Reprinted from the

Proceedings of the
GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Google*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

A Superoptimizer Analysis of Multiway Branch Code Generation

Roger Anthony Sayle
OpenEye Scientific Software
roger@eyesopen.com

Abstract

The high level of abstraction of the multi-way branch, exemplified by C/C++'s switch statement, Pascal/Ada's case, and FORTRAN's computed GOTO and SELECT, allows an optimizing compiler a large degree of freedom in its implementation. Typically compilers generate code selected from a few common patterns, such as jump tables and/or trees of conditional branches. However, for most switch statements there exists a vast number of possible implementations.

To assess the relative merits of these alternatives, and evaluate the selection strategies used by different compilers, a switch-statement "superoptimizer" has been developed. This exhaustively enumerates code sequences implementing a specified multiway branch. Each sequence is evaluated against a simple cost model for code-size, average and worst-case performance (optionally using profile information). Backend timings for several architectures are parameterized by microbenchmarking. The results and insights from applying this superoptimizer to a corpus of "real-world" examples will be discussed.

1 Introduction

Although numerous optimizing compilers were implemented prior to its publication, the first paper to describe the issues of code generation for multiway branches was published by Arthur Sale in 1981 [Sale81]. This now-classic paper compared several implementation strategies for the University of Tasmania's Pascal compiler for the Burroughs B6700/7700 series. Ultimately this evaluation recommended the use of either a table jump or a balanced search tree of comparisons. It is interesting, and perhaps slightly disappointing, that over a quarter of a century later the switch code generation in the GNU GCC compiler is little changed from Sale's original exposition.

Over the last 25 years, microprocessor architecture has advanced considerably, the cycle penalties of memory latency and branch misprediction have changed, and several novel implementations have been proposed.

2 Definitions

A `switch` statement (or multiway branch) is a construct found in most high-level languages for selecting from one of several possible blocks of code or branch destinations depending on the value of an index expression. It can be considered a generalization of the `if` statement that conditionally selects between two possible blocks of code depending upon the value of a Boolean expression. In this work, we restrict the index expression to be of an integral type.

Mathematically, a switch statement behaves like a function or mapping, $\mathscr{F} : Z \rightarrow M$, that for each element of the set Z, the set of all K-bit values where K is a positive integer, selects a single outcome from the set M of destination labels. The function $\mathscr{F}$ is surjective, such that $|M| \leq |Z| = 2^K$.

The semantics of Pascal's and Algol's multiway branch, the case statement, permit that any index value not handled by the case statement may result in undefined behavior. We do not consider such partial functions, and instead restrict $\mathscr{F}$ to be a total function, even though this may restrict some the potential optimizations allowed in those languages. Instead the function $\mathscr{F}$ is made total by mapping any value $z \in Z$ that is not in the original domain to a unique *default label*, $m_d \in M$.

By this definition any pure const function of a single integer argument, such as `factorial` or `Fibonacci`, may be represented or canonicalized using a single switch statement. This ability can be generalized using a currying-like approach such that any pure const function can be represented by nested switch statements, or a single switch statement with a sufficiently large K.

When $|M| = 1$, the switch statement $\mathscr{F}$ is said to be *unconditional*. When $|M| = 2$, then $\mathscr{F}$ is said to be *binary*.

In practice, we are mostly interested in switch statements where the size of the codomain M is small, with only a few labels, $|M| \ll |Z|$. In such cases, a significant fraction of the elements of Z map to a default label $m_d \in M$. In such cases, it makes sense to talk of the set of case values $V \subseteq Z$ that are defined as the values that don't map to m_d, *i.e.* $v \in V$ iff $\mathscr{F}(v) \neq m_d$. We can thus represent $\mathscr{F}$ as a set of ordered pairs $\mathscr{F}_V \subseteq V \times (M - m_d)$ such that for all $(v_i, m_i) \in \mathscr{F}_V$, every v_i is associated with a single label m_i, such that $\mathscr{F}(v_i) = m_i$.

Additionally, when interpreted as either signed or unsigned integers, sequentially consecutive values in Z often map to the same destination label. Thus the switch statement $\mathscr{F}$ can also be efficiently represented by the set of ordered triples $\mathscr{F}_R \subseteq V \times V \times (M - m_d)$ such that every $(l_i, h_i, m_i) \in \mathscr{F}_R$ represents a contiguous case range, where $l_i \leq h_i$ and for all v_i such that $l_i \leq v_i \leq h_i$, we have $\mathscr{F}(v_i) = m_i$ and the case ranges are maximal, *i.e.* $\mathscr{F}(l_i - 1) \neq m_i$ and $\mathscr{F}(h_i + 1) \neq m_i$.

We shall refer to the number of case values, $|\mathscr{F}_V|$, and the number of case ranges, $|\mathscr{F}_R|$.

3 Switch Lowering

This section lists several possible compilation strategies for implementing a multiway branch. The first few are well known and commonly employed by existing compilers, whilst the later strategies are perhaps more novel. For each implementation strategy, the high-level C (possibly using GCC extensions) and the equivalent Intel x86 assembler are given. We use the conventions that the index variable is `x`, the default label is called `L0`, and the case target labels are `L1` ... `L`*n*. In this work, we assume the target labels are opaque, and have no useful mathematical relationships (though some early compilers could rearrange code to simplify the task of switch implementation).

