the JIT is in development, and a system for runtime optimization of statically compiled binaries is in the research phases.

4.2 Differences from the Proposal

The biggest difference between the proposal and the LLVM implementation is the lack of an LLVM to RTL conversion pass. For our research purposes, we use a C back-end, which provides much of the same functionality as a full fledges RTL back-end, but is much slower. We expect that this component can be added upon demand.

Another big difference between the current implementation and the proposal is the interface between the `cc1` program and the mid-level optimizer. For expediency of implementation we currently have the two tools as separate executables, although this obviously incurs more overhead than linking the two components together. Once the subject of including C^{++} code in GCC is better decided, we can look to resolve this issue.

4.3 Support for Developers

One of the strengths of the LLVM infrastructure is that it has some interesting utilities for constructing passes, finding bugs in those passes, and building a compiler around a selection of these passes. This strength is important for two reasons: it allows new people to get into the system and get productive relatively fast, and it also allows experienced developers to be more productive than they otherwise would. The most important features are: a strong consistency checker, a "pass manager," and a tool we call `bugpoint`.

The LLVM infrastructure includes a stringent checker for LLVM code, which ensures that type relationships, SSA properties (e.g., all definitions dominates their uses), and other LLVM invariants haven't been violated by a transformation. This checker is automatically run after passes when in development mode to ensure that these passes are not corrupting the input for other passes that are run. Additionally, when in development mode, an automated memory leak detector is automatically enabled, which detects violations of the LLVM representation's ownership model. This lightweight checker is implemented using only a few additions to constructors and destructors for the classes which make up the representation, no garbage collector is necessary.

The LLVM "Pass Manager" provides a structured environment for passes to execute in. Transformations in LLVM use a declarative syntax to indicate which other passes are prerequisites (e.g. `break-critical-edges`), which analyses are required (e.g. natural loop information, alias analysis, value numbering, interprocedural mod/ref info, etc.), and which analyses are preserved or destroyed by the transformation being run. This structured pass model makes it easier for developers to fit code into the system, and it also makes construction of tools (e.g. `gccas` and `gccld`) a simple matter of handling command-line arguments and selecting a sequence of passes to run.

`bugpoint`, another useful tool, is best described as an "automated test-case reducer." Given an LLVM program (or fragment) and a list of passes to run, it attempts to reduce the test-case (and list of passes) to the minimum which still exposes a problem. `bugpoint` can currently diagnose passes which crash/assert during optimization and passes which misoptimize the program (by executing the resultant program with a code generator, assuming a deterministic program)[6]. If a test-case causes a pass to crash, `bugpoint` is usually able to reduce the test-case down to the few LLVM instructions and basic block which cause the problem. If a pass (or combination of passes) miscompiles the test-case, it can isolate a single function which is being miscompiled. The `bugpoint` tool is possible because of the modularity of the pass manager and the ability to read, write, and modify a representation of whole programs.

[6]A third mode, for debugging back-end bugs, is planned.

4.4 Surprises and Insights from LLVM

Through the experience of developing LLVM, we have developed several insights which may be useful to a broad audience. First, implementing a type-safe linker for C is a non-trivial exercise. C programs often rely on implicit prototypes for called functions, or use prototypes that are blatantly wrong. We have also seen cases where global data is declared to have different types in different translation units (which, in practice, behaves similarly to a COMMON block in FORTRAN). A normal binary linker does not typically have problems with these issues, but they must be handled explicitly with a type-safe linker. On the other hand, this information is often useful to the programmer, like the "`lint`" tool.

When performing interprocedural analysis, having as much of the program available as possible increases the precision of the analyses. For this reason, we have compiled several libraries to LLVM form that allow them to be analyzed and optimized with the program. This has several interesting consequences: first, the library code itself can be specialized and optimized with the program (for example, optimizing `qsort` by inlining the comparison functions, so indirect calls do not need to be used). Second, this dramatically reduces the need for ad-hoc annotations on functions indicating properties such as "`const`" and "`pure`". Instead, simple interprocedural analyses can be used, which have the advantage of applying to user code as well as the built-in functions.

Finally, we have found that investing in making the system easier to develop for, and de-

Source Filename	`wc -l` **LOC**	**GCC CSE 1**	**LLVM Pass Times IC**	**GER**	**GCSE**	**Sum**	**# LLVM Pass xforms IC**	**GER**	**GCSE**
combine.c	11103	0.70s	.431s	.027s	.141s	.599s	16182	141	2734
expr.c	10747	0.52s	.141s	.009s	.072s	.222s	6540	41	2870
cse.c	8779	0.50s	.187s	.012s	.061s	.260s	10925	59	1894
reload1.c	7117	0.37s	.058s	.008s	.034s	.100s	5735	86	1830
c-decl.c	6968	0.42s	.022s	.005s	.031s	.058s	3299	3	2221
insn-recog.c	6957	0.34s	.082s	.004s	.090s	.176s	5238	0	654
loop.c	6648	0.33s	.013s	.001s	.003s	.017s	1671	7	264
c-typeck.c	6604	0.46s	.028s	.005s	.026s	.059s	4481	14	1993

Table 1: Transformation timings for source files from the SPEC CPU2000 176.gcc benchmark

bug in, has been worth it. In particular, the `bugpoint` tool can narrow down a test-case from thousands of lines of C code to a dozen lines of LLVM code in a few seconds: doing the same manually would take *much* longer. Making the development environment detect problems early is also extremely valuable to developers, making them more productive and making it easier to bring new people on. Having a modular system also helps keep people from getting overwhelmed when they first start on the project.

4.5 Optimizer Performance

The LLVM representation allows for efficient transformations and analyses, both for aggressive interprocedural transformation and traditional optimizations. In order to quantify this performance, we compared the performance of the GCC "`cse`" pass with the performance of the LLVM transformations closest to it (see Table 1). For these tests, we compiled the 8 largest single `.c` files in the SPEC CPU2000 176.gcc benchmark (which is based on the GCC 2.7.2.2 source code). The numbers were collected on a 1.7GHz AMD 2100+ Athlon processor.

The timings for the `cse` pass were collected when compiling with GCC 3.2 and the `-O3` option. The actual timings were acquired as the average of 5 runs with the `-ftime-report` option and the compiler configured for a `i686-pc-linux-gnu` target. The `cse 2` pass was ignored, the timings just include the first invocation of the `cse` pass.

For the LLVM timings, we chose to use a combination of the **I**nstruction **C**ombining, **G**lobal **E**xpression **R**eassociation, and **G**local **C**ommon **S**ubexpression **E**limination passes. The combination of these three phases is believed to be strictly more powerful than the `cse` pass. The Instruction Combining pass supersumes value numbering, constant folding and trivial dead code elimination phases, plus it performs a variety of transformations similar to the GCC "combine" pass (described below). The reassociation pass transforms chained occurrences of commutative operations to promote better code motion. The GCSE pass is a well known technique to remove common subexpressions. The table shows the execution time for each pass as well as the sum of the three. The table also shows the number of transformations that each pass makes (instructions combined, instructions reassociated, common subexpressions deleted).

From the table, we can see that the LLVM optimizations always run in less time than the `cse` pass, and with the exception of the "combine.c" case, took about half as much time. Despite being faster overall, the LLVM transformations are more powerful than the `cse` pass, which only operates on extended basic blocks. The slowest individual transformation by far is the instruction combination pass, which uses a work-list driven approach to perform "peephole" style optimization on the SSA graph (giving it global transformation powers) for a large collection of algebraic identities (such as folding "$(A - (A\&B))$" into "$(A\& \sim B)$"), that the `cse` pass does not perform. Together, the three transformations are quite effective.

In addition to simple scalar optimizations, LLVM is designed to support aggressive interprocedural analyses and optimizations at link-time. As an example, we consider the Data Structure Analysis algorithm, a context-sensitive flow-insensitive memory analysis framework. On the same hardware as above it is capable of analyzing entire programs in seconds: 2.5s the `povray` and 1.2s for the `255.vortex` programs, which are about 136,000 and 67,000 lines of C code respectively [13]. Other simpler algorithms may obviously run much more quickly.

5 Related Work

There is a vast amount of related work on interprocedural optimization in research and commercial compilers [1, 8, 2, 9, 3]. To avoid major changes to the build process, all of these compilers combine the program together at link-time in a very high-level representation, before any substantial optimization is performed. Most often, this representation takes the form of the source language Abstract Syntax Tree (AST) with source language-specific nodes removed. Once the program is combined at link-time, optimization for the entire program commences, starting with interprocedural optimizations.

In contrast, the approach described here immediately optimizes and translates the program to a low-level, but strongly-typed, intermediate representation which is suitable for optimization both at compile- and link-time. Because substantial optimization is performed at compile-time, the interprocedural optimizers have less work to perform at link-time, reducing the amount of time a recompilation requires. Previous work [13, 7, 10, 12] has shown that a low-level representation with type information can support aggressive high-level analyses and transformations.

Another successful class of interprocedural optimizers target very low-level optimizations. These "smart-linkers" typically operate at the level of the machine code, performing optimizations such as interprocedural register allocation and code layout optimizations [16, 14, 5]. Although these tools have been successful, and require little or no modification to the source compiler, they are not capable of performing high-level optimizations at all. Also, these optimizations can all be performed in our framework, because code generation occurs for the entire program at a time, exposing the necessary interprocedural information.

Within the GCC project, several projects in development or recently merged onto the mainline are relevant. In particular, the `ast-optimizer` project and its `tree-ssa` subproject aim to improve optimization in GCC by migrating optimizations from the target-specific RTL representation to a target-independent AST representation. The representation proposed in this paper is similar to the `tree-ssa` GIMPLE representation in some ways (both are language-independent, SSA based, and do not allow nested expressions), but they are different in many other ways.

In particular, the GIMPLE representation is not capable of representing the entire translation unit being compiled: a lot of information about the program is stored only in global variables, or are immediately emitted to the output assembly file. Also, the GIMPLE representation has operations which are closer to the source level. For example, variable definitions can have their address taken, which makes the def-use chain representation much more complex in the GIMPLE representation. On the other hand, the `tree-ssa` project is much better integrated into GCC, is written in the C language, and does not require the introduction of a completely new intermediate representation.

6 Conclusion

This paper presents the design for an aggressive, but realistic, interprocedural optimization component for the GNU Compiler Collection. This design is capable of supporting a broad range of whole-program optimization techniques, is reasonable in terms of compilation time, and has already been implemented. We hope our efforts will accelerate the process of making GCC produce code which is more competitive with commercial compilers, and perhaps LLVM can be directly adopted as an optional part of the compiler itself. We encourage members of the community who are interested in the proposed architecture or LLVM itself to contact the authors with any feedback, questions, or ideas.

References

[1] J. Amaral, G. Gao, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure: A tutorial. The International Conference on Parallel Architeture and Compilation Techniques (PACT2000), Oct. 2000.

[2] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. In *Proc. SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 301–312, Montreal, June 1998.

[3] D. Blickstein, P. Craig, C. Davidson, N. Faiman, K. Glossop, R. G. S. Hobbs, and W. Noyce. The gem optimizing compiler system. *Digital Technical Journal*, 4(4):121–136, 1992.

[4] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[5] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. 2003 ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems*, Feb 2003.

[8] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, (Q4), 1999.

[9] A. Holler and Hewlett-Packard Company. Compiler optimizations for the PA-8000. In *Proc. IEEE International Computer Conference*, 1997.

[10] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See `http://llvm.cs.uiuc.edu`.

[11] C. Lattner and V. Adve. LLVM Assembly language reference manual, `http://llvm.cs.uiuc.edu/docs/LangRef.html`.

[12] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[13] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[14] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[15] R. Stallman. *The GNU C compiler*. Free Software Foundation, 1991.

[16] D. Wall. Global register allocation at link-time. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, 1986.

The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC

Vladimir N. Makarov
Red Hat
`vmakarov@redhat.com`

Abstract

A new model to describe the pipeline characteristics of processors is proposed in the article. The model is based on the usage of regular expressions. The model is compared to the one used in GNU C compiler (GCC) for long time. The article also describes the pipeline hazard recognizer generated from the new model currently implemented in GCC and instruction scheduler which uses the pipeline hazard recognizer. The current implementation of the pipeline hazard recognizer is based on the usage of *deterministic* and *nondeterministic* finite state automata.

Examples of usage of the new model, the pipeline hazard recognizer, and the instruction scheduler based on it are given. Possible future directions of developing them to use them for different algorithms of instruction scheduling and software pipelining are discussed.

Introduction

To increase the productivity of computer systems the modern processors can execute several instruction simultaneously. It is achieved by using several functional units and/or pipelined functional units. Of course the instruction execution could start only if the input data are ready and enough processor functional units necessary for the instruction execution are available. If at least one of the two conditions is not satisfied, a processor stall might occur and the instruction execution might be delayed. The delay because the first condition is not satisfied is called data delay. The delay because of the second condition is called resource delay.

A special component in an optimized compiler, called the instruction scheduler, is responsible for decreasing the data and resource delays and (as a consequence) to increase the parallelism of instruction execution. It is achieved mainly by changing the original order of instructions, although more powerful code transformation (like instruction cloning, partial register renaming and forward substitution, and instruction mutation) could be used. An important component of the instruction scheduler responsible to find the resource delays is called the pipeline hazard recognizer.

There is big variety of processors even for one architecture. Therefore writing the pipeline hazard recognizer manually is not wise. This is especially true for portable compilers. Therefore many compilers have a model to describe pipeline characteristics of the target processors and usually a generator of pipeline hazard recognizers. The model language can be a subset of the compiler implementation language (like C used to describe the reservation tables) or a special language designed for this task.

GCC as a compiler ported to most platforms had such a model and generator for long time. This model has its drawbacks. It can not accurately describe many modern processors. As a consequence the generated code is worse than it could be with the same instruction scheduler. The more pipeline irregularity the processor has, the more is the impact of an instruction scheduling inaccuracy. Another drawback is the inconvenience of description. The model is oriented to describe which instructions a functional unit executes instead of the more natural model in which the reservation of the functional units by given instruction is described.

GCC pipeline hazard recognizer is a part of the instruction scheduler itself. It is driven by tables generated from the description. The tables are just a simple translation of the description. The more complex the pipeline description is, the slower the pipeline hazard recognizer is. The modern processor becomes more complex and the slow speed of the pipeline hazard recognizer becomes a problem.

To solve this drawback, the new model and implementation of the pipeline hazard recognizer have been proposed. The model is based on the usage of regular expressions describing all the reservations of functional units by instructions. The corresponding implementation of pipeline hazard recognizer is based on the usage of finite state automata.

Each state of the automaton encodes all current and planned reservations of functional units. If there is an arc from one state to another state marked by an instruction, then the instruction can be issued in a given state and there will be no conflicts on functional unit usage with the instructions issued earlier. The destination state encodes all current and planned functional unit reservations after issuing the instruction. Each state also has an arc marked by *cycle advancing*. The destination state in this case is the state after increasing the simulated processor cycle. Transitions by the arc finally result in freeing functional units.

So the instruction scheduler should only check the presence of the arcs marked by the instruction from given state to find a resource delay. After issuing the instruction the instruction scheduler should change the current state to the destination state. If no instruction can be issued, the instruction scheduler should change the current state to the destination state into which an arc marked by ‘cycle advancing’ enters and increase the simulated processor cycle.

This approach is not new. It has been described in [Bala, Proebsting]. What is a really new thing in the approach described in the article is usage of *alternatives* in the reservations. The alternatives can be treated *deterministically* and *nondeterministically*.

The deterministic treatment of the alternative is to try the first alternative reservation and, if there is a conflict on any functional unit reserved by previously issued instructions, try the next alternative. The nondeterministic one is to try all alternative reservations concurrently.

The first section of the article describes in more detail the description model and the corresponding pipeline hazard recognizer used in GCC for a long time. It also describes the drawbacks of such an approach. In the second section, the proposed model is described. The third section describes the generation of the pipeline hazard recognizer from the proposed model and its interface to the instruction scheduler. The fourth section contains examples of descriptions as deterministic and nondeterministic ones. The fifth section describes an algorithm, called the first cycle multipass instruction scheduling. The algorithm improves instruction scheduling by evaluation of more than one instruction schedule. Usage of the fast pipeline hazard recognizer makes it practical. In the sixth section, the possible future directions of developing the proposed approach are discussed.

1 The old GCC processor pipeline description model

This section is based on the documentation of Gcc internals [Gcc]. Practically all processor parallelism for GCC is described with the aid of one type of constructions—`define_function_unit`—in a Gcc machine description file. Each usage of a functional unit by a class of instructions is specified with a `define_function_unit` expression (see Table 1).

`(define_function_unit NAME MULTIPLICITY SIMULTANEITY TEST READY-DELAY ISSUE-DELAY [CONFLICT-LIST])`	
`NAME` is a string giving the name of the functional unit.	`MULTIPLICITY` is an integer specifying the number of identical units in the processor. If more than one unit is specified, they will be scheduled independently.
`SIMULTANEITY` specifies the maximum number of instruction that can be executing in each instance of the functional unit simultaneously.	`TEST` is an attribute test that selects the instructions we are describing in this definition. Note that an instruction may use more than one functional unit.
`READY-DELAY` is an integer that specifies the number of cycles after which the result of the instruction can be used without introducing any stall.	`ISSUE-DELAY` is an integer that specifies the number of cycles after the instruction matching the `TEST` expression begins using this unit until a subsequent instruction can begin. A cost of `N` indicates an `N-1` cycle delay.
`CONFLICT-LIST` is an optional list giving instructions with which additional conflicts occur.	

Table 1: The old description model construction.

As an example, consider a classic RISC machine where the result of a load instruction is not available for two cycles (a single "delay" instruction is required) and where only one load instruction can be executed simultaneously. This would be specified as:

```
(define_function_unit "memory" 1 1
       (eq_attr "type" "load") 2 0)}
```

For the case of a floating point function unit that can pipeline either single or double precision, but not both, the following could be specified:

```
  (define_function_unit "fp" 1 0
         (eq_attr "type" "sp_fp") 4 4
         [(eq_attr "type" "dp_fp")])
  (define_function_unit "fp" 1 0
         (eq_attr "type" "dp_fp") 4 4
         [(eq_attr "type" "sp_fp")])
```

A special utility in Gcc generates different tables of bit vectors, macros, and some functions (mainly for dealing with conflict lists), which are used by the pipeline hazard recognizer embedded into the instruction scheduler.

The current GCC instruction level parallelism description model has serious drawbacks. The biggest one is that the description model is not powerful enough. Each functional unit is believed to be reserved at the start of instruction's execution. The model also does not permit alternatives in the reservations. This is a big constraint for accurate descriptions of modern processors. As a consequence of inaccurate descriptions, the machine dependent files of Gcc contain a lot of code to fix it. For example, the SPARC machine-dependent files contained about one thousand lines of C code.

Another important drawback of the model is the unnatural way of description when a developer should write a unit and condition which selects instructions using the unit. My experience shows that writing all units reservation for an instruction (an instruction class) are more natural.

The pipeline hazard recognizer of resource delays has a slow implementation. The Gcc schedulers support structures which describe the unit reservations. The more complex the pipeline description, the slower the pipeline hazard recognizer. Such implementation would become even slower when we enable to reserve functional units not only at the instruction execution start. The slow implementation becomes critical for the modern processor (especially VLIW and EPIC).

2 The proposed processor pipeline description model and its implementation

As the old processor pipeline description, the proposed pipeline description should be placed in the machine description files of Gcc. There are several constructions to describe the processor. The order of all such constructions in the machine description file is not important. All constructions are Lisp like construction because the machine description file has Lisp like syntax. Please don't be confused—it is just an implementation form of the description model. The syntax of the major constructions is given on Table 2.

`(define_automaton AUTOMATON-NAME)`	`AUTOMATA-NAME` is a string giving the name of the automaton.
`(define_cpu_unit UNIT-NAMES AUTOMATON-NAME)`	`UNIT-NAMES` is a string giving the names of the functional units. `AUTOMATON-NAME` is a string giving the name of the automaton to which the unit is bound.
`(define_insn_reservation INSN-NAME DEFAULT-LATENCY CONDITION REGEXP)`	`DEFAULT-LATENCY` is a number giving the latency time of the instruction. `INSN-NAME` is a string giving an internal name of the instruction. It is good practice to use the instruction class names as described in the processor manual. `CONDITION` defines what RTL instructions are described by this construction. `REGEXP` is a string describing the reservation of the cpu's functional units by the instruction (the syntax is given in table 3).
`(define_reservation RESERVATION-NAME REGEXP)`	`RESERVATION-NAME` is a string giving the name of REGEXP.
`(exclusion_set UNIT-NAMES UNIT-NAMES)` `(presence_set UNIT-NAMES PATTERNS)` `(absence_set UNIT-NAMES PATTERNS)`	`UNIT-NAMES` is a string giving names of functional units. `PATTERNS` is a string giving patterns of functional units separated by a comma. Currently a pattern is one unit or units separated by white-spaces.

Table 2: The major constructions of the proposed description model.

To describe a processor, first we should define an automaton with the construction `define_automaton`. We can have more than one automaton in a machine description file. All the automata should have unique names. The automaton name is used in the construction `define_cpu_unit`.

It is good practice to use separate automaton to describe a processor of a given architecture. For example, the machine description file for SPARC architecture could have one automaton for UltraSparcII and another one for UltraSparcIII.

We could also use more than one automaton to describe a single processor. Sometimes the generated finite state automaton used by the pipeline hazard recognizer is large. If we use more than one automaton and bind functional units to the automata, the summary size of the automata is usually less than the size of the single automaton.

Each functional unit used in the description of instruction reservations should be described by the construction `define_cpu_unit`.

The construction `define_insn_reservation` is the major construction to describe pipeline characteristics of an instruction. The reservations are described by regular expressions according to the syntax on Table 3.

`regexp = regexp "," oneof`
`       | oneof`
	`,` is used for describing the start of the next cycle in the reservation.

`allof = allof "+" repeat`
`      | repeat`
	`+` is used for describing a reservation described by the first regular expression and the second regular expression etc.

`oneof = oneof "|" allof`
`      | allof`
	`|` is used for describing a reservation described by the first regular expression or the second regular expression etc.

`repeat = element "*" number`
`       | element`
	`*` is used for convenience and simply means a sequence in which the regular expression is repeated `NUMBER` times with cycle advancing (see '`,`').

`element = cpu_unit_name`
`        | reservation_name`
`        | result_name`
`        | "nothing"`
`        | "(" regexp ")"`
	`cpu_unit_name` denotes reservation of the named cpu functional unit. `nothing` denotes no unit reservations.

Table 3: Syntax of the regular expressions.

As an example, consider a superscalar RISC machine which can issue three instructions (two integer instructions and one floating point number instruction) on a cycle but can finish only two instructions. To describe this, we define the following functional units.

```
(define_cpu_unit "i0_pipeline, i1_pipeline")
(define_cpu_unit "f_pipeline,port0, port1")
```

All simple integer instructions can be executed in any integer pipeline and their result is ready in two cycles. The simple integer instructions are issued into the first pipeline unless it is reserved, otherwise they are issued into the second pipeline. Integer division and multiplication instructions can be executed only in the second integer pipeline and their results are ready correspondingly in 8 and 4 cycles. The integer division is not pipelined, i.e. the subsequent integer division instruction can not be issued until the current division instruction finished. Floating point instructions are fully pipelined and their results are ready in 3 cycles. To describe all of this we could specify

```
(define_cpu_unit "div")
(define_insn_reservation "simple" 2
   (eq_attr "cpu" "int")
   "(i0_pipeline|i1_pipeline), (port0|port1)")
(define_insn_reservation "mult" 4
   (eq_attr "cpu" "mult")
   "i1_pipeline, nothing*2, (port0|port1)")
(define_insn_reservation "div" 8
   (eq_attr "cpu" "div")
   "i1_pipeline, div*7, div + (port0|port1)")
(define_insn_reservation "float" 3
   (eq_attr "cpu" "float")
   "f_pipeline, nothing, (port0|port1))
```

In our example we see that the unit reservations for different instructions contain common parts. In such case, we can simplify the pipeline description by defining an abbreviation by the construction `define_reservation`. To simplify the description in our example we could use a reservation as follows

```
(define_reservation "finish" "port0|port1")
(define_insn_reservation "simple" 2
       (eq_attr "cpu" "int")
       "(i0_pipeline | i1_pipeline), finish")
```

Some processors (especially VLIW ones) have many constraints which are quite difficult to describe only by the constructions mentioned above. The three constructions `exclusion_set`, `presence_set`, and `absence_set` make description easy.

The first construction (`exclusion_set`) means that each functional unit in the first string can not be reserved simultaneously with a unit whose name is in the second string and vice versa. For example, the construction is useful for describing processors (e.g. some SPARC processors) with a fully pipelined floating point functional unit which can execute simultaneously only single precision floating point instructions or only double precision floating point instructions.

The second construction (`presence_set`) means that each functional unit in the first string can not be reserved unless at least one of the pattern in the second string has been reserved. This is an asymmetric relation. For example, it is useful to description that VLIW `slot1` is reserved after a reservation `slot0` or `slot1` is reserved only after a `slot0` and unit `b0` reservation. We could describe it by the following constructions:

```
(presence_set "slot1" "slot0")
(presence_set "slot1" "slot0 b0")
```

The third construction (`absence_set`) means that each functional unit in the first string can be reserved only if each pattern in the second string is not reserved. This is an asymmetric relation. For example, it is useful for description that VLIW `slot0` can not be reserved after a `slot1` or `slot2` reservation or that `slot2` can not be reserved if `slot0` and unit `b0` are reserved or `slot1` and unit `b1` are reserved. We could describe it by the following constructions:

```
(absence_set "slot2" "slot0, slot1")
(absence_set "slot2" "slot0 b0, slot1 b1")
```

All functional units mentioned in a set should belong to the same automaton.

There are other constructions to describe pipeline characteristics of processors. But for the sake of brevity they are not described in this article.

A special utility (the generator) generates the automaton based pipeline hazard recognizer in a separate file. The instruction scheduler communicates with it through a procedural interface. The major procedure gets an automata state and an instruction as parameters and returns information on whether the instruction can be issued or not. If it can be issued then the procedure changes the state to reflect the instruction issue.

Each state of the automaton encodes all current and planned reservations of functional units. If there is an arc to another state marked by an instruction, then the instruction can be issued in the given state and there will be no conflicts on functional unit usage with the instructions issued earlier. The destination state encodes all current and planned functional unit reservations after issuing the instruction. If the instruction parameter is null, it means that the simulated processor cycle should be advanced. Each state has an arc marked by `cycle advancing`. The destination state in this case is the state after incrementing the simulated processor cycle. Transitions by such arcs result in the freeing of all functional units.

The DFA pipeline hazard recognizer is believed to not be as flexible as the old Gcc recognizer. This is not true. It is easy to get information from the automata. For example, the generator also generates many other procedures like querying the reservation of functional units for a given automaton state, finding the minimal reservation delay needed to issue an instruction in a given state, checking that no one instruction can be issued in given state and so on.

The *nondeterministic* treatment of alternatives means trying all alternatives concurrently. Some of them may be rejected by reservations in the subsequent instructions. Actually, the nondeterministic treatment of alternatives is enough to describe deterministic alternatives. For example, let us look at the following reservation with deterministic treatment of alternatives.

```
(define_reservation "deterministic"  "u1|u2")
```

It means that we reserve `u1` and, if it is not possible (because `u1` has been already reserved), we reserve `u2`. We can describe it with the following constructions

```
(define_reservation "nondeterministic"
       "u1|u2+u1_present")
(presence_set "u1_present" "u1")
```

Here we use a reservation with nondeterministic treatment of the alternative. What variant of alternative should we use? The processors are deterministic devices, so alternatives should usually be treated deterministicaly (this is the default treatment). Let us look at a dual instruction issue processor which has two integer units. One integer unit `IU1` can execute any integer instruction and another one (`IU2`) can execute any integer instruction except multiply. In the first example, the processor always issues instructions into `IU1` if it is free. The processor could be described by using deterministic alternatives as follows

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "IU1 | IU2")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "IU1")
```

Actually the processor has a bad design because if an integer instruction is followed by multiply instruction the two instructions can not be issued simultaneously. The improved processor should always issue an integer instruction into `IU2` if it is not busy. We could describe this using deterministic alternatives as follows

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "IU2 | IU1")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "IU1")
```

On the other hand we could use nondeterministic treatment in the example too. The result automaton would be the same. But nondeterministic treatment could better reflect the processor's behaviour if the processor had an instruction look ahead buffer to find the best assignment of functional units to instructions in the buffer. Another example of usage of the nondeterministic treatment of alternatives for Itanium and Itanium2 processors is described in the next section.

Generally speaking, the same processor can be described differently. I would distinguish two kind of descriptions. One is the *structural* description which describes (almost) all processors functional units mentioned in processor's documentation. Another one (*behavioural*) aims to describe only pipeline hazards (sometimes with the aid of non-existing functional units). The first one is usually more verbose and the resulting automata are bigger. The second one is simpler and the resulting automata are smaller. But it is better to follow the documentation (in other words to use a structural description) because it makes understanding the description easier for other people.

3 Generation of the pipeline hazard recognizer

Here is a brief description of the phases of the generator of pipeline hazard recognizers and the more interesting tasks solved by the generator. First, the generator of pipeline hazard recognizer translates the pipeline description into an internal representation.

Then it checks the correctness of the automaton pipeline description. The most nontrivial task is to check the correctness of assignments of functional units occurring in a reservation to the automata. There is no such problem for reservations without alternatives [Bala]. Let us consider the following description:

```
(define_cpu_unit "div" "div")
(define_cpu_unit "decode" "rest")
(define_insn_reservation "div" 3
       (eq_attr "cpu" "div") "decode + div*3")
```

The corresponding automata are given on Figure 1. The figure also contains the single automaton as if all units were assigned to one automaton. They behave analogously to the single automaton with the two functional units `decode` and `div`. It means that transition marked by an instruction exists in the single automaton if and only if there are transitions marked by the instruction between the corresponding states of all two automata. Instead of changing only one state for a single automaton, the pipeline hazard recognizer changes the states of the two automata simultaneously. Although a number of the states is hidden in the pipeline hazard interface and there is only one state in the interface, in reality the interface state is represented by two states and pipeline hazard recognizer internally manipulates the states of the two automata.

Let us consider a more advanced dual instruction issue processor with a faster division unit.

```
(define_cpu_unit "decode1" "a1")
(define_cpu_unit "div,decode2" "a2")
(define_insn_reservation "div" 2
       (eq_attr "cpu" "div")
       "(decode1|decode2) + div*2")
```

For automata `a1` and `a2` we have correspondingly the following functional unit reservations for the instruction `div`

```
decode1|nothing
nothing|decode2 + div*2
```

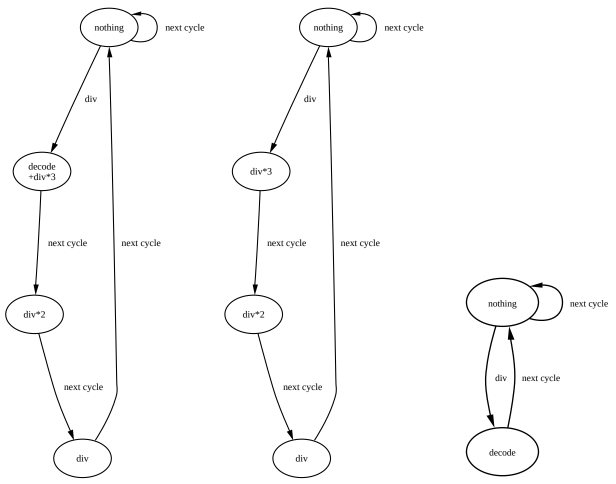

Figure 1: The single automaton and the two automata of the single issue processor.

Figure 2 contains the single automaton (as if all units were assigned to one automaton) and the corresponding two automata.

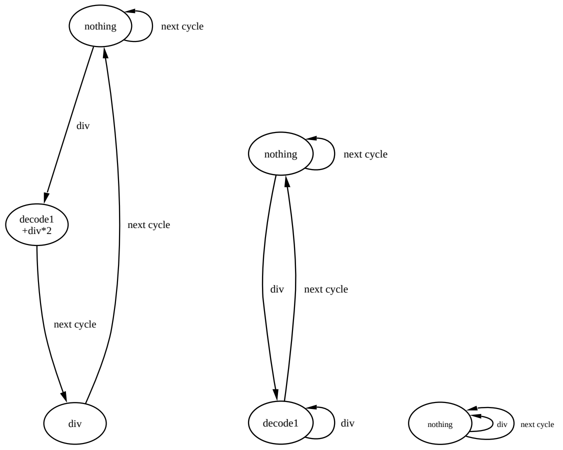

Figure 2: The single automaton and the two incorrect automata of the dual issue processor.

The two automata are not equivalent to the single automata. For example, we could issue any number of division instructions on one cycle according to the two automata. The simple solution of this problem could be the usage of the requirement to assign all functional units occurring in the same reservation to the same automaton. It is a very severe constraint to assign functional units to automata which results in the impossibility of decreasing automata size in many cases even if we have reservations without alternatives. Instead of it, the current implementation uses a less severe requirement. If a functional unit reservation (`div` in our example) is present on a particular cycle of an alternative for an instruction reservation, then some unit from the same automaton must be present on the same cycle for the other alternatives of the instruction reservation. The requirement is not too complicated to be understood and it still helps to considerably decrease automata size in many cases. Let us consider the following distributions of the functional units (The corresponding automata are given on Figure 3):

```
(define_cpu_unit "decode1,decode2" "a1")
(define_cpu_unit "div" "a2")
(define_insn_reservation "div" 2
       (eq_attr "cpu" "div")
       "(decode1|decode2) + div*2")
```

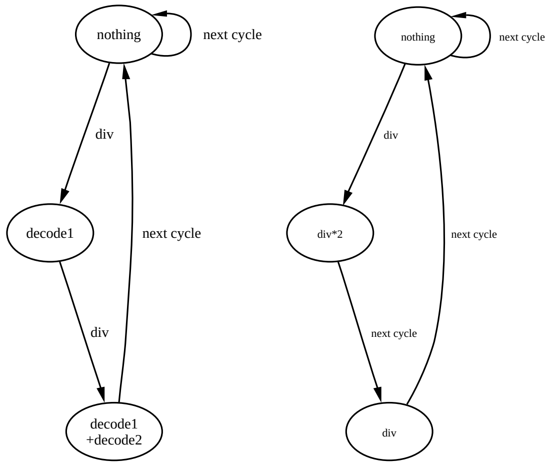

Figure 3: The two correct automata of the dual issue processor.

We see that the automata on figure 3 behave analogously to the single automaton.

After checking the description, the generator of the pipeline hazard recognizer creates the automata and, if the alternatives are treated nondeterministicaly, transforms nondeterministic finite state automata into deterministic ones.

After creating the automata, the generator does a minimization of the finite state automata by merging automaton states (I should mention that Gcc experience shows importance of some preliminary minimization during building the automata because even if the minimized is small the automata before the minimization could be huge). The minimization task is a bit complicated. If we have functional units in the description whose reservation may be queried for a given state. Let us consider a processor with different functional units for multiply and for the rest of the integer instructions

```
(define_insn_reservation "int" 1
       (eq_attr "cpu" "int") "decode + int")
(define_insn_reservation "mult" 1
       (eq_attr "cpu" "mult") "decode + mult")
```

The corresponding automata before and the after the minimization are given on Figure 4. If we want to know whether functional unit `mult` is reserved in the second state of the minimized automaton, we can not get this information from just the state. The simplest solution of the problem could be prohibiting the minimization for automata with queried units. Unfortunately such a solution is not reasonable because automaton minimization is an important optimization which permits to considerably decrease the size of the automata in many cases. Instead of the simplest approach we use minimization with modified state equivalence. The new state equivalence takes queried functional units in the corresponding reservations into account. This approach still permits to considerably decrease the automata size in many cases.

After the minimization, the generator forms tables, compresses them by different algorithms

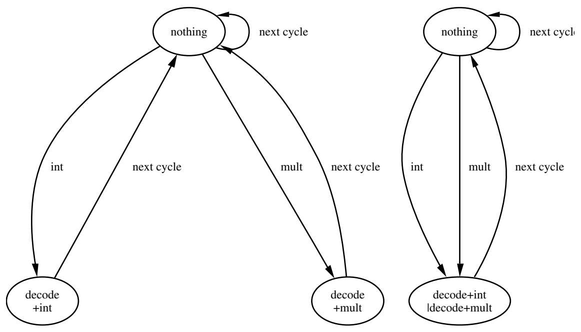

Figure 4: The automaton before and after the minimization.

(like a comb vector algorithm) and outputs them (and functions accessing them (including functions which are interface functions of the pipeline hazard recognizer)) into a C file for further compilation.

The biggest problem of the usage of the DFA approach is the size of the automata. How big can the automata be? For example, Gcc for Intel IA64 has four automata (two for Itanium and two Itanium2) with 24K states and 170K arcs. But this is an extreme case. Itanium and Itanium2 have extremely complicated pipeline characteristics. The IA64 automata are also used for VLIW packaging (bundling instructions). Therefore the IA64 automata have many queried units.

To solve the big automata size problem, it is better to split an automata into several ones and not to use queried units as it was mentioned in above. Now automaton splitting should be done manually by assigning functional units to the automata. Automatic splitting of an automaton into several automata with total size less than the size of the original automaton is a challenging research work.

4 Usage of the proposed model and the pipeline hazard recognizer

The first public usage of DFA based instruction scheduling was for UltraSparc. The previous implementation of the pipeline hazard recognizer contained about 1000 lines of machine-dependent C code for tuning the old pipeline hazard recognizer generated from a non-DFA pipeline description. The DFA description of Sparc which resulted in all this code has been gone and the instruction scheduling has been improved. Table 4 contains a comparison of SPECfp95 run a 500 Mhz UltraSparcIIe box. The average improvement of EEMBC [EEMBC] for a 233 Mhz UltraSparcII box was 5.5%.

Benchmarks	Ratio	Ratio
101.tomcatv	12.6	13.4
102.swim	22.4	22.7
103.su2cor	5.95	6.04
104.hydro2d	6.04	6.05
107.mgrid	7.73	8.65
110.applu	7.52	7.65
125.turb3d	13.7	13.8
141.apsi	10.9	11.0
145.fpppp	11.0	11.4
146.wave5	14.7	15.0
SPECfp95 (Geom. Mean)	10.4	10.7

Table 4: Sparc GCC with non-DFA and DFA pipeline hazard recognizers.

The usage of a DFA description for SH4 is an example of the importance of accurate pipeline descriptions for processors which have complicated pipeline constraints: such as SH4. Improvement of instruction scheduling with the DFA pipeline hazard recognizer for SLALOM benchmark [Slalom] on a 200Mhz SH4 box was about 12-13%.

A good example of usage of *nondeterministic* automata is the description of Itanium and Itanium2 processors. The IA64 architecture is an extension of a typical VLIW architecture. Instructions can only be placed in specific slots (syllables in IA64 terminology) of a VLIW instruction (bundle in IA64 terminology). To place an instruction in the current bundle or the next bundle, sometimes one or two `NOP` instructions should be issued first. Gcc already had pipeline hazard recognizer for the Itanium processor. It was written manually on C because the old description model was not powerful enough. The code was big and complicated. It was tuned very well to achieve good instruction scheduling. The code tried to insert such `NOP` instructions.

