Loongcc - A Compiler based on Open64 for MIPS64 Compatible Loongson 3 Architecture

Ling Kun Wang Tao Liu Ying Huang Lei Hu Shiwen Zhang Mang Zhao Hongjian Lu Tingyu Lian Ruiqi

State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

{lingkun,wangtao2010,liuying2007,leihuang,hushiwen,zhangmang,zhaohongjian,lutingyu,lianruiqi}@ict.ac.cn

Abstract

Loongson (academic name: Godson) is a family of general-purpose MIPS CPUs developed by Institute of Computing Technology (ICT), Chinese Academy of Science. Loongson 3 is the newest generation of Loongson processor family and compatible with MIPS64 ISA. It is designed for boosting general and high performance computing. Our primary objective is to build a high performance and production-quality compiler for Loongson 3 processor family based on Open64 . The work reported in this article can be divided into two folds. (1) Aiming at exploiting the architectural features effectively, we have developed a set of Loongson 3 specific optimizations , such as branch-to-trap, conditional-move, multiply-add, quad-word loading, structure-layout optimization, software prefetching, memory pool optimization, CG loop invariant code motion, etc. We have also employed polyhedral algebra based loop optimization techniques and integrated a 256-bit instruction vectorizer. (2) For high robustness, after a large set of debugging and extending efforts, the compiler has obviously higher successful rate and is capable to compile Linux kernel, GCC and bootstrap.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors

General Terms performance, robustness, code generation, optimization

Keywords Loongson 3, Loongson 3B, Loongson Compiler, polyhedral algebra based code optimization, SIMD extension support, auto-vectorization

1. Introduction

Loongson 3 is the latest generation of Loongson CPU series. It is a multi-core CPU designed for high performance desktops, servers and clusters. The first release of Loongson 3 family is Loongson 3A with four 64-bit, four-issued, super scalar GS464 cores. Each core has a 64KB private L1 data cache and 64KB private L1 instruction cache, the four cores share a 4MB L2 cache. It works at 1G Hz with a peak performance of 16GFLOPS [1]. Loongson 3A also introduced plenty of new features which compiler can take advantage of. These features include: multiply-add instructions, conditional move instructions, load quad-word support, light weight trap instructions, software prefetch support, etc.

Loongson 3B is the latest Loongson 3 CPU. Based on the new features introduced in 3A, Loongson 3B increases to 8 cores. The 8-core Loongson 3B processor is composed of 2 nodes. Each node has four cores and four L2-cache banks. The interconnection network takes the 128-bit AXI standard interface with cache coherence policy. Loongson 3B have also extended 3A's GS464 cores to GS464v cores by replacing the two floating-point functional units (FPU1 and FPU2) in each core with two vector/floating-point functional units (VPU1 and VPU2). Each VPU can perform 4 double-precision or 8 single-precision float point multiply and add (MADD) operations or at most 32 fixed-point operations per cycle. With VPUs, Loongson 3B can achieve a peak performance of 128GFLOPS [2] while working at 1G Hz. To take full advatage of VPUs, Loongson 3B has extended its ISA by adding a powerful 256-bit SIMD instruction extension. SIMD extensions in CPU has been proven to be a highly effective approach for enhancing the performance on applications that exhibit fine-grained data parallelism [3–6]. Loongson 3B's SIMD extension has up to 300 instructions which contains commonly used SIMD operations [7] . In order to efficiently maintain 256-bit vector data in register, Loongson 3B also has introduced 128-entry vector registers. To attain backward compatibility with Loongson's floating point instructions, Loongson 3B reuses the lower 64-bit of the first 32 vector registers for floating-point registers [8].

We try to build a high performance and production-quality compiler: *Loongcc* for Loongson 3. Loongcc is based on the port version of Open64 for MIPS III [9]. To achieve the performance goal, we have introduced 8 major optimizations for Loongson 3. In addition, we have also equipped Loongcc with a polyhedral algebra based optimizer for loop-nest optimization and a three-level programming support for Loongson 3B 256-bit SIMD extension To achieve the robustness goal, we have extended the compiler to support bootstrapping and the compilation of Linux kernels, GCC, Mplayer, Binutils, etc. According to SuperTest [10], the correctness of Loongcc is better than GCC.

1.1 Performance

The high performance objective in this work is focused on speeding up general desktop applications, core algorithms in scientific computing and streaming applications. Our contribution related to performance improvement can be divided into the following three categories:

Firstly, We have implemented and well tested a set of optimizations based on Loongson 3 architecture.

- **Effectively utilizing Loongson 3 CPU features**: we have introduced branch to trap transformation, conditional move in-

OPEN64 *Workshop* 2012-06, Beijing
Copyright © 2012 ACM [to be supplied]. . . $10.00

struction optimization, multiply-add instruction optimization, load quad-word instruction optimization, etc. Branch to trap transformation will help to improve the efficiency of pipeline usage by replacing branching instructions using trapping mechanism. Conditional move optimization can reduce the case of bifurcation and merge corresponding basic blocks using conditional move instructions introduced in Loongson 3. Multiply-add optimization can substitute one multiplication and one subsequent addition with a single MIPS madd instruction. Load quad-word optimization can load four machine words into two registers at the same time.

- **Locality optimization**: we have introduced structure layout optimization, software prefetching, memory pool reorganization to improve and make use of the locality of the applications. Structure layout optimization improves the locality of structure's fields accessing by rearranging the layout of them. Prefetch instruction insertion has the power of decreasing cache miss rate for contiguous array accessing. Memory pool partition and reorganization is on the purpose of increasing the data locality of heap memory .
- **CG optimization**: we have also implemented a loop-invariant code motion (LICM) module for CG Loop optimization. It can help reduce redundant operations which are introduced in CG expanding.

Secondly, taking advantage of memory hierarchy is one of the key performance features . How to reduce the cache complexity [11] of programs on Loongson 3B is a critical issue for compilers. We have employed polyhedral algebra based methods to achieve this goal. Polyhedral based optimization techniques are good at modeling the time-space characteristics of memory accessing sequence on memory hierarchies. Therefore, based on an open source package PLUTO [12], we have implemented an polyhedral optimizer into our Loongcc. This module is still at preliminary stage. The major contributions related to polyhedral framework can be divided into 4 aspects: (1) Integration of polyhedral framework; (2) Accepting more complex WHIRL structures; (3) Avoiding integer overflow problems by modifying the intermediate WHIRL structures and optimize polyhedral operations; (4) Regularizing loops, e.g. loop normalization and branch hoisting to help the vectorizer to generate more vectorized code.