3.1 Unconditional Branch

The simplest of switch statement implementations and one of the fundamental building blocks is the unconditional jump. This is used for switch statements that contain only a default outcome, or during switch lowering after all other cases have been handled.

```
goto L0;
```

```
    jmp L0
```

Unconditional jump instructions typically have low overhead on modern processors, where branch prediction is unnecessary and the instruction prefetch machinery can avoid stalling the execution pipeline. Additionally, basic block reordering can frequently eliminate the jump altogether, by placing the destination basic block at the site of the jump (if it has only a single incoming edge) or by duplicating the basic block otherwise (if it has more than one incoming edge) [Mueller02].

3.2 Sequential Tests

The simplest universal implementation strategy for switch statements is to check for equality against each case value sequentially.

```
if (x == 1)
  goto L1;
if (x == 0)
  goto L2;
```

```
    cmpl $1, %eax
    je L1
    testl %eax, %eax
    je L2
```

On many architectures, the performance of integer comparisons may be dependent on the value being compared. As shown in the assembly example above, the i386 provides special instructions for comparison against zero. On MIPS, comparisons against zero and one are cheaper than comparisons against signed 16-bit constants, which are cheaper than general 32-bit constants.

3.3 Jump Tables

A common method for implementing a switch statement is via an indirect jump (also known as a table jump). This uses the switch variable to index an array of destination addresses, and then branch to that location.

In the general case, to keep the size of the jump table reasonable, the minimum case value needs to be subtracted, and the result compared against the table range (as an array bounds check) before indexing the jump table. Fortunately, the minimum case value is often one or two allowing the subtraction to be omitted for a minor increase in the size of the jump table [Sayle01].

```
unsigned int t = x - 10;
if (t > 5)
  goto L0;
static const void *T1[6] =
  { &&L1, &&L2, &&L3,
    &&L4, &&L5, &&L6 };
goto *T1[t];
```

```
        subl $10, %eax
        cmpl $5, %eax
        ja L0
        jmp *T1(,%eax,4)
T1:     .long L1, L2, L3
        .long L4, L5, L6
```

An obvious free parameter and difference between compilers is the density threshold used to decide whether to use a table jump or not. When the case values are sequentially consecutive values that each jump to distinct labels the decision to use a table jump is obvious. However, as the *density* (approximately the number of case values divided by the difference between their upper and lower bounds) decreases, the memory to performance trade-off eventually reaches a critical limit where the additional cost in bytes provides insufficient benefit in clock cycles.

A less obvious difference is the lack of uniformity in the way that switch table density is calculated. One obvious approach is to use the number of case values, $|\mathscr{F}_V|$, as the numerator. However, as mentioned in the next section, ranges of consecutive case values that branch to the same label can be efficiently as range tests, that are independent of the number of case values they consider. Hence, density may correlate better with the number of case ranges than with the number of case values. The approach currently used by GCC is to count non-singleton case ranges as twice as expensive as singleton case ranges. This provides a better measure of density for comparison against other implementation strategies. In practice, the real non-singleton to singleton cost ratio is less than two, even approaching one for switches consisting of consecutive adjoining ranges.

3.4 Range Tests

When two or more consecutive integer values branch to the same target label, it is possible to test for a range values rather than each value independently. One obvious possibility is via a pair of conditional branches such as `(x>=lo) && (x<=hi)` where `lo` and `hi` are the lower and upper bounds of the range respectively. An often more efficient way of doing this is to subtract the lower bound, and then perform an unsigned comparison against the integer constant `hi-lo` which achieves the same thing with a single comparison. Obviously, if `lo` is zero the subtraction can be omitted. Warren [Warren02] also mentions the efficient use of shifts to perform range tests for suitable high and low bounds.

```
if (x < 8)
  goto L1;
```

```
unsigned int t = x >> 3;
if (t == 0)
  goto L1;
```

3.5 Balanced Binary Trees

A more efficient use of compare and branch comparisons than the sequential testing described above is perform a binary search on the target case values. By comparing against a (median) pivot value, the case values can be partitioned into those values less than or greater than the pivot. This reduces the time to perform a multiway branch from $O(n)$ with the sequential method to $O(\log n)$, where n is the number of case values.

Each comparison in such a binary search refines the upper or lower bounds of the remaining partitions. Whilst many compilers perform only signed or only unsigned comparisons depending upon the type of the switch expression, it is potentially beneficial to use both forms in a single search. Another more commonly implemented improvement is to use the established upper and lower bounds to avoid comparisons. For example, if a binary search has already confirmed that the index value is greater than three, but less than five, there is no point in explicitly comparing against the value four. A naïve implementation of binary tree lowering using these bounds may inadvertently emit conditional branches to unconditional branches and similar inefficient idioms. These may be avoiding by either using the usual CFG jump optimizations as a clean-up step, or by improvements in the binary tree lowering code.

Depending upon the architecture, it may be beneficial (or not) to perform three-way branches at each partition step, reusing the condition codes (or register) from a single comparison against an integer constant for both

an equality test and an ordering test. Currently, GCC always performs three-way branches, whilst LLVM always performs two-way (binary) branching. Even with three-way branching there is the decision of whether to perform the equality/inequality comparison before or after the ordering comparison. Unless the pivot value occurs frequently, it is theoretically better to perform the partitioning first and incur the cost of the equality comparison on only one of the following paths.