The nondeterministic automaton permits to easily describe where to insert such `NOP` instructions. The DFA descriptions have been written for Itanium and Itanium2 processors. Each processor has been described by two automata. The first (nondeterministic) automaton described the instruction reservations with an optional issue of one or two `NOP` instructions before the instruction. So the pipeline hazard recognizer followed all possibilities of inserting `NOP` instructions. This automaton is used for the first and second instruction scheduling in Gcc. The second automaton is deterministic. It is used to bundle instructions on the final phase of Gcc. Bundling instructions is to insert `NOP`s and *template selectors*. Inserting `NOP`s was a dynamic programming algorithm which tests all alternatives in inserting `NOP`s before the instructions and choses the best ones. It uses the second automaton and information about new processor cycle start points prepared by the previous instruction scheduling. Templates are defined by querying the functional units of the second automaton.

Such implementation permitted to speed up all Gcc run (with -O2) up to 45% for Itanium. The code has been improved by 2% (see Table 5) for SPECInt2000 benchmark on a 733 Mhz Ita-

Benchmarks	Ratio	Ratio
164.gzip	176	177
175.vpr	192	203
176.gcc	236	235
181.mcf	142	144
186.crafty	248	243
197.parser	168	171
252.eon	149	147
253.perlbmk	201	207
254.gap	163	167
255.vortex	232	233
256.bzip2	182	188
300.twolf	247	265
Est.SPECint2000	191	195

Table 5: Itanium Gcc with non-DFA and DFA pipeline hazard recognizers.

Benchmarks	Ratio	Ratio
164.gzip	345	361
175.vpr	444	454
176.gcc	460	477
181.mcf	252	249
186.crafty	480	497
197.parser	366	368
252.eon	274	273
253.perlbmk	449	463
254.gap	326	331
255.vortex	509	512
256.bzip2	362	376
300.twolf	506	559
Est. SPECint2000	388	399

Table 6: Itanium Gcc with non-DFA pipeline hazard recognizers vs. Itanium2 Gcc with DFA pipeline hazard recognizer.

nium box.

Unfortunately, there is no implementation of Gcc for Itanium2 using a non-DFA pipeline hazard recognizer. Therefore we could only compare Itanium compiler using the non-DFA pipeline hazard recognizer with the Itanium2 compiler using the DFA-pipeline hazard recognizer. The compiler speed up is about 55% for such a comparison. The SPECInt2000 benchmark results of Gcc (with usage -O2) on a 900Mhz Itanium2 box are given in Table 6.

5 The first cycle multipass instruction scheduling

The usage of the fast DFA pipeline hazard recognizer permits to implement instruction scheduling algorithms trying several schedules and choosing the best one. The traditional instruction scheduling algorithms try only one instruction schedule. The schedule is chosen by a fixed set of heuristics. Usually the major heuristic is a heuristic based on the critical path length [Muchnick, Morgan]. This heuristic works fine for classical RISC processors. For super-scalar RISC or VLIW processors, a greedy algorithm [Muchnick] trying to issue the maximal number instructions on each processor cycle might work better.

The first cycle multi-pass instruction scheduling has been designed to integrate the best of the both approaches. The idea of the algorithm is to choose an instruction whose issue can result in the issue of a maximal number of instructions on the current simulated processor cycle. The highest priority instruction should be among these instructions. In other words, the algorithm guarantees that the instruction with the highest priority will be issued on the current cycle (although necessarily not the first in the cycle). On the other hand, it tries to maximize the number of issued instructions on the cycle. The second highest priority instruction might be not issued on the same cycle even if it could be issued with the highest priority instruction. If it happens, the second highest priority instruction will be issued on the next cycle.

```
function MaxIssues (ReadyArray, var ReadyTry,
                    State, var Index) : integer
begin
  if no one instruction can be issued in State
  then return 0; fi

  Best := 0;

  for i := 0 to length (ReadyArray) do
    if not ReadyTry [i] then
      Insn := i-th of ReadList;
      TempState := State;
      if Insn can be issued in TempState then
        change TempState as if Insn were issued;
        ReadyTry [i] = true;
        n := MaxIssues (ReadyArray, TempState,
                        TempIndex);
        if n > 0 || ReadyTry[0]
        then n := n + 1; fi;
        if Best < n then
          Best := n;
          Index := i;
        fi;
        ReadyTry [i] := false;
      fi
    fi
  end
  return Best;
end

function ChooseReady (ReadyArray, State) : Insn
begin
  ReadyTry := array of length (ReadyArray)
              initialized by false;
  if MaxIssues (ReadyArray, ReadyTry,
                State, i) == 0
  then return the first instruction in
       ReadyArray;
  else return i-th instruction in ReadyArray;
  fi
end
```

Figure 5: The first cycle multi-pass instruction scheduling algorithm.

To find the instruction to issue, the algorithm tries permutations of an array of ready instructions sorted by their priorities. The algorithm might try too many permutations. Therefore the speed of the pipeline hazard recognizer is critical. The number of all permutations is $n!$, where n is number of the ready instructions.

This number can be huge and some heuristics are used to limit the processed permutations. The recursive version of the algorithm (without the heuristics) is given in Figure 5.

The algorithm is written on a Pascal/Modula like language. The function `ChooseReady` gets the array of the ready instructions sorted by their priorities and a DFA state reflecting the current and future functional unit reservations and returns a ready instruction which should be issued. The function calls another function `MaxIssues` to find the best instruction. The recursive function `MaxIssues` gets the ready instruction array, the information about already issued ready instructions as a boolean array, and the current DFA state reflecting issuing the ready instructions. The function returns the maximal number of instructions which can be issued in the given conditions and the index of the instruction which should be issued first to achieve this number. The function checks only the instruction sequences which contain the first ready instruction.

How much can the algorithm improve the code? The improvement can be significant especially for VLIW processors. For example, the test twolf from SpecInt2000 has been improved by 12% for an Intel Itanium2 machine. The overall SpecInt2000 has been improved by 2%. It should be mentioned that a modified algorithm, limiting number of the permutations being checked, was used. The modification was necessary to make the algorithm fast (as a small fraction of all the instruction scheduler work time) so as to be practical for use in industrial compilers.

The algorithm tries all the possibilities to improve the schedule in the scope of one processor cycle. It can be generalized to improve code in scope of a basic block. So the algorithm can be considered as an intermediate step in the algorithm making an optimal or close to optimal instruction schedule.

6 Future directions

The pipeline hazard recognizer based on the proposed model of description and its DFA implementation could be developed in the following ways:

- The same approach in the implementation of the old pipeline hazard recognizer could be used for an implementation of the proposed model. It means a slower but more compact pipeline hazard recognizer. Such an implementation could be useful for debugging and for complicated cases when the automata are too big.
- Some optimization algorithms need to define a DFA state before issuing an instruction having a DFA state after issuing the instruction. It is necessary for trace scheduling [Fisher]. It could be useful for VLIW slot assignment (instruction bundling) too when we have a final DFA state at the end of the basic block and we move backward querying functional unit reservations in order to place instructions into VLIW slots. This kind of algorithm requires reversed automata generation [Bala].
- Some algorithms need a union of DFA states. The union of two DFA states is a DFA state which reflects the union of functional unit reservations (in other words, a simultaneous reservation of functional units) from the both DFA states. It is necessary when we need to know the worst case in a joint point of control flow graph. Perfect software pipelining [Allan] and some interblock instruction scheduling are such kind of algorithms.

The union of states is also necessary for the most widely used kind of software pipelining - modulo scheduling [Allan]. To implement modulo scheduling we need to make a union of the state after instruction issue and the states gotten from the it by advancing the simulated processor cycle by `II * n`, where `II` is the initiation interval and `n` is 1, 2, 3 and so on. `N` could be constrained by a value which results in the state designating no functional unit reservations.

Actually, we could implement the union of states as simply a set of the states. But it results in a slower implementation. On the other hand, adding states to an automaton which are the union of all automaton states might result in the generation of a huge automaton. So this approach requires additional research.

The first cycle multipass instruction scheduling tries all possibilities to improve the schedule in the scope of one processor cycle. It can be generalized to improve code in scope of a whole basic block. It means an optimal or close to optimal instruction scheduling. Optimal instruction scheduling is a *NP*-hard task. But we could decrease the number of all considered instruction schedules by heuristics and by using dynamic programming to reuse the results of optimal instruction scheduling of a subsequence of the instructions. The DFA pipeline hazard recognizer would be important part of the optimal instruction scheduling implementation because of its speed.

Regular expressions in the current implementation describes automata forming *direct acyclic graphs* (DAGs). It is not an adequate model to accurately describe *out-of-order speculative execution* processors. Usually they have register renaming buffers, retire queues and so on. *Generic regular expressions* or *context free grammars* could be an accurate description model for such processors. The single question is "is it worth implementing?" From my point of view, such an accurate description will not give significant improvement of instruction scheduling for the processors. But it could be a good research work.

7 Acknowledgments

I would like to thank Robert Morgan and Norman Rubin. Communication with them gave me my original interest in automaton based pipeline hazard recognizers.

I am grateful to my current and former colleagues at RedHat for their interest in and support of this project. Among them are Richard Henderson who had the biggest impact on the design of the description model, Jeff Law who provided resources for this work through different contracts, Jason Eckhardt for the exchange of interesting ideas in this field, David Miller who wrote the UltraSparc description and had proven the advantages of the approach.

I should name many contributors to Gcc who have affected this work. The full list could be very long. So I only name Jim Wilson at RedHat, Jan Hubicka at SUSE, David Edelhson at IBM, Geoffrey Keating at Apple, Naveen Sharma at HCL Technologies, Dan Nicolaescu at University of California, Irvine. This is the power of the open source community!

Last but not least, I would like to thank my son, Serguei, for the help in editing the article.

References

[Bala] V. Bala and N. Rubin, *Efficient Instruction Scheduling Using Finite State Automata*, International Journal of Parallel Programming (1995).

[Proebsting] T. Proebsting and C. Fraser, *Detecting pipeline structural hazards quickly*, Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1994) p. 280–286.

[Gcc] *A Gcc Manual*, Published by the Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 02111–1307 USA.

[EEMBC] EEMBC, `http://www.eembc.org`

[Slalom] Slalom, `http://www.scl.ameslab.gov/Publications/SLALOM/FirstScalable.html`

[Allan] V. Allan and others, *Software pipelining*, Computing Survey, Sept. (1995).

[Muchnick] Steven S. Muchnick, *Advanced compiler design implementation*, Academic Press (1995), ISBN 1–55860–320–4.

[Morgan] Robert Morgan, *Building an Optimizing Compiler*, Digital Press, ISBN 1–55558–179–X.

[Fisher] J. A. Fisher, *Trace scheduling: A technique for global microcode compaction*, IEEE Trans. Computing 30 (1981) p. 478–490.

Design and Implementation of a Graph Coloring Register Allocator for GCC

Michael Matz
SuSE Linux AG
`matz@suse.de`

Abstract

Historically the register allocator used in GCC is a two phase allocator differentiating between local and global pseudo registers, which doesn't itself produce spill code, and therefore is limited in code quality if spilling is needed. This paper describes a new register allocator for GCC based on graph coloring. After a short overview of the concepts of them in general, including some of the improvements (if used in the implementation) we discuss the actual implementation of the allocator including design decisions and justification for them. This includes parts which aren't explained in the usual scientific papers but needed in a real world multi-target allocator.

1 Introduction

While compiling a program often the need arises to have a place wherein to store certain values. One example is the storage for the result of calculating a common subexpression. To actually make use of it in the later occurance it must be remembered somewhere. One possibility would be memory, but as the fastest storage for most real machines are CPU registers, those are the more natural choice. But the CPU registers (also hardware registers, or hardregs) are limited to a comparatively (to the amount of available memory) small set, which makes it unlikely to actually find a hardreg which doesn't yet hold a value.

The traditional solution is the use of pseudo registers (pseudoregs). While generating code for the program (if for initial generation or optimization doesn't matter) the compiler assumes there is an unlimited set of registers, and if it needs a new one it simply creates it. Now we obviously have to create another pass in the compiler (which has to be fairly late in the translating process), which creates a mapping from pseudoregs to hardregs. It is called register allocation for obvious reasons. This mapping must be injective if constrained to all occurring set of pseudoregs which are live at the same time (so that each hardreg only contains the value for one pseudoreg at a time), which means, that it doesn't necessarily exist trivially. In that case the register allocator needs to change the intermediate code to make use of storage in RAM to hold some of the pseudo registers at least during a part of their life time, which we call to spill a pseudoreg to RAM.

1.1 Current Situation in GCC

The traditional implementation in GCC consists of two passes:

- The first one allocates hardregs to pseudoregs which are only defined and used in one basic block (called local-alloc). This constraint makes the creation of the live range for those pseudoregs trivial (it consists of the start and end point of it, which corresponds to the first def and last use in that block), limits the set of pseudo regs to deal with to those which also are used in that block, and leads to efficient algorithms of creating the mapping to hardregs.

- The second (global-alloc) deals with the other pseudoregs, which are defined and used in different basic blocks. Their live range can span multiple blocks, and most often can not be described simply by their borders. This pass allocates hardregs to those pseudos (it also maintains a conflict graph), constrained to the already done allocation for local pseudos. It also can override decisions of local-alloc if it sees fit.

Both of these passes don't change the code. Instead they simply produce a mapping (in `reg_renumber[]`) which simply doesn't contain a hardreg for a pseudo for which it wasn't able to find one. Then follows a pass called `reload`, which uses this mapping to change the instructions accordingly. Pseudos without hardreg get a place on the stack, and the instructions are modified to refer to their memory location. While doing this `reload` also performs a validity check against constraints from the machine description. If this check fails, the operands which were failing are "reloaded" to make them valid (hence the name of that pass). This for instance then also includes creating explicit load and store instructions for those pseudos which have only stack storage, if the insns which used them can't deal with memory operands. That is, the process of spilling pseudos is implicit in forcing instructions to be valid.

Those reload instructions themselves also need register resources. If the reload was caused by a stack reference, there is a high possibility that it was storage for a pseudo which didn't get a hardreg, which further means that it's also probable that there isn't any free hardreg. So `reload` needs to deallocate some of the currently live pseudos in order to free up some hardregs. For instance consider this instruction:

$$p1 \leftarrow p1 + p2$$

Suppose $p1$ and $p2$ didn't get a hardregs, and the add instruction doesn't accept memory operands. Furthermore suppose that there are no hardregs free during that instruction. Now `reload` conceptually creates this instruction internally

$$[sp + 4] \leftarrow [sp + 4] + [sp + 8]$$

notices that it is invalid and creates reload insns for the memory operands. sp here means obviously the stack pointer and $[adr]$ means the memory at address adr. The add instruction here requires registers as operands, so we need to use some, say $h1$ and $h2$. The code now looks conceptually like:

$$\begin{aligned} h1 &\leftarrow [sp + 4] \\ h2 &\leftarrow [sp + 8] \\ h1 &\leftarrow h1 + h2 \\ [sp + 4] &\leftarrow h1 \end{aligned}$$

So we need to deallocate all pseudos live during this insn which formerly used $h1$ or $h2$. This in turn means that some pseudos now get stack storage instead of a hardreg, therefore the process of reload needs to be repeated until it stabilizes (during which more and more pseudos which initially got a hardreg could be spilled again). In an optimizing compilation `reload` actually calls back into global-alloc right before repeating reloading, in the hope, that some of the newly spilled pseudos could get a different hardreg instead of none at all.

That the emission of spill code is external to the register allocator itself, and that it is done on a per instruction basis leads to non-optimal spill code in some situations. This (and curiosity ;-) lead to the implementation of a more traditional graph coloring register allocator for `GCC`.

2 Graph Coloring Register Allocators

This section describes graph coloring register allocators in general and introduces some improvements to the naive first versions.

2.1 A First Version

As explained above the problem to which we seek a solution is to find a mapping from a set of pseudoregs into a set of hardregs under the constraint that pseudos simultaneously live must not be mapped to the same hardreg. Or more abstractly the constraint is, that certain pairs of pseudos may not get the same hardreg (for whatever reasons). Such pseudos are called to be in conflict. The set of conflicts forms a relation over the pseudos, which is symmetric and irreflexive. The visualization of a set together with such a relation is simply an undirected graph without loops. The nodes represent the pseudoregs, the edges the conflicts, the graph is called conflict graph. In the context of register allocation we talk about webs, instead of nodes.

Now the problem is to assign each node a hardreg such that no neighbor of the node has the same hardreg. This is exactly the formulation of the graph coloring problem (with hardregs being our colors), which explains the name for the class of register allocators working under this model.

Note that pseudos not only conflict with other pseudos, but also with hardregs. The reasons can be that due to machine constraints some hardregs are already used in the intermediate representation before register allocation. Or some pseudos only are permitted a certain set of hardregs (which can be modeled by making them conflict with the inverse set). To make this fit into our model we also include a node for each hardreg into the graph, which already are assigned a color; they all conflict with each other.

Now it's well known that graph coloring is NP-complete, so a full solution isn't feasible for a compiler. We have to implement approximate solutions with better runtime behavior.

The first thing is to make use of Kempe's observation (see [Kempe]), namely that nodes with fewer than N neighbors (where N is the number of available colors) can be trivially colored. We can remove such nodes from consideration, which in turn might make other nodes have fewer than N neighbors. The removed nodes are remembered on a stack. The process of pruning the graph in this way is called **simplify**. If we managed to empty the whole graph in this way we can take one node at a time from the top of stack, put it back into the graph and trivially color it (it's guaranteed to have less than N neighbors).

There are two reasons why simplifying the graph might not completely empty it. First it's only a heuristic, and second the graph itself might not be colorable with N colors at all from the beginning. Either way we might end up with an intermediate graph in which all nodes have N or more neighbors (those nodes are called constrained).

To make it simplify-able again we have to change portions of the conflict graph. This is

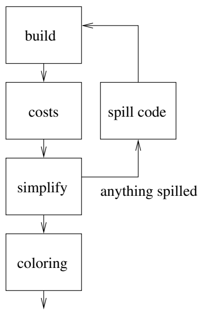

Figure 1: Flow graph of register allocators

done by choosing one of the nodes, the one with the lowest spill cost, remembering it for spilling, and remove it from the graph, in much the same way as if it were trivially colorable. Somewhen this makes other nodes simplifyable again, and in this manner we continue until the graph is empty. If there were spilled node we now add spill code, and repeat the whole allocation process. The next time the conflict graph will be simpler, as all spilled nodes are now split into several nodes, whose conflicts is only a subset of the original ones.

This leads to an allocator like in Figure 1. The **build** phase analyzes the intermediate representation of the program and creates the conflict graph. For choosing which nodes to spill if the need arises, we have to associate a cost for spilling to each node, so we can select the cheapest. Those are calculated by **costs**. The **spill code** phase is only entered if **simplify** had to remove some nodes by marking them as spilled. Otherwise all nodes were simplifyable, and **coloring** is entered, which pops the stack of simplified nodes and colors each one individually. The simplest (and fasted) mean to add spill code is to spill at each reference to a spilled node. Before each use insert a load from, and after each def[1] insert a store to the memory place allocated for the spilled pseudo. See [Cha81], although this includes also a coalescing phase.

[1]definition

2.2 Improvements

There are various improvements to the above simple allocator. Namely in how it deals with copy instructions, in the process of coloring the graph itself, and how spill code is emitted. I'll only describe those which are actually implemented in `GCC`.

After initially building the conflict graph, addition of code often changes it only locally. Therefore it is not necessary to completely rebuild the graph for each colorization round. Instead we **rebuild** the conflict graph incrementally, which is much faster, especially if only few pseudos were spilled.

Coloring and Copies

Copy instructions ensure that the two involved pseudo regs get the same value. Hence they are not a cause for a conflict between those two. To the contrary: if they don't conflict because of other reasons, it even is worthwhile to assign them the same hardreg, as by doing that the copy instruction itself becomes redundant. For instance in a situation like this:

$$\begin{aligned} p1 &\leftarrow \ldots \\ p2 &\leftarrow p1 \\ p3 &\leftarrow p4 + p1 \\ p5 &\leftarrow p4 + p2 \end{aligned}$$

Suppose that $p4$ is defined earlier. Normally $p1$, $p2$, $p3$ and $p4$ all conflict (except $p3$ and $p1$). But the definition of $p2$ is a copy from $p1$, and there are no other defines for it. So $p1$ and $p2$ don't conflict. Furthermore if we could ensure that both get the same color, $p4$ would only conflict with two instead of three nodes.

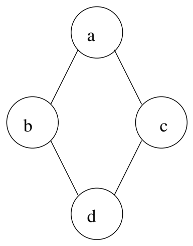

Figure 2: Diamond graph

aggressive coalescing: After building the conflict graph, but before measuring the costs we first try to merge all nodes for pseudos which are involved in one move. Merging them ensures, that they will get the same color. It can only be done if the nodes do not conflict. The resulting conflicts of the merged node are obviously the union of the individual conflicts. As merging nodes may prevent other nodes from being conflict free, nodes associated by the most costly moves should be handled first.

To see a problem in the coloring process look at the graph in Figure 2 and suppose there are only two colors. Here the **simplify** phase doesn't find any node having fewer than N neighbors, and ergo selects one for spilling (the rest is then simplified). Now there is definitely spill code added. But there's a trivial coloring, namely when nodes a and d, resp. b and c get the same color. But we can't know if this holds, until we actually color the nodes, which is only begun when we anyway know, that we succeeded. That is, the decision to spill a node is done too early, which leads us to (see [Briggs94]):

optimistic coloring: Instead of marking a node for spilling in **simplify** we simply also put such nodes on the stack (they are conceptually potentially spilled). No matter if there are such nodes or not, we go to the **coloring** phase. This one works as usual for the stack of nodes. If it colors a simplified node it still is guaranteed to get a color. And if it encounters a potentially spilled node it also tries to find a free color. If it succeeds, good, if not, only then is it actually marked for spilling. It often succeeds, namely in the case, where all the ($\geq N$) neighbors do not need all the N colors at the same time (i.e. some of them are colored equal).

The above mentioned coalescing, which is called aggressive because it tries to coalesce all copies, sometimes results in a much more constrained graph than without coalescing. When nodes are merged whose conflicts are nearly disjoint the resulting node will have much more conflict than the nodes individually. Possibly more than N, which makes it a potential spill candidate instead of a trivially colorable one. It can even make it definitely spill, where without coalescing the individual nodes would not have been spilled (at the expense of leaving a copy instruction around). A solution for this is (see also [GA96]):

iterated coalescing: Two pseudo nodes are only coalesced, if the resulting set of conflicts is smaller than N elements (this is conservative coalescing), and a pseudo to a hardreg node is only coalesced if all conflicts of the pseudo will be colored, or conflict already with the hardreg. This ensures that the graph doesn't become more constrained due to coalescing than it was. To not miss to coalesce too many copies coalescing is tried repeatedly between simplifying and choosing potential spill candidates. There are quite many work lists for nodes and moves, and the exact circumstances when they change their state are a bit involved, so interested readers are referred to the paper, as this is not anymore the method of choice in my implementation.

The method of iterated coalescing still is a bit too conservative. It effectively ensures that the graph remains at least as colorable after coalescing, but misses the positive effect which coalescing can sometimes have one coalescing.

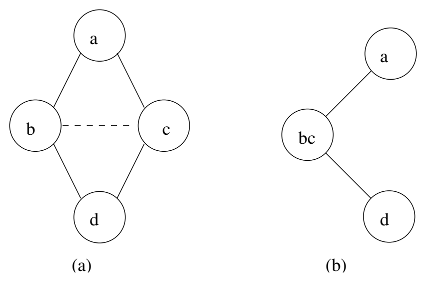

Figure 3: Diamond graph with b and c connected by a move

For instance referring to figure 3(a) if nodes b and c were coalesced the resulting graph (in 3(b)) is trivially colorable without any potential spill. But b and c wouldn't be coalesced under any conservative scheme (when N is two). In general it holds, that if two nodes are coalesced, those nodes which conflict with both have one conflict less after merging. This is the positive impact.

The problem that iterated coalescing (and conservative coalescing) are trying to solve is to prohibit coalesced nodes from becoming spilled. They do this by limiting merging before the fact, but that isn't necessary. It would be better to only act if the danger of spilling a merged node has become real (see [Park]):

optimistic coalescing: All moves are aggressively coalesced before **costs**. Then the normal **simplify** and **coloring** phases are run. When a node which is a merged node now definitely gets no color (i.e. would be spilled) we first split the merged nodes into its ingredients again, and try to color them individually. All parts which still need spilling are spilled. From the parts which get a color only the most costly will be colored right away, the other parts are put under the stack (so they are tried to be colored after all the other nodes), as the building of the color stack expected to only color one node. This splitting of the merge is simply an undo of the merge operation, i.e. all conflicts again point to their initial nodes. Conceptually instead of spilling the node we actually have split it. But compared to general splitting we know already good split points (namely the original copy instructions) and don't even need to insert them.

The example of figure 3 shows, that it sometimes is good for colorability if nodes are merged. It isn't necessary that there actually is a copy instruction. This idea is used by:

extended coalescing: After aggressive coalescing we also try to merge other nodes if it looks feasible. The candidate pairs are those, whose one pseudoreg is target and the other is source in the same instruction, and which do not conflict. Being mentioned in the same instruction makes it probable that the two sets of conflicts have many elements in common, so the merged node will not have that many more conflicts. If we then are unlucky and can't color it we unmerge the nodes again and go on.

Shrinking the Spilled Set

One of the parameters which influences the outcome of our graph colorizer in any way is the heuristic for choosing the next potential spill candidate among a set of remaining nodes (which are all constrained) (the other parameter is which color to choose for a node among those which are still free). The heuristic best for one graph may be bad for another one.

To become a bit more independent from that heuristic Bernstein et.al. ([Bernstein]) proposed a **best-of-three** strategy. For a set of heuristics the graph is colored each time from the beginning with one heuristic, and the overall cost of all spilled nodes is measured. Then finally that colorization with the lowest such cost will be used.

The other parameter is the choice of color among the free ones for a certain node. The usual choices are first-fit and rotating. Another more complex one is **biased coloring** ([Briggs92]): the number of choices depends on how many colors are used by the neighbors, so one goal for coloring a node would be to not unnecessarily enlarge this set for the still uncolored neighbors of it. For that we look at the colored neighbors of all out yet uncolored neighbors. Those colors are anyway already unavailable to them, so would be a good choice for us.

Cheap Spill Code

The improvements up to now had the goal to make the set of spilled nodes as small as possible. The next few items deal with emitting the cheapest spill code once this set is fixed after one colorization round.

First notice that some pseudo regs contain constants (also values loaded from argument stack slots count as constants for this purpose), or values which are provably constant over the lifetime of that pseudo. This make spilling them easy ([Cha81, Briggs92, Briggs94]):

rematerialization: Such pseudos are called rematerializable as the expression calculating their value at each point during their lifetime is known, and hence, once they are overwritten could easily be "rematerialized." To spill such nodes instead of inserting load from stack instructions, one inserts the rematerialization instructions (depending on the value, for instance load with a constant). Stores are not needed for these nodes (as we know their value). Rematerializing a node is worthy if it's cheaper to create these value-load instructions than the mem-loads and mem-stores. More advanced methods of rematerialization also detect expressions over other pseudoregs, like in this example:

$$p3 \leftarrow p1 + p2$$
... code not changing $p1$ or $p2$
$$p4 \leftarrow p1 + p3$$
$$p4 \leftarrow p4 + p2$$

If $p3$ is spilled and $p1$ and $p2$ are not, and an add instruction on registers is cheaper than a load from memory, then we can instead recompute the value of $p3$ before its use. If we operate on SSA form the required analyzation to prove that $p1$ and $p2$ are not changed during the lifetime of $p3$ are relatively easy. Before actually doing such rematerialization it needs also to be ensured that the lifetime of the operands are not extended, i.e. that all operands are live during the lifetime of the spilled node.

Now we look at this code:

code defining $p1$ and $p2$
$$use \leftarrow p1$$
$$use \leftarrow p2$$
$$use \leftarrow p1$$
no further use of $p2$

Suppose that $p1$ is spilled and before the first use up to its definition are no instruction in which a pseudoreg dies. Naively there would be a load instruction added before each use of $p1$. But adding it before the first use doesn't help colorability at all. As there are no deaths between that use and the def the number of used hardregs remains constant there. Inserting a load is not going to help colorability of $p1$.

Therefore we only **spill at deaths**, we only insert loads if we encounter a death of another non-spilled pseudo. For inserting loads we walk backwards the instruction stream, note which nodes need a load, and emit all loads as soon as we reach a def (or the basic block border).

Of course we don't emit the loads directly after the death, but instead right before the instruction which most recently used the spilled pseudos. Otherwise we could end up with code like:

$$\begin{aligned} p1 &\leftarrow p1 + p2 \\ p3 &\leftarrow [sp + 4] \\ p4 &\leftarrow [sp + 8] \\ p5 &\leftarrow [sp + 12] \\ p1 &\leftarrow p1 + p3 \\ p1 &\leftarrow p1 + p4 \\ p1 &\leftarrow p1 + p5 \end{aligned}$$

$p3$, $p4$ and $p5$ were spilled, and $p2$ died at the first shown instruction, where we also inserted all loads together. So the register pressure at the second add instruction is still four. The correct position of the load is right before their uses, but actually emitting them is only triggered by encountering a death, which then leads to this code:

$$\begin{aligned} p1 &\leftarrow p1 + p2 \\ p3 &\leftarrow [sp + 4] \\ p1 &\leftarrow p1 + p3 \\ p4 &\leftarrow [sp + 8] \\ p1 &\leftarrow p1 + p4 \\ p5 &\leftarrow [sp + 12] \\ p1 &\leftarrow p1 + p5 \end{aligned}$$

The next improvement is **interference region spilling** ([Bergner]): if we don't find a color for a node (i.e. it's spilled) we up to now totally removed that node from the graph (by placing it into memory everywhere except for very small ranges around the instructions which needed it). But we also could simply assign any hardreg to this node, and only *remove the edges* to any now really conflicting neighbor.

Practically this is done by choosing a color for all spilled nodes. While emitting the spill loads we also track all hardregs which are currently in use. Remember that we walk backwards. If we encounter a use of a spilled web whose

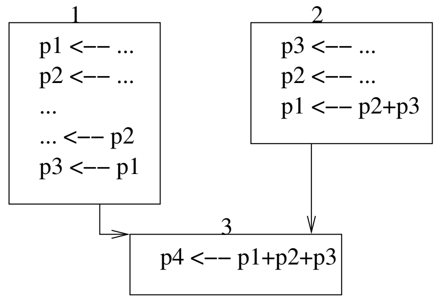

Figure 4: Example for interference region spilling

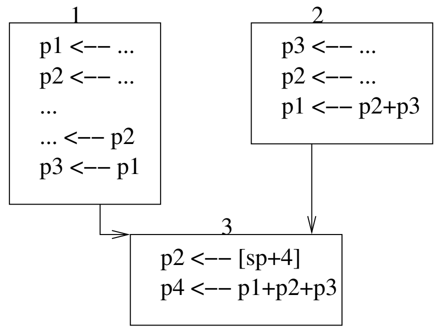

Figure 5: Example for interference region spilling (after inserting loads)

color is in use, we deal with it like described above (i.e. waiting for a death and then inserting a load before the using instruction). If on the other hand its color is currently *not* in use we mark it specially as potentially needing a load. If we go further up and notice a definition for a node marked in this way, and its color didn't become used meanwhile, we simply remove that mark (it's not live before the definition anyway). If we encounter the use of another (non-spilled) node we set its color as used. If we currently have some potential load candidates, whose color now is used, we emit loads for those. This process effectively only adds spill instructions if there is danger that two nodes with the same color are live at the same time.

To better see the effect look at figure 4 for an example situation. $p1$, $p2$ and $p3$ all conflict, and we only have two hardregs. $p1$ got hardregs 0, $p3$ hardreg 1 and $p2$ was spilled. We choose hardreg 1 for it. We begin with the use in block 3, both colors are needed ($p1$ and $p3$ are used), so we need to insert a load (we reached the block border). Then we analyze block 1. Initially hardreg 1 is not used ($p3$ is not live), so we only mark $p2$ as potentially needing a load. While we go upward hardreg 1 doesn't become used, but we encounter a def of $p2$. So we simply forget about it. In block 2 hardreg 1 is used during lifetime of $p2$, but we don't encounter a death until the def, so no load is added here. We end up with the code in figure 5.

If we had spilled by the former method we also had inserted a load into block 1 (if there is any death in the "…"). With interference region spilling we need to insert stores for each definition which reaches one of the uses for which a load was added. In the above case after both defs.

To further reduce cost of spill code we also do **web splitting** ([Mass]). If we can't find a color for a web, i.e. we are going to split it, we first try if we can split this web around other webs, or other webs around that one, in a cheaper way than splitting. Look at this code:

$p1 \leftarrow ...$
$p2 \leftarrow ...$
... code1 without using $p1$
$... \leftarrow p2$
... code2 without using $p1$
$... \leftarrow p2$
... code3
$... \leftarrow p1$

Suppose $p2$ is spilled (there are other uses of $p1$ which makes it more costly to spill $p1$ than $p2$) and $p1$ already colored. Now instead of doing that, we notice that during the lifetime of $p2$ there are no references to $p1$ (which requires something like a containment graph, which can also be used to implement the conflict graph). This makes it possible to completely split $p1$ around $p2$, so that it isn't anymore live during $p2$. This even guarantees, that the number of conflicts for $p2$ reduces, something which normal spilling can't do generally. The result would then look like:

$p1 \leftarrow ...$
$[sp + 4] \leftarrow p1$
$p2 \leftarrow ...$
... code1 and code2
$... \leftarrow p2$
$p1 \leftarrow [sp + 4]$
... code3
$... \leftarrow p1$

Note that the load of $p1$ is not for the later use of it (like in spilling), but rather because the lifetime of $p2$ ended. That is, generally stores for split webs are created before each def of webs around which they are split. Loads for them are created right after each death of the split around webs. A web can also die over a certain edge, not only explicitely at a use.

One minor improvement is **spill coalescing** ([GLnew]): It can happen, that there are uncoalesced copy instructions remaining, where both pseudos of the copy insn are spilled, but do not conflict. This would create a memory-memory move which often is less than desirable. Therefore we can run another aggressive coalescing pass for just the spilled webs in order to remove such copies. This also reduces the needed frame size a bit.

Another situation which sometimes arises is helped by **spill propagation**: There are three pseudos, $p1$ connected to $p2$ by a copy and $p2$ connected to $p3$ by a copy. They don't conflict, like here:

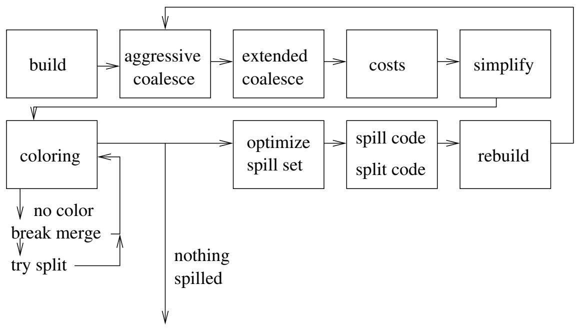

Figure 6: Flow graph of the final allocator

$$\begin{array}{c}\text{... with def of } p1\\ p2 \leftarrow p1\\ \text{...}\\ p3 \leftarrow p2\\ \text{... with use of } p3\end{array}$$

Now suppose, that $p1$ and $p3$ were spilled, but $p2$ colored. It might be better to also spill $p2$ to memory, if then the two copy instructions can be removed by coalescing all three pseudos together. In a sense this "propagates" the spill to a colored pseudo (which initially is counter intuitive).

The last improvement here is **spill coloring**: After the set of spilled webs is finalized a coloring pass is run on the subgraph induced by the spilled nodes, with an unlimited number of colors (i.e. when a node doesn't get a color, the maximum number is simply incremented and it gets the new color). Then a stack slot is allocated for each such color, instead of for each spilled node. This greatly reduces the needed stack frame size for spilling.

The final flow graph of the register allocator can be seen in figure 6.

3 Taking it to `GCC`

Now we look how to fit all the above descriptions into the framework of `GCC`. The definite reference is of course the source code (in files `ra*.c, ra.h` and `pre-reload.*`), and to not obsolete the paper as soon as some parts of the allocator are changed we don't follow the source too closely here.

3.1 Constraints Imposed by `GCC`

Classes and Constraints

`GCC` not only targets an ideal machine with a set of N completely equivalent registers, whose instruction set is totally orthogonal, which doesn't expect certain conditions from the operands of instructions, but instead it targets real machines with sometimes awkward constraints. The ones which influence the register allocator are described here.

`GCC` has the concept of **register classes**: The set of all hardware registers for a machine is divided into named smaller set of registers (`ALL_REGS` for the whole set and `NONE_REGS` for the empty set are defined by all machines). They are not disjoint. The register operands of instructions can specify which hard registers they accept by mentioning such register classes.

The instruction templates for a machine can specify **constraints** and can consist of more than one alternative per template. Each of the instructions in the intermediate representation match one template in the machine description. For register allocation purposes each template has many alternatives, where each of them can have a different set of requirements on the operands. For instance it's possible that the generic "add" template has two alternatives, one accepting registers of class `CLASS1` and the other accepting registers of class `CLASS2`.

The are also other types of constraints, for instance to limit the range of constant operands (so as to fit into an immediate field in the instruction), or matching constraints, which force one operand to be equal to another. In that way two address machines can be implemented. For instance the generic "add" pattern has three operands (one target, and two sources), and if the machine has only an instruction which adds a register to the other source register, this can be specified by constraining the first source operand to be the same as the target operand.

Such matching constraints can easily be made valid by a pass before register allocation, by adding copy instructions for the matching operands (possibly using new pseudo registers). Similar with constraints which don't affect register operands (constants e.g.).

Additionally some hardregs are not available for register allocation at all, as they have special uses (e.g. the static chain pointer for nested functions, or the PIC register on some machines).

The machine descriptions also have the possibility to limit the set of hardregs for a pseudoreg just based on its mode (the `HARD_REGNO_MODE_OK` macro).

Subregs and Wide Regs

Another possibility in `GCC` is the use of **subregs**. Subregs are references to a part of a register (or other values, but in those we aren't interested). This makes such code possible:

$$\begin{gathered} p1:[SI+0] \leftarrow p2 \\ p1:[SI+4] \leftarrow p3+p2 \\ p4 \leftarrow p1+p5 \end{gathered}$$

Here the notation $p1:[SI+x]$ means the subreg of $p1$ of mode `SImode` on byte offset x inside $p1$. Suppose that $p1$ is a `DImode` pseudo. The code does define $p1$ piecewise (first the lower half, then the high half), and then uses the $p1$ in its whole. The interpretation of subregs is bitwise.

A special kind of subregs are **paradoxical** subregs. Those are subregs in a wider mode than the inside register provides. I.e. it accesses bits which aren't provided (or are undefined).

Furthermore not all machines allow subregs to be taken from all hardware registers. For instance on Alpha the floating point register can hold 64-bit integers. But it's not possible to access the low or high 32 bit of that value by simply looking at the low or high 32 bit of the register. Therefore some registers are not allowed for references which involve a subreg reference.

And finally there is the notion of **multi word** hardregs. Those are references to hardregs in a mode which is wider than this hardreg. Such references implicitly use the next few adjacent hardregs (as much as needed). For instance a `DImode` reference to hardreg 0 (which for this example shall be SImode maximum) also uses hardreg 1.