Thirdly, Although many CPUs have SIMD extensions, how to provide a flexible and easy-to-use way for programmers is still a challenge for compiler and toolchain developers. There are three level programming methods which are commonly used for SIMD extensions currently. They are programming in assembly, calling intrinsic functions and using auto-vectorization. Comparing to assembly programming, intrinsic function is a higher level programming interface using function calls to operate on vector data within C/C++ code style, while auto-vectorization can automatically vectorize the existing standard C/C++ code. These different levels of support can help not only senior developers to write advanced vectorized code manually, but also help to automatically take advantage of SIMD extensions without modifying original C code. Based on Open64's framework, we have also implemented Loongson 3B 256-bit SIMD extension support for these three level methods. The implementation includes: (1) assembly support for SIMD extensions, (2) an interface of Loongson 3B 256-bit intrinsic functions support, (3) an 256-bit auto-vectorizer.

1.2 Robustness

The high robustness objective in this work is achieved as following. We have used SuperTest C/C++ as the unit test. It has more than 18-thousand test files,which are intended to verify compiler's C/C++ support. We have also used several large scale and commonly used C/C++ code bases to expose and fix defects of Loongcc. Such code bases include Linux MIPS kernel, bootstrap, Mplayer video player, GCC, Binutils, bzip, etc. These applications also helped us to find previous Loongcc's missing or deficient support for commonly used language extensions, like inline assembly code, position-dependent code generation, fully MIPS N64 ABI (Application Binary Interface) supporting, etc, which can hardly found by simple testsuites.

We will describe the design and implementation of performance enhancement in Section 2 and 3, and robustness improvement in Section 4. The experimentation results for both performance and correctness enhancement will be reported in Section 5.

2. Design and Implemenation for Well Tested Optimizations

As mentioned in Section 1, we have accomplished a set of optimizations. These optimizations have been well tested and show performance improvement using SPEC CPU 2006 benchmark which is a collection of commonly used desktop applications. We will explore the design and implementation details in this section, and show the experimentation results in section 5.1.1.

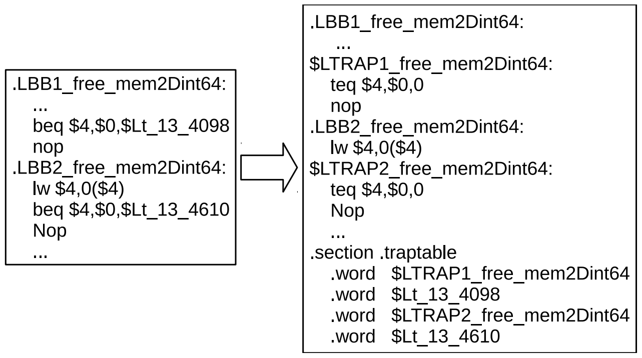
2.1 Optimizations based on Loongson CPU features

As the experience of CPU designing and developing grows in ICT CAS, Loongson 3 have extended its micro-architecture for new features and instructions according to its vaster applications. Making full use of these features by compiler will help programmers speedup their applications easier.

2.1.1 Branch to Trap Transformation

This idea came from an aggreesive branch eliminating experiment on 255.*vortex* from SPEC CPU 2000. According to the edge profiling result, It has lots of if branchs which will never taken. To get an ideal performance result, we manually eliminated all these never taken branches, and got a speedup of about 20% . The result illustrates that the branch predictor of Loongson may mispredict under some circumstances, which cause instruction pipeline flushing and performance degradation. However, Loongson 3 also provides light weight conditional trap instructions, whose advantage is that there is no need to predict the target PC and thus less execution penalty (when the trap condition does not occur). If the condition is not satisfied, the trap instruction will act like a nop instruction, else it will trap into kernel. Taking advantage of trap instruction for never taken branches can avoid the penalty of mis-prediction, and keep the program correct at the same time.

Our implementation includes two parts: compiler side and Linux kernel side. On the compiler side, Loongcc firstly instruments all the branches to get the edge profiling, and then use Feedback-Directed Optimization (FDO) to replace branches which are less taken with trap instructions, at the same time generate a *.traptable* assembly section in order to record the mapping relation between each trap instruction and its target address of the original branch. Figure 1 is an example of the replacement. The code comes from *464.h264ref*. The left part is the original assembly code, and the right part is the transformed code. Each less taken branch instruction is replaced by a trap instruction, the third operand *0* of the trap instruction tells the kernel to use the modified trap handler. A lable is inserted at the same time to identify the branch location. Loongcc will also generate a *.traptable* at the end of the PU. On the kernel side, we firstly modified the load ELF mechanism, the *load_elf_binary* function. When loading ELF kernel to memory, kernel will find out the *.traptable* section, and save it into the private data fragment of the process's memory space, The trap handling algorithm of Linux kernel, the *do_tr*



```
.LBB1_free_mem2Dint64:
   ...
   beq $4,$0,$Lt_13_4098
   nop
.LBB2_free_mem2Dint64:
   lw $4,0($4)
   beq $4,$0,$Lt_13_4610
   Nop
   ...
```

```
.LBB1_free_mem2Dint64:
   ...
$LTRAP1_free_mem2Dint64:
   teq $4,$0,0
   nop
.LBB2_free_mem2Dint64:
   lw $4,0($4)
$LTRAP2_free_mem2Dint64:
   teq $4,$0,0
   Nop
   ...
.section .traptable
   .word   $LTRAP1_free_mem2Dint64
   .word   $Lt_13_4098
   .word   $LTRAP2_free_mem2Dint64
   .word   $Lt_13_4610
```

Figure 1. Branch to trap optimization compiler-side example: *464.h264ref*

function, has also been modified. Algorithm 1 shows the detail of the modification. When trapping, kernel will firstly check whether a *.traptable* exist. If yes, kernel will search for the target address inside the traptable, if corresponding entry is found, and then use the address stored in the table to replace $PC register, else kernel will continue use the original trap handler. Although the process becomes more complicate, for the less taken branches, since the trap instruction will not flush the pipeline and trap into the kernel. we can still get performance improvement.

Algorithm 1 Branch to trap optimization kernel-side trap handling mechanism

```
function do_tr(...)
 1: if Current process does not have trap-table then
 2:     use normal trap handling strategy
 3: else
 4:     trap_instruction_address = value of $EPC register
 5:     if trap_instruction_address not in trap-table then
 6:         use normal trap handling strategy
 7:     else
 8:         change $EPC register to the target value stored in the
            trap-table
 9:     end if
10: end if
```

2.1.2 Conditional Move Instruction Generation

Loongson 3 introduces a set of conditional move(cmov) instructions. They are a set of instruction for data move between registers. Whether a register move operation will be executed depends on the predicate register specified in the instruction. If the value in predicate register is true, then the operation will be executed, otherwise it will be treated as a NOP instruction.