On many processors there is a significant asymmetry in the cycle counts for taken vs. not-taken cycle counts (occasionally even when the outcome is correctly predicted). This difference means that a perfectly balanced binary tree will not have the best worst-case behavior. Instead appropriately skewed binary trees can lead to better performance. This is one reason why the sequential testing strategy described above is sometimes competitive for switches with four or more values.

```
  switch (x) {
  case 100:  goto L1;
  case 200:  goto L2;
  case 300:  goto L3;
  case 400:  goto L4;
  }

  if (x == 200)
    goto L2;
  if (x > 200) {
    if (x == 300)
      goto L3;
    if (x == 400)
      goto L4;
  }
 else {
   if (x == 100)
     goto L1;
 }

      cmpl $200, %eax
      je L2
      jbe LT1
      cmpl $300, %eax
      je L3
      cmpl $400, %eax
      je L4
      jmp L0
LT1:  cmpl $100, %eax
      je L1
```

An often overlooked aspect of performing binary comparisons both in sequential comparisons and in binary search trees is that it can sometimes be advantageous to compare against integer values (and ranges) not in the case set, V, *i.e.* those that map to the default label m_d. For example, when ranges on both adjoining sides of a 'gap' branch to the same destination, testing for the gap allows the adjoining ranges to be merged. Likewise for binary switch tables, it can occasionally be easier to identify the (complementary) default values than the (positive) case values.

3.6 Bit Tests

A relatively recent innovation in switch statement implementation is the use of bit tests [Sayle03a]. This technique allows several case values that share the same label to be tested simultaneously by representing each case by its own bit.

This technique requires that the range of case values not be larger than the number of bits in a machine word. Like jump tables, it is often necessary to perform a subtraction followed by a bounds check to eliminate values outside of the allowed range.

```
if (x > 8)
  goto L0;
unsigned int t = 1 << x;
// cases 1, 4, and 8
if ((t & 274) != 0)
  goto L1;
// cases 0, 2, and 5
if ((t & 37) != 0)
  goto L2;
goto L0;

    cmpl $8, %eax
    ja L0
    movl %eax, %ecx
    movl $1, %eax
    sall %cl, %eax
    testl $274, %eax
    je L1
    testb $31, %al
    je L2
```

The performance of bit testing can be improved by testing the most frequent bit patterns (masks) first. If all cases are equally probable, this equates to testing the masks that have more bits set before those that have a few bits (or just a single one) set.

3.7 Tabular Methods

For completeness, we briefly describe the applicability of table-driven methods for implementing switch statements. In these techniques, the task of implementing a multiway branch is reduced to a static search problem. A good review of tabular methods, including linear search, binary search, and multiplicative binary search, is given by Spuler [Spuler94]. The major utility of tabular approaches is in reducing code size, especially when the table search implementation can be shared between multiple switches and placed in the compiler's run-time library. In the Java virtual machine, the `lookupswitch` bytecode provides a tabular search implementation.

```
static const int T1[4] =
  { 4, 6, 9, 11 };
static const void *T2[4] =
  { &&L1, &&L2, &&L3, &&L4 };
for (int i = 0; i < 4; i++)
  if (x == T1[i])
    goto *T2[i];
```

3.8 Decrement Chains

An interesting implementation technique implemented by some compilers, but not previously described in the literature, is the use of decrement (or subtraction) chains. On some architectures, a decrement instruction followed by a test for zero is more convenient than a comparison against an arbitrary constant. On i386 this leads to dense code, and on some RISC architectures this provides a useful instruction for the branch delay slot.

```
x--;
if (x == 0)
  goto L1;
x--;
if (x == 0)
  goto L2;
x--;
if (x == 0)
  goto L3;
```

```
    decl %eax
    je L1
    decl %eax
    je L2
    decl %eax
    je L3
```

On MIPS, comparisons against one and zero are cheaper than other values that require loading of integer constants into registers. Hence on this target, it is convenient for dense sequential cases to place the constant two into a register, then repeatedly compare against zero and one between subtractions of two.

This strategy may potentially benefit from the use of decrement and branch instructions on those architectures that support them.

```
      movl %eax, %ecx
      loop LT1
      jmp L1
LT1:  loop LT2
      jmp L2
LT2:  loop L0
      jmp L3
```

3.9 Index Mapping

Index mapping is a table-based technique similar to jump tables but with a wider range of applicability. Instead of indexing directly into a sparse table of labels (typically each four bytes long), this method uses a narrower lockup table of bytes to first transform the index variable into dense zero-based range.

Switch statements with up to 256 unique labels can be handled by a byte-wide look-up table. This technique can also be extended to switch statements with even more unique labels by using two-byte shorts instead.

```
unsigned int t = x - 10;
if (t > 7)
  goto L0;
static const char T1[8] =
  { 0, 1, 1, 2, 0, 2, 2, 1 };
t = (unsigned char)T1[t];
if (t == 0)
  goto L1;
if (t == 1)
  goto L2;
```

```
    movzbl T1(%eax), %eax
```

Often index mapping is immediately followed by a regular table jump. This idiom, called *double dispatch*, has the advantage that the range of values in the first table

is known and therefore the usual bounds test on the following table jump can be omitted. On the SPARC architecture, which usually needs to multiply the table jump index by four to access the correct destination word, this multiplication can be precomputed and stored in the byte map (provided that the switch statement has less than 64 unique destinations).

Mathematically, index mapping provides a *minimal perfect hash function* when each target label is unique and there are no ranges.

On many CISC processors, such as the x86 family, binary index mapping (*i.e.* when there is only a single non-default label and the byte array contains only ones and zeros) can take advantage of instructions that compare directly against memory. On the i386, for example, the `cmpb $0, T1(%eax)` instruction avoids the usual load followed by a compare or test.