3.2 Meaning for the Allocator

The constraints on registers result in a set of allowed hardregs for each register reference. The set of allowed hardregs for a whole web consists of the intersection of the sets for all individual references making up that web.

It's possible that one web consists of references with conflicting constraints, i.e. with disjoint allowed hardregs. For instance a pseudo register used in integer (e.g. bitwise logic) and floating point context (e.g. addition with a float constant). Such a web would have an empty set of possible hardregs. A possible solution is to either fake this set (by ignoring the conflicting reference), and thereby leave the work of fixing up the instructions to `reload`, or to spill away the conflicting reference while building the web.

As probably already became clear, each web has its own set of allowed hardregs (in the `usable_regs` member of `struct web`). Most often it will not contain all hardregs. This has implications for the predicate `is-trivially-colorable` used in the **simplify** phase. The number N is meaningless here. Instead a web is trivially colorable if the weight of its conflicts is less than the number of registers in `usable_regs`. For that to work there may only be conflict edges between webs whose possible hardregs have a non empty intersection. This of course makes sense, as if it were empty the two webs anyway couldn't get the same color, so conflicts between them are pointless. This also reduces the necessary conflict edges.

One difficulty are multi-word pseudos. The webs have an `add_hardregs` member which contains the number of additionally required hardregs (at maximum). To generally ensure that there is a hole of two consecutive hardregs in a block of N, it would be required that there are less than $N/2$ neighbors (which itself wouldn't be allowed to use multiple regs). If we had exactly $N/2$ conflicts all even colors could be taken, leaving no block of size two. But this is clearly an overly conservative heuristic.

Instead the `add_hardregs` member simply is counted as another conflict. So the actual predicate is:

$$trivial_i := |usable_i| > add_i + \sum_{n \in neighbors_i} 1 + add_n$$

This is an optimistic predicate, which means that even webs which were simplified could not get a color (only when they are multi word regs).

The possibility of subregs means for us, that a pseudo may sometimes be live only partially (this is a cause of much of the complexity in the actual implementation). This can also result in partial conflicts, i.e. something like "the lower 32bit of $p1$ conflicts with $p2$." Such conflicts are useful to create a good allocation for multi word pseudos, as now partial overlap is allowed (so that for instance only three hardregs are needed for two pseudos each needing two regs). Partial webs are instances of the normal `struct web` but they have their `parent_web` member set. The `subreg_next` members form a linked list between the whole web and its parts.

3.3 The Conflict Graph

The most important structure in a graph coloring register allocator is obviously the conflict graph but up to now we haven't talked about it, because conceptually it's not very interesting in the context of describing the general methods of register allocation.

It is implemented in GCC by these structures:

```
struct conflict_link
{
  struct conflict_link *next;
  struct web *t;
  struct sub_conflict *sub;
};
struct sub_conflict
{
  struct sub_conflict *next;
  struct web *s;
  struct web *t;
};
struct web
{
  ...
  struct conflict_link *conflict_list;
  ...
};
sbitmap igraph;
sbitmap sup_igraph;
```

That is, each web has a linked list of its conflicts. Only whole webs have this list, subwebs (those corresponding to subregs) don't. The targets of those conflicts (in `conflict_link.t`) are also whole webs. This allows fast iteration over all conflicts without having to care for the details of sub-conflicts. If between web a and b only sub-conflicts occur, then those are remembered in a second linked list, which hangs off of the edge between a and b. I.e. there is one `conflict_link` instance in as conflict list, with `.t` being b, which has its `.sub` member pointing to a list of sub-conflicts which note which parts of a resp. b exactly are conflicting (`.s` points to a part of a or a itself, `.t` to b or a part of it). The bitmaps `igraph` and `sup_igraph` are used to test two webs for conflicts. `igraph` contains the exact conflicts between parts, `sup_igraph` lists for whole webs, if they them-self or any parts of them conflict. This is a bit suboptimal. If we had a mean to go from the indexes of two webs to the corresponding `conflict_link` instance for their connecting edge (a hash table for instance) we wouldn't need `sup_igraph`. If one considers coalescing (which also involved merging conflicts, which we must be able to break up again) such a mean is not totally trivially implemented, though.

Actually **building** the conflict graph is implemented in `ra-build.c`. We use an incremental graph builder which at the same time does an liveness analysis, builds webs and creates (preliminary) conflicts (it's in `build_web_parts_and_conflicts()` and sub-functions). It works use by use. Per use it goes backward the instruction stream (following all edges backward), until it reaches a def for the register we currently analyze. On that way it remembers the defs encountered for the current use (from those the real conflict lists are build later), connects uses and defs of the same reg as that use in a UNION-FIND structure, and fills some house keeping information (for instance if an edge is crossed the use is remembered as live over it).

The currently analyzed use is placed into an instance of **struct curr_use**. Partial liveness is supported by having a bit field (the `.undefined` member) where each bit corresponds to one byte of the use. A bit is set if the byte is still undefined. When a def is encountered the bits which correspond to that def are cleared. If that results in no more left bits we have reached the first def which (partially) defines the use on that path. The set bits also represent the part of the use, which is still live. This is used for creating sub conflicts. Partial liveness could also be represented by a set of ranges, which bits are live. A variation of that scheme is used in [Bitwidth], although they only split the bits into three sections (a set of leading and trailing dead bits, and a section of middle bits, which are live). To correctly represent live information under this scheme we would need to treat some subreg references as read-modify-write, like it's done in the conservative data flow pass in `GCC`. This makes it less attractive again.

The advantage of such a builder compared to a more traditional bit-set based liveness analyzer is the simplicity (we deal with only one use at a time), that it's possible to precisely track partial liveness for subregs (something which is not that easily done with bitmaps) and that we can easily rebuild the graph for only those uses, which need it. After spilling was done not the whole graph needs to be rebuilt, but only those webs, which were changed, and their former neighbors. A bit-set based analyzer also needs to iterate until the solution stabilizes. This is not needed here. And that we can note conflicts already *while* still building webs also is attractive.

With some optimizations (like skipping whole basic blocks if the current pseudo isn't mentioned in them) the part of building webs and preliminary conflicts actually was nearly as fast as the traditional bitmap based liveness analyzer in `GCC`.

There is a pass necessary which actually creates the `struct web` instances and the conflict lists from the UNION-FIND structure and the preliminary conflicts (which are all based on the defs and uses, for each of whom an instance of `struct web_part` is created. This is done in `make_webs()` and subfunctions.

The rest of initializing the webs is also in `ra-build.c`. Among it are determining the spill cost of a web, if it's rematerializable, collecting the copy instructions and so on. It probably had better been named `ra-analyze.c` ;-)

3.4 Putting it Together

The implementation of the register allocator consists of different files which roughly reflect the structure described in section 2.

Besides `ra-build.c` which builds not only the conflict graph but also most of the other information about webs and program structure (as described above), there is

`ra-colorize.c`
which is all about changing (like in coalescing) and coloring the conflict graph, including optimizations which shorten the set of spilled webs. This includes the work list management. The structure is fairly close to the allocators in the published papers, except for three things:

- **selectable** algorithm: Most of the improvements in the coloring process are selectable at runtime. For instance it can be switched between optimistic or iterated coalescing, or biased coloring can be activated or not.

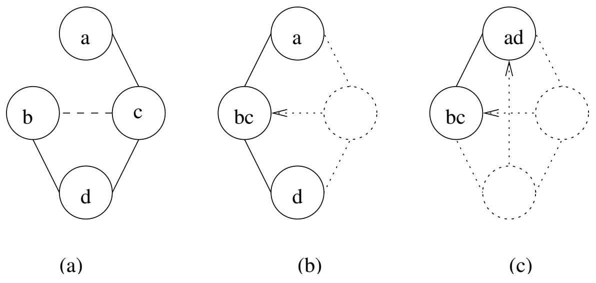

Figure 7: Coalescing of nodes

- **hard trying** to color certain webs: due to irregularities in connection with multi-word pseudos, and with spill temporaries, or other generally difficult webs (which includes those during whose lifetime no death occurs) it's possible that there is no color free for a web which absolutely must have a color (this happens extremely seldom and only on register constrained machines). In that situation it is tried to temporarily mark one of its already colored neighbors as spilled, and try again to find a color. This is done until a color is found or no more colored neighbors are left. After that those temporarily spilled neighbors are tried to be colored again. If they don't get a color they are left in the spilled state.
- **recoloring spills**: after the graph is colorized and the set of spilled webs is determined, each spilled web is tried to be recolored. For this the cost for the spilled web getting a color is measured (it consists of the sum of spill-costs of all neighbors overlapping that color). If the smallest cost is smaller than the web spill cost, this recoloring is done, and the neighbors which now conflict are spilled instead. This can reduce the overall spill cost of the graph.

One particularly ugly problem is how to implement splitting up merged nodes for optimistic coalescing. Refer to figure 7. Starting with graph (a) first nodes b and c are merged, then nodes a and d. The final graph has only one edge left which was not in the original graph. Now suppose we want to split node bc. We may not yet remove edge $b - ad$, because also the original $b - d$ edge was mapped to it. Only when we also split node ad we remove it, and then we also *have* to remove it in order to not constrain the graph more than necessary.

From that description it becomes clear that the only real solution would be to add reference counters to edges. But that would bloat the size of each edge. That's not desirable as there are potentially very many edges in a conflict graph. The reference counter would also only be ever needed for edges which weren't in the original graph, as only those are candidate for removal.

Currently we don't refcount[2] the edges, but instead "repair" the graph after having split nodes. First we remove all edges incident to split nodes which weren't in the original graph (we have an easy way to test that as each node has a list of those), and then we look for other coalesced nodes that would have added that edge also (in which case we reinsert it into the graph). This process is relatively slow, so we will move to a refcounting implementation eventually (the work has already started for that).

`ra-rewrite.c`
is responsible for actually changing the program to include any spill code. Its behavior is also selectable at runtime, and it can use spill-everywhere (separately for uses and defs), traditional spill at deaths or spill at interference regions. It also implements the code for splitting webs around other webs which can theoretically be used with together with all spill methods. Unfortunately interference region spilling and web splitting use separate data structures and can't currently be used together. They will be usable together once the implementation is done.

Besides the improvements from section 2 for reducing the number of inserted loads during spilling, the actual implementation also has a naïve implementation of optimizing dead stores. It goes backward the insn stream remembering to which locations it wrote to. For each encountered use we delete all locations which overlap that use from the list. If it is about to insert a store it first checks if that location is still in the list, and omits the store if it is.

One thing which should be mentioned is that we defer the creation of real stack slots until the very end of allocation. Until then we create new pseudo regs to hold the value of spilled (or split) webs. These pseudos are not to be confused with normal pseudo regs, as they conceptually represent stack slots or real registers. We do this for two reasons:

- We want to be able to track also liveness for stack slots (in order to merge or color them), and sometimes we are able to actually give them back a hard register. This usually happens when multiple rounds of spilling were needed and a spill method which produces long living temporaries was used.

  In that case it happens that first a web is spilled which then didn't relax the situation as much as hoped, so other webs are also spilled. This in turn can make the spilling of the first web unnecessary, and by creating a web also for stack slots we are able to make use of that. Those *stack-pseudos* or *stack-webs* as we call them in the allocator are handled specially in a number of situations. For instance they are colored after all normal webs. If they don't get a color, they are not spilled again

[2]count the number of references of ... ;-)

(this is implemented by coloring them with an impossible color). This also needs changes in the functions which check validity of constraint, so that stack pseudos are accepted for memory references and for registers.

Some machines have requirements on the addresses they accept, for instance a limited range of offsets from a base registers. Emitting an address reference on them can possibly lead to emitting more than one instruction, which actually constructs the address by doing arithmetics on some new pseudo registers. On those machines we can't defer creating stack slots completely, as creating new pseudo regs means we must redo our register allocation. For those machines we actually emit real stack references for all the stack-webs which did not get a color. I.e. we defer stack slots only by one round, not until the very end.

- The other reason is stack slot coloring as described in Section 2 (as "spill coloring"). When we have webs for all stack slots (i.e. for the stack pseudos) including all conflicts, we can color them easily and reduce the frame size. I.e. we allocate stack space not for each stack pseudo, but instead only for each color needed for them.

The rewriting phase is also responsible for resetting the conflict graph and associated information into a state that is usable as a starting point for the next round. For instance all coalescing has to be undone, and the edges added for that have to be removed (as *all* coalescing is undone this is considerable easier than what was described above). We also need to mark which webs have to be rebuilt (namely those which changed their layout).

The file `ra-debug.c` contains some useful functions for debugging the allocator including a new format of outputting the immediate format (RTL) of `GCC`, which is much more compact and easier to read (although it lacks some information) than the traditional lisp like format. It should somewhen be extended to be usable also for the other passes in `GCC`, and be merged with the format of the scheduler debug dumps (which uses something similar).

To actually scan the instruction stream for all (interesting) references to registers we use functions from `df.c`. For each such reference we build one instance of `struct web_part` which creates an indirection in one of the highly used data structures, so it might somewhen be advisable to do this on our own.

The last big part in the allocator is implemented in
`pre-reload.c`.
As written at the very begin the *reload* pass is responsible for actually emitting spill code in the old register allocator, and for fixing up any invalid instructions (those whose operands do not match their constraints). The spilling code we do add ourself now, but we could still produce invalid instructions (for instance operands don't match where they have to, or an operand is in a register which isn't in the required class). This would make reload emit fixup code. As this code is emitted locally without having the big picture of a conflict graph or similar means this often results in spilling some other pseudo registers, and reloads method for adding spill code is undesirable.

Therefore the goal must be to never leave the register allocator with possibly invalid instructions. One requirement is to allocate pseudos to a register which is accepted by all the instructions that reference it. To that end prereload collects the possible register classes for each register reference. Another requirement is to not violate matching constraints, which is done by pre-reload emitting copy instructions before or after the invalid instruction, and change the operands of it to actually match. It also makes sure that constraints which don't involve pseudo regs are fulfilled, like constants be of a certain range, or decomposing multi-level indirect memory access (i.e. the address is a memref[3] itself) if necessary.

The techniques it uses are heavily inspired by reload itself, but as pre-reload works on pseudo regs, the actual implementation can be quite a bit simpler.

The use of pre-reload can not make totally sure, that no invalid instructions are generated. Which register class is acceptable for one operand can depend on which register another operands was put in and this is only known, once allocation finished, so in some situations we have to give up in the allocator and assume something. This means, that reload will still be needed, but only extremely seldom (we once had only about 10 reloads while building cc1 IIRC). This makes me hope that reload can be implemented in a much simpler way than now, for instance by simply emitting fixup instruction as invalid operands are encountered, instead of first collecting all reloads of all instructions. Reload inheritance probably would also not be useful anymore.

Finally `ra.c` holds it all together and contains some initialization functions plus the main loop.

4 Numbers

For comparing the performance we give some numbers of runs of the SPEC2000 performance test suite, with the old allocator and the new one.

Table 8 shows the result on a 1.53 GHz Dual

[3]memory reference

Name	T_{old}	S_{old}	T_{new}	S_{new}
164.gzip	223	627	223	627
175.vpr	420	334	431	324
181.mcf	864	208	874	206
186.crafty	127	787	129	774
197.parser	439	410	438	411
252.eon	170	766	171	759
253.perlbmk	275	654	274	656
254.gap	205	537	201	546
256.bzip2	371	405	359	418
300.twolf	831	361	819	366
168.wupwise	294	544	290	551
171.swim	973	319	1036	299
172.mgrid	481	374	485	371
173.applu	624	337	599	351
177.mesa	233	602	229	610
179.art	1607	162	1664	156
183.equake	327	398	323	403
188.ammp	707	311	721	305
200.sixtrack	334	329	327	337
301.apsi	925	281	963	270

Figure 8: SPEC2000 results for Athlon 1800+

Name	$100 * \frac{(T_{new} - T_{old})}{T_{old}}$
164.gzip	-1.38889 %
175.vpr	-3.7037 %
176.gcc	-0.465116 %
181.mcf	1.82648 %
186.crafty	3.8835 %
197.parser	1.43885 %
252.eon	5.7971 %
253.perlbmk	-3.15315 %
254.gap	-1.17647 %
256.bzip2	-1.5873 %
300.twolf	-5.66616 %
168.wupwise	0.840336 %
171.swim	0.915751 %
172.mgrid	-4.17755 %
173.applu	-1.42518 %
177.mesa	-5.67686 %
179.art	0.555556 %
183.equake	1.0989 %
188.ammp	0.444444 %
200.sixtrack	0.593472 %

Figure 9: Relative SPEC2000 performance on AMD64

Athlon (Athlon 1800+). The T column shows the runtime in seconds (smaller is better), the S column the SPEC score (bigger is better). Note in particular bzip2, twolf and applu, which show some nice improvements. With the tested version of the allocator there were also some quite severe regressions as shown in the table. I've not yet analyzed them in detail.

Table 9 shows the runtime of the SPEC2000 tests compiled with the new register allocator compared with the old one on an AMD64 machine (i.e. with twice as much general purpose registers as x86). As can be seen crafty and eon regress quite much, but the potential of the allocator can be seen in the other results.

5 Acknowledgments

I wish to thank Daniel Berlin who started the `new-regalloc-branch` and created an initial implementation, and Denis Chertykov who created `pre-reload`, for their help in implementation and fruitful discussions.

I also would like to thank SuSE and AMD for letting me work on the register allocator.

And Cafebar 8006 for keeping me awake ;-)

6 Availability

The current development version of the register allocator is available in the `new-regalloc-branch` in `GCC` CVS. See

`http://gcc.gnu.org/cvs.html`

References

[Bergner] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, *Spill Code Minimization via Interference Region Spilling*, Proc. of the 1997 ACM SIGPLAN Conf. on PLDI, pp. 287–295. June 1997

[Bernstein] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter, *Spill code minimization techniques for optimizing compilers*, SIGPLAN Notices, 24(7):258 263, July 1989

[Bitwidth] S. Tallam, R. Gupta, *Bitwidth Aware Global Register Allocation*, TR, Dept. of Computer Science, U. of Arizona, July 2002

[Briggs92] P. Briggs, *Register Allocation via Graph Coloring*, PhD thesis, Rice University, Houston, Texas, April 1992

[Briggs94] P. Briggs, K. D. Cooper, and L. Torczon, *Improvements to graph coloring register allocation*, ACM TOPLAS, Vol 16, No.3, pages 428–455, May 1994

[Cha81] G. J. Chaitin, et. al., *Register allocation via coloring*, Computer Languages, 6:47–57, Jan. 1981

[GA96] L. George and A. Appel, *Iterated Register Coalescing*, ACM Trans. on Prog. Lang. and Systems, 18(3):300–324, May 1996

[GLnew] Allen Leung, Lal George, *A New MLRISC Register Allocator*, `http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/ra.html`

[Kempe] A. B. Kempe, *On the geographical problem of the four colors*, Am. J. Math. 2, 193–200, 1879

[Mass] MSCP group, *The massively scalar compiler project*

[Park] Jinpyo Park and Soo-Mook Moon, *Optimistic register coalescing*, IEEE PACT, pages 196–204, 1998

GENERIC and GIMPLE: A New Tree Representation for Entire Functions

Jason Merrill
Red Hat, Inc.
`jason@redhat.com`

1 Abstract

The tree SSA project requires a tree representation of functions for the optimizers to operate on. There was an existing functions-as-trees representation shared by the C and C++ front ends, and another used by the Java front end, but neither was adequate for use in optimization. In this paper, we will discuss the design of GENERIC, the new language-independent tree representation, and GIMPLE, the reduced subset used during optimization.

2 Introduction

For most of its history, GCC has compiled functions directly to RTL (Register Transfer Language) on a statement-by-statement basis. RTL has been a very useful intermediate language (IL) for low-level optimizations, but has significant limitations that keep it from being very useful for higher level optimizations:

- Its notion of data types is limited to machine words; it has no ability to deal with structures and arrays as a whole.

- It introduces the stack too soon; taking the address of an object forces it into the stack, even if later optimization removes the need for the object to be addressible.

GCC also has another IL: its abstract syntax tree representation. In the past, the compiler would only build up trees for a single statement, and then lower them to RTL before moving on to the next statement. This began to change in GCC 3.0: CodeSourcery, LLC modified the C++ compiler to store entire functions as trees and only lower them to RTL as part of compiling to assembly. As part of the same work, they introduced the first tree-level optimization pass, the inliner. Inlining at the tree level partially addressed the second limitation of RTL mentioned above, since C++ objects passed as arguments to a function are usually passed by address.

The tree SSA project is intented to expand on this by performing a full set of optimizations at the tree level. But to do this, we needed to refine how we use trees to represent whole functions. The result is GIMPLE, and its superset GENERIC.

3 Existing Tree ILs

The C++ compiler work was later extended to work with the C compiler as well, but never became a language-independent tree IL. Initial work on tree-ssa was based on the C front end trees, but they were unsuited for use in optimization.

The main shortcoming of C trees, from an optimization standpoint, is that they are highly context-dependent. Many `_STMT` codes just serve as placeholders for calls to `expand_` functions and rely on the RTL layer to keep track of scoping. For a tree IL to be useful for optimization, things such as the target of a `break` or `continue` statement, or the scope of a C++ cleanup, must be made explicit.

The other preexisting tree IL is the one in the Java front end. Java made an effort to use backend tree codes whenever possible, added a few new tree codes to the backend, and retained a few in the front end. GENERIC is largely based on Java front end trees, adjusted to be entirely language independent.

4 GENERIC

The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees. To this end, it was necessary to add a few new tree codes to the backend, but most everything was already there. If you can say it with the codes in `gcc/tree.def`, it's GENERIC.

Early on, there was a great deal of debate about how to think about statements in a tree IL. In GENERIC, a statement is any expression whose value, if any, is ignored. A statement will always have `TREE_SIDE_EFFECTS` set (or it will be discarded), but a non-statement expression may also have side effects. A `CALL_EXPR`, for instance.

It would be possible for some local optimizations to work on the GENERIC form of a function; indeed, the adapted tree inliner works fine on GENERIC, but the current compiler performs inlining after lowering to GIMPLE.

If necessary, a front end can use some language-dependent tree codes in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE and doesn't expect them to work with any (hypothetical) optimizers that run before the conversion to GIMPLE.

5 GIMPLE

GIMPLE is a simplified subset of GENERIC for use in optimization. The particular subset chosen (and the name) was heavily influenced by the SIMPLE IL used by the McCAT compiler project at McGill University [SIMPLE], though we have made some different choices. For one thing, SIMPLE doesn't support `goto`; a production compiler can't afford that kind of restriction.

GIMPLE retains much of the structure of the parse trees: lexical scopes and control constructs such as loops are represented as containers, rather than markers. However, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values.

Similarly, in GIMPLE no container node is ever used for its value; if a `COND_EXPR` or `BIND_EXPR` has a value, it is stored into a temporary within the controlled blocks, and that temporary is used in place of the container.

The compiler pass which lowers GENERIC to GIMPLE is referred to as the "gimplifier." The gimplifier works recursively, replacing complex statements with sequences of simple statements. Currently, the only way to tell whether or not an expression is in GIMPLE form is by recursively examining it; in the future there will probably be a flag to help avoid redundant work.

6 Interfaces

The tree representation of a function is stored in `DECL_SAVED_TREE`. It is lowered to GIMPLE by a call to `simplify_function_tree`.

If a front end wants to include language-specific tree codes in the tree representation which it provides to the back-end, it must provide a definition of `LANG_HOOKS_SIMPLIFY_EXPR` which knows how to convert the front end trees to GIMPLE. Usually such a hook will involve much of the same code for expanding front end trees to RTL. This function can return fully lowered GIMPLE, or it can return GENERIC trees and let the main gimplifier lower them the rest of the way; this is often simpler.

The C and C++ front ends currently convert directly from front end trees to GIMPLE, and hand that off to the backend rather than first converting to GENERIC. Their gimplifier hooks know about all the `_STMT` nodes and how to convert them to GENERIC forms. I worked for a while on a genericization pass which would run first, but the existence of `STMT_EXPR` meant that in order to convert all of the C statements into GENERIC equivalents would involve walking the entire tree anyway, so it was simpler to reduce all the way. This may change in the future if someone writes an optimization pass which would work better with higher-level trees, but currently the optimizers all expect GIMPLE.

A frontend which wants to use the tree optimizers (and already has some sort of whole-function tree representation) only needs to provide a definition of `LANG_HOOKS_SIMPLIFY_EXPR` and call `simplify_function_tree` and `optimize_function_tree` before they start expanding to RTL. Note that there actually is no real handoff to the tree backend at the moment; in the future there will be a `tree_rest_of_compilation` which will take over, but it hasn't been written yet.

Note that there are still a large number of functions and even files in the gimplifier which use "simplify" instead of "gimplify." This will be corrected before the project is merged into the GCC trunk.

You can tell the compiler to dump a C-like representation of the GIMPLE form with the flag `-fdump-tree-simple`.

7 GIMPLE reference

7.1 Temporaries

When gimplification encounters a subexpression which is too complex, it creates a new temporary variable to hold the value of the subexpression, and adds a new statement to initialize it before the current statement. These special temporaries are known as "expression temporaries," and are allocated using `get_formal_tmp_var`. The compiler tries to always evaluate identical expressions into the same temporary, to simplify elimination of redundant calculations.

We can only use expression temporaries when we know that it will not be reevaluated before its value is used, and that it will not be otherwise modified (these restrictions are derived from those in [Morgan] 4.8). Other temporaries can be allocated using `get_initialized_tmp_var` or `create_tmp_var`.

Currently, an expression like `a = b + 5` is not reduced any further, though in future this may be converted to

```
  T1 = b + 5;
  a = T1;
```

to avoid problems with optimizers trying to refer to variables after they've gone out of scope.

7.2 Expressions

In general, expressions in GIMPLE consist of an operation and the appropriate number of simple operands; these operands must either be a constant or a variable. More complex operands are factored out into temporaries, so that

```
  a = b + c + d
```

becomes

```
  T1 = b + c;
  a = T1 + d;
```

The same rule holds for arguments to a `CALL_EXPR`.

The target of an assignment is usually a variable, but can also be an `INDIRECT_REF` or a compound lvalue as described below.

7.2.1 Compound Expressions

The left-hand side of a C comma expression is simply moved into a separate statement.

7.2.2 Compound Lvalues

Currently compound lvalues involving array and structure field references are not broken down; an expression like `a.b[2] = 42` is not reduced any further (though complex array subscripts are). This restriction is a workaround for limitations in later optimizers; if we were to convert this to

```
  T1 = &a.b;
  T1[2] = 42;
```

alias analysis would not remember that the reference to `T1[2]` came by way of `a.b`, so it would think that the assignment could alias another member of `a`; this broke `struct-alias-1.c`. Future optimizer improvements may make this limitation unnecessary.

7.2.3 Conditional Expressions

A C `?:` expression is converted into an `if` statement with each branch assigning to the same temporary. So,

```
  a = b ? c : d;
```

becomes

```
  if (b)
    T1 = c;
  else
    T1 = d;
  a = T1;
```

Note that in GIMPLE, `if` statements are also represented using `COND_EXPR`, as described below.

7.2.4 Logical Operators

Except when they appear in the condition operand of a `COND_EXPR`, logical 'and' and 'or' operators are simplified as follows: `a = b && c` becomes

```
  T1 = (bool)b;
  if (T1)
    T1 = (bool)c;
  a = T1;
```

Note that `T1` in this example cannot be an expression temporary, because it has two different assignments.

7.3 Statements

Most statements will be assignment statements, represented by MODIFY_EXPR. A CALL_EXPR whose value is ignored can also be a statement. No other C expressions can appear at statement level; a reference to a volatile object is converted into a MODIFY_EXPR.

There are also several varieties of complex statements.

7.3.1 Blocks

Block scopes and the variables they declare in GENERIC and GIMPLE are expressed using the BIND_EXPR code, which in previous versions of GCC was primarily used for the C statement-expression extension.

Variables in a block are collected into BIND_EXPR_VARS in declaration order. Any runtime initialization is moved out of DECL_INITIAL and into a statement in the controlled block. When gimplifying from C or C++, this initialization replaces the DECL_STMT.

Variable-length arrays (VLAs) complicate this process, as their size often refers to variables initialized earlier in the block. To handle this, we currently split the block at that point, and move the VLA into a new, inner BIND_EXPR. This strategy may change in the future.

DECL_SAVED_TREE for a GIMPLE function will always be a BIND_EXPR which contains declarations for the temporary variables used in the function.

A C++ program will usually contain more BIND_EXPRs than there are syntactic blocks in the source code, since several C++ constructs have implicit scopes associated with them. On the other hand, although the C++ front end uses pseudo-scopes to handle cleanups for objects with destructors, these don't translate into the GIMPLE form; multiple declarations at the same level use the same BIND_EXPR.

7.3.2 Statement Sequences

Currently, multiple statements at the same nesting level are connected via COMPOUND_EXPRs. This representation was chosen both because of precedent and because it simplified the implementation of the gimplifier. However, it makes transformations during optimization more complicated, and there is some concern about the memory overhead involved.

The complication is mostly encapsulated by the use of iterators declared in tree-iterator.h. The representation may be extended in the future, perhaps to use statement vectors or a double-chained list, but the iterators should also avoid the need for any changes in the optimizers.

7.3.3 Empty Statements

Whenever possible, statements with no effect are discarded. But if they are nested within another construct which cannot be discarded for some reason, they are instead replaced with an empty statement, generated by build_empty_stmt. Initially, all empty statements were shared, after the pattern of the Java front end, but this caused a lot of trouble in practice, and they were recently unshared.

An empty statement is represented as (void)0.

7.3.4 Loops

All loops are currently expressed in GIMPLE using LOOP_EXPR, which represents an infinite loop. Loop conditions, break and continue are converted into explicit gotos.

A future loop optimization pass may represent canonicalized loops using another tree code, perhaps DO_LOOP_EXPR, but this has not been implemented yet.

7.3.5 Selection Statements

A simple selection statement, such as the C if statement, is expressed in GIMPLE using a void COND_EXPR. If only one branch is used, the other is filled with an empty statement.

Normally, the condition expression is reduced to a simple comparison. If it is a shortcut (&& or ||) expression, however, we try to break up the if into multiple ifs so that the implied shortcut is taken directly, much like the transformation done by do_jump in the RTL expander. Currently, this is only done when it can be done simply by adding more ifs; in the future, this transformation will handle more cases and use goto if necessary.

The representation of a switch is still unsettled. Currently, a SWITCH_EXPR contains the condition, the body, and a TREE_VEC of the LABEL_DECLs which the switch can jump to, and case labels are represented in the body by CASE_LABEL_EXPRs. In future, we may want to move even more information about the cases into the SWITCH_EXPR itself, and reduce the CASE_LABEL_EXPRs to plain LABEL_EXPRs.

7.3.6 Jumps

Other jumps are expressed by either GOTO_EXPR or RETURN_EXPR.

The operand of a GOTO_EXPR must be either a label or a variable containing the address to jump to.

The operand of a RETURN_EXPR is either NULL_TREE or a MODIFY_EXPR which sets the return value. I wanted to move the MODIFY_EXPR into a separate statement, but the special return semantics in expand_return make that difficult. It may still happen in the future.

7.3.7 Cleanups

Destructors for local C++ objects and similar dynamic cleanups are represented in GIMPLE by a TRY_FINALLY_EXPR. When the controlled block exits, the cleanup is run.

TRY_FINALLY_EXPR complicates the flow graph, since the cleanup needs to appear on every edge out of the controlled block; this reduces our freedom to move code across these edges. In the future, we will want to lower TRY_FINALLY_EXPR to simpler forms at some point in optimization, probably by changing it into a TRY_CATCH_EXPR and inserting an additional copy of the cleanup along each normal edge out of the block.

7.3.8 Exception Handling

Other exception handling constructs are represented using TRY_CATCH_EXPR. The handler operand of a TRY_CATCH_EXPR can be a normal statement to be executed if the controlled block throws an exception, or it can have one of two special forms:

- A `CATCH_EXPR` executes its handler if the thrown exception matches one of the allowed types. Multiple handlers can be expressed by a sequence of `CATCH_EXPR` statements.
- An `EH_FILTER_EXPR` executes its handler if the thrown exception does not match one of the allowed types.

Currently throwing an exception is not directly represented in GIMPLE, since it is implemented by calling a function. At some point in the future we will want to add some way to express that the call will throw an exception of a known type.

8 Example

```
struct A { A(); ~A(); };

int i;
int g();
void f ()
{
  A a;
  int j = (--i, i ? 0 : 1);

  for (int x = 42; x > 0; --x)
    {
      i += g()*4 + 32;
    }
}
```

becomes

```
void f() ()
{
  struct A * a.1;
  int iftmp.2;
  int T.3;
  int T.4;
  int T.5;
  struct A * a.6;

  {
    struct A a;
    int j;

    a.1 = &a;
    __comp_ctor (a.1);
    try
      {
        i = i - 1;
        if (i == 0)
          iftmp.2 = 1;
        else
          iftmp.2 = 0;
        j = iftmp.2;

        {
          int x;

          x = 42;
          while (1)
            {
              if (x <= 0)
                goto break_label;

              T.3 = g ();
              T.4 = T.3 * 4;
              T.5 = i + T.4;
              i = T.5 + 32;

              x = x - 1;
            };
          break_label:;
        }
      }
    finally
      {
        a.6 = &a;
        __comp_dtor (a.6);
      }
  }
}
```

9 Rough GIMPLE Grammar

```
function:
  FUNCTION_DECL
    DECL_SAVED_TREE → block
block:
  BIND_EXPR
    BIND_EXPR_VARS → DECL chain
    BIND_EXPR_BLOCK → BLOCK
    BIND_EXPR_BODY
      → compound−stmt
compound−stmt:
  COMPOUND_EXPR
    op0 → non−compound−stmt
    op1 → stmt
stmt: compound−stmt
  | non−compound−stmt
non−compound−stmt:
  block
  | loop−stmt
  | if−stmt
  | switch−stmt
  | jump−stmt
  | label−stmt
  | try−stmt
  | modify−stmt
  | call−stmt
loop−stmt:
  LOOP_EXPR
    LOOP_EXPR_BODY
      → stmt | NULL_TREE
  | DO_LOOP_EXPR
    (to be defined later)
if−stmt:
  COND_EXPR
    op0 → condition
    op1 → stmt
    op2 → stmt
switch−stmt:
  SWITCH_EXPR
    op0 → val
    op1 → stmt
    op2 → TREE_VEC of LABEL_DECLs
jump−stmt:
    GOTO_EXPR
      op0 → LABEL_DECL | ,*, ID
  | RETURN_EXPR
      op0 → modify−stmt
            | NULL_TREE
label−stmt:
  LABEL_EXPR
      op0 → LABEL_DECL
  | CASE_LABEL_EXPR
      CASE_LOW → val | NULL_TREE
      CASE_HIGH → val | NULL_TREE
      CASE_LABEL → LABEL_DECL
try−stmt:
  TRY_CATCH_EXPR
    op0 → stmt
    op1 → handler
  | TRY_FINALLY_EXPR
    op0 → stmt
    op1 → stmt
handler:
  catch−seq
  | EH_FILTER_EXPR
  | stmt
catch−seq:
  CATCH_EXPR
  | COMPOUND_EXPR
      op0 → CATCH_EXPR
      op1 → catch−seq
modify−stmt:
  MODIFY_EXPR
    op0 → lhs
    op1 → rhs
call−stmt: CALL_EXPR
  op0 → _DECL | ,&, _DECL
  op1 → arglist
arglist:
  NULL_TREE
  | TREE_LIST
      op0 → val
      op1 → arglist

varname : compref | _DECL
lhs: varname | ,*, _DECL
pseudo−lval: _DECL | ,*, _DECL
compref :
  COMPONENT_REF
    op0 → compref | pseudo−lval
  | ARRAY_REF
    op0 → compref | pseudo−lval
    op1 → val

condition : val | val relop val
val : _DECL | CONST

rhs: varname | CONST
   | ,*, _DECL
   | ,&, varname
   | call_expr
   | unop val
   | val binop val
   | ,(, cast ,), varname

(cast here stands for all valid C
typecasts.  Use of varname here seems
odd; it may change to val.)

unop: ,+, | ,-, | ,!, | ,~,

binop: relop | ,-, | ,+, | ,/, | ,*,
  | ,%, | ,&, | ,|, | ,«, | ,», | ,^,

relop: All tree codes of class ,<,
```

References

[SIMPLE] L. Hendren and C. Donawa and M. Emami and G. Gao and Justiani and B. Sridharan, *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*, Lecture Notes in Computing Science no. 757 (1992) p. 406-420

[Morgan] Robert Morgan. *Building an Optimizing Compiler*, Digital Press (1998).

Tree SSA
A New Optimization Infrastructure for GCC

Diego Novillo
Red Hat Canada, Ltd.
`dnovillo@redhat.com`

Abstract

Tree SSA is a new optimization framework based on the Static Single Assignment (SSA) form that operates on GCC's tree representation. Tree SSA is designed to be both language and target independent and allow high-level analyses and transformations that are difficult or impossible to implement with RTL. One of the main goals of the project is to produce an analysis and optimization infrastructure based on proven algorithms and techniques available in the literature. In this paper we describe the design and implementation of the Tree SSA framework, provide preliminary results and discuss possible applications and future work.

1 Introduction

Currently, optimizing transformations in GCC operate on a single intermediate representation, namely RTL (Register Transfer Language). Parse trees generated by the front end are almost immediately converted into RTL and passed on to the optimizer (Figure 1).

Being a low-level representation, RTL works well for optimizations that are close to the target (e.g., register allocation, delay slot optimizations, peepholes, etc). However, many optimizing transformations need higher level information about the program that is difficult (or even impossible) to obtain from RTL (e.g., array references, data types, references to program variables, control flow structures). Over time, some of these transformations have been implemented in RTL, but since the data structure is not really suited for this, the end result is code that is excessively convoluted, hard to maintain and error prone.

In this paper we describe an optimization framework based on GENERIC and GIMPLE, two high-level intermediate representations (IR) derived from GCC parse trees [5]. Language-specific constructs are removed from the input parse trees to obtain GENERIC. In turn, GENERIC trees are broken down into a simpler three address representation called GIMPLE which is used for optimization.

Optimizing GIMPLE is appealing because, (a) it facilitates the implementation of new analyses and optimizations closer to the source, (b) it simplifies the work of the RTL optimizers, potentially speeding up the compilation process or improving the generated code, and (c) it can be done in a largely language and target-independent way. The latter is an important feature for a compiler like GCC that targets many different architectures and languages.