We implemented the conditional move instruction generation in CG phrase, and it is invoked before control flow optimization. The generator will firstly analyse the branches in control flow, and pick up the one that can generate cmov instructions, then corresponding basic blocks will be merged. A verification phase was also added to ensure that the semantics stay the same after the optimization.

2.1.3 Multiply-add Instruction Generation

Loongson processor family exhibits excellent performance in float-point operations, to which floating-point Multiply-Add (MADD) instructions contribute a lot. Issuing one MADD instruction equals to issuing one single or double precision multiply operation and one add operation of the same type, while the former is 2-cycles shorter than the latter one.

To effectively utilize this hardware feature, we firstly extended WN_Lower phase to recognize as many MUL and ADD/SUB/NEG operations as possible, combine and transform them into MADD type whirl nodes. And then, we carefully selected some of them to be expanded into MADD instructions in CG phase, according to a cost model constructed from instruction execution latency. Finally, we aggressively scheduled those instructions in consideration of balancing different ALUs and keep them in-fly.

2.1.4 Load Quad-word Instruction Generation

Loongson 3 introduced a set of load quad-word (LQ) instructions which can load 128-bit data to two 64-bit general purpose registers or lower 128-bit part of vector register. These instructions are faster than two 64-bit load double-word instructions when 16-byte alignment can be guaranteed.

We have implemented a load quad-word instruction generator in CG loop optimization phase, after loop unrolling. It will combines several continuous array accesses to a totally 128-bit LQ memory access when the alignment constraint is satisfied.

2.2 Optimizations based on memory access locality

Memory access penalty is one of the key performance bottlenecks to thwart the performance of computers. Loongson also has this problem. We have designed and implemented a few optimizations to improve the program's locality or eliminate unnecessary memory accesses.

2.2.1 Structure Layout Optimization

As a general purpose high performance processor, Loongson 3 also suffer from the memory wall. Improving data locality can help increase the efficacy of cache utilization, and this is an effective way to bridge the gap between CPU and memory. This optimization tries to improve the locality by rearranging the layout of the structure fields.

We have implemented a layout transformation framework based on fields access affinity graph introduced by Forma [13], and extended it to represent not only intra-instance fields affinity, but also inter-instance fields affinity. The locality of the application can then be improved based on these affinity informations. We firstly implemented a method to gather information of adjacent memory accesses and field affinities by instrumenting WHIRL IR, the information contains precise field affinity among fields of structure instances. A predictor based on such information is also implemented. It can predicate the characteristic of structure field memory accesses using type escape and compatibility analysis. These analysises also make the transformation safe. Since by default all possible data structure transformations are marked as unsafe, only those proved by escape analysis and compatibility analysis will be marked as safe. The analysises make sure that the structure type will not pass as a parameter to any outer functions which are not compiled by loongcc, and all fields are accessed directly using field name. Structures which have fields been accessed using offset will be marked unsafe. Then loongcc rearranges layout of structure fields which is marked as safe based on the predicated affinity information. The above framework was implemented in the Very high WHIRL phase of IPL phase.

2.2.2 Software Prefetch

Loongson 3 makes a great improvement in memory access module, whose prefetch instruction does not occupy an entry in CP0[1] queue.

[1] A co-processor in MIPS which is used to handle interrupts, configuration options, and some way of observing or controlling on-chip functions like caches and timers.

```
  U4U4LDID 158 <1,4,.preg_U4> T<142,anon_ptr.,4,C> # wayar {class 221}
 U4U4ILOAD 0 T<117,anon_ptr.,4> T<142,anon_ptr.,4> {class 21775}
U4STID 119 <1,4,.preg_U4> T<8,.predef_U4,4> # <preg>
  U4U4LDID 119 <1,4,.preg_U4> T<8,.predef_U4,4> # <preg>
  U4U4LDID 133 <1,4,.preg_U4> T<8,.predef_U4,4> # <preg>
 U4ADD
U4STID 139 <1,4,.preg_U4> T<8,.predef_U4,4> # <preg>
```

Figure 2. Loop Invariant Code Motion in CG example: *regwayobj::getway* in *464.h264ref*

This feature offers a more effective manner to prefetch data into L1 cache and hide load latency.

Currently, our software prefetch scheme can handle three massive-memory-access situations as follows. Firstly, for regular arrays residing in nested loops, we implemented our basic prefetch support based on Open64's existing framework in LNO phase [14], which generates prefetch instructions for array elements appearing in next iterations. Secondly, for pointer chains traversing situation, we added a module to pre-compute the address of next pointer, and then generated prefetch instruction for it. Thirdly, for arrays appearing as fields of structures, we simplified the complex array base by promoting it to a temporary variable, hence those arrays in structures could be viewed as regular array and prefetch instructions could be generated. Finally, we also extended several modules in CG phase for instruction selection, instruction schedule and assembly generation, so that prefetch instructions would achieve better performance.

2.2.3 Memory Pool Optimization

General-purpose heap memory allocators often does not pay much attention to exploit the affinity of allocated heap objects, however this affinity can help to increase access locality of the heap objects. The optimization is based on Wang's work of DPA [15], while our implementation is static and planted in compiler.

We provide a new memory pool management library, which can manage several memory pools and allocate space from some specified memory pools. We also added a pool allocation transformation phase to the IPA phase in compiler. In such a transformation phase, we use FDO to find the hot callsites for the heap memory allocator and record their calling relationships got from the call graph in IPA phase. Then after some validity check, we replace the qualified callsites by calling the new allocator, and also some replacements on the corresponding locations to free the space.

2.3 Optimization based on redundant operation elimination

2.3.1 Loop Invariant Code Motion in CG

Loop Invariant Code Motion (LICM) is a traditional optimization, which is implemented in compiler's middle end usually. Although Open64 compiler has a powerful LICM in the middle end, There may still be some loop invariant codes when come to CG. These codes are composed of codes which escaped the motion in middle end and the ones produced during lowering WHIRL to CG IR [16]. Figure 2 shows a WHIRL node within the loop nest body of *regwayobj::getway* from *473.astar*. *wayar* will not change in the loop, however the LICM in WOPT does not promote the *ILOAD* node. A preliminary analysis indicates that the alias analysis or less analysis of ILOAD may be the reason.