3.10 Condition Codes

Depending upon the architecture, and upon the frequency of case values, it may be beneficial to manipulate explicit Boolean expressions rather than the more common 'short-circuit evaluation' conditional branches.

```
char t1 = (x == 6);
char t2 = (x == 11);
if (t1 | t2)
  goto L1;

      cmpl    $6, %eax
      sete    %dl
      cmpl    $11, %eax
      sete    %al
      orb     %al, %dl
      je      L1
```

In the example above, the i386 architecture manipulates the condition codes as bytes, but on many RISC architectures with multiple condition code or predicate registers, they can be manipulated directly. As an example, the PowerPC architecture provides a suitable `cror` instruction.

3.11 Imperfect Hashing

Hashing functions provide a method of handling sparse switch statements. By quickly calculating an auxiliary value and branching or multiway branching on its value, we can 'divide and conquer' the original switch. A review of suitable (*i.e.*, cheap) hash functions is described by Dietz [Dietz92]. More recently, Erlingsson *et al.* have described a practical method for quickly selecting a suitable bitwise-and hash function called "Multiway Radix Search Trees (MRST)" [Erlingsson96].

Modern processor architectures contain a number of instructions that perform useful hashing functions. Examples include `popcount`, `ffs` (find first set), `clz` (count leading zeros), and `ctz` (count trailing zeros).

One simple use of imperfect hashing is handling switch statements where the index expression has a multi-word type (such as `long long`). In such cases, the implementation could be considered a switch of the high-word, dispatching to switches on the low-word. This is especially advantageous on narrow processors such as AVR.

The MRST scheme that extracts consecutive bits from the index value for the hash (*i.e.* where hash functions of the form `t = (x >> C1) & C2`) is particularly well suited to architectures that have bit-field extraction instructions, such as the ARM and the PowerPC. For the special case where the field is a single bit, *i.e.* `C2` is one, the comparison can be performed using a bit-wise 'and' instruction, or special-purpose bit-test instruction where available.

3.12 Perfect Hashing

In the field of switch statement code generation, *perfect hashing* describes any transformation of the original integer index expression that is a *bijection*. A bijective integer operation has the property that every output value is generated by a unique input value, enabling the existence of an inverse. Such an operation just permutes the integer values, but as mentioned above, these permuted case values may allow more efficient implementation than its original isomorphism.

Examples of suitable bijective operations (permutations) on K-bit words include logical inversion (`~x`), integer negation (`-x`), addition and subtraction of an integer constant (modulo 2^K), bitwise-exclusive-or (xor)

of an integer constant, byte-swap, bitwise rotation, and even multiplication by any odd number (proof left to the reader).

```
switch (x) {
case 0:
case 4192257:
case 8384514:
case 12576771:
case 16769028:
case 20961285:
  goto L1;
}
```

```
unsigned int t = x * 2049;
if (t < 6)
  goto L1;
```

```
    movl %eax, %edx
    sall $11, %eax
    addl %edx, %eax
    cmpl $5, %eax
    ja L1
```

A far more frequent application of perfect hashing is the use of rotation (instead of subtraction) to reduce the range of target case values.

```
switch (x) {
case 16: goto L1;
case 32: goto L2;
case 48: goto L3;
case 64: goto L4;
}
```

```
int t = (x >> 4) | (x << 28);
switch (t) {
case 1:  goto L1;
case 2:  goto L2;
case 3:  goto L3;
case 4:  goto L4;
}
```

```
    rorl $4, %eax
```

Instruction set support for multimedia functions can be a rich source of suitable bijective instructions. For example, the IA-64's `mux1.rev`, `mux1.mix`, `mux1.shuf`, and `mux1.alt` can all potentially be used to improve the performance of a switch statement implementation.

It should be clear that the modulo-2^K subtractions, used in range tests as bounds checks for table jumps and bit tests and functionally in decrement chains, are just a special case of the use of subtraction as a perfect hash function.

3.13 Safe Hashing

Somewhere between perfect (bijective) hashing and imperfect hashing is safe hashing. These are many-to-one index transformations with the fortunate property that all index values that map to the same output value have the same outcome (target label).

Examples of suitable safe hashing functions are division, bitwise logical and arithmetic shifts, bitwise-and, and bitwise-or.

```
switch (x) {
case 0:
case 1:  goto L1;
case 2:
case 3:  goto L2;
case 4:
case 5:  goto L3;
}
```

```
unsigned int t = x >> 1;
switch (t) {
case 0: goto L1;
case 1: goto L2;
case 2: goto L3;
}
```

```
    shrl %eax
```

Another example of the potential benefit of safe hashing is the classic `Has30Days` function.

```
switch (x) {
case 4:   // April
case 6:   // June
case 9:   // September
case 11:  // November
  return true;
}
```

```
unsigned int t = x | 2;
switch (t) {
case 6:
case 11:
  return true;
}
```

3.14 Conditional Moves

Modern processors can often avoid the penalty of conditional branching by the use of conditional move instructions.

```
void *t = &&L0;
if (x == 1)
  t = &&L1;
if (x == 2)
  t = &&L2;
if (x == 3)
  t = &&L3;
goto *t;
```

```
      movl    %eax, %ecx
      movl    $L0, %edx
      movl    $L1, %eax
      cmpl    $1, %ecx
      cmove   %eax, %edx
      movl    $L2, %eax
      cmpl    $2, %ecx
      cmove   %eax, %edx
      movl    $L3, %eax
      cmpl    $3, %ecx
      cmove   %eax, %edx
      jmp     *%edx
```

4 Superoptimization

The term *superoptimization* was first coined by Henry Massalin in [Massalin87] and describes the task of finding the optimal code sequence for a single, loop-free sequence of instructions. Whilst most compiler optimizations attempt to improve code by applying heuristic transformations, superoptimization performs an exhaustive search in the space of valid instruction sequences. An outstanding example of superoptimization is the GNU superoptimizer [Granlund92]. A less rigorous but practical application of superoptimization is the task of generating instruction sequences for multiplication by integer constants, known in GCC as the routine `synth_mult` [Bernstein86, Lefevre01].