We believe that modularizing the compiler and using well-known published algorithms will help developers maintain and improve GCC, and flatten the learning curve required for ex-

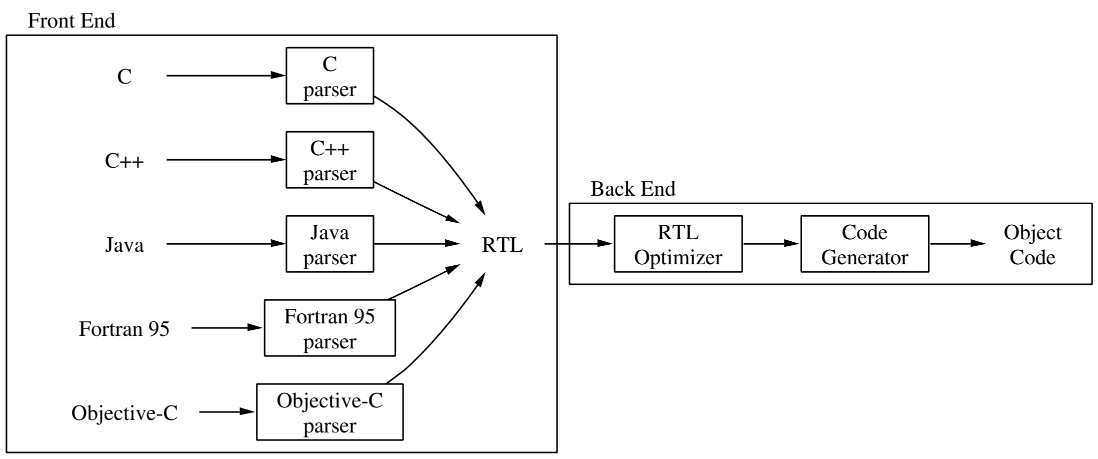

Figure 1: Existing compilation process in GCC.

ternal developers to contribute optimization passes. Furthermore, by reducing the amount of RTL code generated, we also expect to reduce compilation times and improve the quality of the final code.

2 Overview

There are three main components to the basic infrastructure: the gimplifier, the control flow graph (CFG) and the SSA module (Figure 2).

- The gimplifier is responsible for converting the input GENERIC representation into GIMPLE. It also provides functions for generating GIMPLE statements and testing whether a given statement or expression is in GIMPLE form.

- The Control Flow Graph (CFG) is a directed graph that models the execution of the program. Each node in the CFG, called a *basic block*, represents a non-branching sequence of statements (execution starts with the first instruction in the group and it leaves the block only after the last instruction has been executed). The edges of the graph represent possible execution paths in the flow of control (conditionals, loops, etc.).

- Static Single Assignment (SSA) is a relatively new representation that is becoming increasingly popular because it leads to efficient algorithmic implementations of data flow analyzers and optimizing transformations [3].

  The SSA module finds all the variables referenced and builds the SSA form for the function. It provides all the necessary functions and data structures to compute, among other things, aliasing, reaching definitions, and reached-uses information. It is also responsible for converting the function back to normal form right before calling the RTL expanders.

Figure 3 shows the proposed integration between GIMPLE and RTL optimizations as implemented in the `tree-ssa` branch. Notice that the interface between GENERIC and GIMPLE may involve some language-dependent transformations, but those issues are beyond the scope of this paper.

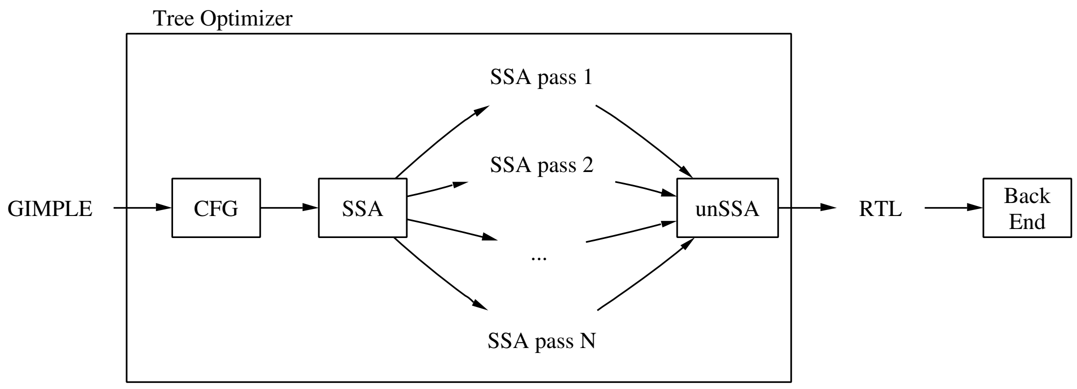

Figure 2: Overview of the tree optimization process.

3 GIMPLE Trees

Although GCC parse trees provide very detailed information about the original program, they are not suitable for optimization:

1. **Lack of a common representation**. There is no single tree representation shared by all the front ends. This means that each language would require a different implementation of the same infrastructure. This would be a maintenance nightmare and would make it very difficult to add new front ends to GCC.

2. **Side effects**. Parse trees are allowed to have side effects. This means that the tree analysis and optimization phases would have to understand the semantics of every source language, which takes us to the multiple implementation scenario described above.

3. **Structural complexity**. Parse trees may combine in arbitrarily complex patterns, which may obfuscate the control flow of the program. For instance, the following expression is represented in a single parse tree

   if ((a = (b > 5) ? c : d) > 10)

   . . .

   When building the control flow graph for this code fragment, the compiler must realize that the predicate for the `if()` statement contains different flows of control of its own. Furthermore, this expression requires more than one basic block to be represented.

To overcome these limitations, we use two new tree-based intermediate representations called GENERIC and GIMPLE. GENERIC addresses the lack of a common tree representation among the various front ends. GIMPLE solves the side-effect and complexity problems that facilitate the discovery of data and control flow in the program. All the analyses and optimizations are done on the GIMPLE representation.

Figure 4 illustrates the differences between GENERIC (Figure 4(a)) and GIMPLE (Figure 4(b)). Notice how in the GIMPLE version individual expressions are simpler and more regular in structure. For instance, with the exception of function calls, a statement in GIMPLE form is guaranteed to have no more than three variable references. GIMPLE expressions are also guaranteed to contain no side-effects (for example, the post-increment operation in line 5 of Figure 4(a) has been explicitly exposed by

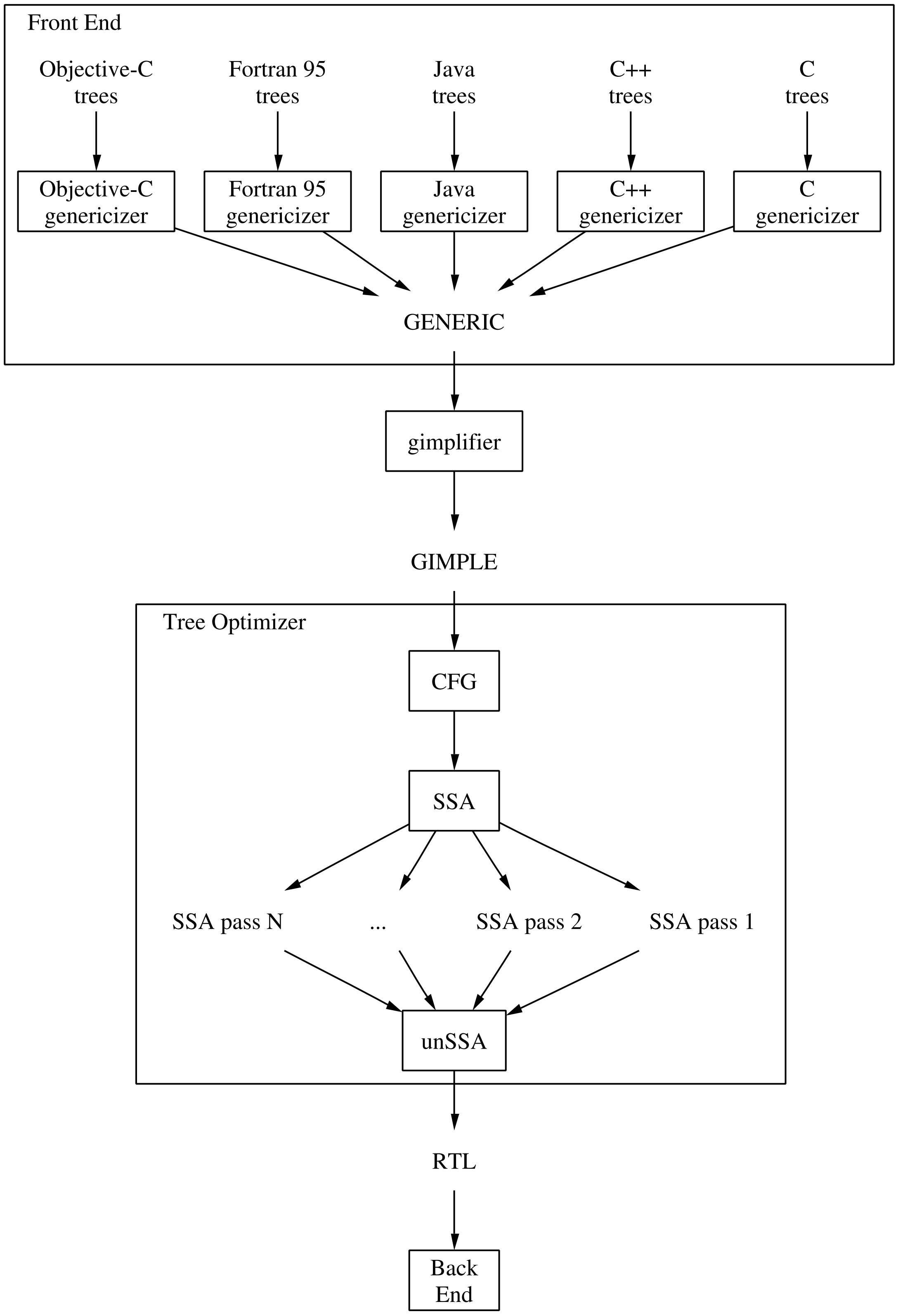

Figure 3: Proposed integration of GIMPLE and RTL optimizers.

```
a = foo ();
b = a + 10;
c = 5;
if (a > b + c)
  c = b++ / a + (b * a);
bar (a, b, c);
```

(a) GENERIC form.

```
a = foo ();
b = a + 10;
c = 5;
T1 = b + c;
if (a > T1)
  {
    T2 = b / a;
    T3 = b * a;
    c = T2 + T3;
    b = b + 1;
  }
bar (a, b, c);
```

(b) GIMPLE form.

Figure 4: A program in GENERIC and GIMPLE forms.

the conversion to GIMPLE form).

4 The Control Flow Graph

To take advantage of the existing flow graph code for RTL, the GIMPLE flow graph uses the same data structures for basic blocks and edges. This allows the GIMPLE CFG to use all the functions that operate on the flow graph independently of the underlying IR (e.g., dominance information, edge placement, reachability analysis). For the cases where IR information is necessary, we either replicate functionality (e.g., flow graph cleanup) or have introduced hooks (e.g., loop discovery).

The flow graph builder will also remove superfluous control expressions of the form `if (0)`, `if (1)` and `switch (CST)`. The edges leading to the unreachable arms of the conditionals are removed, which in turn causes the unreachable arms to be removed. These statements are also completely linearized by replacing the conditional with the clause that is always executed.

4.1 Statement manipulation

Although GIMPLE trees have a much simpler structure than GENERIC and the original parse trees, they still contain certain elements that are of no interest to a typical optimization pass. GIMPLE is a container-based data structure. As such, statements inside constructs like loops, conditionals and lexical scopes are contained in sub-trees. Within each lexical scope, individual statement nodes are chained together using compound expression (CE) nodes. For instance, the body of function `baz` in Figure 5 contains two statements, the lexical scope starting at line 5 and the `return` statement at line 13. In turn, the lexical scope at line 5 contains 3 statements (lines 8, 9 and 10). Notice how all the statements in each lexical scope are joined using CE nodes.

One way to traverse this function is to use the traditional call to `walk_tree` with a callback function to do the processing. However, this approach not only visits more nodes than necessary, but it also makes it difficult to distinguish a statement from an expression contained

```
1   baz ()
2   {
3     int i, j;
4
5     {
6       int k;
7
8       k = foo ();
9       i = k + 2;
10      j = i * k;
11    }
12
13    return j;
14  }
```

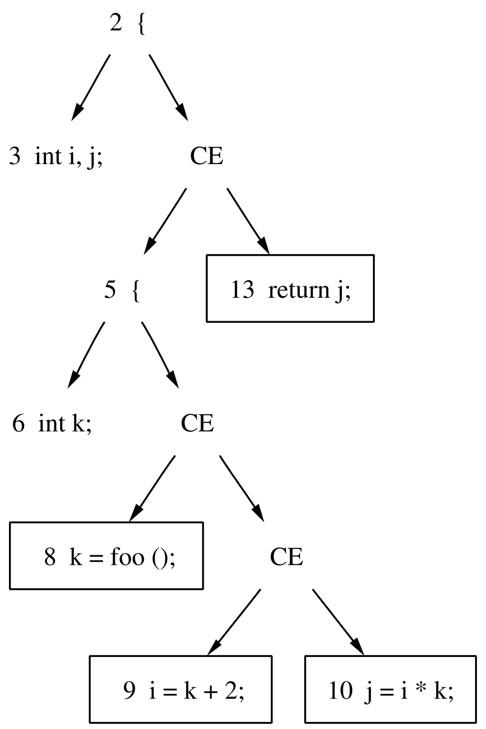

Figure 5: A GIMPLE program and its tree representation.

in a statement[1].

To traverse the statements of a function in GIMPLE, one must follow the compound expression nodes in the body of the function. We have implemented an iterator data structure, called *tree statement iterator* (TSI), to facilitate this process. Note that TSIs don't guarantee that every single statement will be visited. A traversal starting at line 5 in Figure 5 will only visit lines 5 and 13. It is up to the caller to detect when a lexical scope or control statement is being visited and recursively visit its body.

While TSIs are convenient for traversing lexical scopes, they are not suited for traversing statements inside basic blocks. Notice how function `baz()` contains a single basic block. A proper traversal should visit lines 8, 9, 10 and 13, which could be done using TSIs, but the caller would have to be responsible for handling lexical scopes and determining basic block boundaries. This is provided by *block statement iterators* (BSI). Thus, once the flow graph for the function has been built, traversing all the statements in the function can be done with the double nested loop:

```
FOR_EACH_BB (bb)
  for (i = bsi_start (bb); !bsi_end_p (i); bsi_next (&i))
    process_stmt (bsi_stmt (i));
```

BSIs can also be used for backward traversals as well as statement manipulation. Currently, statements can be removed, inserted inside blocks (before and after other statements) and inserted on edges.

[1]GENERIC and GIMPLE do not distinguish statements from expressions as is done in the C and C++ front ends.

```
 1  a = foo ();
 2  b = a + 10;
 3  c = 5;
 4  T1 = b + c;
 5  if (a > T1)
 6    {
 7      T2 = b / a;
 8      T3 = b * a;
 9      c = T2 + T3;
10      b = b + 1;
11    }
12  bar (a, b, c);
```

(a) Original GIMPLE program.

```
 1  a_1 = foo ();
 2  b_1 = a_1 + 10;
 3  c_1 = 5;
 4  T1_1 = b_1 + c_1;
 5  if (a_1 > T1_1)
 6    {
 7      T2_1 = b_1 / a_1;
 8      T3_1 = b_1 * a_1;
 9      c_2 = T2_1 + T3_1;
10      b_2 = b_1 + 1;
11    }
12  b_3 = φ(b_1, b_2);
13  c_3 = φ(c_1, c_2);
14  bar (a_1, b_3, c_3);
```

(b) Same program in SSA form.

Figure 6: Static Single Assignment form.

5 Static Single Assignment form

The Static Single Assignment (SSA) form [3] is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable. Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment.

Figure 6 shows the program from Figure 4(b) and its corresponding SSA form (Figures 6(a) and 6(b) respectively). Notice that every assignment in the program introduces a new version number for the corresponding variable. Every time a variable is used, its name is replaced with the version corresponding to the most recent assignment for the variable.

Now consider the use of variable `b` in the call to `bar()` (line 12). There are two assignments to `b` that could reach line 12; the assignment at line 2 and the assignment inside the `if` at line 10. To solve this ambiguity, SSA introduces a new artificial definition for `b` by means of a ϕ (phi) function. This new definition merges both assignments to create a new version for `b` (b_3). The semantics of the ϕ function dictate that b_3 will take the value from one of the function's arguments. The specific argument returned by the ϕ function is not known until runtime.

6 Real and virtual operands

The SSA form is not suited for handling non-scalar variable types. For instance, given an array `M[100][100]`, not only would the compiler need to keep track of 10,000 different version numbers for `M`, but it may also be impossible to determine whether two references `M[i][j]` and `M[k][l]` are the same variable or not. Structures, unions and aliased variables present similar problems.

One alternative to handling non-scalar types would be to simply ignore them. After all, if the operands are not converted into SSA form, they would not be considered for optimization. However, that would also mean that statements referencing nothing but non-scalars would appear dead to the optimizers. Also, situations like scalar variables aliased by a structure field would also be missed.

To address this problem, operands referencing non-scalar variables are considered references to the base object for that variable. For instance, references to `M[i][j]` and `M[k][l]` in the previous example would be considered references to `M`. Since these operands need to be treated separately by the optimizers, they are known as *virtual operands*, as opposed to the *real operands* for scalar variables. Therefore, every GIMPLE statement *S* contains four distinct sets of operands:

DEF(S). If *S* is an assignment statement, this is the variable at its left-hand side.

USES(S) is the set of all the variables used (or loaded) by *S*.

VDEFS(S) is the set of all the virtual variables defined (or stored) by *S*. VDEF operators represent non-killing definitions because they may or may not occur at run time. A VDEF operator is of the form `V = VDEF <V>`, which means that a new value for `V` is created from `V`'s old value.

VUSES(S) is the set of all the virtual variables used by *S*.

Virtual operands are also used to handle situations where the program is altering variables in ways that are difficult or impossible to determine statically. In these cases, the data flow framework needs to gather enough information to prevent the optimizers from missing a potential data dependency. In all these cases, virtual operands are used. Some of the more common situations include:

1. **Aliasing**. If two variables `a` and `b` may alias each other, then the compiler selects one of them to serve as the representative for all the aliased references. Every reference to either variable is then considered a virtual operand using the alias representative.

2. **Call clobbering**. Function calls may modify addressable local variables and globals in unknown ways. This is handled using a similar approach. Variables that may be call clobbered are considered alias of an artificial variable called `.global_var`. This variable is considered modified by function calls and by assignments to any of the variables associated with it.

3. **Inline assembly**. Much like function calls, inline assembly may modify local variables in ways that the optimizers do not understand. Variables listed in the *outputs* or *clobbers* list of GCC's `asm` statement, are considered VDEF operands.

The programs in Figures 7, 8 and 9 illustrate how virtual operands are used to handle non-scalar variables, aliasing and call clobbering. All the example functions have been renamed into SSA already. Notice how the VDEF operators link new SSA versions for a variable with its previous version. This creates def-def links that are used in passes like dead-code elimination to determine all the potentially live assignments.

```
double foo (int i, int j, int k, int l)
{
  double T1, T2, f;
  double M[100][100];

  /* References to an element of 'M' are
     considered references to the whole
     matrix.   */
  # M_2 = VDEF <M_1>
  M[i][j] = ...

  /* VDEFs are non-killing definitions,
     that's why the new definition
     created for M_3 is linked to M_2 in
     the previous assignment.     */
  # M_3 = VDEF <M_2>
  M[k][l] = ...

  # VUSE <M_3>
  T1_4 = M[i][j];

  # VUSE <M_3>
  T2_5 = M[k][l];
  f_6 = T1_4 + T2_5;

  return f_6;
}
```

Figure 7: Virtual operands for handling non-scalar variables.

7 Representing pointers

In GIMPLE there are no multi-level pointers. This is a very appealing property that allows the compiler to keep track of a pointer p and its dereference *p as two separate, but related, variables. The relation between p and *p is quite straightforward:

1. Every store to p implies a store operation to *p, because now p is pointing to a different memory location.
2. Every store or load of *p implies a load operation from p, because p is read to determine what memory location to use.

```
int foo (int i, int j, int *p)
{
  int a;

  if (i_1 > j_2)
    {
      /* Whenever 'p' changes, '*p' must
         also change.   */
      # (*p)_4 = VDEF <(*p)_3>
      p_5 = &a;
    }

  /* Since '*p' may alias 'a', instead
     of renaming the operand 'a', we
     create a virtual definition for its
     alias '*p'.    */
  # (*p)_7 = VDEF <(*p)_4>
  /* 'p' is needed to access '*p'.      */
  # VUSE <p_5>
  a = i_1 + j_2;

  # VUSE <(*p)_7>
  return *p;
}
```

Figure 8: Virtual operands for handling aliases.

8 Conversion into SSA form

Converting the program into SSA form consists of three main phases:

1. may-alias computation, which determines what variables are referenced in the function and whether they may be aliased or not,
2. insertion of ϕ nodes at basic blocks reached by more than one definition of the same variable, and,
3. statement renaming, which rewrites every operand and virtual operand using the appropriate SSA version numbers.

The following sections highlight the more important aspects of the conversion into SSA

```
float F;

float foo(float f)
{
  /* Since 'F' is call-clobbered,
     instead of renaming 'F' in the
     statement, we rename the virtual
     operand .GLOBAL_VAR. */
  # .GLOBAL_VAR_2 = VDEF <.GLOBAL_VAR_1>
  F = f_3 + 2;

  /* Function calls clobber the variable
     .GLOBAL_VAR which in turn indicates
     that 'F' is also clobbered.       */
  # .GLOBAL_VAR_3 = VDEF <.GLOBAL_VAR_2>
  bar ();

  /* Uses of 'F' are converted to
     virtual uses of .GLOBAL_VAR.   In
     this statement we are using the
     value of 'F' potentially
     modified by the call to bar().      */
  # VUSE <.GLOBAL_VAR_3>
  return F;
}
```

Figure 9: Virtual operands for handling call clobbering.

form. A more detailed description of the process can be found in the literature [3, 1, 6].

8.1 Computing may-alias information

This pass collects all the variables referenced in the function and determines may-alias sets for each one. Currently, alias information is type-based. A points-to analyzer is implemented, but it is not fully functional yet.

8.2 Inserting ϕ nodes

This pass inserts ϕ nodes at the dominance frontier of blocks with live variable definitions. The algorithm implements the semi and fully pruned forms suggested by Briggs et. al. [1] to reduce the number of ϕ nodes in the program. The basic idea is that if a variable is not live after being defined in block b, then it is not necessary to insert a ϕ node at the dominance frontier of b.

Since computing global live-in information is more expensive than local live-in, this pass uses a heuristic based on the total number of ϕ arguments. If this is is above a certain threshold[2], the compiler builds a fully pruned form.

8.3 Rewriting statements and dominator-based optimizations

The renaming process is done using a depth-first traversal of the flow graph's dominator tree [3]. During this traversal it is possible to apply very simplistic transformations that take advantage of the order in which basic blocks are visited [6].

These transformations, also known as *dominator-based optimizations*, include constant propagation, redundancy elimination, copy propagation and propagation of predicate expressions. These optimizations are only supposed to do simple cleanup work that catches most of the simple cases. The key property is that they must work fast because they are piggybacked on top of the renaming process (which is linear in the number of statements).

1. Constant propagation. When a constant assignment of the form $a_i = C$ is found, it is stored in a hash table. Successive occurrences of a_i are replaced with C. No folding nor control flow simplification is done, only constant replacements. Copy assignments are similarly optimized.

2. Redundancy elimination uses a similar idea. When an assignment of the form $a_i = b_j \oplus c_k$ is found, the expression $b_j \oplus c_k$

[2]Currently 32.

is stored into a hash table. Successive occurrences of $b_j \oplus c_k$, *within the same sub-tree*, are replaced with a_i. Notice that this transformation is valid only when replacing redundant expressions dominated by the original assignment, otherwise it would be possible to insert a_i in a control flow path where it is never evaluated.

3. Propagation of predicate expressions. When a conditional statement of the form `if (`$a_i == C$`)` is found, the assignment $a_i = C$ is inserted into the hash table for constants and copies when processing the "then" clause of the conditional. This will cause the constant/copy propagator to replace a_i with C in that sub-tree.

8.4 Conversion back to normal form

Once all the SSA optimizations have been applied to the function, all the SSA version numbers and ϕ nodes must be removed to return the code to its original form. This process consists mainly in converting all ϕ nodes into copy operations. Some of the more important aspects of this pass is avoiding superfluous copy operations. We implement the standard conversion into normal form described in the literature [1, 6].

9 Implementation status

Currently, the basic framework is almost finished. Two front ends (C and C++) have been fully converted to emit GIMPLE trees and the regression test suite presents similar results to those of mainline GCC. Readers interested in testing the current implementation and/or contributing to its development are invited to visit the Tree SSA web page at `http://gcc.gnu.org/projects/tree-ssa/`. This page contains information for retrieving a copy of the development branch in CVS, status of the implementation and a list of "to-do" items.

In terms of performance, the branch still lags behind mainline. This is hardly surprising as we have mostly worked on correctness issues. Performance is going to be the focus of the next phase of development. We have been tracking performance using SPEC95 and SPEC2000. Daily results of these experiments can be found at `http://gcc.gnu.org/benchmarks/`.

In addition to the optimizations performed while renaming into SSA form and the flow graph restructuring, there are four optimization passes implemented.

1. Sparse Conditional Constant Propagation (CCP) [7] is an efficient formulation of the constant propagation problem that is also capable of finding constant conditionals and unreachable code. This optimization is currently enabled by default at `-O1` and above.

2. Partial Redundancy Elimination (PRE) [2] finds expressions that are computed more than once and re-writes them so that their values are computed once and reused as necessary. In addition to removing completely redundant computations, PRE has the ability to make partially redundant computations fully redundant, thus combining the effects of global common subexpression elimination and loop invariant code motion.

3. Dead Code Elimination (DCE) [3] removes all statements in the program that have no effect on its output (assignments to variables that are never used again, conditional expressions with empty bodies, etc). This optimization is currently enabled by default at `-O1` and above.

4. Copy Propagation (CP) is the same optimization applied while converting the program into SSA form, but implemented as a separate pass.

We are also implementing a memory checker, called *mudflap*, that instruments every pointer and array reference in the program with boundary checks [4]. It is a combination of compile-time instrumentation and run-time library. The instrumented code contains calls to the run-time library that will be triggered when the program attempts one of several illegal operations, such as accessing an array out of bounds, freeing the same block of memory more than once, accessing unallocated memory, leaking memory, etc.

Mudflap is not yet integrated into the SSA framework, so no static analyses are done to prevent inserting superfluous instrumentation. Optimization of mudflap instrumentation is currently underway.

10 Conclusions

The Tree SSA project provides a new optimization framework to make it possible for GCC to implement high-level analyses and optimizations. Currently, the framework is in active development and some optimizations have already been implemented. The goals of this project include:

- Provide a basic set of data structures and functions for optimizers to query and manipulate the tree representation.
- Simplify and, in some cases, replace existing optimizations that work on the RTL representation but are not really suited for it. By simplifying the work for the RTL optimizers we aim to improve compile times and code quality.
- Implement new optimizations and analyses that are either difficult or impossible to implement in RTL.

By basing all the analyses and transformations on widely known published algorithms, we are also trying to improve our ability to maintain and add new features to GCC. Furthermore, the use of standard techniques will encourage external participation from groups in the compiler community that are not necessarily familiar with GCC.

Acknowledgments

I would like to thank Red Hat for funding the Tree SSA project and to all the developers who have contributed to it. In particular, I would like to thank the regular contributors to the project: Jeff Law and Andrew MacLeod for their work on the base infrastructure and optimizers; Jason Merrill for his work on GENERIC and GIMPLE; Frank Eigler for his work on Mudflap; Sebastian Pop for the original expression simplifier, tree unparser, and tree browser; Daniel Berlin for his work on points-to alias analysis and PRE; Steven Bosscher and the G95 team for their work on integrating G95 with GIMPLE; and Andreas Jaeger, Phil Edwards, and Andrew Pinski for testing the branch on a regular basis.

References

[1] P. Briggs, K. D. Cooper, T. J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software—Practice and Experience*, 28(8):859–881, 1998.

[2] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for

partial redundancy elimination based on SSA form. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, 1997.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] F. Ch. Eigler. Mudflap: Pointer Use Checking for C/C++. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[5] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[6] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[7] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

Porting GCC to the IBM S/390 platform

Hartmut Penner *Ulrich Weigand*
IBM Deutschland Entwicklung GmbH
Schönaicher Str. 220, 71032 Böblingen, Germany
`{hpenner, uweigand}@de.ibm.com`

Abstract

IBM's mainframe architecture S/390 is the living architecture with the longest heritage, defined in a time when assembler programming was predominant and compilers were in their childhood. Hence in porting GCC to S/390 we had to cope with certain architecture features that were difficult or impossible to model in GCC's architecture-independent framework. These include 31-bit addressing mode, instruction-dependent address formats, limited availability of address displacements and immediate literals, and the condition code handling. These problems notwithstanding, the S/390 back end matured over the last couple of years to make GCC a stable and competitive compiler for the S/390 platform. In this paper we want to share how we managed to handle most of the mentioned architecture features. We also want to point out areas that promise room for further improvement in the back end itself and suggest middle-end modifications that would benefit our platform in particular.

1 Introduction

1.1 From System/360 to zSeries

In the early 1960s IBM defined the System/360 architecture. This architecture was designed to serve for a whole family of systems. The difference the distinguished systems of that family had was the way the instruction set was implemented. The System/360 architecture defined 16 32-bit general purpose registers, 4 64-bit floating point register and a 24-bit address space. Shortly afterwards, virtual addressing was added to the architecture. In 1970, System/370 was introduced, providing an enhanced instruction set. Around 1982 370/XA brought 31-bit addressing, in 1988 370/ESA introduced support for multiple address spaces. In the 1990s the ESA/390 architecture was introduced; subsequent machines added over time the relative branch instructions as well as the IEEE floating-point instruction set.

In 2000 the first IBM eServer zSeries machine came out, introducing a major architecture update. The z/Architecture remained upward compatible to ESA/390, but provided full 64-bit support, extending the general purpose register size to 64-bit and adding a 64-bit addressing mode in addition to the traditional 24-bit and 31-bit modes. This means in particular that both 64-bit and 31-bit applications can run under a 64-bit operating system (if that provides the required support). However, it is also possible to operate a zSeries machine in ESA/390 mode in order to run legacy 31-bit operating systems.

1.2 GCC S/390 port history

Within the S/390 firmware development we were searching in 1997 for a C compiler fulfilling specific requirements. We needed a compiler that could be link-compatible to the internally used pl.8 compiler, which was developed at IBM Research a decade ago. Also it should provide the ability to use embedded assembler code. One of the authors was asked to look into the then existing System/370 port of GCC, to evalute whether this could be adapted for the intended use. This port was not very stable at this time, but it could easily be shown that it could be used as a base. Since in firmware development there is no reason for backward-compatibility, we decided to set a certain level of architecture as given, and started internally with a S/390 port, producing only code for latest CMOS based systems.

When work on the upcoming Linux for S/390 port started in 1998, the compiler port developed by the firmware team could be used for the Linux port. The success of this new operating system proved to be beneficial for GCC on S/390 as well, since the Linux development team was then rapidly driving the efforts to develop the GCC port further to use the ELF linkage format and eventually to exploit the 64-bit z/Architecture. In 2001, the S/390 GCC port was finally donated to the Free Software Foundation, with the authors in charge as maintainers, one of us (Hartmut Penner) representing the S/390 hardware, the other (Ulrich Weigand) the Linux for S/390 constituency.

2 Architectural overview

Before going into details of the GCC back end implementation, we will start by giving a short overview of the relevant features of the zSeries architecture as well as the ABI used by the Linux for zSeries port.

2.1 zSeries instruction set

The zSeries architecture as a typical CISC architecture provides an extensive instruction set. It has a full set of I/O related instructions, dealing with a channel based I/O subsystem. For system programming there exists a full set of instructions which enables operation systems to retrieve all information about the system running on and do communication with a service element. The `START INTERPRETATIVE EXECUTION` instruction provides efficient virtualization capabilities, with the possibility to define very precisely which instructions are to be intercepted. Many of these architectural facilities were defined over the last 40 years, putting all the experience of the years before into the definition. However, even though the above mentioned fields are very interesting, we want to concentrate in this paper on the small subset of instructions a compiler normally deals with. For a complete reference of the ESA/390 or z/Architecture see [1] or [2].

The zSeries architecture defines 16 general-purpose registers and 16 floating-point registers. Depending on the architecture mode, the general-purpose registers have a width of 32 or 64 bits. It is a classical 2-address architecture, where for most instructions the first source operand is also used as destination. Each instruction has a length of 2, 4, or 6 bytes, and up to now more than 30 instruction formats are defined. For most ALU operations there exist two instruction types, one using two register operands (RR), the other a register and a memory operand (RX). Logical operations are also available with two storage operands (SS) or a storage and a immediate operand (SI).

More formally, the general instruction set of the zSeries architecture usable by a compiler can be divided into following classes of instructions:

RR	r1 = r1 op r2
RX	r1 = r1 op [x+b+d]
RI	r1 = r1 op ch
RS	r1 = r1 op [b+d]
SI	[b+d] = [b+d] opl cb
SS	[b1+d1] = [b1+d1] opl [b2+d2]

where we use the following elements:

r	General or floating-point register
x	Index register (register `%r1`–`%r15`)
b	Base register (register `%r1`–`%r15`)
d	Displacement, 12-bit constant (0–4095)
cb	8-bit constant, unsigned
ch	16-bit constant, signed or unsigned
op	Arithmetical or logical operation
opl	Logical operation (including move)
[addr]	Content addr is pointing to

If running in zSeries architecture mode, an address is 24, 31, or 64 bits wide, depending on the addressing mode a program operates in. The S/390 architecture mode provided only the 24-bit and 31-bit addressing modes. Here, the most significant bit of a 32-bit address is sometimes used to distinguish between 24-bit and 31-bit bit mode in 'mixed' environments. The displacement for address generation in the instruction itself is only 12 bits. Together with the fact that this displacement is unsigned, this causes some problems for defining a ABI and implementing a efficient compiler, especially when dealing with large stack frames, a downward growing stack, large GOT tables, etc. The impact of this for implementing the compiler will be shown later.

In order to provide conditional execution, zSeries uses a 2-bit condition code as part of its program status word. Most non-move or non-branch instructions, depending on the result of their operation, set this condition code. The actual value a specific instruction sets is defined for each instruction individually, and only to a certain extend a clear classification can be made. The architecture has branch instructions that decide whether a branch is taken depending on whether the current condition code equals one of the values provided in the form of a 4-bit branch condition mask as part of the instruction.

2.2 Linux for zSeries ABI

The Linux port on S/390 and zSeries uses a variant of the ELF ABI. For a full definition of the architecture-dependent parts see [3] and [4]; the following gives a short overview of the most important features. While the processor architecture does not define a stack, the ABI chooses by convention the general purpose register `%r15` for use as stack pointer. The stack grows downwards; the low 96 bytes (160 bytes on 64-bit) are reserved as register save area for use by called subroutines. Registers `%r0`–`%r5` are clobbered across function calls, while `%r6`–`%r15` are saved. Parameters are passed in registers and a parameter area on the stack.

Apart from the stack pointer `%r15`, the following general purpose registers may be used for special purposes: `%r14` holds the function call return address, `%r13` is used to point to a per-function literal pool, `%r12` points to the Global Offset Table in position-independent code, and `%r11` is used as frame pointer in functions that perform dynamic stack allocation (otherwise, the stack pointer is used as frame pointer as well).

The following short "hello world" example shows a typical 31-bit routine. Comments under each line give the semantics of the instruction, using the abbreviated syntax used by GCC in its scheduling printouts.

```
  stm  %r6,%r15,24(%r15)
# {[%r15+24]=%r6;[%r15+28]=%r7;...}
  bras %r13,.L2
# {%r13=.L1;pc=.L2}
.L1:
.LC0:  .long   .LC2
```

```
.LC1:  .long   printf
.L2:
  ahi  %r15,-96
# {%r15=%r15-96;clobber %cc}
  l    %r2,.LC0-.L1(%r13)
# {%r2=[%r13+.LC0-.L1]}
  l    %r14,.LC1-.L1(%r13)
# {%r14=[%r13+.LC1-.L1]}
  basr %r14,%r14
# {pc=%r14;%r14=pc+2}
  lm   %r6,%r15,120(%r15)
# {%r6=[%r15+120];%r7=[%r15+124];...}
  br   %r14
# {pc=%r14}
```

Note how the function prolog saves registers, sets up the literal pool pointer and allocates a new stack frame. The function proceeds to load the address of the `printf` routine as well as the address of the string constant from the literal pool and performs the call. The epilog simply restores all saved registers (thereby resetting the stack pointer and removing the current stack frame) and returns to the caller by branching to the address provided in `%r14`.

3 GCC and the zSeries architecture

While most of the features of the zSeries architecture can be easily modelled using the standard mechanisms available to a GCC back end, we have found some that require extra effort to implement correctly. This section describes how we addressed these issues in the current S/390 back end: literal handling, 31-bit addressing mode, and instruction-dependent address formats.

3.1 Literal handling

Literals, i.e. values determined at compile time, play an important role in most functions generated by a compiler. They include constant values of various types (e.g. integer, floating point, or string constants) provided in the source code as well as address constants generated by the compiler itself, used to reference code or data labels.

However, the original S/390 architecture did not provide instructions that could use literal values as immediate operands. While it was possible to load an immediate integer in the range 0–4095 into a register using the `LOAD ADDRESS` instruction, all other literal values required loading from memory.

On the other hand, accessing a memory location to load a literal from requires to express the address of that location first. Similarly, branch instructions need to be able to reference the branch target address. Again, the original S/390 architecture did not provide instructions that could use immediate address constants, neither as absolute nor as pc-relative values. The only way to specify an address, for any purpose, was to use the standard effective address generation mechanism that computes a target address as the sum of the contents of a base register, an index register, and an immediate displacement in the range 0–4095.

To overcome these restrictions, the usual coding conventions for S/390 applications required to reserve one general purpose register to always hold the address of the start of the routine currently executing. This way, targets for branches within the routine could be expressed via immediate displacement relative to that function base register, and by placing a pool of literal constants immediately adjacent to the routine's code section, the same mechanism could be used to load literals from memory.

The obvious disadvantage of this method is that it requires the total size of a routine's code section plus its literal pool not to exceed a single page (4 KB), to ensure every address within both code and literal pool remains addressable via the function base register. When these limits are exceeded, a function has to be split into multiple fragments, each consisting of up to 4096 bytes of code and literals required by the fragment. On every branch between two different fragments, the base register has to be reloaded to point to the beginning of the current fragment. This can incur significant runtime overhead.

Fortunately, over time several extensions to the S/390 architecture were implemented that provide relief to those constraints. With the second generation of S/390 machines, starting from 1992, the *relative and immediate instruction* facility provided a set of instructions that allow on the one hand the use of immediate integer constants with several operations, and on the other hand the use of pc-relative addressing modes for a number of branch instructions. However, due to the requirement that the new instructions fit within the overall scheme of S/390 instruction types, these literals were limited in range. This means that only integer values in the range of -32768–32767 are allowed as immediate operands, and pc-relative branch targets can only specific a range of up to 64 KB before or after the current instruction.

With the advent of the 64-bit z/Architecture in 2000, the latter restriction was once again loosened: the new *relative long* instructions accept pc-relative targets in a range of up to 4 GB before or after the current instruction. The `LOAD ADDRESS RELATIVE LONG` instruction finally allows the use of pc-relative addressing for other accesses besides branches.