We introduced an LICM phase in CG , which can identify such loop invariant, evaluate the benefit of the code motions and move invariants out of the loops if needed. For the WHIRL node of *wayar* in Figure 2, CG will expand it to load operation. With the variable liveness information, CG-LICM find out that the variable will not be changed in the inner loop, then it will be marked as loop invariant, and will be promoted to the outer loop.

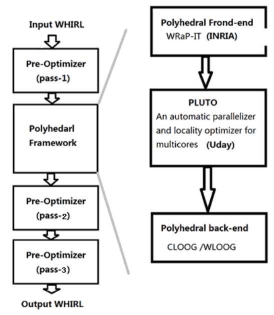


Figure 3. The integration of polyhedral framework

3. Polyhedral Framework and SIMD supporting

Polyhedral based optimization has a higher level view to the iteration space, data space and dependence of loop-nests, thus has strength to transform the code towards optimization objectives such as locality, parallelism etc. Certainly, the objective can also be scheduling the code into vectorizer friendly format. We report the framework design and implementation approaches in this section.

3.1 Polyhedral Framework

Basically, polyhedral based loop optimization is rather high level technique. More precisely, it is a systematic approach to optimize affine loop-nest/arrays program structures [17]. Comparing to traditional loop optimization techniques, it has two major advantages: (1) capable to capture the exact data dependence between all iteration execution instances, thus, has more freedom to reorganize the iteration domain of statements and re-schedules the execution towards better locality and parallelism [18–22]. (2) capable to combine numbers of iteration space interleaving techniques, such as loop fusion, tiling, shewing, permutation, auto-vectorization into one optimization pass. Namely, It can construct a synthesized optimization objective and calculate the corresponding transformations accordingly.

There are a few open source libraries supporting polyhedral optimization techniques, such as LeTSeE [24, 25], PLUTO [12, 26], Omega [27], Polly [28], Graphite [29]. We chose PLUTO as the initial baseline optimizer and used WRaP-IT as the WHIRL level polyhedral representation. The implementation efforts can be divided into 4 aspects:

1. Integration of polyhedral framework: we have solved two major problems. The first problem can be phrased as "inconsistency between the input and output WHIRL to polyhedral framework". The input WHIRL is translated by WRaP-IT and SCoP converter into PLUTO's internal polyhedral representation. PLUTO then optimizes the code purely on this internal representation and relies on CLoog and Wloog to generate the optimized WHIRL code, as shown in Figure 3. However, there was an important issue that was not handled. Namely, the loop variables and global parameters were not updated along the code transformation process. For example, when PLUTO performs a loop tiling, new loop variables and bounds will be created, however, these new symbols are not updated in the symbol table. The second problem can be phrased as "inconsistency of alias and DU information". Similar to the first problem, the previous implementation of WRaP-IT does not maintain the alias and DU information along the code transformation process which may produce wrong code. We have solved both problems by collecting, updating and maintaining the symbol table, alias and DU management information.

2. Accepting more complex WHIRL structures: the classical polyhedral optimizer has well-defined restriction to the input program. In fact, it only handles so-called Static Control Parts (SCoPs) in the program. A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time) [36]. As formulated in Cédric's paper [36], the affine restriction for loop bounds and if conditionals can be removed by using over-estimation techniques. At the time of writing this report, we haven't finished the implementation of this extension. However, we have modified a few crux places of WRaP-IT to make the polyhedral front-end identifies more SCoP, such as converting arithmetic expressions inside the conditionals to logic ones, supporting set operations of polyhedra, extracting array Load/Store from complex WHIRL expression, etc. Although these modifications are merely engineering work, the improvement of identifying SCoPs is helpful.
3. Reducing integer overflow from polyhedral operations: As known integer programming and polyhedral operations are NP-complete. The intermediate expression swell is a severe problem during these computations. Namely the integer coefficients grow quickly for large input. Although multiple precision integer libraries can keep the precision, it increases compilation time exponentially. Therefore, we still use machine integer encoding the polyhedral coefficients, and try to reduce the cases of integer overflow. Basically, we used two approaches. (1) We try to simply the input WHIRL such that the coefficients in the linear program become smaller, e.g. simplifying the array addressing expression. (2) We have tuned the integer programming solver and the polyhedral libraries, such as adjusting the order of input inequalities, and the order of polyhedral computation.
4. Tiling 45-degree timespace loop for vectorization: first, if we linearize the $\mathbb{N}$-dimensional iteration space associated to a statement in an affine loop-nest, and linearize the $\mathbb{M}$-dimensinal data space associated to an array A accessed by the same statement, we could depict the set of binary iterations between iterations and array locations in a 2-D figure. Namely, one axis of the figure coordinates the iterations space, the other coordinates the data space accordingly. Any line in this figure with 45-degree will represent a set of array accessing which is executed consecutively in both data space and iterations space in terms of Lexi-graphical order (Figure 4). In other words, if a sequence of array accessing satisfies the 45-degree property, we call it vectorization friendly. The example shown in Figure 5 is an good example that the loop can be vectorized for SIMD instructions. We name the set of lines has degree in between 45-90 degree a data scaling area and, the set of lines has degree in between 0-45 degree an iteration scaling areas. Figure 6 and 7 are toy examples for each case. Currently, in polyhedral framework we only extract the loop with 45-degree and regularize it. By passing a flag embedded in the WHIRL representation of loops, we guide the vectorizer to generate SIMD instructions.

3.2 Intrinsic function and Auto-Vectorization Support

To get more performance boost from Loongson 3B processor and provide easier way for using SIMD extension , We based on our intrinsic support for Loongson 2F MMI SIMD instruction extention [34], provide all the three level SIMD programming support for Loongson 3B , including assembly, intrinsic function and auto-vectorization. These three levels have different level of programming flexibility and performance tuning support. Assembly is the most flexible way to programming, but error prone. Intrinsic function is much easier than assembly. Auto-vectorization is the most convenient way to use SIMD extension, it generates SIMD code based on common standard sequential C/C++ code, but the speedup depends on the code and input of the program. senior programmers can use assembly or intrinsic functions for high performance, while programmers can also use compilers auto-vecterization features to achieve comparable performance.