Unlike most traditional superoptimizers, the superoptimizer implementation described in this work needs to handle explicit control flow instructions. In addition to the strict linear instruction sequences enumerated in previous efforts, a representation of conditional branches introduces instructions with two successor instructions; one for a taken branch and one for a not-taken branch. In this work, we restrict the representation on control flow graphs to binary trees. Looping constructs (such as those in inlined tabular implementation methods) are represented by a single "macro instruction" hiding the internal backward edges from the superoptimizer. Although there may be some benefit in extending the current enumeration strategy to general directed acyclic graphs (DAGs), the restriction to binary trees helps limit the combinatorial explosion associated with exhaustive search.

The multiway branch superoptimizer works by a divide-and-conquer approach, repeatedly simplifying or transforming the remaining switch cases that need to be implemented. At each node in the search/tree, the problem state consists of the set of case nodes that remain to be considered, and the set of potential values the index expression may have. At the start of the recursive search, the root of the tree, the set of case values are those of the original switch, and the set of potential values are all 2^K values representable in the index type. The terminal nodes at the leaves of the tree are unconditional jump nodes or table jump nodes. In theory, the value range propagation pass of a compiler may be able to refine the initial set of potential index values.

Some example solution trees enumerated by the superoptimizer are given in Figure 1. These depict some of the possible balanced binary tree implementations for a switch statement with four unique case values.

5 Cost Model

An important aspect of superoptimization is the objective function that is used to assess the merit of each solution. When optimizing for code size, a size in bytes can be associated with each node in a solution tree, and the total code size is simply the sum of each node size in the tree. When optimizing for performance, a cycle count can be associated with each edge in the tree. For conditional branches, different timings can be associated with the taken and the not-taken edges. The worst-case performance is calculated by determining the maximum edge cost from the root to any leaf node. By associating a frequency with each original case value, we can determine the average or expected cost of a multiway branch. The expected average cost of a switch statement is the sum over all leaves of the probability of the leaf multiplied by the cost of the leaf. In practice, the use of floating point probabilities can be avoided by using integer

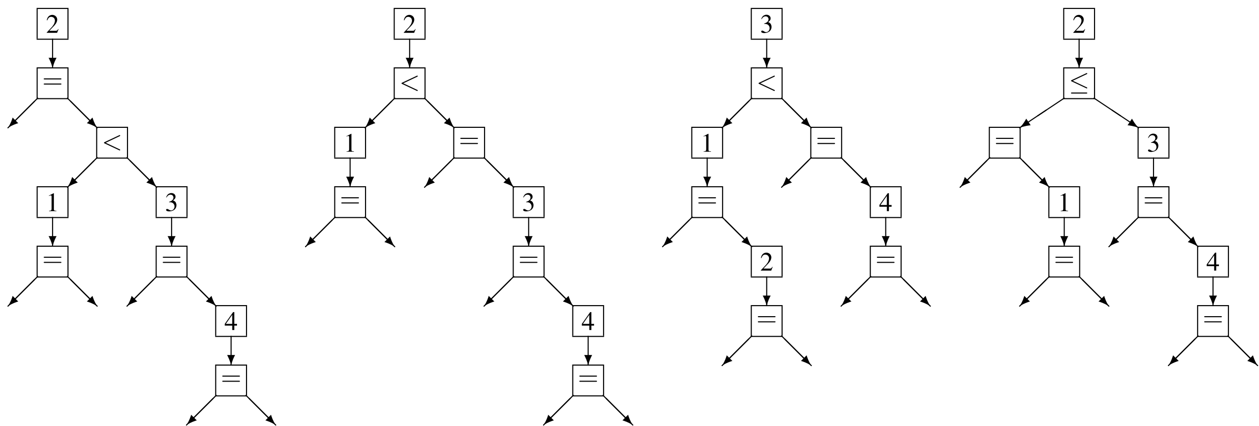

Figure 1: Four alternate balanced binary tree solutions for implementing a unique switch statement with case values one, two, three, and four. Digits represent comparison instructions; relational operators represent conditional branch instructions with the taken edge below to the left and the not-taken edge below to the right.

frequencies. These frequencies can be taken from profiling information or assumed to be equal when profile information is not available.

In practice, rather than purely minimize average performance, worst-case performance, or code size, it is often more useful to use *Pareto frontier* methods to perform multiobjective optimization. If two implementations have the same performance, then the shorter solution is preferable. Often when optimizing for size, we're not interested in the absolute shortest code if the performance is terrible. Likewise, when optimizing for performance, solutions that use an unrealistic amount of memory may be inappropriate. The restriction of jump tables based on density can be seen as an instance of this. Likewise, if the training data used to generate profile edge data doesn't cover all of the cases, the performance of cases with a zero frequency are not used in estimating the average case performance. Whenever two implementations have the same average case behavior, we can avoid potential issues with pathological input values by preferring the one with the better worst case performance.