Now, how does the GCC back end cope with those restrictions? We have chosen to support in the S/390 back end only processors that provide the relative and immediate instruction facility. This means that we can use the pc-relative branch instructions for all intra-function branches as long as the code section of the routine does not exceed 64 KB. While we also use immediate operands wherever possible, it is still necessary to maintain a literal pool for constants exceeding the allowed range. This pool is addressed via a base register (usually `%r13`) pointing to the start of the literal pool. To set up the pool base register, we use a `BRANCH RELATIVE AND SAVE` instruction, followed by the literal pool itself. Executing that instruction transfers control to the instruction following the pool, while at the same time loading the pool start address into the base register.

Address literals whose use cannot be avoided via pc-relative instructions are placed into the literal pool. However, if we are generating position-independent code for use in Linux shared libraries, we do not want to place absolute addresses into the literal pool, as those would require relocations to be applied by the dynamic loader to the text segment. This is undesirable as it prevents that page from actually being shared across multiple processes using the same library. To solve this issue, we instead place the *offset* from the start of the literal pool to the required address into the pool. Every user of that pool entry needs to add the pool start address back to that offset, which can usually be done implicitly as part of normal address generation, using the offset loaded into an index register together with the pool register as base register.

This method works fine as long as the routine's code size does not exceed 64 KB and its literal pool size does not exceed 4 KB. For the vast majority of routines these conditions hold true, which is why we have chosen to optimize for this common case. However, the compiler certainly has to be able to cope with cases where either or both of these limits are exceeded.

If a routine's code section exceeds 64 KB, determining whether the target of any particular branch within the function is out of range or not is nontrivial, as this recursively depends on the sizes of other branch instructions that lie in between. Fortunately, this analysis is performed by GCC's *branch shortening* pass, which we are able to use unmodified for our target. We simply need to provide GCC common code with information about the length of each instruction via the `length` attribute.

Once the branch shortening pass has determined which branches cannot be implemented via a pc-relative branch instruction, our machine-dependent reorganization pass replaces each of those out-of-range branches by a branch using a register as target, preceeded by an instruction loading the branch target address from the literal pool into that register:

```
l    %r14,.LCtarget-.Lpool(%r13)
br   %r14
```

As previously mentioned, when generating position-independent code, we place an offset to the branch target label into the literal pool instead:

```
l    %r14,.LCtarget-.Lpool(%r13)
b    0(%r14,%r13)
```

This replacement is simple and incurs only relatively low overhead. However, if the literal pool overflows its maximum size of 4096 bytes, things get much more difficult. Fortunately, this happens only extremely rarely; the cases where we have seen this to occur typically involve extremely large routines, unlikely to be found in source code written by hand, but sometimes occurring as a result of automatically generated code.

However, if literal pool overflow does occur, we still need to handle it correctly. What we do then is to partition the function into smaller *chunks*, each requiring a partial literal pool whose size does not exceed 4096 bytes.

At every transition between different chunks, we insert instructions to reload the pool base register with the start of the literal pool of the current chunk. Those reload instructions thus need to be inserted before the first instruction of every chunk as well as after every code label that is being branched to from an instruction located outside the chunk. Unfortunately, performing this reload operation is difficult, as we cannot use a pc-relative instruction to do so, we cannot use any arithmetical operations as those would clobber the condition code register which might be live at the point the reload is inserted, and we cannot even load anything from the literal pool because we do not know to *which* pool chunk the base register currently points—the same label might be the target of instructions residing in multiple different chunks. We solve this problem by using the following sequence of instructions:

```
basr  %r13,0
la    %r13,.Lchunk-.(%r13)
```

which resets the pool base register to the current instruction address, and adds the offset from there to the current pool chunk start address using a `LOAD ADDRESS` instruction to avoid clobbering the condition code. This technique unfortunately imposes further requirements on the pool chunks: every pool chunk must be placed within the function text section, following the corresponding code chunk, and the size of that code chunk must not exceed 4096 bytes to avoid overflowing the range of the LA instruction.

Once we've succeeded in dividing the function into chunks and inserting the pool base register reload instructions, we can then proceed to replace all references to the normal constant pool by explicit references to the current pool chunk, assuming the base register is set up properly. Position-independent code provides an additional challenge, however. Recall that in this scenario we are using offsets relative to the pool start address instead of absolute address literals. Now, when we've split the pool into multiple chunks, which pool chunk are those references supposed to be relative to? We've initially tried to set up things so that every offset is always relative to the chunk where it resides. Unfortunately this does not work, as due to constant propagation it is possible for an offset to be loaded into a register in a completely different chunk from where that register is finally used. Thus we've decided to keep the master literal pool present, even it is empty after all constants have been distributed to pool chunks, so that its start address can remain to serve as anchor for address literal offsets. To make this work, every *explicit* use of the literal pool base register `%r13` needs to be replaced by another register holding the master anchor address. That address can be computed on the fly using the current pool chunk address and an offset from the start of that chunk to the anchor; this offset is by convention always stored at the very start of each pool chunk:

```
l     %r14,0(%r13)
la    %r14,0(%r14,%r13)
```

Two final obstacles remain before literal pool splitting can be considered a general solution. The first is the fact that literal pool splitting introduces additional instructions at various points throughout the instruction stream. This can cause branch splitting information to become invalid, as some branches that were originally in-range can now exceed their allowed ranges. On the other hand, branch splitting works by placing branch target addresses into the literal pool, which can cause the pool to overflow. To solve this interdependency, we iterate branch splitting and attempting to split the literal pool until both operations succeed simultaneously. This is guaranteed to always terminate, as every branch that we decided to split at any one point will remain split forever, and thus the number of unsplit branches is strictly decreasing throughout this iterative process.

The final obstacle is that we require a temporary register for both branch splitting and literal pool splitting (for the case of anchor reloading). Fortunately, the live ranges introduced are very short, and span just the newly added instructions together with the immediately following instruction from the existing instruction stream. However, at this point in the code generation process (after reload), all registers might in fact be live at the point where we need to insert additional code. Thus, we currently reserve one register (`%r14`) for use for those purposes. Note that the ABI defines `%r14` to hold the function return address, which means it is always clobbered across function calls, but apart from that restriction the register would be free for arbitrary use inside a routine. We are not doing that, however, in order to have this register available for use in branch splitting and literal pool splitting. The only problem with that is that the decision whether we need to use `%r14`—and thus need to save and restore the register in the function prolog/epilog code—can be made only during machine-dependent reorg, long after the function prolog and epilog code was generated. Therefore, we always generate code to save and restore registers `%r13` and `%r14`, and remove that code during machine-dependent reorg once it has proven to be unnecessary.

Up to now, we have exclusively discussed code generation for S/390 machines in 31-bit mode. On z/Architecture machines, many of the problems described in this section disappear due to the availability of the *relative long* family of instructions. First of all, the `BRANCH RELATIVE LONG` instructions allow pc-relative branches within the range of 4 GB. By restricting the maximum allowed size of any single executable or shared object to 4 GB, we can thus use those instructions for nearly every branch. (The only occasion where we still might need branch splitting is in the case of `BRANCH ON COUNT` instructions, which lack a relative-long variant.)

Also, the `LOAD ADDRESS RELATIVE LONG` instruction allows us to directly load arbitrary address literals, without requiring literal pool entry, in a position-independent manner. This means that we never need to handle offsets relative to the literal pool base, and the whole issue of reloading the anchor register after pool splitting disappears. Also, as we can use `LARL` to load the literal pool start address, literal pools no longer need to reside in the text section, but can be moved to the read-only data section. This also simplifies inserting pool base reload instructions in the case of literal pool splitting. However, the core problem that the literal pool cannot exceed 4096 bytes remains.

The solution described in this section allows GCC to correctly handle every valid source code, even if it causes code or literal pool sizes to exceed their optimum limits. However, there is still a lot of room for improvement to optimize the code that is generated once that overflow happens. We are currently working on some minor improvements. In particular, we'll remove the whole complex of pool anchor reloading for position-independent code by representing address literals as offsets relative to the gobal offset table (like on other platforms) instead of relative to the literal pool. This requires some new relocation types to be implemented in binutils first. Once this is done, we can try to finally make register `%r14` available for regular use. This would require that every branch instruction reserves one register to be used for branch splitting if necessary, but even so overall register pressure should benefit.

The major problem with optimizing literal pool overflow situations, however, is to determine how to split the function into chunks. An optimal solution here would try to minimize the frequency of inter-chunk branches at run time. To try to tackle that problem will require control flow data including basic block boundaries and branch probabilities; unfortunately GCC currently no longer maintains that information at the point in time where literal pool splitting has to be performed (in machine-dependent reorg).

3.2 31-bit addressing mode

For historical reasons, the S/390 architecture does not have a 32-bit addressing mode, but uses 31-bit addressing. This means that while base and index registers used in address generation are regular 32-bit registers, the most significant bit is ignored when computing the effective address. (Note that this does not apply on zSeries in 64-bit addressing mode; most of the problems discussed in this section disappear in that environment.)

For the compiler, this causes two issues that need to be considered. As for every 31-bit address there are two equally valid 32-bit pointer representations, one with the high bit set and one with the high bit cleared, care must be taken when comparing pointer values for equality. To simplify this process, GCC tries to always represent pointers using the representation with the high bit cleared. However, some machine instructions store address values with the high bit set; most importantly the `BRANCH AND SAVE` family of instructions does so. A `BAS` instruction transfers control to another address, and at the same time stores the current instruction address (with the high bit set) into a register. GCC uses those instructions for two purposes: to implement function calls, and to set up the literal pool. Since both the call return address and the literal pool start address are normally used only for compiler-internal purposes, GCC does not bother to normalize these values by clearing the high bit. However, in some cases these values are visible externally, and extra care needs to be taken:

- The call return address can be retrieved by doing a stack backtrace, e.g. via the function `__builtin_return_address`. This will yield values with the high bit set, which the caller needs to normalize; this is handled by the `__builtin_extract_return_address` function. However, as this built-in does nothing on most platforms, we have seen several cases where applications didn't work on S/390 because they forgot to use it.
- The literal pool start address is used as anchor to compute the addresses of local variables in position-independent code. As these can be externally visible, the compiler needs to make sure this address computation will normalize the resulting pointer. This is done by using an `UNSPEC` operation that enforces the use of `LOAD ADDRESS` (instead of, say, a normal 32-bit addition operation) to perform the calculation. The `LA` instruction will always return a 31-bit value with the high bit cleared.

The second main problem caused by the 31-bit addressing mode is that address generation is a distinctly different operation from regular addition. As mentioned above, the `LOAD ADDRESS` instruction performs a 31-bit addition operation, adding the values of base and index register and an immediate displacement, and returning a 31-bit value. The `ADD` instruction, in contrast, performs a full 32-bit addition operation. The decision whether to use `ADD` or `LOAD ADDRESS` needs to take into account a number of issues:

- We *must not* use `LOAD ADDRESS` to perform integer addition, as the high bit of the result is not computed.
- Where the result of an addition operation is used as address, we can use `LOAD ADDRESS`, and it is in fact often the preferred method to minimize pipeline stalls.
- Some passes of the compiler (reload) insert address computation operations into the instruction stream, making the implicit assumption that they do not clobber the condition code. We *must* use `LOAD ADDRESS` in these cases.
- In some cases, in particular when computing local addresses in position-independent code (see above), we rely on the property that `LOAD ADDRESS` clears the high bit, so we must not use regular addition instead.

This has been a problematic area during the development of the S/390 back end; we have tried various ways of simultaneously meeting all these requirements, not always completely successfully. As an example for the difficulties involved, consider the question whether there should be an `LA` pattern that accepts all RTL instructions of the form `(set (reg) (plus (reg) (reg)))`. If this pattern exists, there is the danger that it might be incorrectly used to implement an integer addition. If it does not exist, there is the danger of reload failures as reload will create such instructions anyway. The current S/390 back end tries to solve this as follows:

- The `add` instruction patterns accept insns that explicitly clobber the condition code.

- The `la` instruction patterns accept insns that do not clobber the condition code, provided that it is safe to assume the result is being used as an address. This assumption can be made if one of the registers involved is the literal pool base register, the global offset table base register, or is known to point into the stack frame (stack register, frame register, argument pointer register etc.). The instruction will also accept addresses using an `UNSPEC` to enforce clearing the high bit.

- A second set of `forced_la` patterns accept all syntactically valid load address insns, without employing the sanity check mentioned above. Those use a special pattern that will never be accidentally generated by other parts of the compiler (e.g. combine), so that those patterns will only match in case they were explicitly generated by the S/390 back end.

- When reload tries to load a `plus` expression that would not be accepted by a regular `la` pattern, this is handled via the secondary input reload mechanism. This means that the `reload_insi` expander is called, which in turn will compute the address using `forced_la` patterns if necessary. That way, reload will never fall back to generating add operations by itself.

- To optimize for using `LA` where possible, a set of peephole2 patterns tries to transform `add` instructions into `la` instructions. This is only done when considered profitable.

A completely different option to solve the 31-bit addressing mode problems might be to employ the `PSImode` mechanism to explicitly represent a 31-bit data type. However, we have tried this solution and found that it typically generated less efficient code due to superfluous `SImode <-> PSImode` conversions inserted at various points by the middle end. Improving the `PSImode` support might make this option viable at some point in the future, though.

3.3 Instruction specific address formats

A fundamental assumption of GCC, in particular the reload pass, used to be that memory addresses are represented in the same format in all instructions. This means that if a particular RTL expression represents a valid address for one instruction, it is supposed to be valid for *all* other instructions as well. The most important place where this assumption is made is the `find_reloads` routine. This routine is supposed to check whether an RTL instruction matches the constraints imposed by the insn pattern, and if it doesn't, determine the most efficient way to modify the instruction stream by inserting additional reload insns to correct the problem. In doing so, `find_reloads` first tries to make sure that all memory addresses mentioned in the instruction are valid. This pass is performed in the same way for all instructions, and does not even look at the constraint string. This means there is no way to impose different conditions as to whether a memory address is valid or not, depending on which instruction is involved.

Unfortunately, the S/390 architecture uses two different formats to specify memory addresses in instructions. The most general address format allows to to specify a base register, an index register, and a displacement (in the range of 0–4095). These are added up to compute the effective address. Some other instructions, however, do not allow the use of an index register; instead, they compute the effective address simply as the sum of a base register and the displacement. (The two formats are commonly called X and S instruction operands, respectively.) However, the back end has only two choices when asked to validate an address RTX: either to never accept addresses with index register, or to always accept them. The first option causes very inefficient code to be generated, while the second option can potentially cause invalid operands for S-type instructions to be produced.

We have tried various ways of coping with this problem, but with limited success. It is possible to try to avoid invalid S-operands by checking for their presence in the instruction predicate of affected instruction patterns. However, this is not reliable, as an address operand that initially does not use an index register can be modified into one that does by the reload pass, e.g. due to register elimination or displacement overflow. While we could in addition to the predicate use a constraint letter to check for valid S-operands, this does not solve the problem: if a non-standard constraint does not match, reload will not know how to fix the problem, causing compilation to abort. We were able to overcome this by relying on undocumented—and arguably incorrect—behaviour of reload when interpreting the 'o' constraint; but this hack was not only fragile, it also didn't allow full flexibility in generating efficient code.

We finally solved this issue by introducing two new features to the reload pass, starting with GCC version 3.3—the `EXTRA_MEMORY_CONSTRAINT` and `EXTRA_ADDRESS_CONSTRAINT` target macros. These were inspired by the way reload was able to handle *offsettable* memory constraints. A memory operand is called offsettable, if it stays a valid memory operand when a small additional displacement is added to the address, so that every byte of the object comprising the operand can be addressed. As an example, the RTX

```
(mem:DI (plus:SI (reg:SI 1 %r1)
                 (const_int 4092)))
```

is a valid memory operand on S/390, but it is not an offsettable operand, because only the initial four bytes of the `DImode` operand are addressable before the displacment exceeds the maximum value of 4095. In some cases, instructions cannot accept non-offsettable operands, and GCC allows to specifc this using the 'o' constraint letter. If, after reload has performed all required modifications, a memory address marked with that constraint turns out to be non-offsettable, reload will generate a load-address operation to reload the address into a single register; this register can then be used as offsettable memory operand.

The `EXTRA_MEMORY_CONSTRAINT` target macro now allows the back end to specify other classes of memory operands that require similar treatment by reload. By declaring that a constraint letter describes an extra memory constraint, the back end promises that `EXTRA_CONSTRAINT`, when called to verify whether an expression satisfies this constraint, will:

- accept only memory operands, and
- accept all memory operands whose address consists of one single base register.

This allows the reload pass to handle such operands correctly: if a memory operand does not pass the `EXTRA_CONSTRAINT` check, reload is able to fix the problem by loading the address into a base register. Similarly, the `EXTRA_ADDRESS_CONSTRAINT` target macro allows the back end to define constraints that work like the standard 'p' constraint to denote address operands, but accepts only a subset of all valid addresses (again including all those that consist of solely a base register so that reload can fix the operand up if required).

The `EXTRA_MEMORY_CONSTRAINT` macro is used by the S/390 back end to define the 'Q' constraint to handle S-operand instructions; this allows the use of these instructions without abusing reload, and also provides flexibility to mix S-operand instructions with others in the same instruction pattern, choosing the best alternative depending on the specific situation. The `EXTRA_ADDRESS_CONSTRAINT` macro could be used by the S/390 back end to implement the full range of options to specify the count operand for shift instructions (this is not currently implemented yet, however).

4 Performance considerations

The previous section described issues relating to correctness of the generated code which required special handling. However, for GCC to be a competitive compiler on the zSeries platform, we need to not just generate correct, but also efficient code. This section details two areas where we found we could achieve significant performance benefits by exploiting specific features of the zSeries architecture: condition code handling and instruction scheduling.

4.1 Condition code handling

The S/390 architecture uses a *condition code* to implement conditional branches. The condition code consists of two bits stored in the program status word. Various arithmetical, logical, and comparison instructions set the condition code, while branch instructions make use of it to decide whether the branch is to be taken or not. As opposed to many other platforms, the S/390 condition code is not composed of single bits with specific semantics. Instead, the two bits of the condition code combine to represent a condition code value in the range 0–3. Branch instructions use a 4-bit branch condition mask to decide whether branching is performed. The current condition code selects one of the four mask bits, and if this bit is one, the branch is taken. The relationship between the condition code value and the mask position is given by the following table:

Condition Code	Mask Position Value
0	8
1	4
2	2
3	1

For example, the instruction `bcr 12,%r1` branches to the address given in register `%r1` if the current condition code is either 0 or 1. (The GNU assembler also accepts mnemonics instead of explicit mask values; as this branch typically represents a *less-or-equal* decision, it can equivalently be written as `bler %r1`.)

However, the numerical values 0–3 the condition code can assume have no fixed meaning. Instead, every instruction that sets the condition code is free to define the semantics of the condition code values it may set. In early versions of the S/390 back end we therefore used only the condition codes set by explicit comparison instructions (which are very regular), and completely ignored that other instructions may set the condition code as side effect of some other operation. This works, but can obviously cause code to be generated that is significantly less efficient. In particular, some important instructions the S/390 architecture provides (e.g. `TEST UNDER MASK`) could not be exploited at all.

To improve this situation, we have rewritten the condition code handling parts of the S/390 back end to use an explicit `CCmode` register to represent the condition code (instead of using `cc0`). The various different semantics that instructions can impose on the condition code values are represented via different machine modes of that register. The following list tries to give an overview of the typical uses of the condition code:

- Comparison operations (signed)

0	Operands equal
1	First operand low
2	First operand high
3	Operands unordered (floating point)

This condition code semantics is represented by the `CCSmode` mode. It is used by instructions like `COMPARE`; some other instructions (e.g. `LOAD AND TEST`, `SHIFT RIGHT SINGLE`) set their condition code according to this mode as well, assuming an implied comparison of their single operand against zero.

- Logical comparison operations (unsigned)

0	Operands equal
1	First operand low
2	First operand high
3	n/a

This condition code semantics is represented by the `CCUmode` mode. It is used by the `COMPARE LOGICAL` family of instructions.

- Arithmetical operations

0	Result zero; no overflow
1	Result less than zero; no overflow
2	Result greater than zero; no overflow
3	Overflow

This is used by the `ADD` and `SUBTRACT` instructions. Unfortunately, due to the fact that the case of signed arithmetic overflow is signalled via condition code 3, and in that case no comparison of the result against zero is performed, in most cases we cannot use the condition code set by those instructions. However, if one of the operands is a compile-time immediate constant, we may be able to determine at compile-time that if the operation overflows, the result *must* always be greater or less than zero, respectively. Those situations are represented by the `CCAPmode` and `CCANmode` modes. (Note that some languages, like C, guarantee that arithmetic on signed data types must not overflow. Unfortunately, this information is lost at the RTL level. Having some means to pass this fact to the back end would enable us to make use of the `ADD` condition code in many more cases.)

- Logical operations

0	Result zero; no carry
1	Result not zero; no carry
2	Result zero; carry
3	Result not zero; carry

This is used by `ADD LOGICAL` and in slightly modified form by `SUBTRACT LOGICAL`; we represent these cases by the `CCL1mode` and `CCL2mode` modes. We use the logical variants of the add and subtract operations in cases where the result of the operation is compared against zero, and we are not sure whether overflow happens. They can also be used to implement carry propagation for multi-word additions.

- Zero test

0	Result zero
1	Result not zero
2	n/a
3	n/a

The logical operations (`AND`, `OR`, `EXCLUSIVE OR`) use these condition code semantics, which we represent by `CCTmode`. What is important here is that some of the condition code modes mentioned above can also be used to implement a test against zero (e.g. `CCSmode, CCUmode`). We therefore implement such tests using a virtual condition code mode `CCZmode` that is allowed to match against all such modes, using a semantics of condition code 0 if result equals zero, and condition code nonzero if the result is nonzero.

The condition codes described above are all used by a number of different instructions, and share a certain amount of regularity. However, other instructions use the condition code in completely different ways. As an example we describe here an important instruction of the S/390 architecture, `TEST UNDER MASK LOW`, and how we can make use of this instruction within the GCC framework. `TEST UNDER MASK LOW` takes the low 16 bits of a register operand and compares them bit-for-bit against a mask provided as immediate operand. The sole effect of the instruction is to set the condition code, depending on whether the operand bits selected by the mask are ones or zeros:

0	Selected bits all zeros; or mask bit all zeros
1	Selected bits mixed, and leftmost is zero
2	Selected bits mixed, and leftmost is one
3	Selected bits all ones

This instruction is very useful to generate efficient code for a number of frequently used bit-test operations. The following `if` statement, for example:

```
if ((flags & 0x80) &&
    !(flags & 0x4))
```

can be translated into a single `TEST UNDER MASK LOW` operation followed by a conditional branch:

```
# Mask selects both 0x80 and
# 0x04 bits for testing
tml  %r1,0x84
# Branch if leftmost bit is one,
# and the other zero
brc  2,.Lxxx
```

Starting with GCC 3.3, the S/390 back end is in fact able to generate this optimal code sequence. This is made possible by the fact that the combiner pass notices the two subexpressions of the `if` clause can be combined into

```
if ((flags & 0x84) == 0x80)
```

The S/390 back end now uses the `SELECT_CC_MODE` macro to inform combine that it is possible to implement this particular comparison operation using the `CCT2mode` mode, causing the following (simplified) instruction sequence to be emitted:

```
(set (reg:CCT2 33 %cc)
     (compare:CCT2
       (and:SI (reg/v:SI 40)
               (const_int 132 [0x84]))
       (const_int 128 [0x80])))

(set (pc)
     (if_then_else
       (ne (reg:CCT2 33 %cc)
           (const_int 0 [0x0]))
       (label_ref 18)
       (pc)))
```

These in turn later generate the assembler code shown above. Note that the use of `CCT2mode` causes the branch instruction to use condition code 2 for equality (and all other condition codes for inequality); this is very different from how most other branches are handled.

Overall, the `CCmode` facilities of the GCC middle end allow to make use of the S/390 condition codes in many important cases; no changes outside the S/390 back end were necessary to exploit them. However, we have noticed some areas where common code changes would be required to further improve the generated code. One of these is to allow a condition code computed by one instruction to be reused across multiple branches; the sequence

```
if (x == 5)
  ...
else if (x < 5)
  ...
```

currently performs two distinct comparison operations, although the optimal implementation would use a single `COMPARE` to set the condition code, followed by two branch instructions evaluating it.

4.2 Instruction scheduling

The time required to run a certain program depends on the number of instructions and the time each specific instruction takes. Besides that, in most modern implementations of computer architectures, dealing with a pipelined and/or superscalar processor implementation, the cycles an instruction takes as part of an instruction stream depends heavily on the issue order. For some architectures (e.g. VLIW) an inappropriate scheduling of instructions will lead to a significant performance decrease.

Also on the recent z900 machines, some of the single-cycle instructions will in fact take from 1 to 5 cycles, depending on the order this instruction is issued within an instruction stream. The reason for this can easily be seen if we take a close look at the single-issue pipeline all instructions are executed on. (See [5] for a more detailed description of the z900 pipeline.)

After instruction fetching, the instruction pipeline consists of 6 stages. This pipeline is designed so as to ensure that register-memory (RX) instructions perform the best way possible.

DC Decode instruction, latch registers for address generation.

AA Address generation, by adding base, index register and displacement from instruction text.

C1 Cache access, TLB access.

C2 Send memory data to execution unit.

E1 Execute.

WR Writeback result to register file.

Regardless whether an instruction actually uses a memory operand or not, latching of base and index registers is done in the decode stage. Likewise, the address generation stage as well as the C1 and C2 stages are used for all instructions, even though they would be required for memory operands only. Together with the fast L1 data cache, this enables register-memory instructions to be as fast as register-register instructions.

Due to the single cycle E1 stage for most simple instruction, true data dependency does not cause a pipeline stall. This leads to a theoretical cpi of 1 for most compiler generated instructions, assuming an infinite cache. Also, since this pipeline is short, the penalty for mispredicted branches is comparatively small.

The main instruction-issue related problem left by this design is the address generation interlock (AGI). If a register used in the AA stage (e.g. base register) is changed in an instruction shortly before, the pipeline will be stalled for up to 4 cycles. This is due to the fact that the AA stage needs to wait for the WR stage to update the register needed.

(Please see Figure 1.)

This AGI lets most applications suffer a performance degradation in the double-digit percentage range. If we look at code examples like the PLT code generated for ELF shared libraries, the impact is even bigger. Over the last generations of S/390 systems attempts to reduce this impact led to building certain kinds of bypasses into the pipeline. Especially the *load* and *load address* type instructions, which generate all their side-effects in the early stages of the pipeline and which are frequently used in pointer intensive code, got those bypasses. The result of a *load address* type instruction is generated in the AA stage and ready after C1, and can be bypassed with a 1 cycle delay to the AA stage of a directly following instruction.

```
                 0  1  2  3  4  5  6  7  8  9  10 11
ar r2,r3         DC AA C1 C2 E1 WR
l  r2,0(0,r2)       DC             AA C1 C2 E1 WR
ar r4,r2                           DC AA C1 C2 E1 WR
```

Figure 1: Address Generation Interlock, first example

```
                 0  1  2  3  4  5  6  7  8  9  10 11
la r2,0(r2,r3)   DC AA C1 C2 E1 WR
l  r2,0(0,r2)       DC    AA C1 C2 E1 WR
ar r4,r2                  DC AA C1 C2 E1 WR
```

Figure 2: Address Generation Interlock, second example

(Please see Figure 2.)

The result of a *load* type instruction is ready after the C2 stage and can be bypassed with a 2 cycle delay to the AA stage of a directly following instruction.

(Please see Figure 3.)

All other instructions suffer a 4 cycle penalty if setter and user are issued back to back. To avoid this, we use in the recent GCC implementation the new DFA based scheduler.

To describe the behavior of the pipeline, we only need to define the last two stages. Down below we shortly show part of description of the z900 pipeline.

```
(define_automaton "z_ipu")
(define_cpu_unit "z_e1"   "z_ipu")
(define_cpu_unit "z_wr"   "z_ipu")

(define_insn_reservation "z_la" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "type" "la"))
  "z_e1,z_wr")

(define_insn_reservation "z_load" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "type" "load"))
  "z_e1,z_wr")

(define_insn_reservation "z_int" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "atype" "reg"))
  "z_e1,z_wr")

(define_insn_reservation "z_agen" 1
  (and (eq_attr "cpu" "z900")
       (eq_attr "atype" "agen"))
  "z_e1,z_wr")
```

The 4-cycle hazard of the pipeline due to AGI, the 1-cycle bypass for the *load address* type instructions and the 2-cycle bypass for *load* type instructions are described using the `define_bypass` construct.

```
(define_bypass 5 "z_int,z_agen"
       "z_agen,z_la,z_load" "s390_agen_dep_p")

(define_bypass 3 "z_load"
       "z_agen,z_la,z_load" "s390_agen_dep_p")

(define_bypass 2 "z_la"
       "z_agen,z_la,z_load" "s390_agen_dep_p")
```

With all this in place, GCC does a good job scheduling within a basic block. The places where we still see for certain code a non-optimal scheduling are as follows:

At the beginning of a basic block, the state of the DFA is reset. With GCC 3.4, the second scheduling pass is placed after basic block reordering. Since the reordering will lead to a high probability that a basic block is entered from the immediately preceding basic block,

```
                  0  1  2  3  4  5  6  7  8  9  10 11
l  r2,0(0,r3)     DC AA C1 C2 E1 WR
l  r2,0(0,r3)        DC       AA C1 C2 E1 WR
ar r4,r2                      DC AA C1 C2 E1 WR
```

Figure 3: Address Generation Interlock, third example

this could be used to improve scheduling. Instead of resetting the state at the beginning of the basic block, the state from the end of the last basic block scheduled could be used as initial state.

This uncovers another problem with the current way the DFA is defined. The `define_bypass` mechanism only influences `insn_cost`, which is used to set up the priority a insn is scheduled with. Also `insn_cost` is used to find out when a insn is ready, depending on the instructions already scheduled in the current basic block. However, this information is not actually part of the state of the DFA itself, and due to that the detection of AGI hazards cannot be achieved solely by looking at this state.

If GCC will use more and more of the DFA-based algorithms for scheduling, like global scheduling, the DFA should be built to model all resources. In our specific case, in order to detect AGIs, this needs to include the general register file. To model the AGI behaviour, we need to define a RR type instruction allocating the source register in the E1 stage and allocating the destination register in the AA, C1, C2, E1, WR stages. A RX type instruction allocates the address registers in the AA stage, the source register in the E1 stage and the destination register in the AA, C1, C2, E1, WR stages. In case of a *load* type instruction the destination register is only allocated in the AA, C1 and C2 stage, for a *load address* type instruction in the AA and C1 stage. Having this in place, the DFA would be sufficient for detecting the AGI hazard.

This all would need some kind of new syntax, in order to refer to the registers an instruction is using. Also, it would definitely not work before register allocation, since the number of states and transition could not be handled. Even after register allocation, it remains to be seen whether the the number of states and transisiton is managable. In our case, each instruction may use up to 16 registers, and will use up to two for addressing.

5 Conclusion

GCC on the IBM mainframe is a mature compiler that is in widespread use as the system compiler for all Linux on zSeries distributions. The efficiency of the generated code is competitive with other compilers for our platform.

However, there is still room for improvement. We will continue to work on the S/390 back end in order to fully exploit all features the architecture provides. We also remain committed to add support for future generations of the zSeries processor as soon as those become available.

References

[1] *ESA/390 Principles of Operation*, IBM Document Number SA22-7201-07, 2000. `http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9ar007`

[2] *z/Architecture Principles of Operation*, IBM Document Number SA22-7832-01,

2000.
`http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dz9zr001`

[3] *LINUX for S/390 ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00, 2001.
`http://oss.software.ibm.com/linux390/docu/l390abi0.pdf`

[4] *LINUX for zSeries ELF Application Binary Interface Supplement*, IBM Document Number LNUX-1107-00, 2001.
`http://oss.software.ibm.com/linux390/docu/lzsabi0.pdf`

[5] E.M. Schwarz et al. *The microarchitecture of the IBM eServer z900 processor*, IBM Journal of Research and Development Vol. 46 No 4/5, 2002.

Building and Using a Cross Development Tool Chain

Robert Schiele
`rschiele@uni-mannheim.de`

Abstract

When building ready-to-run applications from source, a compiler is not sufficient, but libraries, an assembler, a linker, and eventually some other tools are also needed. We call the whole set of these tools a development tool chain. Building a native tool chain to build applications for the compiler's platform is well documented and supported. As clusters become more and more widespread, it becomes interesting for developers to use the enormous CPU power of such a cluster to build their applications for various platforms by using cross development tool chains.

We describe how a development tool chain is structured and which steps have to be taken by its parts to build an executable from source. We also evaluate whether the characteristics of each step imply that a special version of this tool is needed for the cross development tool chain. Furthermore, we explain what has to be done to build a complete cross development tool chain. This is more involved than building a native tool chain, because intrinsic dependencies that exist between some parts of the tool chain must be explicitly resolved. Finally, we also show how such a cross compiler is used and how it can be integrated into a build environment on a heterogeneous Linux/Unix cluster.

1 Motivation

1.1 Unix Standard System Installations

Although in recent years some Unix vendors stopped shipping development tools with their operating systems, it is still quite common on most systems to have a C compiler, an assembler and a linker installed. Often system administrators use these tools to compile applications for their systems when binary packages are not available for their platform or when the setup of the binary package is not applicable to their local setup. For such scenarios, the system compiler is quite sufficient.

1.2 Development Usage

Although this so-called system compiler can also be used by a software developer to build the product he is developing on and is often done, this is in most cases not the best solution.

There are several reasons for not using the system compiler for development:

- In development you often have a large number of development machines that can be used in a compiler cluster to speed up compilation. Tools for this purpose are available, as `distcc` by Martin Pool, `ppmake` from Stephan Zimmermann with some improvements from my side, or many other tools that do similar things. The problem is that when using the system compiler, you can only use other development machines that are of the same architecture and operating system because you cannot mix up object files generated for different platforms.

- As a developer, you normally want to support multiple platforms, but in most cases, you have a large number of fast machines for one platform, but only a few slow machines for another one. If you used only the system compiler in that case, you would end up in long compilation times for those platforms where you only have a few slow machines.

- Last but not least, you often also want to build for a different `glibc` release etc. than the one installed on your system for compatibility reasons. This is also not possible for all cases with a system compiler pre-configured for your system's binutils release and other system specific parameters.

1.3 Compiling for a Foreign Platform

We can solve all those problems by making clear to ourselves that a compiler does not necessarily have to build binaries for the platform it is running on. A compiler where this is the case, like the system compiler, is called a native compiler. Otherwise, the compiler is called a cross compiler.

We also need a cross compiler for bootstrapping a new platform that does not already ship a compiler to bootstrap a system with. But this cannot really be a motivation for this paper, as people that bootstrap systems most likely do not need the information contained in this paper to build a cross development tool chain.

In the following section we will show some basic principles of a development toolchain, how the single parts work and whether their characteristics require them to be handled specially when used in a cross development tool chain. In section 3, we will show what must be done to build a complete cross development tool chain and what are some tricks to work around some problems. In section 4, we show how to integrate the cross development tool chain into build systems to gain a more efficient development tool chain. Finally, we will find some conclusions on our thoughts in the last section.

2 How a Compiler Works

To understand how a compiler works and thus what we have to set up for a cross compiler, we need to have a look at the C development tool chain. This is normally not a monolithic tool that is fed by C sources and produces executables, but consists of a chain of tools, where each of these tools executes a specific transformation. An overview of this tool chain can be found in Figure 1. In the following, I will show those parts and explain what they do.

This section is not intended to provide a complete overview on compiler technology, but does only discuss some principles that help us to understand why cross development tool chains work the way they do. If you would like to have some detailed information about compiler technology, I recommend reading the so-called Dragon book [ASU86].

2.1 The C Preprocessor

The C preprocessor is quite a simple tool. It just removes all comments from the source code and processes all commands that have a hash mark (#) on the first column of any lines. This means, for example, it includes header files at the position where we placed `#include` directives, it does conditional compiling on behalf of `#if...` direc-

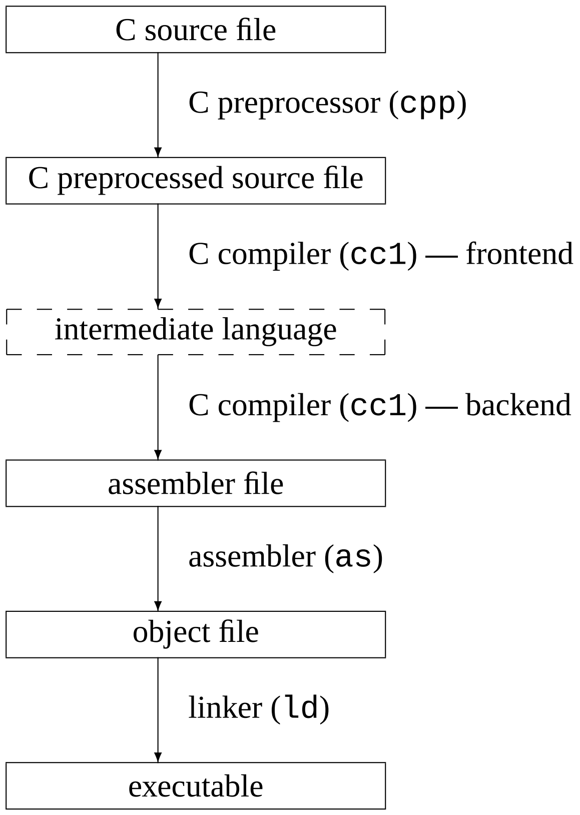

Figure 1: tool chain

tives and expands all macros used within the C source code. The output of the C preprocessor is again C source code, but without comments and without any preprocessor directive.

Note that most programming languages other than C do not have a preprocessor. It should be noted that preprocessor directives and especially macros make some hackers to produce really ugly code, but in general, it is a quite useful tool.

It can easily be seen that the C preprocessor itself should not be platform dependent, as it is a simple C-to-C-translator. But in fact, on most systems the preprocessor defines platform-specific macros like e.g. `__i386__` on an ia32 architecture, and it must be configured to include the correct platform specific header files. Apart from that, in many compilers the preprocessor is integrated into the actual C compiler for performance reasons and to solve some data flow issues. Because of these reasons, the C preprocessor is actually not really platform-independent.

2.2 The C Compiler

The actual C compiler is responsible for transforming the preprocessed C source code to assembler code that can be further processed by the assembler tool. Some compilers have an integrated assembler, i.e. they bypass the assembler source code, but compile directly to binary object code.

We can divide the compiler into a front end and a back end, but you should note that in most cases these two parts are integrated into one tool.

2.2.1 The Compiler Front End

The front end is responsible for transforming the C source code to some proprietary intermediate language. This intermediate language should be ideally designed to be independent of both the source language and the destination platform to allow easy replacements of the front end and the back end. Because of that reason the front end is independent of the destination platform.

2.2.2 The Compiler Back End

The back end does the translation of the intermediate language representation to assembler code. As the assembler code is obviously platform-dependent, the back end is as well.

This results in the fact that although the front end is platform-independent, the whole C compiler is not because it is an integration of both the front end and the back end, where the latter is not independent.

2.3 The Assembler

The assembler is the tool that translates assembler code to relocatable binary object code. Relocatable means that there are no absolute addresses built into the object code, but instead, if an absolute address is necessary, there are markers that will be replaced with the actual address by the linker. The object code files include a table of exported symbols that can be used by other object files, and undefined symbols that require definition in a different object file. As both the input and the output of this tool is platform-specific, the assembler obviously depends on the platform it should generate code for.