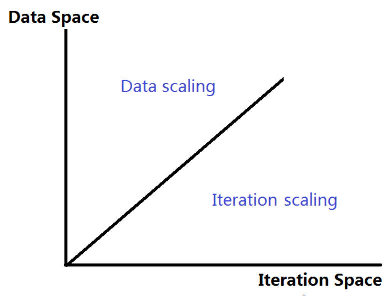


Figure 4. Time-Space relation of array accessing

```
for $i$ in $1 \cdots \mathbb{N}$ do
    $A[i] = A[i] + A[i]$
```

Figure 5. Example of 45-degree

```
for $i$ in $1 \cdots \mathbb{N}$ do
    $A[2 * i] = A[2 * i] + A[2 * i]$
```

Figure 6. Example of data scaling

```
for $i$ in $1 \cdots \mathbb{N}$ by 2 do
    $A[i/2] = A[i/2] + A[i/2]$
```

Figure 7. Example of iteration scaling

Figure 8 illustrates ours implementation based on Open64's existing framework, the gray boxes are what we have done plenty of works for Loongson 3B SIMD intrinsic support. The right part of the figure is the detailed framework of auto-vectorization module in LNO. For assembly support, as Open64 already has it for X86 embedded assembly, there was only some extending in WGEN and CG for Loongson 3B.

Our implementation can be divided into three categories, they are common framework, intrinsic functions and auto-vectorization. Common framework extending is mainly about changes in common part of Open64 compiler. The implementation of Intrinsic functions is mainly in the CC142, WGEN front end, and CG. The implementation of auto-vectorization is mainly in LNO and CG. Thanks to Open64's rich assertion support, it makes compiler debugging a pleasance.

Common framework extending The common frameworks in Open64 associated with SIMD support are the WHIRL IR description and target-dependent information. To make it easy for extending and re-targeting, Open64 tries to make them independent from other parts of the compiler. So extending these two components is mainly about adding new items according to the Loongson 3B SIMD extension. We firstly extended WHIRL IR for the attribute

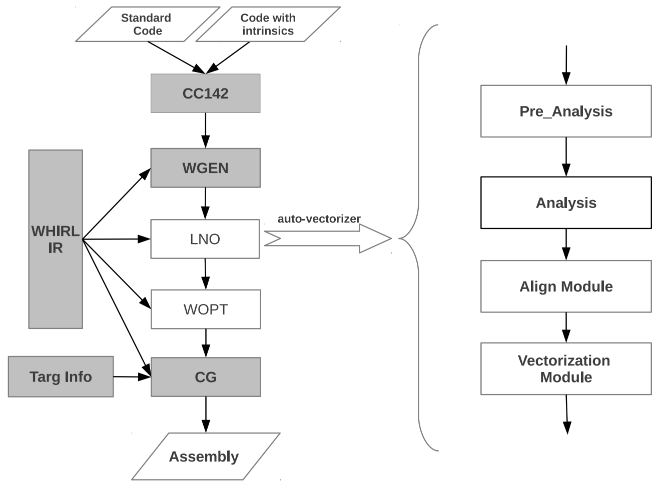


Figure 8. The integration of SIMD support framework

of 256-bit vector mtypes and corresponding INTRINSIC_OPs for all SIMD instructions, and then extend the target information for SIMD instructions and VPU description. For the 128 entry 256-bit vector registers, Loongcc uses the 0-31 registers for floating-point data only, and added a new 256-bit vector register class for the 32-127 registers.

Intrinsic function Intrinsic is currently a powerful tool for senior programmers to write high performance parallel library or the hot code of the application, it also helps keeping them from error prone works like register allocation and instruction scheduling which advanced compilers, like Open64, can handle it more effectively. For Loongson 3B , it is quite necessary to provide such an interface for programmers. We choose to reuse the Open64's INTRINSIC_OP framework for Loongosn 3B intrinsic support. As INTRINSIC_OP is only used for unknown and no side effect operations.

Front End extension For intrinsic support, because it is a programming language extensions, we have to extend the front end, CC142 and WGEN, to check and recognize it. We extended CC142 and gspin to support the 256-bit vector data types and built-in functions recognition. While in WGEN, we mainly extended gspin to WHIRL conversion support for intrinsic function. It includes conversions about the gspin data type to WHILR vector mtypes and builtin function to INTRINSIC_OP.

CG extension In CG, we extend the expanding for INTRINSIC_OPs during WHIRL to CGIR, and the register allocator for vector regs. To make our compiler support all loongson CPUs, we firstly implement a backward compatible register allocator for 256-bit vector registers. This allocator is based on Open64's existing powerful allocator, with some modifications about new vector TN class, vector register allocation, register spilling. Moreover, Because Loongson 3B use two 128-bit memory access instruction for 256-bit vector register load and store, we implement a partly-load framework for it .

LNO auto-vectorization The most attractive way to use SIMD extension for ordinary programmers is auto-vectorization. Currently, auto-vectorization can be divided into two categories according to whether it bases on Loops or basic block [35]. The former one, which is based on loop, tries to analyze loops firstly, and do vecterization for several adjacent loop iterations. This type of auto-vectorization is mainly derived from vector computer [37]. The latter one, which is based on basic block, is mostly derived from Super-word Level Parallelism(SLP) [38]. It tries to collect similar operations in a basic block, which usually results from loop unrolling, then analyses memory alignment and vectorizes the similar operations. Our implementation is based on Open64's existing vectorization framework for X86 which is based on loops. It supports up to SSE4.2 128-bit instruction extensions. The right part of Fig 8 is the four main phases of the vectorizer. We ported it for Loongson 3B's 256-bit SIMD extension, and modified the basic analysis by providing corresponding vector width, mtypes, vectorizable operations, then extended it for 256-bit SIMD operation generation. This work also includes adding some common WHIRL operators and opcodes in common whirl files, and in other phases like WOPT. CG also needs WHIRL to ops expanding support for these WHIRL operations.

4. Robustness

In order to improve the robustness, we have used Loongcc to build many real world large scale Linux applications for Loongson platform. These applications include Linux kernel, bootstrap, Mplayer(a GUI based video player), GCC and Binutils(commonly used GNU toolchain components), etc. The following of this section will mainly focus on Linux kernel building and bootstrap. The works for the other applications are similar.

4.1 Linux Kernel Build

Using loongcc to build Linux kernel can not only check its robustness, but also improve the programming interface which a productive compiler should provided to programmers. So far as we know, only a few Non-GNU compilers, like ICC [39], LLVM [40] can successfully build Linux kernel version 2.6 and above. We mainly overcame three major difficulties during the Linux kernel building: (1) Adjusting GCC options to Loongcc options and tracing them. (2) Enhancing and adding compiler support for rich and flexible embedded assembly, GNU C extensions, position dependent code generation, MIPS N64 ABI support, etc. (3) Fixing the defects exposed during the Linux kernel building.

Firstly, thousands of files will be built using carefully adjusted GCC options for Linux kernel, we should convert these options to the corresponding Loongcc ones. We should also trace the full command line for debugging purpose. To achieve this goal, we enhanced the existing kopencc shell script for temporary file keeping, command line parsing and recording, and object file replacing. With the help of the enhanced script, we can quickly find out files which Loongcc has failed to built and the compiling options.