An interesting additional cost metric not considered in the present study is the optimization of code to reduce power consumption [Tiwari94]. Much like microbenchmarking, the cost of each node could be parameterized by chip-level simulation.

6 Microbenchmarking

In order to both parameterize the cost model described previously and confirm the predictions made by the superoptimizer, it is possible to measure the performance of possible code sequences on real hardware.

The use of microbenchmarking is even applicable to targets where real hardware isn't available. The methodology can be used with virtual machines, such as a JVM, or with simulators, such as GNU sim or SIMH. One application of this technique was to evaluate the benefits of the VAX's `casel` instruction by running VAX Net/BSD on SIMH.

The advanced branch prediction and memory caching strategies performed by modern processors can often significantly skew and bias the results of microbenchmarks, where memory (for jump tables) is often held in cache, and where hot branches can often by predicted from history tables. To avoid (or more accurately to account for) these effects, explicit cache flushing can be used [Whaley08].

7 Benchmark Corpus

To evaluate whether the advanced switch statement implementations described above are applicable on real-world code, a survey of existing switch statement usage was undertaken. By using a specially instrumented version of GCC, the switch statements were extracted from

the source code of 5,953 packages of the Debian Linux distribution.

These 5,953 Debian packages contained, or more accurately compiled, 522,681 switch statements. These switch statements contained 3,024,896 case ranges, giving a mean of 5.79 case ranges per switch statement. A case range is defined a consecutive sequence of case values that map to the same non-default target label. The largest switch statement contained 4,541 case ranges. Table 1 summarizes the distribution of switch sizes, as measured by the number of case ranges.

# Cases	Count	Fraction
1	61193	11.71%
2	115980	22.19%
3	103623	19.83%
4	65671	12.56%
5–8	102898	19.69%
9–16	48575	9.29%
17–32	17462	3.34%
> 32	7279	1.39%

Table 1: Distribution of Switch Sizes

As shown by these figures, the vast majority of switch statements contain relatively few case ranges. Over a third have only one or two cases, and less than two percent have more than 32 cases. However, the distribution has a long tail with 19 switch statements having more than a thousand case ranges.

Of the 522,681 switch statements, 424,600 (81.24%) are *unique*, where every case range branches to a distinct destination label. These 'unique' switch statements don't benefit from bit tests or safe hashing methods. There were 78,925 (15.10%) *binary* switch statements that had a single destination label other than the default. There were 94,523 (18.08%) switch statements that had non-singleton ranges, *i.e.* consecutive case values that branched to the same non-default destination.

Table 2 is a summary of the ranges or spans of the observed switch statements. These figures show that over eighty percent of all switch statements span a range of less than 32 values. This reveals that techniques such as index mapping and bit testing should frequently be applicable.

Range	Count	Fraction
1	51804	9.91%
2	62757	12.01%
3	70489	13.49%
4	49311	9.43%
5–8	80043	15.31%
9–16	66019	12.63%
17–32	47947	9.17%
33–64	31998	6.12%
> 64	62313	11.92%

Table 2: Distribution of Switch Ranges

Table 3 provides an overview of switch statement density. Rather than the more usual (floating point) measure of *density*, calculated as n/r where r is the range of values and n the number of values, we instead report its integral reciprocal *sparsity*, which is defined as $\lceil r/n \rceil$. Using this measure, all switch statements with a density greater than $33.\dot{3}\%$ have a sparsity less than or equal to 3.

Sparsity	Count	Fraction
1	286427	54.80%
2	85380	16.34%
3	24429	4.67%
4	16701	3.20%
5	16081	3.08%
6	10374	1.98%
7	7509	1.44%
8	5301	1.01%
9	5459	1.04%
10	3271	0.63%
> 10	61749	11.81%

Table 3: Distribution of Switch Densities

These figures show that extending the range of applicability of jump tables by decreasing the density threshold has rapidly diminishing returns. GCC currently uses a density threshold of 10% when optimizing for speed, and 33% when optimizing for size. LLVM currently uses a more conservative density threshold of 40%.

8 Results

An example x86 solution found by the described superoptimizer is given below.

```
switch (x) {
```

```
    case 20:
    case 21:
    case 23:
    case 24: goto L1;
  }
```

```
  cmpl $22, %eax
  je L0
  subl $20, %eax
  cmpl $4, %eax
  ja L0
  jmp L1
```

A more interesting example is taken from compiling the Linux kernel on IA-64. A frequent idiom is to switch on powers of two.

```
  switch (x) {
  case 1:  goto L1;
  case 2:  goto L2;
  case 4:  goto L3;
  case 8:  goto L4;
  ...
  case 16384:  goto L15;
  case 32768:  goto L16;
  }
```

The fastest solution found uses count-trailing-zeros as a imperfect hash function, which in turn makes use of the Itanium's `popcount` instruction.

```
  int t = __builtin_ctz(x);
  static const void *T1[32] = {
    &&LT1, &&LT2, &&&LT3, ...
    &&L0, &&L0, &&L0, &&L0 };
  goto *T1[t];
LT1:
  if (x == 1)
    goto L1;
  goto L0;
LT2:
  if (x == 2)
    goto L2;
  goto L0;
  ...
```

The switch statement lowering code in the current version of LLVM differs from that used by the current version of GCC in several minor respects. Instead of making the decision whether to use a table jump or sequence of bit tests once for the whole switch statement, LLVM uses a divide-and-conquer approach, allowing a sequence of binary comparisons to lead to leaf nodes that perform table jumps or bit tests to implement the remaining comparisons. This allows more than one jump table to be used to implement a single switch statement. The threshold for switching from comparisons to jump tables is slightly lower for LLVM, at a maximum of three comparisons vs. four for GCC. LLVM also has a stricter density requirement for jump tables than GCC.