2.4 The Linker

The linker can be considered the final part in the development tool chain. It puts all binary object code files together to one file, replacing the markers by absolute addresses and linking function calls or symbol access to other object files to the actual definition of the symbol. Some of those object files might be fetched from external libraries, for example the C library. We do not explain how linking to shared objects works, as it just makes things a bit more complicated, but does not make a real difference on the principles that are necessary to understand the development tool chain. The result of this tool is normally an executable. For the same reasons as with the assembler, the linker clearly depends on the destination platform.

More detailed information on the principles of linkers can be found in [Lev00].

3 Building the tool chain

As we now have some basic knowledge about how a development tool chain is structured, we can start building our cross development tool chain. We can find both the C preprocessor and compiler in the `gcc` package [GCC], which is the most commonly used compiler for Linux and for many other Unix and Unix-like platforms.

We use the assembler and linker from the GNU binutils package [Bin]. As an alternative linker for ELF platforms, there is the one from the elfutils by Ulrich Drepper, but this one is in a very early point in its life cycle, and I would not currently recommend using these tools for a productive environment. For the GNU assembler, there are also various alternatives available, but as changing an assembler does only a straightforward translation job and thus, no improvements of the results are to be expected, it is not worth integrating another assembler into the tool chain.

These are all tools for our tool chain, but we are still missing something: As every C application uses functions from the C library, we need a C library for the destination platform. We will use `glibc` [Gli] here. If we wanted to link our applications to additional libraries, we would need them also, but we will skip this part here. The essential support libraries for other `gcc` supported languages like C++ are shipped and thus built with `gcc` anyway.

The following examples are for building a cross development tool chain for a Linux system with `glibc` on a PowerPC. The cross compiler is built and will run itself on a Linux system on an ia32 architecture processor. Although something might be different for other system combinations, the principles are the same.

3.1 The Binutils

The simplest thing to start with is the binutils package because they neither depend on the `gcc` compiler nor on the `glibc` of the destination platform. And we need them anyway when we want to build object files for the destination platform, which is obviously done for the `glibc`, but even `gcc` provides a library with some primitive functionality for some operations that are too complex for the destination platform processor to execute directly.

From a global point of view we have dependencies between the three packages as shown in figure 2.

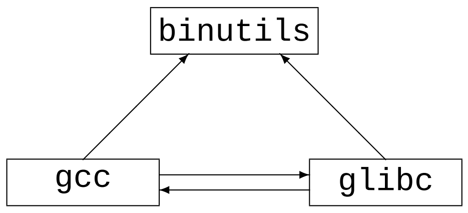

Figure 2: Dependencies between the packages

So we fetch a binutils package, unpack it and create a build directory somewhere—it is recommended not to build in the source directory—where we then call

```
../binutils-2.13.90.0.20/configure
    --prefix=/local/cross
    --enable-shared
    --host=i486-suse-linux
    --target=powerpc-linux
```

We set the prefix to the directory we want the cross development tool chain to be installed into, we enable shared object support, as we want that on current systems and we tell `configure` the host platform, i.e. the platform the tools are running on later, and the target platform, i.e. the platform for which code should be generated by the tools later. Afterwards, we run a quick `make`, `make install`, and the binutils are done.

As long as there is not a hard bug in the used binutils package, this step is quite unlikely to fail, as there are no dependencies to other tools of the tool chain we build. For the following parts we should expect some trouble because of intrinsic dependencies between `gcc` and `glibc`.

From this point on, we should add the `bin/` directory from our installation directory into `$PATH`, as the following steps will need the tools installed here.

3.2 A Simple C Compiler

Now we run into the ugly part of the story: We need a C library. To build it, we obviously need a C compiler. The problem is now that `gcc` ships with a library (`libgcc`) that in some configurations depends on parts of the C library.

For this reason, I recommend building the C library and all the other libraries on a native system and copying the binaries to the cross compiler tool chain or using pre-built binaries, if possible. If you build a cross compiler that compiles code for a commercial platform like Solaris, you have to do so anyway, as you normally do not have the option to compile the Solaris `libc` on your own. If you decide to build the C library with your cross compiler, continue here, otherwise skip to building the full-featured compiler.

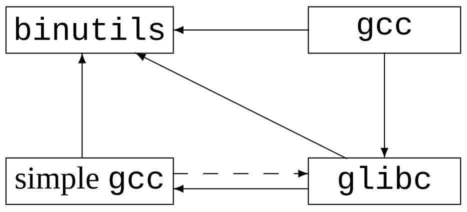

Figure 3: Dependencies with simple C compiler

We cannot build a full-featured compiler now, as the runtime libraries obviously depend on the C library. This cycle in the dependency graph can be seen in figure 2. We can resolve this cycle by introducing a simple C compiler that does not ship these additional libraries, so that we get dependencies as shown in figure 3. But because of the reason mentioned above, for most configurations we cannot even build a simple C only compiler. That means we can build the compiler itself, but the support libraries might fail. So we just start by doing

```
CFLAGS="-O2 -Dinhibit_libc"
    ../gcc-3.2.3/configure
    --enable-languages=c
    --prefix=/local/cross
    --target=powerpc-linux
    --disable-nls
    --disable-multilib
    --disable-shared
    --enable-threads=single
```

and then starting the actual build with `make`. The `configure` command disables just everything that is not absolutely necessary for building the C library in order to limit the possible problems to a minimum amount. Sometimes it also helps to set the `inhibit_libc` macro to tell the compiler that there is no libc yet, so we add this also. In case the build completes without an error, we are lucky and can just continue with building the C library after doing a `make install` before.

Otherwise, we must install the incomplete compiler. In this case, the compiler will most likely not be sufficient to build all parts of the C library, but it should be sufficient to build the major parts of it, and with those we might be able to recompile a complete simple C compiler. We have to iterate between building this compiler and the C library, until at least the C library is complete.

The installation of an incomplete package can be either done by manually copying the built files to the destination directory, by removing the failing parts from the makefiles and continuing the build afterwards, or by just touching the files that fail to build. The last option forces `make` to silently build and install corrupted libraries, but if we have this in mind, this is not really problematic, as we can just rebuild the whole thing later and thus replace the broken parts with sane ones.

The simplest way of installing an incomplete compiler when using GNU `make` is calling `make` and `make install` with the additional parameter `-k` so that `make` automatically continues on errors. This will then just skip the failing parts, i.e. the support libraries.

3.3 The C Library

After having built a simple C compiler, we can build the C library. It has already been said that this might be necessary to be part of an iterative build process together with the compiler itself.

To build the `glibc` we also need some kernel headers, so we unpack the kernel sources somewhere and do some basic configuration by typing

```
make ARCH=ppc symlinks
    include/linux/version.h
```

Now we configure by

```
../glibc-2.3.2/configure
   --host=powerpc-linux
   --build=i486-suse-linux
   --prefix=
    /local/cross/powerpc-linux
   --with-headers=
       /local/linux/include
   --disable-profile
   --enable-add-ons
```

and do the usual `make` and `make install` stuff.

Note that the `-host` parameter is different here to the tools, as the `glibc` should actually run on the target platform and not, like the tools, on the build host. The `-prefix` is also different, as the `glibc` has to be placed into the target specific subdirectory within the installation directory, and not directly into the installation directory. Additionally, we have to tell `configure` where to find the kernel headers and that we do not need profiling support, but we want the add-ons like `linuxthreads` enabled.

In case that building the full `glibc` fails because building the C Compiler was incomplete before, the same hints for installing the incomplete library apply that where explained for the incomplete compiler. Additionally, it might help to touch the file `powerpc-linux/include/gnu/stubs.h` within the installation directory, in case it does not exist yet. This file does not contain important information for building the simple C compiler, but for some platforms it is just necessary to be there because other files used during the build include it.

After installation of the `glibc` (even the incomplete one), we also have to install the kernel headers manually by copying `include/linux` to `powerpc-linux/include/linux` within the installation directory and `include/asm-ppc` to `powerpc-linux/include/asm`. The latest kernels also want `include/asm-generic` to be copied to `powerpc-linux/include/asm-generic`. Other systems than Linux might have similar requirements.

3.4 A Full-featured Compiler

After we have a complete C library, we can build the full-featured compiler. That means we do now again a rebuild of the compiler, but with all languages and runtime libraries we want to have included.

With a complete C library, this would be no problem any more, so we should manage to do this by just typing

```
../gcc-3.2.3/configure
   --enable-languages=
       c,c++,f77,objc
   --prefix=/local/cross
   --disable-libgcj
   --with-gxx-include-dir=
       /local/cross/include/g++
   --with-system-zlib
   --enable-shared
   --enable-__cxa_atexit
   --target=powerpc-linux
```

and again doing the build and installation by `make` and `make install`.

4 Using the Tool Chain on a Cluster

We now have a full-featured cross development tool chain. We can use these tools by just putting the `bin/` path where we installed them to the system's search path and calling them by the tool name with the platform name prefixed, e.g. for calling `gcc` as a cross compiler for platform `powerpc-linux`, we call `powerpc-linux-gcc`. The tools should behave in the same way the native tools on the host system do, except that they produce code for a different platform.

But our plan was to use the cross compiler on a cluster to speed up compilation of large applications. There are various methods for doing so. In the following we will show two of them.

4.1 Using a Parallel Virtual Machine (PVM)

We receive most scalability by dispatching all jobs that produce some workload to the nodes in the cluster. `make` is a wonderful tool to do so. A long time ago, Stephan Zimmermann implemented a tool called `ppmake` that behaved like a simple shell that distributed the commands to execute on the nodes of a cluster based on PVM. He stopped the development of the tool in 1997. As I wanted to have some improvements for the tool, I agreed with him to put the tool under GPL and started to implement some improvements. You can fetch the current development state from [ppm], but note that the documentation is really out of date and that I also stopped further development for several reasons.

If you want to use this tool, you just have to fetch the package, build it and tell `make` to use this shell instead of the standard `/bin/sh` shell by setting the `make` variable `SHELL` to the `ppmake` executable. Obviously you have to set up a PVM cluster before make this work. Information on how to set up a PVM cluster can be found at [PVMa]. To gain something from your cluster you should also do parallel builds by specifying the parameter `-j` on the `make` command line.

For example, if you had a cluster consisting of 42 nodes configured in your PVM software and `ppmake` installed in `/usr/`, you call

```
make -j 42
    SHELL=/usr/bin/ppmconnect
    ...
```

instead of just

```
make ...
```

CVS head revision replaced `ppmconnect` by the integrated binary `ppmake`.

There is also a script provided in the package that does most of these things automatically, but I do not like the way this script handles the process, so I do not use it personally, and such it is a bit out of date recently.

Note that there is a similar project [PVMb] by Jean Labrousse ongoing which aims at in integrating a similar functionality directly into GNU `make`. You may want to consider looking at this project also.

You should note that it is necessary for this approach that all files used in the build process are available on the whole cluster within a homogenous file system structure, for example by placing them on a NFS server and mounting on all nodes at the same place. Additionally, it is necessary that all commands used within the makefiles behave in the same way on all nodes of the cluster. Otherwise, you will get random results, which is most likely not what you want. This means you should always call the platform-specific compiler explicitly, e.g. by `powerpc-linux-gcc` instead of `gcc`, and the same releases of the compiler, the linker and the libraries should be installed on all nodes.

4.2 Using with `distcc`

The biggest disadvantage of the method described above is that it relies on central file storage and on identical library installations on all nodes. You can prevent these constraints at the cost of limiting the amount of workload that will be distributed among the nodes in the cluster to the compilation and assembling step. Preprocessing and linking is done directly on the system where the build process was started and thus not parallelized. Only compilation jobs are parallelized, all other commands are directly executed on the system, where the build process was invoked. Although this limits the amount of workload that really runs in parallel, this is in most cases not a real problem, as most build processes spend most of their time with compilation anyway.

The advantage of this approach is that you only need to have the cross compiler and assembler on each node. Include files and libraries are necessary only on the system on which the build is invoked.

Such an approach is implemented in Martin Pool's `distcc` package [dis]. This tool is a replacement for the `gcc` compiler driver. Preprocessing and linking is done almost in the same way the standard compiler driver does, but the actual compile and assemble jobs are distributed among various nodes on the network.

Although this solution obviously gives not the same amount of scalability, as not all jobs can be parallelized, it is for most situations a better solution, as from my experience it seems that many system administrators are not capable of installing a homogenous build environment on a cluster of systems.

5 Conclusion

Finally, we can conclude that it is not really difficult to build and use a cross development tool chain, but in most cases, building the whole tool chain is not as simple as described in the compiler's documentation because building cross development tool chains is not as well tested as building native tool chains are. Thus, you should expect numerous minor bugs in the code and in the build environment. But with some basic knowledge about how such a system works and, thus, what the source of those problems is, in most cases they can be easily fixed or worked around.

At least if you have an amount of systems for office jobs idling almost all of their time, it is worth investing some time for building up such an infrastructure to use their CPU power for your build processes.

As this is a tutorial paper, its contents are intended for people that do not have extensive konwledge on the topic described to help them understanding it. If you think something is unclear, some information should be added or you find an error, please send a mail to `rschiele@uni-mannheim.de`.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Bin] GNU Binutils. `http://sources.redhat.com/binutils/`.

[dis] distcc: a fast, free distributed C and C++ compiler. `http://distcc.samba.org/`.

[GCC] GCC Home Page—GNU Project—Free Software Foundation (FSF). `http://gcc.gnu.org/`.

[Gli] GNU libc. `http://sources.redhat.com/glibc/`.

[Lev00] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2000.

[ppm] SourceForge.net: Project Info—PVM Parallel Make (ppmake). `http://sourceforge.net/projects/ppmake/`.

[PVMa] PVM: Parallel Virtual Machine. `http://www.epm.oml.gov/pvm/`.

[PVMb] PVMGmake. `http://pvmgmake.sourceforge.net/`.

Optimal Stack Slot Assignment in GCC

Naveen Sharma *Sanjiv Kumar Gupta*

System Software Group

HCL Technologies Ltd

Noida, India–201301

{naveens, sanjivg}@noida.hcltech.com

Abstract

Several microprocessors, used in digital signal processing and embedded devices, have limited displacement (4-6 bit) in "register + offset" addressing mode. In some cases, only auto increment/decrement addressing modes are available. Hence, while accessing data on local frame, there are number of explicit instructions whose sole purpose is to reach the desired data. This paper describes the impact of layout of local variables on performance and code size for these architectures. It also describes the techniques for optimal assignments of stack offsets such that instructions for address arithmetic for access of local variables are minimized. The implementation of the techniques in GCC is also discussed. Results indicate an improvement of 2%-7% in code size and 5-9% improvement in execution timings for several benchmarks.

1 Introduction

The use of micro-processors in embedded devices has been growing. The complexity of applications that run on these processors has increased proportionately. This makes the use of HLLs such as `C/C++` almost inevitable for writing these applications. Therefore, the compiler has to address the special architectural issues normally found on these processors.

One prominent issue is the restrictive addressing modes in these processors. Many of the architectures have limited offsets, if a 'reg+offset' addressing mode is available or just the auto increment/decrement modes. Accessing data beyond reachable offset incurs extra instructions. While this cannot be avoided in all cases, local frame is one area where we can improve data layout to subsume the address arithmetic. This freedom can lead to subtle benefits both in terms of performance and code size. This flexibility is useful regardless of target; the benefits, however, are most apparenet for processors that have limited displacement capability (such as `SH, ARM-Thumb, PA-RISC`).

GCC currently does not allow to reorder items in local frame. The document first disccusses the problems that arise due to this. The solution strategy and the implementation are discussed subsequently.

2 Problem Description

The stack allocation scheme in GCC needs improvements. In the present scheme, the objects are allocated on top of current frame when an allocation is required. An RTX of the of the form

```
(mem:mode (plus (fp) (const_int
          offset)))
```

is associated with it. A hard offset is thus assigned to it at the very beginning. This scheme results in the following problems.

- Increased code size
- Alignment holes and thus larger runtime frames.
- Performace degradation due to cache thrashing for certain applications.

We first explain the impact of frame layout on code size, taking the example of SH architecture. The SH architecture has a limitation of four bit offset in the 'offset + register' addressing mode `(@(k, rm))`. The 4-bit offset is zero extended and multiplied by 1, 2 or 4, according to the operand size (being a byte, word or long). Hence a maximum of 64 bytes can be accessed from a base register using this addressing mode. In cases where higher offsets need to be accessed, the compiler adjusts the register `rm`, so that a given reference lies within desired displacement. Hence if we want to access location, say, (72, fp) on SH, the assember output looks like:

```
mov    fp , r1 !Extra register
add    #72, r1 !Addition
mov.l  @r1, r2 !Actual Load
```

Notice the worst case costs involved when accessing data beyond addressable offsets in frame.

- The cost to spill the registers for temporary stack base pointer of array/structure/class (spill$_{reg}$).
- The cost to copy the frame pointer. (fp$_{copy}$).
- The cost to add the offset to temporary base pointer (reg$_{add}$).
- The cost to restore the temporary base pointer after use (reg$_{restore}$).

For floating point data SH allows predecrement, post-increment and indexed addressing modes (`r0` being the sole legal index register). Similar problems are imminent there too.

As another example, consider this piece of code in which a large array is placed at beginning of the frame.

```
void func(void)
  {
    float foo[16];
    int l,m,n;
    putval(&l,&m,&n);
    l=m+n;
    func1(l,m,n);
  }
```

GCC produces this code for statement `l=m+n` for SH, when we don't reorder anything on frame.

```
mov     r14,r1   !frame pointer r14 --> r1
add     #68,r1   !reaching "m"
mov.l   @r1+,r5  !m --> r5 and reaching "n"
mov.l   @r1,r6   !n --> r6
mov     r5,r4    !m --> l
add     r6,r4    !m+n -->r4 (stored to "l")
```

Note that frame layout is "foo, l, m, n"; so offsets assigned to these relative to the frame pointer are 0, 64, 68 and 72 respectively.

Ideally, if stack was laid out differently with following layout "l, m, n, foo", GCC generates the following code.

```
mov.l   @(4,r14),r5  ! m --> r5
mov.l   @(8,r14),r5  ! n --> r6
mov     r5,r4        ! m --> r4
add     r6,r4        ! n+r4 --> r4
```

Notice the benefits by this simple reordering. First, a decrease in Code Size beacuse nbecausef instructions, whose sole purpose is to reach data in local frame, are reduced. Secondly Register "r1" in above example remains free to be utilized elsewhere. Thirdly reduction in frame size in the general case because ordered layout will lead to lesser alignment holes. In cases when a large array on local frame is unused, we can significant stack space if we do not allocate it at all. (Array foo in the above example).

Last, if compiler allows frame object to be placed flexibly, the cache performance of applications might also be improved.

We propose two improvements in way GCC allocates local objects. The first improvement is the way the stack slots are represented internally and secondly the algorithms to assign actual offsets to address these problems.

3 Approach to the problem

The problem of offset assignments can be viewed in different ways. We can view this problem as similar to register allocation. Drawing analogy from the fact that compiler generates IL [1] code assuming infinite registers and allocates actual hard registers later, we can generate IL assuming infinite displacement and later map it to machine dependent displacement. While this mapping takes place we try to assign frame items within "fast access window" based on the interference graph of stack slots.[Burlin] describes a technique on lines of graph coloring. However, the approach has some implementation problems. Register allocation has significant differences with offset assignment inspite of apparent sin spiteties. Some obvious differences that need to be taken care are

[1]IL:Intermediate language or RTL in case of GCC

- Size of frame items is variable unlike registers which are of fixed size.
- Spilling has a different meaning than in traditional allocation.
- Graph coloring usually performs better for register sets numbering more than 16. While considering limited displacements, the algorithm seemed expensive.

These and several others problems are described by [Burlin].

The most popular approach for offset assignment is described by [Liao]. This approach is described for auto increment/decrement modes and can be adjusted to accomodate limited displacemenaccommodates occurence of adjacent accesses asoccurrenceto frame layout.

3.1 Solution Strategy

3.1.1 The stack pseudos

It was obvious that current representation of stack slots had several problems. It made reshuffling objects in the stack virtually impossible. An rtx of the form

```
(mem:mode (reg/f/c:Pmode slot))
```

is taken as the representation of a frame object. The slot is a stack address(or a stack pseudo). It is similar to virtual register but with slightly different semantics. We return a rtx of this form for each requested stack slot. Note that the special flag `/c` is used to tell that this is stack address pseudo. The register allocator should not try to allocate any hard reg for this because it is already a known stack slot.After register allocation, we sort the allocated stack slots by size and number of references and convert it to normal `fp+offset` form.

3.1.2 The Access Graph

An access graph is derived from a basic block. It gives the relative benefits of assigining adjacent locations for assigningof local variables. Given a insn sequence, an access sequence can be defined from it. Given an operation `set(r3 op (r1 r2))`, the access sequence is r1, r2, r3. The access sequence for an ordered set of operations is just a concatenated sequence of each individual operation. The access graph G(V, E) is derived from access sequence by adding edges corresponding to adjacent access between variables. Instead of an adjacent access, we take the limited offset window to add the edges. For each repeated adjacent access, update the weight associated with an edge. At the end, we have a possibly disjoint graph, representative of benefits of placing variables within a same displacement window.

This access graph can be extended to model the entire procedure with the help of data flow information.. The access graphs of basic blocks have to merge. Let us consider the scenario shown in Figure 1. Assume that probability of execution of basic blocks B2 and B3 is p2, p3 respectively. Further, since B3 is in a loop let us assume it has frequency of execution `f`. Then the following heuristics apply.

1. For access sequences in B3, the weight assigned while connecting adjacent variable accesses is proportional to `f`.
2. Weights assigned while connecting stack variable accesses between B1, B2 and B3 is proportional to probabilities p2 and p3.

These heuristics ensure that access graph takes into account the locality of accesses across entire procedure. From this information, we can determine placement of variables on the stack to minimize large displacements.

3.1.3 Use Data Flow Information

Another strategy is to use information built by flow analysis pass of the compiler. GCC builds data flow information regarding pseudo registers. This includes the attribute REG_FREQ which is the estimated frequency of the reference of the pseudo. Since stack slots are no diferent, this information is generated for different can use this information for frame layout by placing most frequently referenced variables near the frame.We tried the following heuristics:

1. sort the stack slots by size first
2. place the most frequently referenced variables together near the frame

3.1.4 Stack Reorganization Pass

A stack reorganization optimization pass is introduced after register allocation and is called as a subroutine during the reload phase [2]. This new pass primarily takes care of stack layout of variables. Stack assignments are made for pseudo registers based on locality of usage.It was observed that stack reorganization will have little effect before reload because most of the stack allocations are from within reload. So next possibility was to place it after reload pass. But replacing stack pseudos with their normal form after reload turns out to be complicated because validation of changed rtx's becomes part of stack reorganization, a task that reload is already doing. So calling stack reorganization from within reload turns out out be simpler and reload's code need not be repeated.

The algorithm is based on method given by [Liao]. The algorithm starts with the insn chain

[2]Post register allocation pass that handles the spills

of the function being compiled. The routine Construct_Access_Graph converts into a graph G(V, E) where V is number of variable accesses in a basic block and E is number of edges. An edge will exist between two variables v1 and v2 if they are accessed adjacently and the frequency of the adjacent access is recorded in the edge. Then algorithm uses a greedy approach, where it tries to add the edges with maximum weight adjacent to each other in spanning tree E'. The routine Traverse_And_Assign_Offsets takes this spanning tree as input and assigns offsets to variables in stack.

```
INPUT:  The insn chain of the function.
OUTPUT: Offset Assignment on the Stack.

G (V, E)<-- Construct_Access_Graph (L);
/* G is a graph with local variables
   (V) as nodes and E is the number of
   edges.  */

Es: sorted list of edges in descending order
    of weight.
/* The weight of an edge between <v1, v2> is
   frequency/relative gain of their adjacent
   access.  */

G'(V', E'): V'<--V, E'<--NULL;

while (|E'| < |V| -1 && Es != NULL)
  {
     /* Choose first edge.  */
     e = Es[1];
     /* Remove it from Edge List  */
     Es = Es - e;

     if ((e does not cause a cycle in G')
         &&(e does not cause and node in V'
         to have degree > 2)
        add e to E';
      else
        reject e;
   }
/* Now the best disjoint path cover
   is available. */
Traverse_And_Assign_Offsets(E')
```

3.2 Benchmark Results

The performance improvement by frame reordering depends on the following factors.

1. Size of the local frame.

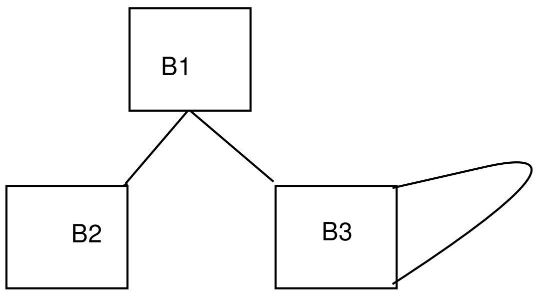

Figure 1: A sample control flow

2. Number of accesses of variables moved near the frame.

3. Frame layout heuristics.

In the best cases, the execution perfromace could go as high as 9%. The results for SH4 processor are shown here. The base version used for benchmark measurements GCC-3.3. The compiler options are '-O2 -ml m4'. A new option namely `-fstack-reorg` is introduced to enable stack reorganization. Table 1 gives size comparisons of stress1.17 files with and without stack reorganization. The Heuristics used are while frame layout are those of section 3.1.3. It is clear that in most cases, we have a decrease in code size. Some benchmarks show slight code size increase due to noise in reload phase.

The execution results for some benchmarks are shown in Table 2. Only those benchmark which have variation in execution timings are shown. One undesirable side effect, which is probably the main cause of performance degradation, is the harm done to loop optimizer because stack addresses are not exposed to it. A loop optimization pass after reload phase could possibly fix this problem.

File Name	size	size (stack-reorg)	decrease (%)
revolt.o	5956	5508	7.52
l3psy.o	15024	13968	7.03
mission.o	16972	15820	6.79
blocksort.o	4960	4640	6.45
advdomestic.o	8152	7640	6.28
explode.o	7916	7468	5.66
advmilitary.o	14844	14140	4.74
dogmove.o	10436	9956	4.60
lndsub.o	13820	13276	3.94
compress.o	4968	4776	3.86
physics.o	9020	8700	3.55
jidctflt.o	928	896	3.45
navion_gear.o	2040	1976	3.14
mhitm.o	22528	21824	3.13
r_segs.o	4384	4416	-0.73
q_shared.o	7966	8030	-0.80
g_phys.o	7396	7460	-0.87
tonal.o	10832	10928	-0.89
regex.o	24012	24268	-1.07

Table 1: Code Size Comparisons

Benchmark	Input Data Size	Gain (%age)
gsm Compression	1.71 MB	8.29
gsm decompression	361 KB	5.60
jpeg(dct int)	3.25 MB	-1.04
jpeg (dct float)	3.25 MB	-0.38
djpeg(dct int)	328 KB	4.73
djpeg (dct float)	328 KB	-2.05
gzip	80 MB	0.01
gunzip	16.2 MB	0.7

Table 2: Execution Timings

4 Acknowledgements

We would like to thank the GCC developer community for help. Their support is invaluable. We specially thank Zack Weinberg, Toshiyasu Morita, and Joern Rennecke for implementation ideas and comments.

References

[Liao] S.Liao and S.Devdas *Storage Assignment to Decrease Code Size*, MIT Departmenet of EECS, Cambridge MA (1995).

[Burlin] Johny Burlin *Optimizing Stack Frame Layout for Embedded Systems*, Information Technology Computing size department, Uppsala University, Sweden.

[GCC] GCC Internals Manual `http://gcc.gnu.org`

Getting the Best From G++

Nathan Sidwell
CodeSourcery LLC
`nathan@codesourcery.com`

Abstract

The 3.0 series of G++ compilers and libraries offers a new multi-vendor ABI and increasing conformance to the C++ standard. The C++ ABI offers increased efficiency for C++ idioms and interoperability with other compilers. Features of the ABI that the G++ user should be aware are described. Both additional and deprecated features in versions 3.2, 3.3 and 3.4 are described. Using various source idioms to aid the G++ optimizers and loading process is shown. The process of tracking the C++ standard as both defect reports and C++0X become available is outlined.

1 The 3.0 ABI

Starting with G++ 3.0 a new C++ ABI is provided. This multi-vendor ABI [2] came from development of an Itanium port of GCC, which included the design of a C++ ABI for the Itanium processor. That ABI was designed by CodeSourcery, EDG, Compaq, HP, Intel, Red Hat and SGI. Although designed for one architecture, the C++ ABI is sufficiently abstracted from Itanium features to allow its use for other processors, and hence the multi-vendor C++ ABI came about.

The 3.0 ABI is a complete redesign of the G++ ABI, which leads to space and speed improvements. The previous G++ ABI had evolved over time as C++ itself stabilized. ABI improvements include,

- Empty structures take zero size when used as a base class.
- Tail padding can be overlaid for non-POD bases and members.
- Derived to base conversions are constant time for both single and multiple inheritance. Conversion to a non-virtual base, requires a fixed adjustment and a single access of the vtable is needed to convert to a virtual base. Having virtual base offsets held in the vtable reduces the object size overhead for virtual inheritance. In most programs virtual inheritance does not increase the size of an object, because nearly all classes with virtual bases have virtual functions too. Previously a virtual base would add a pointer member to each class that derived from it, and base conversion involved following an inheritance path, which could involve several member accesses.
- Pointers to member functions are smaller, and dispatching via them is faster, because the vtable pointer is always at the start of an object.
- Virtual function thunks are all emitted with the thunked to function. This gives better cache coherency, and permits multiple entry point optimizations for thunked functions.[1] These improve the performance of the virtual function calling

[1] G++ does not currently implement multiple entry point optimizations.

mechanism. The thunk mechanism is such that even overriding from a virtual base is fast, with a single adjustment using one access into the vtable.

- Covariant return thunks are specified, and implemented in G++. Again, these are emitted with the overriding function that required their emission, and so have the cache coherency improvements and multiple entry point optimization opportunities of the simpler thunks.
- Dynamic cast hints are generated by the compiler, and improve the speed of `dynamic_cast` in common cases. In most cases the speed of `dynamic_cast` is now linear in the number of bases between the dynamic object type and the target type of the cast.
- Runtime type comparison is constant time, which further improves `dynamic_cast` and catch matching. Previously, type comparison involved string comparison.
- Exception handling is a two phase process. The first phase locates a catch handler, and only when one is found is the stack unwound to that handler. If a handler is not found, `std::terminate` can be called in the throwing context, and hence help debugging.
- A new mangling scheme that uses a compression algorithm. This produces shorter names, and so improves link and load times.

Additional improvements in G++ 3.0 were,

- The `std` namespace became a real namespace, rather than an alias for the global namespace.
- A new implementation of the standard template library, which is properly contained in the `std` namespace.
- Type based aliasing is enabled at optimization level `-O2`.

These changes effect user code to varying extents. Other than speeding up code, the new ABI should result in no user visible changes. Of course, all programs and libraries will need to be recompiled. If the user relied on ABI features, then a program might be effected.

1.1 Shared Libraries

The ABI makes use of a link facility that ELF [3] supports called common data. The common data linkage is used for objects that have no well defined object file in which to place them. The C++ ABI relies on common data linkage to implement the constant time comparison of types. This requires the names of type information objects to be globally visible. Libraries are effected because the type information objects must be visible to user programs. Shared libraries that are resolved at load time by the runtime loader, and those opened explicitly with `dlopen`, as is commonly done for program plugins, are effected in the same way. Static libraries are also effected, but the impact on real programs has not been so great. The link and loading speed of all three kinds of libraries can be improved by the mechanisms described here. A library makes available, or exports, to user programs a set of names. It also has to specify, or import, those names it uses from other libraries. Both importing and exporting use the same mechanisms and the remainder of this paper simply refers to exporting. If the library wishes to `dynamic_cast` or throw exceptions across the library interface, it must export type information names, so that the common linkage is achieved. More complicated export requirements require other types of names to be exported.

Because C++ has no module system, the library programmer cannot indicate at the source level which types, functions and objects are to be exported. The library is forced to export all symbols, to ensure the user can access the exported functionality. There are proposals [5] to add module facilities to the language. It is desirable to indicate a subset of the names as available to users of the library. The currently available mechanism for doing this is symbol versioning [4].

The simplest solution is to export all external names from the shared library. Unfortunately this has two disadvantages. Firstly program load times are increased because the dynamic linker must resolve all these symbols in order to eliminate duplicates with the already loaded program. Secondly, it exposes the internal names of the library implementation that have global scope. Sometimes those names can conflict with the user's names, or those in other libraries used in the program.

The solution to name conflict is to put the internal names into a library specific namespace. For instance, have the exported library functionality in a '`FooLib`' namespace, and the internal names in a '`FooLib::Internal`' namespace. Unfortunately it can be difficult to retro fit such a solution to an existing library that is not namespace aware.

For a simple shared library, where no runtime type information is transfered across its interface, it is simply necessary to export the library interface functions. For a more complicated library, it is necessary to export the type information names, and potentially some of the internal names. This can be done by examining the names in the library object files using `nm` and using a pattern matcher to extract the important ones. The G++ ABI mangles all names with an initial '`_Z`', followed by the mangled name. Certain prefixes are placed between the '`_Z`' and the mangled name, for particular kinds of names. These are,

`TV` Vtables. Pointed to by polymorphic objects and those with virtual bases. These are termed dynamic classes in the ABI.

`TT` Vtable table. Used in constructing and destructing polymorphic objects with virtual bases. Not all polymorphic classes will need a vtable table.

`TI` Type information. Returned by `typeid` operator, pointed to by the vtable.

`TS` Type string. Returned by `type_info::name`, and used for type comparisons.

`GV` Guard variable. Used to guard the initialization of function scope static objects that are dynamically initialized. The name of the static object will be the same as the guard variable without the '`GV`' prefix.

`Th`, `Tv`, `Tc` Virtual function thunks. These are followed by a mangling of the thunk information, and then the mangling of the thunked to function. The second prefix letter indicates whether it is a fixed, virtual or covariant thunk.

The vtable, vtable table, type information and type string are not tightly bound to any particular object file by the language, and so have common data linkage. Potentially any object file that uses them could contain their definition. The C++ ABI has an optimization where the class to which they belong has a non-inline virtual function, the first of which is called a key function. In that case, all these objects are only emitted in the object file that contains the definition of the key function. Other object files will not contain these objects, as it can be determined that their definition will be provided elsewhere. Libraries can be effected by this because, although it might not matter that two instances of a particular object were used in a program, a user program can rely on a definition it knows is in the library.

The type information objects will need exporting, to share type information, as user programs which use the type for `dynamic_cast` or catching, will need to refer to them. Sometimes these are emitted with internal linkage, in which case they refer directly or indirectly to an incomplete type. Such instances should *not* be exported. Type comparison itself uses the address of the type string. It is necessary for that string to be shared by all instances of the same type. If they are not exported, the type comparison algorithms will consider two types with the same name to be different types. Therefore, external names beginning with '`_ZTI`' and '`_ZTS`' should be exported from the library.

If the library exposes inlinable constructors or destructors of dynamic classes to users of the library, it is necessary for the library to export the vtable and vtable table.

If the library exports constructors to the user, all the user callable virtual functions of the class and its ancestors must be exported. Although virtual functions are normally called via the vtable (and therefore their names are not needed, just the index in the vtable), by exposing the constructor it might be possible to determine the dynamic type of an expression at compile time. Should the compiler do that, it may elect to replace a virtual call with a direct call, and hence require the name of the virtual function.

Static objects in inlinable functions that are exposed in library header files will cause problems. The static objects' names must be exported, so that only one becomes live in the final executable. Only static objects with a dynamic initialization expression will have a guard variable.

If the library exports types that can be inherited from, then the type information object, all user callable member functions of the class and all virtual functions and thunks must be exported. The class members will be mangled, following any applicable prefix, as a scoped name of the form '`N<classname> <membername>E`'. Both the `classname` and `membername` components are mangled as a numeric length followed by the name, such as '`6FooLib`'.

Here is an example library header file, showing what needs to be exported, depending on the functionality provided.

```
#include <exception>
#include <new>
namespace NMS {
  namespace Internal {
    // Helper we do not wish to expose
    // Do not export
    class Helper
    {
    public:
      Helper () {......};
      virtual int Frob () throw ();
    };
  } // namespace Internal
  // Export type info _ZTIN3NMS5ErrorE
  // Export type string _ZTSN3NMS5ErrorE
  class Error
    // Import std::exception typeinfo
    :  public std::exception
  {
    friend class Widget;
    // Do not export, library creates
    Error () throw () {}
    public:
    // Do not export, it is inline
    virtual ~Error () throw () {};
    // Do not export, called virtually
    virtual char const *what () const
      throw ();
  };
  // Export Widget if it is inheritable
  class Widget
    // Export direct & indirect bases,
    // if Widget is inheritable.
    :  Internal::Helper
  {
  private:
    // Do not export, library creates
    Widget () throw ();
  public:
    // Do not export, called virtually
    virtual ~Widget () throw ();
  public:
    // Do no export, called virtually
    virtual int Action () throw (Error);
  public:
    // Export, user can call
    // _ZN3NMS6Widget3NewEv
    static Widget *New ()
      throw (std::bad_alloc);
  };
} // namespace NMS
```

Because the only way of constructing a '`NMS::Widget`' object is by calling '`NMS::Widget::New`', users of the library will always have to use the virtual call mechanism to call '`NMS::Widget::~Widget`' and '`NMS::Widget::Action`', so those two functions do not need to be exported. Both `NMS::Error`'s type information and type string need exporting so that user programs can successfully catch such an object.

1.2 Library Compatibility

Linking C++ objects from different compilers involves more than just the C++ ABI. If the programs use the standard library, then the library versions must be compatible too. The multi-vendor ABI does not specify the binary compatibility of the library, as that would be too constraining on implementations. The ABI specifies a small runtime support library, necessary to implement the core C++ language. G++ provides that as a separately selectable `libsupc++`. The full library is also provided automatically as `libstdc++`. The G++ 3.0 implementation is a complete redesign of the library. The new library is more standard conformant, and this has lead to some issues with user code,

- The '`std`' namespace must now be explicitly noted. For example, '`vector<int> foo;`' does not compile. '`vector`', along with everything else, is in the '`std`' namespace. Previously, G++ also found it in the global namespace, so programs compiled whether '`vector<T>`' or '`std::vector<T>`' was used. Another common instance of this problem is using plain '`cout « "Hello World" « endl;`' The solution is to recognize the failure mode and insert '`std::`' appropriately.

- IO is slower. According to the C++ standard, by default, the standard C++ streams, '`std::cin`', '`std::cout`' and '`std::cerr`', have to be synchronized with the standard C file streams, '`stdin`', '`stdout`' and '`stderr`', so that use of corresponding pairs of streams can be intermixed. A clever trick allowed the previous C++ library to overlay its stream classes on the underlying C library's file structure, *but* only for one specific C library. With the change in the G++ ABI, and better standard conformance, that trick became impractical to maintain. The standard allows users to explicitly decouple the C and C++ file IO operations by calling, '`std::ios::sync_with_stdio (false)`' before any IO has happened on the standard streams.

  Another issue with '`std::cin`' and '`std::cout`' is that they are synchronized with each other. C++ requires that, by default, intermixed input and output will display in the correct order. This synchronization can be removed by calling '`std::cin.tie (0)`'. C does not have such fine grained synchronization on '`stdin`' and '`stdout`', these are normally only synchronized at newline characters.