Secondly, Linux Kernel and GCC have plenty of co-designs, like rich and flexible embedded assembly support, useful built-in functions support, convenient GNU extended syntax like variable length structure, attributes, etc. These additional features exist in many part of the code. Furthermore, the currently available Linux kernel for Loongson computer is only well tested and widely used based on MIPS N64 ABI building, and it is position-dependent. We should extend Loongcc for all the features mentioned above, so that the Linux kernel can be successfully built. With the help of Open64's rich assertion support and plenty of designing, programming and debugging, we finally make Loongcc support them based on the documents of these extensions, the difference of assembly generated by Open64 and GCC, and the internal implementation details of GCC.

Finally, excluding the absence of extensions and ABI supporting, there are only several defects that Loongcc exposed during the build process. These defects are mainly about load type conversion during WHIRL2OPS in CG, side effect judge error during Local Register Allocation in CG, and variable argument function inline in INLINE.

4.2 Bootstrap

Both the cross and native bootstrapping have been accomplished for Loongcc. For native bootstrapping , since Loongcc just works like a common user space application, the mainly problem is about defects fixing. While for cross bootstrapping, the fact that Loongcc does cross compilation from X86 host to MIPS target imposes more challenges to support bootstrap than those native compilers, due to different environment settings (architectures, systems, preprocessors, libraries, etc) between the host and target platforms.

At the meantime, Loongcc contains 50,000 source files, about 7 million lines of C/C++ code, and a huge number of optimization modules, which greatly increases workload and difficulty of bootstrap. Considering those challenges and difficulties, we adjusted the front-end of the compiler to identify the available preprocessors and switch them on-fly; and then improved a few optimizations with defects to adapt them with large file compilation and special run-time input; we also applied some techniques to minimize Global Offset Table(GOT) at link time, which effectively avoided its overflow. Currently, bootstrap of Loongcc is successful on both X86 and MIPS platform with all optimization levels (O0-O3), which indicates its excellent robustness.

5. Experimentation

The main part of this section describes experimental results associated to the performance and robustness enhancement illustrated in Section 2. We have used SPEC2006 test suites [32] and Polybench [31] to benchmark the performance, and SuperTest to test the quality and robustness. If not specified, all the testing is conducted on Loongson 3B machines (CPU model ICT Loongson-3B V0.6 FPU V0.1) with 8 cores at 1 GHz.

5.1 Performance

5.1.1 SPEC CPU 2006 Performance

The followings are performance of the 8 selected optimizations mentioned in Section 2. Because Loongson 3B had not yet finally released when the optimizations were firstly implemented, Loongson 3A has all the features of which these optimizations take advantage, and the micro-architecture of Loongson 3A and Loongson 3B are similar, all the results were test on Loongson 3A platform.

The baseline of the followings speedup is the existing peak performance of each benchmark. These peak performance is based on "$-O3$", "$-ipa$" with some additional optimization flags for each benchmark. We used *runspec* to invoke all testcase, ran each testcase at least 3 times using reference input. Because all the test cases are single threaded, all tests run exclusively with Reference as input, there is almost no variation ($\leq \pm 3\%$) for the experiment result, except 481.wrf ($\leq \pm 4\%$), 459.GemsFDTD ($\leq \pm 6\%$), 470.lbm ($\leq \pm 7\%$). To offset these variations, we use the arithmetic mean of all the evaluations as the final result.

Multiply-add Instruction Generation (MADD) 12 cases have speedup after adopting Mutiply-Add optimiztion. The most significant three testcases are 454.calculix 32.93%, 436.cactusADM 18.56%, 481.wrf 12.71%.

Software Prefetch (PREFETCH) 14 cases have speedup after adopting software prefetch. The most significant three testcases are 462. libquantum 128.69%, 437.leslie3d 34.81%, 481.wrf 32.33%.

Load-Quad Word Instruction Generation (LQ) 7 cases have speedup after adopting Load-quad word optimization. The most significant three testcases are 401.bzip2 7.03%, 450.soplex 3.55%, 459.GemsFDTD 2.44%.

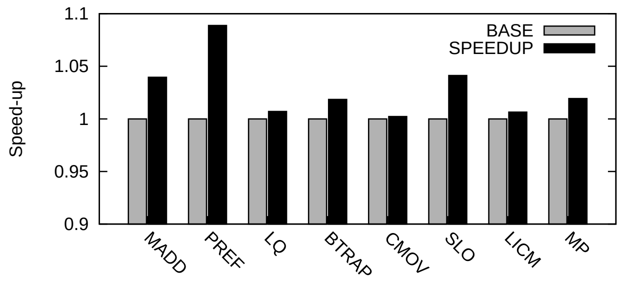


Figure 9. Geomean of all SPEC06 testcase of the 8 selected optimizations

Branch to Trap Transformation (BTRAP) 7 cases have speedup after adopting branch to trap optimization. The most significant three testcases are 400.perlbench 10.60%, 464.h264ref 5.06%, 401.bzip2 5.05%.

Conditional Move Instruction Generation (CMOV) 4 cases have speedup after adopting conditional move optimization. The most significant three testcases are 403.gcc 3.80%, 458.sjeng 1.55%, 481.wrf 1.14%.

Struct Layout Optimization (SLO) The most significant testcase is 462.libquantum 105.94%.

Loop invariant code motion optimization (LICM) 6 cases have speedup after adopting Loop invariant code motion optimization. The most significant three testcases are 447.dealII 9.09%, 473.astar 5.73%, 464.h264ref 4.99%

Memory Pool Reorganization (MP) The most significant testcase is 483.xalancbmk 80.58%.

Fig 9 shows the geomean peak performance improvements of SPEC CPU 2006 whith each of these optimizations. The BASE bar is the geomean of existing peak performance of each test case before the optimization was implemented. The SPEEDUP bar is the geomean of the following performance *perf_max* of each test case.

$perf\_max = max(\ peak\ ,\ peak\ with\ optimization\ on\ )$

For all the Loongson feature based optimizations, multiply-add instruction generation has the most significant geomean speed-up by 4.0%, since plenty of floating-point testcases can take advantage from this feature. Branch to trap transformation, load quadword instruction generation and conditional move instruction generation have improved the performance by 1.8%, 0.7% and 0.2% respectively. For memory locality optimizations ,software prefech can get a speedup of 8.9% for all SPEC CPU 2006 testcases due to memory access latency decreasing, structure layout optimization can also significantly improve the performance by 4.1%, and memory pool optimization 1.9% due to the locality improvement. For loop invariant code motion in CG, a speedup of 0.6% have been attained.