Method	Count	Fraction
binary trees	202633	38.77%
(without ranges)	163548	31.29%
(with ranges)	39085	7.48%
table jumps	151893	29.06%
(without delta)	99932	19.12%
(with delta)	51961	9.94%
sequential tests	153211	29.31%
bit tests	14944	2.86%
(without delta)	11575	2.21%
(with delta)	3369	0.64%

Table 4: GCC Implementation Statistics

Of the 522,681 switch statements in the survey corpus, 5,135 have two singleton case values that branch to the same label, and of these, 1,564 (0.30%) have case values that differ by a single bit and are therefore suitable for `ior` safe hashing.

Of the GCC table jumps, 1,056 (0.20%) are *binary* and have a single non-default label and could therefore be more efficiently implemented with a Boolean index map.

When bit testing, when the number of bits set in the immediate constant is more than half of the available bits, on some architectures it may be cheaper to test the complement bit pattern and reverse the sense of the conditional branch. Analysis of the Debian data set reveals that this occurs extremely infrequently (62 times for binary bit tests, and 516 times for multiway bit tests).

Of the switch statement implement by GCC using binary trees, 29,155 (5.58% of the total) have more than four case ranges, but failed to be implemented by a table jump for density reasons. We shall refer to this set as the *difficult subset*. The largest difficult switch statement, found in the `conversions.c` source file of Debian's `msort-8.42` package, had 1,080 case ranges, covering 1,202 case values branching to 320 unique destination labels.

Of this difficult subset, 1,806 (0.35%) were "powers of two" where each case value had only a single bit set. There were also 3,083 (0.59%) switch statements where every case value had trailing zero bits, and in 2,066 (0.40%) of these, rotating by the number of common trailing zero bits reduced the range (and therefore the density) enough to be implemented by a table jump.

Index mapping turns out to be a particularly effective implementation strategy for dealing with these 29,155 difficult cases. Using GCC's current 10% density criterion, which permits using up to 40 bytes of memory per case value (on a 32-bit machine), index mapping can extend the applicability of table jumps to another 15,346 (2.94%) switch statements (or over half of the difficult cases). Additionally, of the 155,893 switch statements currently implemented by table jumps, 77,958 (14.92%) would have reduced memory consumption using "double dispatch" index mapping. Likewise, 19,790 (3.79%) of the current table jumps have fewer than 5 unique destinations, allowing the table jump to be eliminated completely by index mapping followed by sequential comparisons or a binary search tree.

One approach to improving GCC's implementations of the difficult subset is to use a binary tree of comparisons to split/subset the original switch statement into partitions that are dense enough to be implemented by table jumps. Analysis of the difficult set reveals that 15,131 (2.89%) contain one (or more) subranges that contain enough case ranges and are dense enough to satisfy GCC's table jump criteria. Table 5 gives the histogram of the number of table-jump clusters in the difficult subset. The maximum number of dense clusters observed in a switch statement is 28, found in both in the `inventor_2.1.5` package's `SoFieldConvertors.c++` and in the `netrik_1.15.3` package's `parse-syntax.c`.

Clusters	Count	Fraction
0	14024	48.10%
1	12599	43.21%
2	2005	6.88%
3	356	1.22%
4	67	0.23%
5–8	85	0.29%
9–16	10	0.03%
> 16	9	0.03%

Table 5: Dense Clusters in Difficult Subset

An area of active research in switch statement lowering is how best to select the pivot element to best partition switch statements to take advantage of these dense clusters. An easy-to-implement approach is to use the existing mechanism of selecting the median case value (or frequency-weighted median) as is traditionally done for constructing binary search trees. This has the unwanted effect of occasionally splitting dense clusters, reducing the number of potential jump tables. Bernstein [Bernstein85] suggested splitting at the largest gap between case values. Korobeynikov [Korobeynikov07] implemented a density-balancing metric used to select a reasonable pivot element in LLVM. A common failing with these two "greedy" approaches is that they inadvertently skew binary search trees even when there are no dense clusters to be found. These improve the performance of the rare (15,131 or 2.89%) binary trees that contain dense clusters at the expense of the far more common (187,502 or 35.87%) binary trees that should attempt to be suitably balanced independently of their case value's integer values. A better global approach has been proposed by Delorie [Delorie04] that initially identifies the dense clusters and treats these much like case ranges, as leaves (or potentially internal nodes) when building a balanced binary search tree. These remain heuristics, as optimal switch statement lowering may occasionally require intentional splitting of dense clusters, much like testing frequent cases prior to table jumps can improve average case performance.

9 Future Work

The C# programming language extends the switch statement to allow switching of strings, where each case contains a constant string literal. The implementation of string switches is analogous to the classic computer science problem of keyword recognition. Additional strategies include perfect and imperfect hashing (as used by the GNU utility `gperf`), Patricia tries, or finite state machines (as used by the UNIX utilities `flex` and `lex`).