  C++ IO is more expressive than that provided in C, and because the C++ library is implemented on top of the C library, C++ IO will never be faster than C IO. Work is ongoing in improving IO performance.

- Iterators do not have pointer types. Some code presumes that iterators are implemented as pointer types, and contain code such as '`&myIterator`', expecting to get a '`T **`'. Because the previous library implemented them as such, that code 'worked', even though that implementation is neither required nor guaranteed by the standard. Now iterators are implemented as templated classes, which gives better type safety, but breaks such erroneous code. Code which assumes the underlying representation of an iterator can be forced to work simply by `&*myIterator`, as the `*` operator will provide a reference to the iterated object, whose address can be taken.

2 What is in G++ 3.3

The multi-vendor ABI is very complicated and its first G++ implementation in G++ 3.0 turned out to have some bugs. Several of the defects were discovered in time for G++ 3.2. More issues have been discovered since then, by testing interoperation with other compilers and by using CodeSourcery's testsuite. It is very inconvenient to change the ABI, as that means that all object files and libraries need to be recompiled with the new compiler. Some ABI bugs merely effect inter-operation with other compilers, and are unimportant to a significant user base. Rather than force an ABI change on all users, G++ implements two flags to warn about ABI discrepancies and to select ABI version. The `-Wabi` flag warns when G++ is emitting code or data that is known to be at variance with the multi-vendor ABI. The `-fabi-version=<n>` flag allows the user to specify which set of known ABI fixes to include. The current default version is 1. Whenever an ABI bug is discovered, code for both options is added to the compiler, and the warning code is backported to the previous stable release branch, for a subsequent minor release. Of course, because time machines are nonexistent, it is not possible to backport it to the previously released version. All known ABI fixes can be selected with `-fabi-version=0`. Which fixes that includes depends on the version of G++, so using this value implies that the same version of G++ must be used to compile all the object files and libraries of a program. When a sufficiently stable set of fixes has been implemented, another ABI version number will be added, and `-fabi-version=2` will be selectable. It is likely that G++ 3.4 will implement such an ABI version number, but it is undecided whether that will be made the default value. Version 0 will still be selectable, to obtain all the subsequent fixes added after version 2 has been stabilized.

G++'s implementation of the standard template library has not yet stabilized. Because the library exposes much of its implementation in header files containing class, inline function and template definitions, it is very difficult to improve the library without changing something that effects binary compatibility. There are no planned library ABI changes between the 3.2 and 3.3 releases. However, the 3.4 release will not be binary compatible, and the shared object version number has been incremented. Because it is provided as a shared library, and the version number has changed, users will get a link error, rather than mysterious runtime failures, if they attempt to mix versions.

One of the more significant changes in G++ 3.3 is the removal of the implicit typename extension. The extension was deprecated in G++ 3.2, and elicited a warning at every use. In a template class, names from dependent bases are not visible when the template is defined—they are only looked up at instantiation time. G++ had an extension that made names visible before instantiation, so G++ knew which were types and which were not. The standard requires that those that name a type be referred to using the `typename` keyword and qualified name.

```
template <typename T>
class Base
{
  typedef int Type;
  typedef int Other;
};
typedef unsigned Other;
template <typename T>
class Derived :  public Base<T>
{
  Type a; // Implicit typename use.
  // Standard conforming way.
  typename Base<T>::Type b;
  Other c; // Which Other?
};
```

The implicit `typename` extension became impossible to keep when updating G++'s parser to be more conformant. The extension is also problematic in itself. In the example, when instantiating '`Derived`' for some particular type '`U`', '`Base<U>`' might have a specialization for which '`Base<U>::Type`' is not an `int`, or even a type. Another confusion is shown in the example by the use of '`Other`' in '`Derived`'. If the implicit typename extension is in operation, it will be '`Base<T>:: Other`', whereas without it, it should find '`::Other`'. Having a program's meaning change between two valid interpretations by changing a command line flag (`-pedantic`), is really bad—better to remove the extension.

2.1 Optimization

Previously G++ had a named return value extension to help functions that returned a class by value. Because returning a class value requires a copy of the return value into the area provided by the caller, such functions would invoke a copy constructor just before returning. The idea of the named return value extension was to allow the programmer to use that area directly and avoid the copy. This extension did not work with template functions, and has been removed. In its place is the return value optimization, which notices when a function is returning a temporary by value, and will directly construct the temporary in the return area.

2.2 Exception Specifications

G++ 3.2 had poorer inlining performance than desired. It would not make sensible choices about what to inline, and the inlining process could lead to long compile times and large compiler memory size. This has been fixed by taking advantage of '`throw ()`' exception specifications. If none of the functions called by a particular function can throw exceptions, the inliner can do a better job.

Exceptions specifications can also be used to reduce the size of a program. In the following program, `CLASS1`, `CLASS2`, `FOO` and `BAZ` can be defined to be empty, or '`throw ()`'. The code and exception data sizes for various combinations using G++ 3.2 for i686-pc-linux-gnu producing optimized code is shown in Table 1. The 'Check' column indicates whether the `-fno-enforce-eh-specs` option was used.

```
struct Class1
{
  int m;
  Class1 () CLASS1;
  ~Class1 () CLASS1;
};
```

Exception specification								
CLASS1	CLASS2	FOO	BAR	Check	Code	Data	Total	Overhead
`throw ()`	`throw ()`	`throw ()`	Either	Either	63	0	63	-
`throw ()`	`throw ()`	None	Either	No	83	92	175	178%
None	None	`throw ()`	Either	No	95	88	183	190%
None	None	None	None	Either	103	104	207	226%
None	None	None	`throw ()`	Yes	137	113	250	296%

Table 1: Exception Overhead Example

```
struct Class2
{
  int m;

  Class2 () CLASS2;
  ~Class2 () CLASS2;
};

void Foo () FOO;

void Baz () BAZ
{
  Class1 c1;
  Class2 c2;

  Foo ();
}
```

The worst case is a factor of 4 in program size, however the more common case is probably the penultimate line of the table where none of the functions have an exception specification. The `-fno-enforce-eh-specs` option tells G++ not to add code to a function to check that it is throwing only the exceptions listed in its exception specification. A correct program will only throw such exceptions, so such checking code is behaving as `assert` macros. However it is notoriously difficult to exercise exceptional paths in program flow. The author has used a custom allocation library to rigorously test allocation failures in a command line application, to good effect.

Such a small example might be skewed to give large overheads—it has no real code in it, and nothing can be inlined. Two C++ libraries of about 30,000 lines of code each were examined. One was a low level utility library, and the other a higher level 3D toolkit. Both libraries have been written with exception specifications on every function, most of which were no-throw, but many were allocation failure exceptions. Each library was compiled in three different ways:

- With exception specifications, but with `-fno-enforce-eh-specs` enabled to remove the exception checking code.
- With exception specifications and with exception checking enabled.
- With `throw` defined as a varadic '`throw(...)`' macro, so that the exception specifications were removed. With no exception specifications, checking exception specifications would have no effect on code size, so `-fno-enforce-eh-specs` would make no difference.

Table 2 shows that the overhead is between 11% and 18%. The code size of a checked exception specified library is larger than that of the library without exception specifications, because of the number of non-empty exception specifications. G++ is not clever enough to notice whether functions that have a non-empty exception specification only call functions that can throw the listed exception types—it still

Exceptions		Utility Library				3D Library			
Specs	Checked	Code	Data	Total	Overhead	Code	Data	Total	Overhead
Yes	No	147871	23037	170908	-	219124	49422	268546	-
Yes	Yes	158000	35070	193070	13%	238691	77538	316229	18%
No	Either	150760	39685	190445	11%	224646	83306	307952	15%

Table 2: Library Exception Overhead

emits code to verify. A suitable optimization will be able to remove that extra checking code. The same is not true of the extra code added when there are no exception specifications. That code has been added to destroy local variables that will go out of scope, should an exception be thrown. The compiler cannot determine only from a function declaration with no exception specification that the function will not actually throw an exception, so it must presume the worst and emit appropriate destruction code.

When the body of a function is visible, G++ can determine if it does not throw by noting whether it calls a function that could throw, or contains a `throw` expression. If it cannot throw, G++ will optimize appropriately. In the small example above, such analysis could only be done on '`Baz`', and the specification checking code can be deleted as unreachable, if all the other functions have a '`throw ()`' exception specification. Both 3.2 and 3.3 will remove this unreachable code, but 3.3's compile time performance will be better, as it notices much earlier in the translation process that the checking code is unreachable. G++ cannot currently tell whether an exception will be caught inside the function, so appropriate '`throw (...)`' exception specifications should be added to function declarations and function definitions that contain '`try ... catch`' clauses.

One final note about code size. A static image of 'Hello world' is surprisingly large. For instance, the program '`int main () {return 0;}`' has a static code size of 289,281 bytes on the author's gnu-linux system. Both C and C++ sources gave the same size. The size is a glibc [8] issue, not a GCC problem. Glibc is not designed to be used as a static library, and embedded systems should use an alternative library.

3 What Will be in G++ 3.4

G++ 3.4[2] will feature a much better parser, which correctly deals with more ambiguous parsing situations than G++ 3.3 does. C++ has an ambiguous grammar where a construct can look like both a declaration and an expression, it is not until deep within the statement that the parser can tell which one it is. The previous Bison [6] based parser could not deal with several cases that were reasonably common. Bison parsers deal with LALR(1) [7] grammars, but C++ is not such a grammar. The Bison based parser has some C++ specific hacks to deal with some of the ambiguities. The new parser is a handwritten recursive descent design, with arbitrary back tracking. Here is an example, where as G++ 3.3 fails on every line of '`Foo`', G++ 3.4 will parse them all.

```
struct A
{
  A (int = 0);
};
```

[2]This section describes the development version of G++ as at 28th April 2003. When 3.4 is released, it might differ from what is described here.

```
struct C
{
  C (A, A = A ());
  void oneFish () const;
};
A Foo (int thing1, int thing2)
{
  C redFish (A (), A (1));
  C blueFish (A (thing1), thing2);
  C (A (2)).oneFish ();
  return (A ());
}
```

Having a better parser is good, but it is also more picky about name lookup in template definitions. Names can be looked up during template definition and during template instantiation. Depending on context, a name might be looked up only during definition, or only during instantiation, or both. This is called two-stage name lookup. C++ programs developed only with G++ are more than likely to have template name lookup problems—switching to the new parser will produce compilation errors. There are two cases of interest, one involving dependent bases and the other to do with argument dependent lookup (Koenig lookup).

The dependent base problem is similar to the implicit typename issue that was removed in G++ 3.3. Here is an example,

```
template<typename T>
struct Base
{
  int count;
  int total;
};
int count;
template<typename T>
class Derived :  public Base<T>
{
  // Wrong Thing
  void Flangify ()
  {
    // 3.3 defers both to instantiation
    // time and finds those in Base<T>.
    // Should bind to ::count.
    int ix = count;
    // Error, should not be found.
    int jx = total;
  }
  // Right Thing
  void Flangicate ()
  {
    // Deferred to instantiation.
    int ix = this->count;
  }
};
```

As ‘`Derived::Flangify`’ shows, the compiler will give an error at template definition time, if a non-dependent name is not found. Unfortunately, it could bind to an unintended object, which happens for ‘`count`’. ‘`Derived::Flangicate`’ shows the correct way of forcing name lookup of members to be deferred until instantiation time. The idiom has the advantage of making explicit to the programmer that the name refers to a member. That members in dependent bases are not searched for, unless preceded by ‘`this->`’, is very surprising to programmers unfamiliar with the rule. The intent is to allow more checking and precompilation of template definitions, before instantiation, and only defer to instantiation time those lookups that are demonstrably dependent on a template parameter.

The other place effected by name lookup is in function calling and argument dependent name lookup. When a function is called using unqualified name lookup (something like ‘`foo (arg)`’, but it also happens on overloaded operators), the function is looked up in the current scope as normal and in the classes and namespaces of the arguments’ types. If the arguments’ types are template dependent, that part of the lookup is deferred until instantiation time. The non-dependent part of the lookup is done at definition time, and not repeated at instantiation time. Here is an example,

```
namespace NMS {
  class MyClass {};
  void Foo (MyClass);
} // namespace NMS
```

```
void Foo (int);

template <typename T>
void Bar (T thing)
{
  Foo (1); // #1
  Foo (thing); // #2
}
```

The first call, '`Foo (1)`', will find '`::Foo (int)`' at definition time. The second call, '`Foo (thing)`', will find the global '`Foo`' at definition time, but it will also find '`NMS::Foo (MyClass)`' during the instantiation of '`Bar<NMS::MyClass>`'. The two declarations of '`Foo`' are added to the overload set, upon which overload resolution is performed. Overload resolution could be done at definition time for the first call, as that does not contain any template dependent expressions. At the current time, the development version of G++ still defers lookup for function calls until instantiation time, and therefore does not have the correct two-stage lookup behavior here.

Another impact of this, is that G++ will not mangle some templated names correctly. In some cases the mangling depends on knowing what is a dependent expression and what is not. Without that knowledge, although the manglings are unique, they do not adhere to that specified by the ABI.

A lookup case that G++ still gets wrong is where a name refers ambiguously to a member of a dependent base and of a non-dependent base. At definition time the dependent base will be ignored and the name found unambiguously in the non-dependent base. The ambiguity *should not* be discovered at instantiation time, as the lookup is not repeated. G++ will repeat the lookup at instantiation time and discover an ambiguity.

```
template <typename T>
struct Wump
{
  int zed;
};

struct Gump
{
  int zed;
};

template <typename T>
struct Sneetch :  Wump<T>, Gump
{
  Sneetch ()
  {
    // Both 3.3 and 3.4 find an
    // ambiguity at instantiation.
    // Should bind to Gump::zed.
    zed = 5;
  }
};
```

Because G++ still does not do the correct two-stage lookup for function call, some cases of the first described name lookup issue can still remain undetected. When the intent is to call a member function of a dependent base, the name lookup is incorrectly deferred until instantiation time. Even if the function parameters are template dependent, a non-friend member from a dependent base should not be found—only those names found by argument dependent lookup should be added at instantiation time.

```
template<typename T>
struct Base
{
  void Deflange ();
  void Flange ();
};

void Deflange ();

template<typename T>
class Derived :  public Base<T>
{
    // Wrong Thing
    void Flangify ()
    {
      // 3.4 binds both of these to
      // members of Base<T>
      // Should bind to ::Deflange.
      Deflange ();
      // Error, should not be found.
      Flange ();
    }
    // Right Thing
    void Flangicate ()
    {
      // Deferred to instantiation.
      this->Deflange ();
    }
  };
```

To get two stage lookup correct requires better tracking of the symbol table so that, at instantiation time, it is known what declarations are visible at both the definition context and the instantiation context. Whether this work will be completed by the time 3.4 is released is unknown.

4 Tracking the Standard

The C++ standard is an evolving document. Since the 1998 C++ standard was released, various changes have been made. A Technical Corrigendum 1 (TC1) is in the process of being released. That bundles all of the accumulated changes into single document. Issues can be raised by anyone, and are collated via an email list. Every six months, a global meeting of the ANSI J16 and ISO WG21 [9] committees takes place. These meetings are open to all interested parties, and membership of J16 is not required. Affiliation does effect voting rights. There are three subgroups within those meetings,

- Core Working Group. This group deals with issues in the core language (that documented in clauses 2 to 16). The core defect reports are available [10].
- Library Working Group. This group deals with issues in the libraries (clauses 17 to 27). The library defect reports are available [11].
- Evolution Working Group. This group deals with extensions and other changes to the language and library. The group is currently considering what significant changes should be made for the next version of the standard, code named 'C++0X'.

The output of the core and library working groups are lists of defect resolutions. A report may be deemed to be not a defect (the standard requires no change). Alternatively the standard may require clarification, or require change. The wording of the changes is discussed and goes through a process of drafting until it is ready to be accepted.

G++ aims to track the standard with its collection of defect reports. We do not make a distinction between the 1998 standard and the standard plus defect reports. Active participation in the C++ standards meetings allows the G++ maintainers to both know how defects are likely to be resolved, and to influence that process. When C++0X is released, G++ will probably have a command line switch to select which version of C++ is to be accepted (just as either C89 and C99 can be selected between in GCC).

Many people have suggested extensions that G++ should accept. Often these proposals are of the form 'It would be neat if I could write ...', rather than a complete specification. Such vague descriptions can prove problematical with a language as complicated as C++—all the implications of an extension are not apparent, even after some thought. Without care, extensions can either silently change the meaning of a C++ program, or fail in obscure ways under some circumstances. Several GCC extensions have caused such problems when ported to G++. Some of note are,

- It used to be possible for `__PRETTY_FUNCTION__` to participate in string concatenation. Unfortunately this does not fit well with templates, where the expansion of `__PRETTY_FUNCTION__` depends on the instantiation, much later than string concatenation occurs. GCC has been changed throughout, so that `__FUNCTION__`, `__PRETTY_FUNCTION__` and the C99 defined `__function__` all behave the same way as constant arrays of characters.

- Variable length arrays have a type which is not fixed at compile time. This causes a problem with `typeid`, because there is no fixed `std::type_info` object that can be returned. `typeid` was changed to return the type info for the array member type. Also template deduction suffers, because the type has a size that is not fixed. Template deduction will not deduce variable length arrays by reference. They can still be deduced as pointer types via the normal array to pointer decay rule.

- GCC allows empty structures as a C extension, and gives them a size of zero. C++ allows empty structures, but specifies that their size is not zero. They have a non-zero size to preserve the invariant that no two objects of the same type have the same address. GCC does not keep that invariant for such empty types. Structures that contain empty members will be laid out differently in C and C++.

- The implicit typename extension described above has now removed from G++.

Because C++ extensions can have so many unseen consequences, the G++ maintainers require a very strong argument and implementation in favor of an extension, before accepting it. Incompletely documented extensions lead to problems in maintaining G++ [12].

5 Closing Remarks

C++ support in the 3.x versions of G++ has improved considerably over that in the previous 2.x versions. Improving C++ conformance is not without pain to users who have unknowingly been writing ill-formed C++. G++ aims to smooth the transition by deprecating inappropriate features and giving a warning in one version and then removing the feature in the next version. When a new error message is added, because of better standard conformance, explanatory text might be added to help the user correct their code.

Various improvements and ways that user programs can effect the quality and speed of compilation have been described. Library writers are particularly inhibited by the lack of a module system, and workarounds are shown so that library link time can be reduced.

There are still new optimization opportunities in G++, for instance a multiple entry point mechanism for thunks, so that multiple inheritance is even cheaper in both speed and code size. Better exception tracking can be added to remove unnecessary runtime checks.

References

[1] Programming Languages—C++, ISO/IEC 14882:1998.

[2] Itanium C++ ABI, `http://www.codesourcery.com/cxx-abi/abi.html`.

[3] System V Application Binary Interface, `http://www.caldera.com/developers/gabi/`.

[4] Using GNU ld, `http://sources.redhat.com/binutils/docs-2.12/ld.info/index.html`.

[5] Pete Becker, Draft Proposal for Dynamic Libraries in C++, `http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1428.html`.

[6] Bison Parser Generator, `http://www.gnu.org/software/bison/bison.html`.

[7] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[8] Gnu C Library, `http://www.gnu.org/software/libc/libc.html`.

[9] ISO/IEC Working Group 21 `http://std.dkuug.dk/jtc1/sc22/wg21/`.

[10] C++ Core Issues List, `http://std.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html`.

[11] C++ Library Issues List, `http://std.dkuug.dk/jtc1/sc22/wg21/docs/lwg-active.html`.

[12] Zachary Weinberg, A Maintenance Programmer's View of GCC, GCC Developer's Summit, 2003.

StackGuard: Simple Stack Smash Protection for GCC

Perry Wagle *Crispin Cowan*[*]
Immunix, Inc.[†]
`wagle@immunix.com, http://www.immunix.com/˜wagle`

Abstract

Since 1998, StackGuard patches to GCC have been used to protect entire distributions from stack smashing buffer overflows. Performance overhead and software compatibility issues have been minimal. In its history, the parts of GCC that StackGuard has operated in have twice changed enough to require complete overhauls of the StackGuard patch. Since StackGuard is a mature technology, even seeing re-implementations in other compilers, we propose that GCC adopt StackGuard as a standard feature. This paper describes our recent work to bring StackGuard fully up to date with current GCC, introduce architecture independence, and extend the protection of stack data structures, while keeping the StackGuard patch as small, simple, and modular as possible.

1 Introduction

Despite years of punditry, source code audits, and many layers of proposed technology, buffer overflows are *still* the leading cause of software vulnerability. This paper describes the motives and technical issues of incorporating the StackGuard [6] stack smash defense as a standard feature of GCC.

The *stack smashing* variety of buffer overflow [14] is its most common subtype, and the most readily treatable. A stack smash attack gains control of a thread in an address space by overwriting control information—such as a return address—on its stack.

The common way for the attacker to overwrite values stored on the stack is to use a *buffer overflow*, where large inputs are used to cause more data to be written to an area of memory than space has been allocated. StackGuard protects against stack smash attacks resulting from buffer overflows, but also those resulting from *any* sequential write through memory.

To detect corrupted control information in procedure activation records, StackGuard adds a location that it calls a "canary"[1] to the stack layout to hold a special guard value. Traditionally, the layout of that section of the stack has been determined by function *prologue* and *epilogue* code generators, which are architecture specific. As a result, StackGuard *implementations* have also been architecture specific. As time has passed, these parts of GCC have become more abstract, requiring repeated re-implementations of StackGuard, but the ability to modify stack layout in a platform independent way has been lacking.

A new version of StackGuard has been imple-

[*]This work supported by DARPA Contract F30602-01-C-0172.

[†]nee WireX Communications, Inc.

[1]Alluding to the canary Welsh miners used to detect air problems before the miners could.

mented for modern versions of GCC. It now guards *all* the information in the control region of all procedure activation records generated by the `./gcc` back end of the GCC compiler suite (C, C++, *etc.*). That is, the saved registers and saved frame pointer are are now protected in addition to the return address for every procedure. Stack layout to provide the canary *location* is still left to the architecture specific function prologue and epilogue code generators. The rest of StackGuard is now architecture independent.

This new StackGuard has been successfully used in conjunction with other security hardening technologies to rebuild the Red Hat 7.3 distribution (GCC 2.96-113). The StackGuard patch has also been applied to the source for GCC 3.2-7 used in the Red Hat 8 distribution to rebuild both the compiler and GLIBC.

In accordance with the principle of default deny [15] StackGuard makes a point to apply the guarding technology to *every* procedure in a distribution. In this fashion, StackGuard is a *security optimization* that transforms *all* emitted code to deny a class of attacks. It also shows the soundness of the transformation by showing that the distribution works the same after the transformation as it did before. Picking and choosing which procedures receive the transformation is a *performance optimization* that, based on *assumptions* about the nature of the security threat, trades some security for performance.

StackGuard strives to be the essence of the "guard the control information" security optimization that is capable of being applied to every procedure on a system. Given the negligible performance impact of the complete transformation [5, 7], we have never seen the need to apply any additional performance optimizations. Unfortunately, there persists the mistaken impression that StackGuard produces a significant performance impact [1].[2]

This paper describes the issues to be considered for including StackGuard as a standard feature in GCC. In keeping with good modular design principles, we emphasize the considerations specific to the stack smash detection technique to keep that problem small and separate. In particular, the guarding technology can be used, with compiler support, to guard other regions of memory. Its design and implementation should not be unnecessarily tied to the specific use as a Stack Smash detector, though that's all that is discussed in this paper.

The rest of this paper is as follows. Section 2 describes compiler work to date on the buffer overflow problem. Section 3 describes the design of our proposed feature for GCC. Section 4 describes our current implementation of this design in GCC 3.2. Section 5 presents our performance benchmarks, supporting our claim that this feature is low-cost. Section 6 describes our on-going security testing. Section 7 presents our conclusion. Section 8 describes the availability of the StackGuard technology.

2 Background and related work

Aleph One's paper [14] presented a cook book for the "stack smashing" variety of buffer overflows, in which the attacker overflows a stack buffer to change the return address in an activation record to point to malicious code contained in that self same overflow. In the general case, the attacker wants to inject malicous code, and alter control flow structures so that the program will jump to the malicious code. The stack smash is an elegant attack that

[2]The performance issues shown in the Libsafe paper result from selected benchmarks (quicksort) that emphasize where StackGuard imposes overhead (function calls) and ignores where Libsafe imposes overhead (string functions).

achieves both objectives in a single stroke by exploting a very common programming error (weak bounds checking on fixed sized stack buffers). However, the attacker only *needs* to change control flow, because subvertable code (capable of performing the moral equivalent of "`exec(sh)`" for the attacker) is often already resident in the victim program's address space.

Since Aleph One's paper appeared, there has been a lot of work to defend against buffer overflows, interceding in the operating system [9, 8, 11, 17], system libraries [16, 1], and compilers [6, 12, 13, 10, 19]. These techniques variously try to prevent the modification of control flow paths, prevent the injection of malicious code, or both [7].

Because we are proposing a GCC enhancement, we consider only compiler defenses. Compiler defenses can in turn be divided into array bounds checking (which prevents buffer overflows, described in Section 2.1) and data integrity checking (which detects buffer overflows in time to prevent attacks from succeeding, described in Sections 2.2).

2.1 Bounds Checking

Array bounds checking is the ultimate way to eliminate buffer overflows. Unfortunately, the design and idioms of the C language make it difficult to provide for fully secure array bounds checking while preserving reasonable legacy compatibility and reasonable performance.

The Compaq C compiler for Tru64 UNIX [3] is an example of incomplete protection. The compiler has an option to perform bounds checking, but it only does so on *explicit* array references; pointer references are not checked. Since all arrays passed as function arguments are converted to pointers, this means that array bounds checking is effectively limited to strictly local variables, and the security value of the feature is low.

The Jones and Kelly GCC enhancement [12, 18] is an example of compromised performance. This GCC enhancement provides complete array bounds checking, even for pointer references, and maintains the current size of a pointer as a machine word. They achieve this through an associative lookup on each pointer reference to an array descriptor that stores the base and bounds. Performance penalties are high, approximately 10X to 30X slowdown.

The Bounded Pointers project [13] is an example of compromised compatibility. Rather than associative lookup, Bounded Pointers changes pointers from a single word into a tuple that incorporates base and bounds. This improves performance by eliminating the associative lookup in Jones and Kelly, but also costs compatibility because pointers no longer fit in a single word. Performance penalties are still high at approximately 5X slowdown. It is conjectured that this slowdown could be substantially reduced, but unlikely that the penalty would reach the low percent range.

2.2 Integrity Checking

The first integrity checking mechanism was Snarski's libc [16] that checked the integrity of activation records within libc functions. StackGuard [6] generalized this notion with a compiler enhancement to check the integrity of *all* activation records. These methods ornament activation records as they are built with data structures that cannot survive stack smashing attacks, so that when the function tries to return, it can detect that the activation record has been corrupted. Upon detection, the program issues an intrusion attempt alert and exits, rather than handing control to the attacker.

There have been three major releases of Stack-

Guard. StackGuard 1 was a patch to GCC 2.72, hooking directly into the `prologue` and `epilogue` code generation functions to emit StackGuard canary generation and verification code into function set up and tear down. StackGuard 2 was a complete re-write, providing an enhancement to GCC 2.92, this time implemented as modified RTL generation for function setup and tear down. StackGuard 3, presented here, is another complete re-write to accomodate GCC 2.95 and newer.

There are two significant reimplementations of StackGuard: Propolice and Visual C++.net.

OpenBSD's Propolice implements something very much like the StackGuard defense as an enhancement to GCC, and provides a important and very interesting contrast with its differences:

- To a large extent, Propolice and StackGuard have independently converged, from opposite directions, on doing the guarding code inserts at the RTL level.[3] But they haven't quite met in the middle—Propolice does the inserts at a much earlier pass in the compiler.
- Propolice places the canary word, as a *buffer overflow* detector, *only* at the top of auto variable regions containing "buffers".[4] StackGuard places the canary word, as a *stack smash* detector, at the bottom of *every* control region.
- Propolice uses random canaries. StackGuard uses terminator canaries.
- Propolice provides variable sorting—moving *some* character arrays above all other data types—to make it difficult to overflow into *adjacent* variables. StackGuard makes no assumptions about the starting point of runaway sequential overwrites of the stack, leaving security optimized stack layouts to separate mechanisms, such as Propolice.
- Propolice appears to move significantly towards a universal, architecture independent, stack layout. It even goes as far as to move saved registers into the autovariable region. StackGuard goes to great pains to try to leave the stack layout as close to the way it was as possible.
- Propolice modifies far older versions of GCC than StackGuard.[5]

Propolice's design decisions present different trade-offs than StackGuard:

- By doing the code inserts well before sibling and tail recursion is recognized, Propolice has no way to insert canary checks before the function exits points produced by the external branches. StackGuard makes a point of doing these insertions also.
- By depending on the *coincidental* adjacency of the autovariable region and the control region on the stack, Propolice gives the appearance of guarding the control region from buffer overflows that it detects leaving the autovariable region. But this *implicit* invariant isn't maintained across compositions with other security and performance oriented transformations that affect stack layout.
- The apparent strengths and weaknesses of both terminator and random canaries are discussed in Section 3.1.

[3]Propolice started at the AST level, while StackGuard started at the architecture specific function prologue and epilogue backend level.

[4]Currently, this appears to be defined as character arrays of greater than 4.

[5]OpenBSD's GCC 2.95.3 20010125 *vs.* Redhat 7.3's GCC 2.96-113 and Redhat 8.0's GCC 3.2.2-2.

- Nothing requires string writes to start in a char buffer. When an exploit finds such an opportunity, Propolice will stop it only if it's lucky. StackGuard will stop it by its design that *all* stack smashes should be detected.

- Propolice's buffer overflow detector becomes quite different than StackGuard's stack smash detector when alternate stack layouts, involving multiple stacks, stacks growing upward, heap allocated stacks, *etc.* are considered. Both are useful: Propolice detects buffer overflows that aren't stack smashes, and StackGuard detects stack smashes that aren't buffer overflows.

- By moving saved registers into the autovariable region, Propolice appears to assume that saved registers have the same dynamic scope rules as autovariables. This is not necessarily true for tail calls, where it would be correct to restore saved registers, but not correct, in general, to deallocate autovariables.[6]

- Propolice's changing of the stack layout could disrupt other tools that do stack introspection, such as GDB and JIT-styled JVM's.[7] StackGuard goes to pains to be invisible to such tools.

- It's unknown how well Propolice ports to current versions of GCC. StackGuard strives to be its part of that work, done completely and correctly.

Microsoft has also implemented [4] a feature very similar to StackGuard which they call the "/gs" feature in Visual C++.net. Compiler implementation details are naturally closed source, but the emitted code strongly resembles StackGuard code. The comparison is gone into more detail at various points in later sections.

Section 3 presents the StackGuard 3 design in more detail.

3 Design

The purpose of StackGuard is to do integrity checking on activation records, with sufficient precision and timeliness that a program will never dereference corrupted control information in an activation record, which is written to once on entry to a function, and read from once on exit from a function.

The *threat* is that the attacker has the capability to overwrite control information in some frame on the stack via a sequential write operation—such as a string copy or a memory copy—starting from somewhere lower in memory than the *target*. This permits the attacker to hijack the thread to execute code of the attacker's choice. The desired code might be new code supplied by this particular sequential write into the stack, another sequential write into stack *or non-stack* memory, or else code that is already in the address space that will do what the attacker needs when branched to in this fashion.

We will assume that the attacker does not need to inject code, but can use executable code already in the address space. This is a growing technique in practice, and permits us to focus on the most important part of the attack: overwriting control information, particularly pointers to code, such as return addresses.

The attack works if it can rewrite the control information between the time it was written with correct values to be saved and the time it was later read assuming it contained correct values of things to be restored.

[6] Some really clever tricks would be needed to support tail recursion from functions with autovariables, and people have been known to build compilers that do that.

[7] As has happened with previous (but not current) versions of StackGuard.

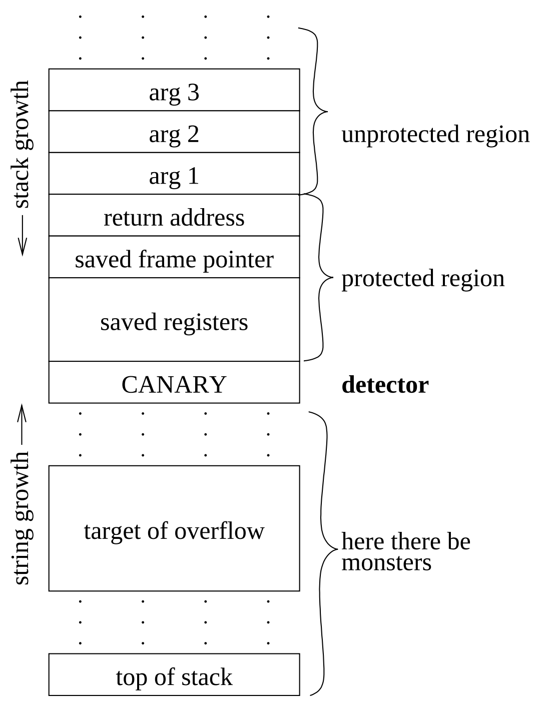

Figure 1: i386 Stack Layout

The *defense* is to insert a **canary location** immediately before the control information in each frame on the stack. See Figure 1. Any sequential write through memory, such as by a buffer overflow, that tries to rewrite the control information will be forced to also rewrite the canary location. Then the remaining problem is to make the value of the canary something that's hard to spoof. The canary location is **initialized** immediately after the control values are saved, and **checked** immediately before the control values are restored.

The control region is protected by virtue of the fact that the canary is checked before each use of the protected information. The arguments sitting above that are used sooner than that, so aren't protected.

3.1 Types of Canaries

There are three kinds of canaries, each with a different strength and weakness:

- **terminator canaries** detect runaway strings, but is a known value.
- **random canaries** detect all sequential memory writes that don't know its secret value.
- **random XOR canaries** are random canaries that might also detect random-access memory writes into the protected region.

Terminator canaries leverage the observation that most *stack* buffer overflows involve string operations, and not the memory copy operations almost always applied instead to heap allocated objects, by using a value composed of four different string terminators (CR, LF, Null, and -1). The attacker can't write the terminator character sequence for the particular string operation being used to memory and then continue writing, because one or more of the terminator characters halt the string operation.

If the exploit gets to overwrite the canary more than once, it can overwrite the protected control information on the first write, and then reconstruct the canary value with consecutive writes. It's not known how rare multiple write vulnerabilities are.

Any memory copy will be able to write the terminator canary value.

Random canaries assume that the exploit can sequentially write any value it wants and keep going. So it forces the exploit to know a 32-bit secret random number thats retrieved from a global variable that's initialized to a different value each time the program is executed. Memory protection techniques can be used to protect the global from writing, such as isolating the global on its own page and bracketing it with "red" (unmapped) pages. The exploit might also be able to get the victim program to tell it what the current random canary value is, having it read from either the stack or from the global.

If the attacker can also deploy an exploit that can read the random canary value from anyplace it might also reside in memory, then both string and memory copies can easily overwrite the correct value (unless it contains the appropriate string termination characters, and thus less entropy).

Random XOR canaries assume that the exploit might be able to random-access write to the location of some of the protected information [2]. So in addition to employing the random canary defense, some or all of the saved control information is exclusive-or "encrypted"[8] with the random canary value, storing the result in the canary location. Then to change the protected control information the attacker needs to deploy an exploit that sets the canary location to the exclusive-or of the random canary value and the new values of the control values used in the full "encryption."

Random XOR canaries have the same weakness as random canaries above.

3.2 Examples of Canaries

All versions of StackGuard have provided terminator canaries. We know of no alternate implementations that provide this type of canary.

All versions of StackGuard up to, but excluding the latest version, provide random canaries. Propolice provides random canaries.

Only the mid-1999 version of StackGuard provided random XOR canaries, protecting only the region containing the return address. When we checked in early 2003, Visual C++.net's `/gs` option performed exactly the same algorithm with the return address, but positioned the canary to also protect (without the "encryption") the saved frame pointer.

[8]Some people are uncomfortable with the use of the word "encryption" to protect integrity instead of confidentiality, hence the scare quotes.

4 Implementation

The first thing a function does on entry is to save the caller's control information on the stack. The region of memory used for this must be protected by a canary location, which is initialized with the desired canary value.

The last thing a function does on exit is to restore the caller's control information from the region of memory set aside for that. But, before that restoration can take place, the canary value must be checked to see if it has changed. If it has, the stack has been corrupted, and the process is killed after a suitable Intrusion Response System has been notified.

The code generators are:

- **determine canary location**—decide on where the canary location is going to be on the stack and how the below operations are going to refer to it.
- **allocate canary location**—make space for the canary in the stack layout in a memory location close to and preceeding the region containing the saved control values to be protected.
- **initialize canary**—give the canary location its correct value before any operation happens that could rewrite it or its protected region of memory.
- **check canary location**—check that the canary location contains its correct value, after any operation happens that could rewrite it or its protected region, but before the saved control information it protects is restored with corrupted values. If

the check fails, invoke the fail stop operation below.

- **deallocate canary location**—remove the space made for the canary on the stack by the allocate canary location operation above.
- **perform fail stop**—send the mangled name[9], the type of canary, the correct value of the canary, and the corrupted new value of the canary to a security fault handler.

The traditional way to implement StackGuard has been to modify the function prologue and epilogue code generators, which are responsible for causing the machine instructions to be emitted that save the caller's frame pointer at the beginning of the frame (if frame pointers are enabled), saving registers, establishing the position independent code pointer if it is enabled, and possibly aligning the stack pointer to some boundary.[10]

We've decided on terminator canaries on the basis of the observation that nearly all *stack* overflows are via string operations.

It should be noted that the order of the application of the code generators is different than the order that the emitted code appears in the generated function. In particular, first, the body of the function is converted to RTL. Then a number of optimizations take place, until the sibling and tail recursion optimization makes its decisions available. Then the **initialize canary location** operation is added to the beginning, and the **check canary location** operation is added to all the exit points. Then some more RTL optimizations are performed until the function prologue adds code to the beginning and invokes the **allocate canary location** code generator, and the function epilogue adds code to all the exit points and invokes the **deallocate canary location** code generator.

[9]The variable containing the unmangled name isn't always initialized at the time it is needed.

[10]The author believes his IA-32 bias here merely adds concreteness to his examples, and doesn't build in bad assumptions.

In this paper, we try to keep a clear distinction between "code generators" and "operations." Code generators might be invoked in an arbitrary order to emit operations that appear in a desired order in the object code.

In an earlier section, we were critical of Propolice's *design*. In the remainder of this section, despite that it really does successfully recompile practically all of the Redhat 7.3 distribution for a production quality distribution, we are critical of StackGuard's *implementation* for the remainder of this section.

4.1 Determine Canary Location

The **determine canary location** code generator is architecture *specific*, since it needs to know how the stack is laid out.

Both the **initialize canary location** and **check canary location** code generators need the architecture specific RTX for referring to the location of the canary. But both are invoked before the architecture specific **allocate canary location** and **deallocate canary location** code generators are. It turned out that the i386 backend placed alignment padding (`padding1`) right where the soft frame pointer points:

```
rtx canary_loc
  = gen_rtx_MEM (SImode,
      frame_pointer_rtx);
```

and that was a nice location for the canary.