Fig 10 shows SPEC CPU 2006 performance built by Loongcc. In order to give a clearer impression of the performance, we compare Loongcc to GCC . For Loongcc, we use a cross build version and peak options with optimizations metioned . For GCC, We use GCC 4.4 native build compiler. In order to fully exploit the performance, all the testcase are built using N32 Application Binary Interface. As far as we know, for the Loongson 3 target, GCC's highest performance comes from the O3 build, so we put the performance of GCC O3 and Loongcc peak together. All the result were tested on Loongson 3B processor running at 1GHz with 4GB memory. We note that there are two testcases, 445.gobmk and 444.namd, of which Loongcc has performance degradation comparing to GCC.

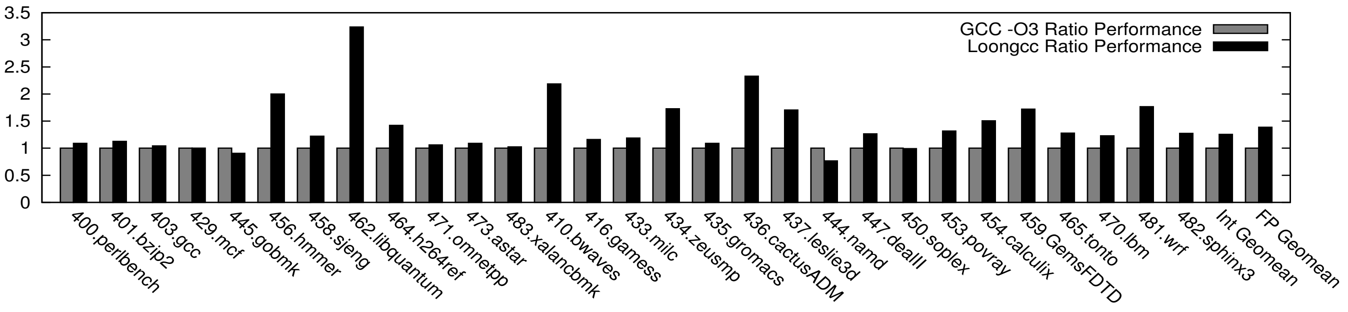


Figure 10. SPEC CPU 2006 Performance compare to GCC on Loongson 3B

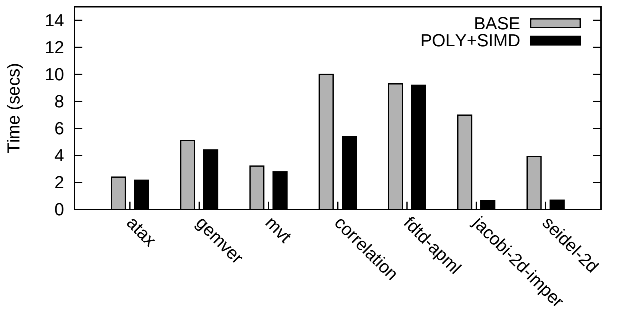


Figure 11. Base and Poly+SIMD Performance of Selected benchmarks from Polybench on Loongson 3B

A preliminary performance analysis has been done on these two cases, showing that 445 have more register spills than gcc, while 444 has more branch mis-predication.

5.1.2 Polyhedral and Auto-vectorization performance

Polybench 3.1 consists of 30 kernels extracted from core numerical algorithms, such as Jacobi method (jacobi-1d/2d), 2-D Gauss-Seidel Iteration(seidel-2d), implicit finite-difference time-domain (ADI, FDTD) method. It is designed to test the efficiency of Polyhedral based optimization and vectorization. As explained in Section 2 the optimization objective of polyhedral based techniques is to reduce cache misses by tiling and fusion; the optimization objective of the auto-vectorizer is to maximally utilize the Loongson 3B SIMD unit. Figure 11 shows the testcases which have performance and their speedup of polyhedral based optimization together with auto-vectorizer optimization using a preliminary version of polyhedral framework and auto-vectorization of Loongcc.

5.2 Robustness

Table 1. Failure rate comparing to GCC on SuperTest benchmark suite

	O0	O2	O3
Loongcc	0.25%	0.45%	0.35%
GCC 4.4	1.23%	1.10%	1.10%

To evaluate the quality of our Loongcc after the intensive robustness enhancement, we have used SuperTest test suite. SuperTest is the world's most comprehensive compiler test and validation suite that is coming from ACEs 30+ years of experience and expertise in compiler construction. The SuperTest suite contains over eighteen thousand source files and provides well over 2 million quality and conformance tests [10]. As shown in the Table 1, Loongcc has a better SuperTest pass rate compare to GCC 4.4 using O0, O2 and O3.

Besides the previous unit test, Loongcc also successfully built and boot a MIPS-64 Linux kernel on Qemu simulators, which is also a good proof for Loongccs robustness [41]. According to the test, Loongcc can successfully build a MIPS Linux kernel that can run on simulator. The other applications that we have tested including Mplayer video player(svn trunck 2010-4-1), GCC(4.6), binutils(svn trunck 2011-07-07).

6. Future Work and conclusion

We have developed a set of optimization methods to improve the memory traffic and utilize the features of Loongson 3. The performance results are satisfactory. Moreover, after intensive efforts for promoting the functionality and quality, our Loongcc is capable to compile very complex systems such as Linux kernel, and the correctness based SuperTest suite is superior to GCC. However, there are two major issues we plan to solve in the near future primarily: (1) to regularize the loop such that its array accessing sequence become 45-degree angled in the time/space coordinates by data/code layout rearrangement. (2) to calculate the combinatorial optimization objective with considering SIMD unit, multi-core and memory hierarchy, based on the architecture of Loongson 3B.

Acknowledgments

Since Loongcc derived from Open64 trunck, we would thank all the Open64 contributors firstly for all their works. We are also grateful to developers of MIPSPro, Pathscale and ORC for their contributions to the open source society. Zhou Shuchang and Gao Zhitao have done plenty of works for the branch to trap transformation and structure layout optimization. Wang Zhenjiang gave us lots of help to make DPA work in Loongcc compiler. Yang Shuxin have done a ORC version of LICM, which makes our LICM design and implementation much more easier. Lai Jianxin, Zhu Qing and Yu Gang in HP shared all their Linux kernel building experience with us. We would also like to thank the anonymous reviewers for their precious remarks which help improve this paper.