In this work, the implementation of a switch statement has been equated with that of a multiway branch. However, for many uses of the switch statement in real code, it is possible avoid branching altogether and replace the switch with one or more table look-ups. For example, the `Has30Days` example presented earlier can be implemented as the following:

```
if ((unsigned)x > 11)
  return 0;
static const int T[12] =
  { 0, 0, 0, 0, 1, 0,
    1, 0, 0, 1, 0, 1 };
return T[x];
```

This mechanism can be extended to more diverse targets through the use of unification.

```
switch (x) {
case 0:  foo(2,4);  break;
case 1:  foo(8,3);  break;
default: foo(5,9);  break;
}
```

. . . can be implemented as. . .

```
int t1, t2;
if (x < 2) {
  static const int T1[2] = { 2, 8 };
  static const int T2[2] = { 4, 3 };
  t1 = T1[x];
  t2 = T2[x];
} else {
  t1 = 5;
  t2 = 9;
}
foo(t1,t2);
```

Another possible area of research is to combine the tabular implementation methods described earlier with dynamic data structures such as splay-trees, self-balancing binary search trees, or move-to-front lists. The advantage of such a hybrid technique is that the performance of the implementation could dynamically adapt to the frequency distribution of the index expression.

Finally, to borrow from Newton's third law, "*For every optimization, there is an equal and opposite pessimization*." For each of the transformations described in this article, it would be nice to perceive the original higher-level switch statement from its optimized (lowered) equivalent. Programmers by necessity often attempt to hand-optimize code to the current generation of computer hardware. The ability for a compiler to automatically undo these (premature) optimizations and then re-lower them using the target's optimizer could potentially avoid the performance problems frequently associated with legacy code.

10 Acknowledgments

The author would like to thank OpenEye Scientific Software for the freedom to investigate code generation issues as a hobby; IBM, HP, SGI, Compaq, Apple, SUN, and Intel for providing hardware; and the GCC community for inspiring this research. Particularly, Jan Hubička, Andi Kleen, DJ Delorie, Martin Jambor, and Kazu Hirata have discussed the issue of GCC's switch statement generation. The author is especially grateful to Martin Michlmayr for actually running the instrumented GCC over the Debian Linux distribution's source code repository.

References

[Bernstein85] Robert L. Bernstein, "Producing Good Code for the Case Statement," *Software – Practice and Experience*, Vol. 15, No. 10, pp. 1021–1024, October 1985.

[Bernstein86] Robert L. Bernstein, "Multiplication by Integer Constants," *Software – Practice and Experience*, Vol. 16, No. 7, pp. 641–652, July 1986.

[Delorie04] DJ Delorie, "Case Clustering" `gcc-patches:2004-07/msg01234`, July 2004.

[Dietz92] H.G. Dietz, "Coding Multiway Branches Using Customized Hash Functions," ECE Technical Report, School of Electrical Engineering, Purdue University, 1992.

[Erlingsson96] Ulfar Erlingsson, Mukkai Krishnamoorthy and T.V. Raman, "Efficient Multiway Radix Search Trees," *Information Processing Letters*, Vol. 60, No. 3, pp. 115–120, November 1996.

[Granlund92] Torbjön Granlund and Richard Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices, Vol. 27, No. 7, pp. 341–352, July 1992.

[Hennessy82] J.L. Hennessy and N. Mendelsohn, "Compilation of the Pascal Case Statement," *Software – Practice and Experience*, Vol. 12, pp. 879–992, September 1982.

[Lefevre01] Vincent Lefèvre, "Multiplication by an Integer Constant," `citeseer:491491.html`, 2001.

[Kannan94] Sampath Kannan and Todd A. Proebsting, "Correction to 'Producing Good Code for the Case Statement'," *Software – Practice and Experience*, Short Communication, Vol. 24, No. 2, pp. 233, February 1994.

[Korobeynikov07] Anton Korobeynikov, "Improved Switch Lowering for the LLVM Compiler Compiler System," *Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering (SYRCoSE'2007)*, Moscow, Russia, May 2007.

[Massalin87] Henry Massalin, "Superoptimizer: A Look at the Smallest Program," ACM SIGPLAN Notices, Vol. 22, No. 10, pp. 122–126, October 1987.

[Mueller02] Frank Mueller and David B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices, Vol. 27, No. 7, pp. 322–330, July 1992.

[Sale81] Arthur Sale, "The Implementation of Case Statements in Pascal," *Software – Practice and Experience*, Vol. 11, pp. 929–942, 1981.

[Sayle01] Roger A. Sayle, "Optimize Tablejumps for Switch Statements," `gcc-patches:2001-10/msg01234`, October 2001.

[Sayle03a] Roger A. Sayle, "Implement Switch Statements with Bit Tests," `gcc-patches:2003-01/msg01733`, January 2003.

[Sayle03b] Roger A. Sayle, "Tune Jump Tables for -Os," `gcc-patches:2003-08/msg02054`, August 2003.

[Spuler94] David A. Spuler, "Compiler Code Generation for Multiway Branch Statements as a Static Search Problem," Technical Report, Department of Computer Science, James Cook University, Australia, January 1994.

[Tiwari94] Vivek Tiwari, Sharad Malik and Andrew Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 4, pp. 437–445, 1994.

[Warren02] Henry S. Warren, Jr., "Hacker's Delight," Addison-Wesley Publishers, 2002.

[Whaley08] R. Clint Whaley and Anthony M. Castaldo, "Achieving Accurate and Context-Sensitive Timing for Code Optimization," *Software – Practice and Experience*, Accepted for publication, 2008.

[Wienskoski06] Edmar Wienskoski, "Switch Statement Case Reordering FDO," GCC Summit 2006, Ottawa, Canada, pp. 235–241, June 2006.