Marking the location volatile:

```
MEM_VOLATILE_P (canary_loc) = 1;
```

was required for the **initialize canary location** code generator to keep its RTL from floating past things that could corrupt the canary.

But, it broke the GCSE pass of the optimizer for the **check canary location** code generator, apparently due to the way the infinite loop in the **perform fail stop** was constructed, and it appears not to be required to keep the RTL sufficiently pinned down.

4.2 Allocate Canary Location

The **allocate canary location** code generator is architecture *specific*, since it runs in the architecture specific function prologue code generator.

For i386, it turns out to be very simple. Currently, the i386 architecture's `ix86_compute_frame_size` function does alignment padding between the autovariable region and the saved control information region. The solution is to add another alignment to `padding1` if it's not big enough to hold the padding.

Since the padding is allocated when the stack pointer is decremented (stack grows downward) to also allocate the autovariables, the allocation has no performance impact at runtime.

4.3 Initialize Canary Location

The **initialize canary location** code generator is architecture *independent*, since it just inserts an assignment into the RTL of the current function immediately after the sibling and tail recursion recognition optimization:

```
emit_move_insn (canary_loc,
 GEN_INT(terminator_canary_host_value));
```

where the canary value happens to be a simple expression[11] not requiring evaluation as an expression:

```
static const int
 terminator_canary_host_value
 = 0x000aff0d;
```

if it wasn't such a simple expression, then you would need to worry more about its temporaries being spilled to the stack *where they can be attacked*. Random and random XOR canary value expressions are largely non-simple, especially when being compiled for position independent code.

The above RTL sequence is inserted before the first non-NOTE RTL in the current function. As remarked in section 4.1 above, designating `canary_loc` as volatile appears to be sufficient to keep the it from floating past something that could corrupt the protected control information, but this isn't very comfortable. I've been hoping to stumble on a good way to insert barriers in the RTL instead of depending on volatile.

Sometimes the machine instruction generated needs a register, and usually it doesn't. Thus the late insertion might confuse late stages of register allocation depending on information from stages earlier than the insertion.

4.4 Check Canary Location

The **check canary location** code generator is architecture *independent*, since it just inserts a conditional branch into the RTL of the current function immediately after the sibling and tail recursion recognition optimization (see the discussion in subsection 4.3 above):

```
emit_cmp_and_jump_insns
```

[11] There are two different sorts of simplicity, one at the source code level (see the talk on TreeSSA), and one at the RTL to ASM conversion level.

```
(canary_loc,
  GEN_INT(terminator_canary_host_value),
  /* comparison = */ comparison, /* EQ/NE */
  /* size       = */ 0,
  /* mode       = */ SImode,
  /* unsignedp  = */ 0,
  /* label      = */ else_label);
```

which is appended to the end of each function, and inserted before each tail-call.

If the expression for the canary is not simple, then you need to make sure that it doesn't grab corruptable temporaries from the same computation in the **initializer canary location** code at the beginning of the function.

At the end of each function, the comparison argument is EQ, because the test is used to branch around the **perform fail stop** code whose generation is described in the section 4.6 below. The `else_label` label jumped to in that case is refers to the normal function epilogue code that hasn't been generated yet.

Before each tail call, the comparison argument is NE, because the branch is to the **perform fail stop** code appended to the end of the function.

Dead code removal works correctly for all of these inserts.

The branch prediction of the EQ case appears to get flipped correctly to put the return on the fast-path.

The machine instruction seems to usually need a register, but sometimes not. The late insertion of this RTL when it needs a register may be causing the code generation in the "getdents" function in GLIBC which has the attribute `((regparm(3), stdcall))` to go awry in the GREG (global register allocation) pass.

4.5 Deallocate Canary Location

The **deallocate canary location** code generator is architecture *specific*, since it runs in the architecture specific function prologue code generator.

For i386, it turns out to be absolutely free. The alignment padding where the location resides is stripped off at the same time as the autovariables.

4.6 Perform Fail Stop

The **perform fail stop** code generator is architecture *independent*, since it mostly just inserts a call to an external function named "`__canary_death_handler`" using the GCC's internal `emit_library_call` procedure.

The `__canary_death_handler` is invoked with information such as the current procedure name, the version of stackguard, the type of canary, what the canary value was supposed to be, and what the canary is now that it has been corrupted.

It's not expected that recovery is possible from a corrupted stack, so if the `__canary_death_handler` returns control from its call, something is very wrong, and the only thing reliable to do is go into an infinite loop. The correct way to recover would be to setup a different stack that returns control to different code.

Exception handling does not work here, since the stack is corrupt. If you like, you might consider this to be a security *fault* as opposed to an exception.

The late insertion of the `emit_library_call` into the RTL might be causing trouble.

4.7 Summary of problems

Moving the RTL code generators for **initialize canary location** and **check canary location** out of the function prologue and epiloque code generators was essential for two reasons. First, the prologue and epilogue were invoked too late to be able to generate the desired RTL. Second, the prologue and epilogue are architecture specific, and architecture independence is highly desired.

However, the movement of these two generators appeared to be blocked in two ways. First, they appeared to only work correctly around the time of the sibling and tail recursion optimization pass. Second, this was fortuituous because this was also the first point where the insert points became available for adding **check canary location** immediately before function exiting branches (that is, before return statements and tail calls).

Ideally the movement of these two generators should proceed to the point that AST is converted to RTL (which would also fix any problems the call to `emit_library_call` might be causing), but that implies that sibling and tail recursion recognition also move to that point.

4.8 Debuggers, Exception Handlers, and Other Stack Crawlers

Previous versions of StackGuard placed the canary location immediately before the return address on the stack. This was quite confusing to programs that did their own ad hoc parsing of the stack, such as GDB, Mozilla's module loading mechanism, and IBM's Java JIT compiler.

All of these became non-problems with the latest version of StackGuard, which places the canary location in a spot where nothing's supposed to be.

The aspell packages for Red Hat 7.3 has a complex enough class system for handling "file not found" exceptions that something throws it off, and it runs off the top of the stack without finding an exception handler, and abort()'s. This appears to be a problem with a dwarf annotation interaction with the old exception handler in GCC 2.96-113, which would probably be fixed (or at least completely different) in current GCC.

Exception handlers should check canaries for each frame as they crawl up the stack soas not to use corrupted information. We're hoping to add such support to the new exception handler in GCC 3.x, just as soon as a distribution that we can build, strenuously test, and release uses it.

4.9 Testing

The assembly output of the StackGuard compiler has been inspected for correct output for many optimization levels, with and without frame pointers, PIC and non-PIC, inlines, and nested function declarations.

A parser of the disassembler output for the StackGuarded version of the main glibc library libc.so.6 was done. Every procedure was correctly StackGuarded, and several tens of tail-call sites were observed.

Previous versions of StackGuard rebuilt Red Hat Linux 5.1, 5.2, 6.0, 6.1, and 7.0. The current version of StackGuard has rebuilt Red Hat 7.2 and 7.3, with a rebuild of Red Hat 8 in progress.

The `getdents` function in GLIBC in the Redhat 8 rebuild has problems. It looks like the late insertion of the canary check causes GREG optimization phase to drive something insane enough to apparently be confused about the sizes of various types. The RTL for the function suddenly becomes quite different starting about halfway through the function after that pass, with tremendous movement of temporary and register initializations.

5 Performance Benchmarks

Formal performance benchmarks are currently under way, but were not complete at press time. Previous performance benchmarks on StackGaurd 2 [5, 7] show very marginal overhead on real loads, especially those programs that actually face network attack. In particular, benchmarks of Apache loaded by webstone, and throughput benchmarks of OpenSSH through the loopback interface, show overhead that is within measurement noise: `http://immunix.org/StackGuard/performance.html`. We expect similar performance from StackGuard 3.

6 Security Benchmarks

Security testing (like total correctness) is always problematic, because you cannot test for security, you can only detect *vulnerability*. As in performance, security testing is still under way at press time. Past security testing of StackGuard [6, 5, 7] shows that StackGuard is effective in its narrow goal of stopping classic stack smashing attacks. Furthermore, unlike some of the kernel-based defenses [8] when StackGuard stops a vulnerability, it is *stopped*, i.e. revising of the attack code does not result in bypassing StackGuard protection.

The major exception to this claim is that some exploits can attack the frame pointer, which was left unprotected in StackGuard 1 and 2. StackGuard 3 fixes this by moving the canary below the frame pointer.

7 Conclusion

StackGuard is a very modest sized patch, with modest performance and legacy compatibility costs, and yet solves a very large problem: chronic stack smashing buffer overflows. Despite having been first innovated in GCC [6], Microsoft has implemented a StackGuard-like feature [4] *as a standard feature* ahead of GCC. We propose that it would be beneficial for the GCC user community if the StackGuard security optimization became a standard compile option in GCC.

8 Availability

StackGuard has always been distributed under the GPL, and is currently available at `http://immunix.org/stackguard.html`.

Copyright assignment to the FSF for the StackGuard patches is in progress.

References

[1] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *2000 USENIX Annual Technical Conference*, San Diego, CA, June 18-23 2000.

[2] “Bulba” and ”Kil3r”. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[3] Compaq. `ccc` C Compiler for Linux. `http://www.unix.digital.com/linux/compaq_c/`, 1999.

[4] Crispin Cowan. Re: In response to alleged vulnerabilities in Microsoft Visual C++ security checks feature. `http://online.securityfocus.com/archive/1/256416`, February 14 2002. Bugtraq.

[5] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[7] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, `http://schafercorp-ballston.com/discex`.

[8] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[9] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, January 2 1997.

[10] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, November 21 2000.

[11] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, Washington, DC, August 2001.

[12] Richard Jones and Paul Kelly. Bounds Checking for C. `http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`, July 1995.

[13] Greg McGary. Bounds Checking in C & C++ Using Bounded Pointers. `http://gcc.gnu.org/projects/bp/main.html`, 2000.

[14] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[15] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[16] Alexander Snarskii. FreeBSD Stack Integrity Patch. `ftp://ftp.lucky.net/pub/unix/local/libc-letter`, 1997.

[17] 'The PaX Team'. PaX. `http://pageexec.virtualave.net/`, May 2003.

[18] Herman ten Brugge. Bounds Checking C Compiler. `http://web.inter.NL.net/hcc/Haj.Ten.Brugge/`, 1998.

[19] "Vendicator". Stack Shield. `http://www.angelfire.com/sk/stackshield/`, January 7 2000.

A Maintenance Programmer's View of GCC

Zachary Weinberg
CodeSourcery, LLC
`zack@codesourcery.com`

Abstract

GCC is considered more difficult to modify or debug than other programs of similar size. This paper will investigate the reasons for this difficulty, from the point of view of a maintenance programmer: someone producing a small patch to fix a bug or implement a feature, without causing new problems for unrelated use. Because the development tree's head is expected to be functional at all times, such incremental changes are normal—even regular contributors are in the maintenance programmer's shoes.

1 Introduction

Who is a maintenance programmer? Anyone working to implement a specific feature, or fix a specific bug, without introducing new problems at the same time. Anyone with limited time to investigate the situation and become familiar with the code.

Maintenance programmers are faced with both technical and procedural hurdles. GCC has a complex task to accomplish, but even so GCC is far more complicated than it needs to be, which makes it harder to modify the code than it should be. Further, once one does successfully make a change, it is hard to get it accepted to the official source tree. The procedural requirements are stringent for good reason, but still discourage people from contributing, and cause patches that were 90% correct to be rejected.

GCC's development process requires everyone to work incrementally and make minimally invasive changes. Although not a formal requirement, it is a consequence of the no-regressions policy for check-ins, coupled with the extreme complexity of the source code. A simple change might turn out to have ramifications everywhere. A few individuals know the compiler inside out; they can pull off hugely invasive changes without breaking anything. Most of us are not that good, so we must take small steps, testing carefully as we go. Furthermore, even regular contributors often have difficulty getting their patches approved. And, of course, we all have lots of demands on our attention, so there is never enough time to work out the perfect design. Therefore, making life easier for a maintenance programmer who might have just one patch to contribute will make regular contributors' lives easier as well.

2 Technical Hurdles

Let's take a moment and look at the GCC source tree from 10,000 feet up. Table 1 breaks down the code by category. There are about 1.6 million lines in total, ignoring comments. Just over half of this is C; there are also substantial bodies of Ada, Java, and C++. Machine description files are written in a domain-specific, Lisp-like language, which accounts for ten percent of the total.

By nature, any program of this size is going to be nontrivial to work with. Furthermore, a

By category		
Core compiler	250,000	
Back ends	410,000	
biggest	40,000	(rs6000)
smallest	2,200	(fr30)
median	6,500	(v850)
Front ends	480,000	
biggest	221,000	(ada)
smallest	2,500	(treelang)
median	59,000	(java)
Runtime libraries	458,000	
biggest	274,000	(java)
smallest	8,200	(objc)
median	11,000	(f77)
Total	1,639,000	

By language		
C	861,000	53%
Ada	298,000	18%
MD	170,000	10%
Java	127,000	8%
C++	105,000	6%
Other	78,000	5%

Physical source line counts, generated using SLOCCount [1]. MD = machine description.

Table 1: GCC 3.3 source code breakdown

compiler is necessarily more complicated than the average program of similar size, since it contains many algorithms and techniques that require arcane theoretical knowledge to understand. SSA (static single assignment) form, for instance, takes a good chapter of exposition to explain. GCC is necessarily more complicated than the average compiler, since it supports so many input languages and target architectures in its official distribution alone. Many other compilers support only one or two targets.

Even so, GCC's code could be much simpler and easier to maintain. This can be put down to three primary causes: incomplete transitions, functional duplication, and inadequate modularity.

2.1 Incomplete transitions

Incomplete transitions occur whenever anyone invents a new, better way to do something, but does not update every last bit of code that used to do it the old way. They might run out of time; they might not have the necessary expertise; they might just not be able to find it. Whatever the reason, an old API cannot be removed from the compiler until there are no remaining uses. An incomplete transition thus means that for an extended period there are two or more ways to do something. One is preferred, but it may not be obvious which. Someone writing new code that needs to do whatever it is, might pick the obsolete technique, further delaying the day when the old API can be removed.

Incomplete transitions are most common in the API for writing architecture back ends. For example, there are two ways to write a machine-specific peephole optimization. Both do pattern matching on the stream of RTL insns constituting the intermediate representation of a function. The old way (`define_peephole`) overrides the normal mechanism for writing out assembly language, substituting its own text. No further optimization can happen to the result. The new way (`define_peephole2`) replaces the matched insns with new ones, which can then be optimized further. For instance, the second instruction scheduling pass sees the result of new peephole optimizations.

The new construct was created in 1999, but of the 37 back ends present in GCC 3.3, only six use it exclusively. Fifteen still use `define_peephole` exclusively, and six more have both. (Ten have no peepholes at all.) Now, peephole optimization is a relatively minor part of a back end. The majority of the architectures that use either variety define fewer than ten. In terms of code generation, using `define_peephole2` is most beneficial for architectures that use instruction scheduling. The maintainers of any given architecture have no real incentive to update it to the newer style. >From a maintenance programmer's point of view, this situation is very bad. The presence of two functionally-equivalent mechanisms for the same basic operation adds complexity and increases the likelihood that something will be broken accidentally.

Peephole optimizations of either variety rarely cause trouble, because the machine-independent code that applies them is small and robust, so it is unlikely to be broken by an unrelated change. However, consider the `cc0` mechanism, which is the older of two possible ways to represent condition codes in a machine description. There are 805 lines of code in the core compiler that are used only by `cc0` architectures, and 79 lines of code used only by non-`cc0` architectures, scattered through 28 files in 121 individual `#if` blocks. This is not a lot of code compared to the total size of the compiler, but it is all in critical places, affecting most of the major optimization passes. Testing on a non-`cc0` architecture will not reveal brokenness in the code used exclusively by `cc0` architectures, or vice versa. The only widely-used architecture that still uses this mechanism[1] is the m68k, and m68k environments are all slow enough that no one wants to test them. It is not surprising, then, that all `cc0` architectures were broken for some time last year.

[1] If `(cc0)` appears only in `define_expand` forms that generate no RTL, that machine description does not use the `cc0` mechanism.

When someone discovers that a target they wanted to test is broken for some other reason, their usual response is not to bother testing that target anymore. This of course means that nothing stops the target from accumulating faults. By the time someone comes along who wants it to work, it may be easier to start from scratch than to fix all the faults. This is especially true for OS-specific configurations, which break more easily than architectures and require relatively little effort to rewrite from scratch, especially if they are similar enough to the generic Unix that GCC takes for its default.

Recent experience [2] suggests that even CPU ports can age to the point where starting over might be easier. The MIPS back end had not been kept up to date for several years; it was overhauled starting in late 2002, with most of the work done by Richard Sandiford and Eric Christopher. This took six months start to finish, with approximately eight thousand lines of code changed, which is comparable to the effort required to write a minimal back end from scratch. Of course, the MIPS back end is not minimal; starting from scratch might have meant abandoning many of the sub-architectures and operating systems that it currently supports.

A primary driver for the overhaul was the desire to avoid use of the macro instructions provided by the MIPS assembler. This can also be seen as a transition, but not of an API; rather, the preferred style for machine descriptions has changed. When the MIPS port was originally written, the macro instructions were a convenient way to simplify the compiler's job. Now they are seen as a hindrance to quality code generation, requiring awkward workarounds in the compiler.

2.2 Functional duplication

Functional duplication occurs when two components both implement some capability that could be shared. A long-standing case exists in the RTL simplification code. When Jeff Law created `simplify-rtx.c` in 1999, he included a comment which gives the flavor of the problem:

> Right now GCC has three (yes, three) major bodies of RTL simplification code that need to be unified.
>
> 1. `fold_rtx` in `cse.c`. This code uses various CSE specific information to aid in RTL simplification.
> 2. `combine_simplify_rtx` in `combine.c`. Similar to `fold_rtx`, except that it uses combine specific information to aid in RTL simplification.
> 3. The routines in this file.
>
> . . . It's totally silly that when we add a simplification that it needs to be added to 4 places (3 for RTL simplification and 1 for tree simplification).

It is worth pointing out that at 8,790 lines of code, `combine.c` is the second longest file in the core compiler. Much of this bulk is `combine_simplify_rtx` and its subroutines.

Functional duplication is less likely to cause breakage than incomplete transitions. Continuing with this example, all the RTL simplifiers are exercised by the normal testing procedure, so it is unlikely that one of them will remain broken for an extended period. However, the answer to the question "Why did this bad optimization happen, when I can see that the code in file A is correct?" may well be "because that transformation is duplicated in file B, only with bugs." Furthermore, this duplication invites people to update one set of simplifiers and not another, which means that whether or not an RTL construct gets simplified depends on which optimizer pass encounters it. And, of course, it causes the compiler's runtime image to be bigger than necessary, which contributes to compiler-speed problems by wasting space in the instruction cache.

Law's comment hints at a deeper cause of functional duplication, namely, that we have two different intermediate representations (trees and RTL). In the past, almost all of the compiler dealt exclusively with RTL so this was not a cause for concern. We now do some optimizations at the tree level, and lots more are planned. It would be useful to share code between tree optimizers and RTL optimizers as much as possible. This has already been done for the control-flow graph, on the `tree-ssa` branch. If the data structure holding an expression to be simplified could be made opaque to the code computing the simplification, the same could be done for the algebraic simplification library.

Functional duplication also occurs when a module exists that logically should be responsible for some task, but is not presently capable of it. Instead of fixing the existing module so that it is capable, often people choose to build something new from scratch, which is easier in the short term. A good example here is the language-independent tree-to-RTL converter (`stmt.c`, `expr.c`, etc.) It is one of the oldest parts of the compiler. It still reflects design decisions made when C was the only supported language, and the tree representation was used for only one source statement at a time. When front ends started being rewritten for whole-function tree representations, no one wanted to update the converter to match.

Instead, each front end that now uses whole-function trees contains duplicated tree-walking logic, so that it can continue to feed the tree-to-RTL converter one statement at a time.

This duplication not only causes the problems described above, but also hinders conversion of other front ends to whole-function processing, because they would have to duplicate this code again. Nor is there agreement on the form of whole-function trees. The maintainers of the C language family developed one such representation; independently, the Java maintainers developed another, incompatible representation. This prevented the tree inliner developed for C from being used for Java. Rather than copy the file over, it has been heavily `#ifdef`ed, which may or may not be an improvement. (The people working on the `tree-ssa` branch have a major goal of developing a proper, language-independent, whole-function tree representation.)

When a transition is finally completed, or duplicate code finally collapsed together, it may still leave vestiges behind. The garbage collector was completed in late 1999, but most of the obstack allocation scheme that it obsoleted stuck around until late 2000. We are still finding traces of it now, in the second quarter of 2003.

Everyone likes deleting code, so why do vestiges stick around? People usually find vestigial code when working on something else. To delete it, they would need to stop whatever they were doing at the time, construct a fresh CVS checkout, delete the vestige, do a full test cycle to make sure nothing broke, then submit the patch and wait for approval. All this time, they would not be working on whatever they originally planned to work on. We will come back to time consumed by procedures later.

Another reason is, it is hard to distinguish code that is left over from code that was never completed, or that was written in anticipation of a use that never materialized. One can usually figure it out from mailing list traffic or CVS logs, but only with practice. However, no matter what its intended function is or was, code that is not being used now should be deleted; even if a future use was planned, it is likely never to happen.[2] If someone does have a use for a body of unused code in the immediate future, they will undoubtedly say so when its removal is proposed.

2.3 Inadequate modularity

Unfortunately, much code that has no apparent function will cause something to break if it is taken out. This is the problem of inadequate modularity. GCC is composed of a lot of logical modules, but the boundaries between these modules are ill-defined and poorly documented. Any given behavior has a good chance of being required by some other module. For instance, the C compiler reads the first line of its input much earlier than would be natural. This is because some of the debugging-information generators want to know what the name of the primary source file is, when their initialization hook runs. These two things may sound like they have nothing to do with each other. But if the C compiler is handed already-preprocessed input, the primary source file is not the file on the command line. It is the file named by the # marker on the first line of the file on the command line. Therefore, in order to initialize the debug-info generator properly, that first line has to be read. [3]

The interface between language front ends and the core compiler is especially prone to this sort of problem. This stems mostly from the ad-hoc way in which the front-end interface has evolved. It has never been documented, yet there are seven different lan-

[2]This is the YAGNI (You Aren't Gonna Need It) principle.

guages using it in the current source tree, plus a few more maintained separately. As languages were added, their developers generally tweaked the tree specification around as they saw fit, without much coordination. It was originally intended to cover the needs of GNU extended C only, and still reflects that in some aspects. For instance, the Java front end has interesting kludges in it to cope with the allegedly language-independent `builtins.def`, which is full of C-specific notions like `va_list`. Or, consider the way each back end specifies its platform's fundamental data types: the `*_TYPE` and `*_TYPE_SIZE` macros. These macros map directly onto the fundamental data types of C; if this is a poor match to the language being implemented, one is in trouble. To be fair, most modern platforms define their most basic ABI in a similar fashion, so one might be in trouble anyway.

The interface between the core compiler and a target-specific back end is also very fuzzy. The most basic parts are in the machine description, which is pretty well defined and documented, but there are lots of little details handled by defining macros, which are then visible to the entire compiler, including the front ends. A naive count finds close to five thousand different macro names defined by header files in GCC 3.3's `config` directory. Some of these are internal to one architecture, and some of the headers are not used during the compiler build itself, but there is no easy way to tell them apart. Since the macros are visible to every part of the compiler, every part of the compiler can use them, and does. A target must define almost all of the macros used by the core compiler, which leads to massive duplication.

There is ongoing work to convert all of these macros to data members or function pointers in a global object called `targetm`, which forces a more structured approach. The people doing the conversion are taking the opportunity to clean up the interfaces and create sensible defaults. Thus there is hope that this problem will dwindle as time goes by. However, the conversion project could drag on for years, becoming another of the incomplete transitions that were discussed above. GCC 3.3 has about seventy members of the `targetm` structure; a complete job will require about five hundred, but most targets will not need to override the defaults for most of them.

The core compilers is not free of modularity problems, either. The RTL optimizers are structured as a pipeline of passes, and what each pass does to the code is reflected in the insn chain. On its face that is a modular design. However, there are undocumented limitations to what each optimization pass can handle, which impose constraints on earlier passes. For instance, the first local CSE pass is a waste of time at this point, because the GCSE pass is more powerful…except that GCSE is not prepared to deal with certain high-level constructs that local CSE eliminates, such as `addressof`. This is doubly unfortunate, because GCSE could do a better job than CSE of handling the high level RTL, if it only knew how. [4]

2.4 Style

We should not neglect aesthetic concerns. Anything that makes code harder to understand, hides bugs from developers. Anything that makes code harder to restructure, hinders developers from resolving the more serious problems discussed above. GCC's primary failing in this domain is by virtue of sheer size. Particularly in the older parts of the compiler, it is common to find a single function so large and convoluted that a human reader cannot remember all its details. Some may have grown by accretion: `expand_expr` for example may have been much smaller when there were fewer kinds of tree to be considered, or when fewer optimizations were attempted at that time. Others are perhaps stylistically inspired by the "Pastel" compiler that predated GCC 1, which was in a language that supported nested functions; very large outer functions would have been more natural in that language. [5] These functions often maintain state in local variables of an outer block; performing the "obvious" refactor of pulling the inner blocks out to their own functions can cause mysterious failures, since the outer variables are no longer visible.

Gigantic controlling expressions in `if` statements are also common. Here the problem is notational. Such expressions often turn out to be performing pattern matching on RTL, in the most straightforward fashion possible in C. If it were possible to write these expressions in the language used for machine descriptions they would be far more readable.

The macros, idioms, and style constraints which permitted us to build GCC with compilers that predate the 1990 C standard should also be seen as an issue of aesthetics. We already enjoy the benefits of most of standard C's features, such as prototyped functions. However, eliminating all these idioms (as we can now do) will make it easier to read the code, and this is not a trivial thing. Just the removal of the macros that cloak the differences between traditional and standard C with regard to variable-length argument lists should be a great step forward.

3 Procedural hurdles

Once again, let's take a moment and look from 10,000 feet up, this time at the process for contributing a patch to GCC. For this purpose we shall postulate a contributor named Alice, who has a copyright assignment on file, but has not yet been granted write-after-approval privileges, and proposes to fix a bug which appears in the GNATS database.

The first step is to get a copy of the development tree (i.e. CVS HEAD). Then the bug must be reproduced and fixed. The potential difficulties with that were covered above.

Next, Alice must carry out a full bootstrap and test cycle. This is not very hard once you know how. Typical first-time gotchas include configuring in the wrong place or with the wrong sort of pathname, and tripping over a Makefile bug; having the wrong version of GNAT installed, so the Ada front end cannot be built;[3] having the wrong version of autoconf installed, so the configure scripts are broken; and finally, having a broken DejaGNU installation, so the test suite reports thousands of spurious failures. Once all these issues are resolved, Alice gets to sit back and wait for at least two hours. Depending on how slow her computer is, it might be more like a full day. There is also the possibility that the test cycle will fail because someone else checked in a patch which broke the compiler.

Assuming that went fine, the patch is now to be submitted for review. Alice may be ignored for weeks on end, depending on how busy the official maintainer of that component is, whether she has submitted patches before, and how important the bug seems to be. Once someone does get around to responding, there is a good chance that the patch will be torn to shreds and sent back for revision, repeatedly. Alice might get frustrated and give up. If she persists, the patch will eventually get approved. Now (since she lacks write privileges) the person who approved it is responsible for committing it and closing the entry in the GNATS database. If Alice keeps submitting good patches, she will

[3]This is not currently a requirement, but Alice is being thorough.

be granted write-after-approval privilege. She can then do these last steps herself.

It is not terribly useful to speculate about the ultimate causes of the procedural hurdles that can be seen in this description. Instead, we will categorize them by nature, as slow or tedious tasks; problems coping with tools; and human error.

3.1 Slow or tedious tasks

One of the most important procedural hurdles is the sheer amount of time it takes to develop a patch and get it committed to CVS. Alice had to wait for review, but let's defer that issue for later. Even people with global write privileges are expected to carry out a full bootstrap and test cycle on at least one target, including all languages, before installation. This takes two hours on a 2GHz P4 with 512MB of real RAM, running Linux 2.4. A slower CPU, less memory, or a less efficient operating system will all cause it to be dramatically slower. The author is personally aware of an environment in active use which is centered around UltraSPARC 5 machines running Solaris 2.5.1. On this platform a cross-compiler build, C and C++ only, takes six hours; an all-language bootstrap would take even longer.

On a sufficiently efficient operating system, the bottleneck for a bootstrap is CPU time expended by the compiler itself. This parallelizes well; on a multiprocessor system, `make -j`N will reliably divide the time for bootstrap by N, up to some limit. Experimentation is usually required to find the best value to use. However, using parallel make can expose missing-dependency bugs in the Makefile. Since the header dependency lists are maintained by hand, it is easy for these bugs to creep in. Some makefiles have not been written with parallel make in mind; for instance, at the time of writing, `make gnatlib_and_tools` does not work at all in parallel mode. Also, DejaGNU has no ability to run tests in parallel, so the entire test suite must be run serially.

Bootstrap time accounts for the majority of time spent waiting for a computer to do something. However, CVS operations should not be neglected in this regard. On a higher-end ADSL connection (1.5Mbps down/384Kbps up) a `cvs update` on the mainline takes fifteen seconds—if it has nothing to do, and there are no modified files. If it has updates to download, or potentially modified files that have to be checked (by sending the full text of the file to the server for comparison) it can take substantially longer. Branches are also slower; on the 3.3 release branch, an update with nothing to do and no modified files takes a minute and a half. Recursive commit and diff operations take a similar amount of time.

Once a patch is fully tested, the contributor must write an explanation of the changes made, for the `gcc-patches` mailing list, and a ChangeLog entry. Working out long ChangeLog entries can be tedious. To some extent it can be automated; for example, a simple perl script can extract the names of all the files and functions touched by a patch and format them in ChangeLog style, leaving one to write the "what was done to each" comment, but that part can still be tedious for a long change. This text has to be copied from the message into all of the relevant ChangeLog files, and into the CVS commit log; it is easy to make a mistake along the way.

All of this places a lower bound on the time it takes to develop or revise a patch. Even the most trivial changes have to go through this process, because they *could* have broken something. The time it took to design and implement the change itself is neglected here. That time cannot be said to have been wasted, except insofar as it may have been harder than necessary to make a change, which was discussed above. Of course, the lower bound is only met if the patch works the first time. If the patch causes a regression in some part of the testsuite that must be fixed, then the bootstrap must be repeated.

And the lower bound is only met if the contributor can commit his or her own patches without approval. Otherwise, there will be some time spent waiting for the patch to be reviewed. It is not uncommon to get no response at all to a patch, or even to repeated inquiries. This is not because anyone hates the patch or its contributor. Most often patches are ignored because everyone with the authority and the experience to review the patch is just too busy that week. A lot of GCC's code is listed as maintained by one of the people with global write privileges, or else has no listed maintainer at all. Either way, the set of people who can approve a change to that component is limited to those with global privileges, all of whom are busy. A related problem is that people who do not have authority to approve patches often refrain from commenting on them, even though their opinions are still valued.[4]

Another contributing factor is that some patches are too hard to review. This happens when a patch tries to do too much at once, or when the person who wrote it did not explain its motivation well enough. What seems simple and obvious for the person who was just immersed in the relevant area, may not be obvious at all to anyone else. Splitting patches into minimal changes and explaining them well are both learned skills. At present, we expect people to pick them up by osmosis, but not everyone can learn like that.

Sometimes a patch is not quite right, and sometimes a patch addresses an issue that clearly needs addressing but does not do it in the way that the reviewers would like. When this happens, the reviewers will send the patch back for revisions. Sometimes they send it back so many times that the contributor gives up hope that it will ever be accepted. Then the patch, which might not have been perfect, but was an improvement over the status quo, gets abandoned.

It does happen that patches are ignored intentionally, in order to reject them without having to offer feedback. In most cases, this happens because everyone who could review the patch feels that they cannot have a productive discussion with the person who submitted it. That might be the submitter's fault—there is just no working with some people—but it is much more likely to be a failure of the community. Fortunately this is rare.

3.2 Coping with tools

The tools which give people the most trouble on a day-to-day basis are DejaGNU and the `autoconf` family. To begin with the most straightforward issue, the GCC testsuite always produces a handful of "unexpected failure" (`FAIL`) results when run. These failures are *not* unexpected in the standard sense of the word. They do not change often. People who build the compiler on a daily basis and/or follow the `gcc-testresults` mailing list will know which unexpected failures are currently normal for a given environment. They are only unexpected in the sense that DejaGNU has not been advised to turn them into "expected failure" (`XFAIL`) results. Regular contributors are used to this. However, someone who does not build the compiler on a daily basis, or follow the test-results list, will not know whether a given unexpected failure is normal or not. If they are running the testsuite to make sure the compiler works, not having made any changes, they may believe there is something wrong with their environment, or a bug that is

[4]This is a variant of the "bikeshed effect." [7]

not already known. If they have made changes, they will not know whether or not their changes caused the unexpected failures. The only way they can be sure, in this latter case, is to do two complete test cycles from the same baseline code, one without the desired patch and one with. This doubles both the testing time and the disk space requirements, since it is necessary to keep both trees around for comparison.

Failures are not marked expected mainly because it is too awkward. At the least, it involves adding special tags to files in the testsuite. For test cases in the `c-torture` framework it involves creating special files containing snippets of Tcl code. What the tags or snippets should be is mostly undocumented. People usually do it by copy-and-paste from another test case. Further, DejaGNU's ability to describe the situations under which a failure is expected is quite limited. For instance, there is no way to specify that a test will fail if the necessary locale definitions are not installed, or that a test may sometimes (depending on system load) take so long to run that it times out.

There is also a general assumption that expected test failures are not going to be fixed anytime soon, whereas unexpected failures have someone looking at them right now. This discourages people from marking tests expected to fail, because they might be fixed soon and then the marking would have to be undone. Yet tests continue to fail "unexpectedly" for months on end.

If one does not have access to a hosted system for an architecture, one can still test some patches that affect it by building a cross compiler to a simulator target. The GDB source tree includes simulators for many popular architectures. It is easy to construct a combined tree including gcc, binutils, the simulator, and a minimal C runtime, in which to test the cross compiler. However, DejaGNU is prone to glitches when used with a simulator target. One common problem is complete failure to find `stdio.h` or `crt1.o`. One suspected cause of this is invoking `configure` by relative instead of absolute pathname.

Autoconf, automake, and libtool have all undergone backward-incompatible revisions in the past few years. One must have exactly the right version of each installed in order to regenerate GCC's `configure` scripts or Makefiles. For instance, all of the configure scripts presently require autoconf 2.13, which is the oldest version in common use. It is old enough that it is left out of the default installation of some newer operating systems, such as Red Hat 8.0. Use of a newer version might cause visible errors when the script is regenerated or run, or more insidiously it might just cause a small handful of features to be misidentified. Since GCC's Makefiles will automatically attempt to regenerate configure scripts that are older than the parent `configure.in`, a user may discover that the first build from a fresh working copy succeeds, but all subsequent builds mysteriously fail. Using the `contrib/gcc_update` script can prevent this problem, but it will not help someone who has modified the configure script.

It is harder to get in trouble just by having the wrong version of automake or libtool installed, because these tools are only run on specific user request. But one may still be stuck with no way to regenerate files under their control. The author has resorted to updating a generated `Makefile.in` by hand on several occasions.

3.3 Human error

From time to time someone checks in a patch which renders the tree unbuildable. Normally it worked just fine for the person who tested it, but breaks in a different environment. The problem may be target-specific, or involve only a language which is not supported by the tester's platform. Or perhaps the patch that was tested is different from what checked in, somehow. Whatever the cause, when this happens, everyone who did a `cvs update` just before starting their bootstrap cycle gets to wonder whether it was their changes that broke the tree.

A few years ago, a CVS checkout taken at a random point in time had a 34% chance of being unbuildable. [6] This is directly attributable to the two-year lapse between the 2.95.0 and 3.0.0 releases. During that time, latent bugs were continually introduced, until any given checkin had a good chance of triggering one. There was no concerted effort to flush these bugs out until the situation became dire enough to hinder day-to-day work. Since the institution of the three-stage development process, in mid-2001, unbuildable CVS checkouts happen only rarely, since the tree is regularly stabilized.

The automated testers operated by Geoff Keating, Phil Edwards, and others have also been instrumental in reducing the incidence of unbuildable source trees. A failure report from one of these testers can be trusted to indicate a genuine problem—no risk of a quirky environment causing issues—and conveniently lists all of the changes that could have been the proximate cause. They also make people aware of bugs immediately, rather than several weeks down the road when they no longer remember the details of their changes. Unfortunately, at present only a few platforms are monitored in this fashion.

Nowadays, build failures are usually addressed immediately, but testsuite regressions tend to linger for weeks on end. The author believes this is largely a matter of perception. Test cases are often contrived rather than reflective of real code, and the failure may seem unimportant. For instance, at the time of writing, half of the unexpected failures in the C testsuite for GCC 3.3 were caused by incorrect warning messages. Nonetheless, a general habit of ignoring persistent unexpected failures is not good practice.

4 Conclusion

Contributors to GCC face both technical and procedural challenges. These can be narrowed down to a short list of causes: incomplete transitions, functional duplication, and inadequate modularity; slow or tedious tasks, coping with tools, and human errors. Some of these problems are easy to address immediately, while others will require long-term, concerted effort. This paper limits itself to discussion of the problems. However, we are confident that solutions can be found.

5 Acknowledgements

This paper is largely based on my personal experience fixing bugs in older versions of GCC for a CodeSourcery client. I am also indebted to Neil Booth, Eric Christopher, and Richard Henderson for sharing their experiences. Michael Ellsworth, Kristen Hrycyk, David Johnson, Mark Mitchell, Jeffrey Oldham, and Nathan Sidwell were kind enough to comment on drafts.

Inspiration crystallized around the following IRC exchange between Phil Edwards and myself:

<**pme**> Every time I read *Snow Crash*, I wonder what a GCC "room" in the metaverse would look like.

<**zwol**> Take an H.R. Giger painting, you know, with the perverse and in-

> sanely complicated biomechanical constructs.
> Now, instead of being all shiny and new, make it old and rusty and overgrown with weeds. Slimy weeds.

A *Snow Crash*-esque view of GCC's code wasn't really what Phil meant, but I would still like to thank him for sparking my imagination.

References

[1] David Wheeler, "SLOCCount, a tool for counting physical Source Lines of Code." `http://www.dwheeler.com/sloccount/`

[2] Eric Christopher, personal communication.

[3] Neil Booth, personal communication.

[4] Richard Henderson, personal communication.

[5] Richard Stallman, "The GNU Project." `http://www.gnu.org/gnu/thegnuproject.html`

[6] Jeffrey Oldham, "March gcc 3.0 and 3.1 Bootstraps Fail 34% of Time." Email message dated 30 March 2001. `http://gcc.gnu.org/ml/gcc/2001-03/msg01319.html`

[7] Poul-Henning Kamp, "A bike shed (any color will do) on greener grass." Email message dated 2 October 1999, as quoted in the FreeBSD FAQ. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/misc.html#BIKESHED-PAINTING`