References

[1] Loongson 3A CPU details. *http : //www.loongson.cn/EN/product_info.php?id* = 35

[2] Loongson 3B CPU details. *http : //www.loongson.cn/EN/product_info.php?id* = 33

[3] HASSABALLAH M, OMRAN S, MAHDY Y B. A Review of SIMD Multimedia Extensions and their Usage in Scientific and Engineering Applications.Comput. J., 2008(6):630649.

[4] TORRES G. Inside the Intel Sandy Bridge Microarchitecture, 2010. *http : //www.hardwaresecrets.com/article/Inside − the − Intel − Sandy − Bridge − Microarchitecture/*1161/1.

[5] WALRATH J. Bulldozer at ISSCC 2011 - The Future of AMD Processors, 2011. *http : //www.pcper.com/reviews/Processors/Bulldozer − ISSCC −* 2011 *− Future − AMD − Processors.*

[6] Intel AVX - Intel Software Network. *http : //software.intel.com/en − us/avx/*

[7] WEIWU HU Y C. GS464V: A High-performance Low-power XPU with 512-bit Vector Extension. HOT CHIPS.

[8] HU W, WANG R, CHEN Y, et al. Godson-3B: A 1GHz 40W 8-core 128GFLOPS processor in 65nm CMOS. Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International.2011

[9] Zhou Shuchang, Liu Ying, Lu Fang, Yin Le, Huang Lei, Li Shuai, Ma Chunhui, Gao Zhitao, Lian Ruiqi, Open64 on MIPS: porting and enhancing Open64 for Loongson II, Open64 Workshop at CGO 2009.

[10] ACE Associated Computer Experts bv ACE Associated Compiler Experts bv ACE Consulting bv, http://www.ace.nl/compiler/supertest.html

[11] Matteo Frigo, Volker Strumpen, The Cache Complexity of Multithreaded Cache Oblivious Algorithms, Theory of Computing Systems (2006) Volume: 45, Issue: 2, Publisher: ACM Press, Pages: 203-233

[12] U. Bondhugula . Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model. PhD dissertation, The Ohio State University ,2010.

[13] Peng Zhao, Shimin Cui, Yaoqing Gao, Raul Silvera, Jose Nelson Amaral. Forma: A Framework for Safe Automatic Array Reshaping, Transactions on Programming Languages and Systems (TOPLAS) 2007, 30(1).

[14] Todd C. Mowry, Monica S. Lam, Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS), 1992

[15] Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On improving heap memory layout by dynamic pool allocation. International Symposium on Code generation and optimization (CGO '10). ACM, New York, NY, USA, 92-100

[16] YANG Shu-Xin, XUE Li-Ping, ZHANG Zhao-Qing. Loop Invariant Code Motion in Code Generator. Computer Science, 2004, 31(11):158-161

[17] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles,Techniques, and Tools (2nd).Addison-Wesley,ISBN-100321486811,2006.

[18] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In Proceedings of CC'10 International Conference on Compiler Construction, LNCS 6011, pages 283-303, Paphos, Cyprus, March 2010.

[19] D. K. Wilde, A library for doing polyhedral operations, International Journal of Parallel, Emergent and Distributed Systems, Volume 15, Issue 3 & 4 December 2000.

[20] Cérib Bastoul, Albert Cohen, Sylvain Cabala, Saurabh Shard and Olivier Temam,Putting Polyhedral Loop Transformations to Work, Putting Polyhedral Loop Transformations to Work. Lecture Notes in Computer Science, 2004, Volume 2958/2004, 209-225,

[21] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem (I) One Dimensional Time. Journal of Parallel Programming 1992, Volume 21, Number 5, 313-347.

[22] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem (II) Multidimensional

[23] WRaP-IT,WHIRL Represented as Polyhedra - Interface Tool. An Open64 Plug-In for Unified Polyhedral Transformations. *http : //www.lri.fr/ girbal/site_Pratt/*

[24] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen and Nicolas Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In Proceedings of International Symposium on Code Generation and Optimization 2007 (pp.144-156). San Jose, CA: ACM.

[25] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen and Nicolas Vasilache. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. Conference on Programming Language Design and Implementation, 2008, vol. 43, pp. 90-100.

[26] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. Programming language design and implementation 2008 (pp. 101-113). New York:ACM.

[27] William Pugh and the Omega team, *http : //www.cs.umd.edu/projects/omega/omega.html/*

[28] Polyhedral optimizations for LLVM, *http : //wiki.llvm.org/Polyhedral$_o$ptimization$_f$ramework*

[29] GRAPHITE: Polyhedral Analyses and Optimizations for GCC, *http : //gcc.gnu.org/wiki/Graphite*

[30] I.I. Luican, H. Zhu, F. Balasa , Reducing the dynamic energy consumption in the multi-layer memory of embedded multimedia processing systems, Workshop on Synthesis and System Integration of Mixed Information technologies (SASIMI 2007), Sapporo, Japan, Oct. 2007.

[31] PolyBench – Homepage of Louis-Noël Pouchet, *http : //www.cse.ohio − state.edu/ ∼ pouchet/software/polybench/*

[32] designed to evaluate the speed and request-handling, www.spec.org/benchmarks.html

[33] Lee, R.B. Accelerating multimedia with enhanced microprocessors. Micro 1995

[34] Ling Kun, Hu Shiwen, Lian Ruiqi, Optimization of the Intrinsic Support in Loongcc Compiler for Loongson CPU's SIMD Extensions. High Performance Computing Technology, 2011.6:38-42

[35] EICHENBERGER A E, WU P, O'BRIEN K. Vectorization for SIMD architectures with alignment constraints. Conference on Programming language design and implementation, 2004:82-93

[36] Mohamed-walid Benabderrahmane and Louis-noel Pouchet and Albert Cohen and Cédric Bastoul, The Polyhedral Model Is More Widely Applicable Than You Think, 2010, LNCS, Paphos, Cyprus, Classement CORE : A, nombre de papiers acceptés : 16, soumis : 56,Springer-Verlag.

[37] ALLEN R, KENNEDY K. Automatic translation of FORTRAN programs to vector form. ACM Trans. Program. Lang.Syst., 1987, 9:491.

[38] LARSEN S, AMARASINGHE S. Exploiting superword level parallelism with multimedia instruction sets. Conference on Programming language design and implementation, 2000.

[39] Feilong Huang. Building Linux Kernel with Intel C++ Compiler for Linux. Intel White Paper. 2008

[40] Jake Edge. LFCS: Building the kernel with Clang. http://lwn.net/Articles/441018/

[41] Qing Zhu , Tao Wang , Gang Yu , Kun Ling , Jian-xin Lai. Build Linux kernel with Open64. Open64 Workshop at CGO 2009