Eliminating Branches using a Superoptimizer and the GNU C Compiler

Torbjörn Granlund
Logic Programming and Parallel Systems Laboratory
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista, Sweden
tege@sics.se

Richard Kenner
Ultracomputer Research Laboratory
New York University
715 Broadway, 10th Floor
New York, NY 10003
kenner@vlsi1.ultra.nyu.edu

1. Introduction

In 1987, Henry Massalin, of Columbia University, described a *superoptimizer* that generates optimal instruction sequences given a function to be performed [1]. The sequences are found by doing an exhaustive search over a subset of the instructions of the machine for which the optimization is made. Little or no mention of this important technique has occurred since. In the present work, we describe an alternative technique for constructing a superoptimizer, which will call the GNU Superoptimizer or GSO.

We believe this optimizer is faster and more versatile than Massalin's original work. We show how GSO was used to add patterns to GCC, the GNU C compiler, so it can eliminate many branch instructions when generating code for the IBM RS/6000. A number of surprising results were obtained, many of which were unknown to the architects of the RS/6000 processor.

In the first section we present some sample results of the superoptimizer. The next section discusses the basic design and structure of the GNU C compiler. Subsequently, we show how the results of the superoptimizer were used to enable the GNU C compiler to produce very compact code sequences on the RS/6000. After a brief discussion of the use of some specific RS/6000 instructions, the final two sections discuss how the GNU superoptimizer works and the limitations of superoptimization, and present a large number of instruction sequences generated by the superoptimizer for the RS/6000.

Although this paper uses the RS/6000 for all its examples, the techniques described here are applicable to most machines.

2. Sample Results of GSO

We first study the output produced by the superoptimizer when it searches for sequences that compute the value of the C expression **`(signed int) a >= 0`**. The input is assumed to be in register 3 and the output is placed in register 5.

The RS/6000 is a three-operand machine. Usually the presence of the letter ''**`i`**'' in an operation code signifies an immediate constant, ''**`il`**'', means that the operation is performed on the low-order 16 bits of a quantity, and ''**`iu`**'' means that the operation is performed on the high-order 16 bits.

Here is the superoptimizer output for the C expression **`(signed int) a >= 0`** on the RS/6000.

```
1:  sri    r4,r3,31
    xoril  r5,r4,1

2:  srai   r4,r3,31
    ai     r5,r4,1

3:  srai   r4,r3,31
    cal    r5,1(r4)

4:  xoriu  r4,r3,0x8000
    sri    r5,r4,31

5:  nand   r4,r3,r3
    sri    r5,r4,31

6:  sfi    r4,r3,-1
    sri    r5,r4,31
```

GSO found six ways to perform the operation, each of which takes two instructions. The first sequence computes **`(a < 0)`** by shifting the sign bit to the low-order bit, and then complements it by doing an **`xor`** with the constant one. The second and third sequences first duplicate

the sign bit into the entire word (thus yielding –1 if **a** is negative, and 0 if **a** is greater than or equal to 0) and then add 1 to get the desired result. The difference between sequence two and three is in the instruction used to add 1. In the fourth sequence, the sign bit is first inverted and then shifted right into the low-order bit. The fifth and sixth sequences one's complement the entire value and then shift the sign bit into the low-order bit. They differ only in the manner in which the one's complement is computed. In the fifth sequence a logical operation is used, while an arithmetic operation is used in the sixth sequence.

These sequences do not use the **carry** bit. Therefore they are machine-independent and can be used on any machine that has logical negation and logical right shift instructions. Indeed, this particular method for computing **(signed int) a >= 0** has been known for quite some time. It is included here as an example of the superoptimizer output.

More interesting sequences involve use of the **carry** bit of a machine. On the RS/6000, the **carry** bit is simply the carry out of the high-order bit of the adder. A subtraction is defined as an addition of the minuend, the one's complement of the subtrahend, and either the **carry** bit or the constant one. The result of this *addition* determines the generated value of the **carry** bit.

When GSO is asked to compute the value of the expression **r3 == 0**, a single two-instruction sequence is found. It is

```
1:  sfi     r4,r3,0
    ae      r5,r4,r3
```

This sequence is quite typical of the type of **carry**-bit manipulations that occur in these types of conditional expressions. It works as follows: The first instruction computes the sum of the one's complement of **r3** and 1. It sets **r4** to the result of this addition, which is **–r3**, and sets the **carry** bit to the carry out of the high-order bit of the addition, which, in this case, has the effect of setting the **carry** bit to 1 if r3 equals zero, and to 0 otherwise. The second instruction sets **r5**, the final result, to **r4 + r3 +** carry **= –r3 + r3 +** carry, which is equal to the **carry** bit from the first instruction, or our desired final result.

More sequences will be presented throughout this paper and a large listing of sequences is presented in Section 8.

3. The GNU C Compiler

The GNU C compiler (GCC) is a highly portable, retargetable, optimizing C compiler. The recently released GCC version 2 supports virtually all general-purpose microprocessors currently in use and includes front ends for C, C++ and Objective-C, with front ends for FORTRAN and Ada under development. The code quality of GCC is quite competitive with commercial C compilers; in most cases the code produced for the Sparc and Motorola 88100 (and possibly others) runs faster than code produced by any other compiler. At least two vendors (NeXT and Data General) distribute GCC as the C compiler for their systems.

Rather than using a fixed intermediate language and compiling it for a variety of machines, GCC instead defines a representation called *RTL*, for *register transfer language*, in which the actual instructions of the target machine are represented. The source program is initially compiled directly into RTL representing the target machine instructions and using an unlimited number of registers. The remaining compilation phases perform optimizations and register allocation. The idea of using RTL and some of the optimizations came from the Portable Optimizer (PO), written at the University of Arizona by Jack Davidson and Christopher Fraser [3].

The implementation of GCC is much more fully described in [2]; here we summarize the points of interest to the remainder of the present paper.

Instructions in GCC, referred to as *insns*, are represented in RTL and are an expression tree consisting of the various RTL operators along with pointers to the next and previous insns and some dataflow information. RTL is usually written in a LISP-like notation; a typical add insn looks like:

```
(insn 11 10 12 (set (reg/i:SI 0)
                    (plus:SI (reg:SI 22)
                             (const_int 10)))
     152 {addsi3} (insn_list 9 (nil))
     (expr_list:REG_DEAD (reg:SI 22) (nil)))
```

This indicates that register 0 is set to the sum of register 22 and the constant integer 10. The unique number assigned to the insn is 11, the next insn is numbered 12, and the previous insn is 10. This insn matches pattern number 152 (named ''addsi3'') in the machine description file. The value used in this insn (register 22) was set in insn number 9 (and this is the first use) and register 22 dies in this insn.

Machine description files contain a list of *patterns*, which describe the instructions available on the machine and how to generate code for basic operations. A typical pattern, if named, says what RTL to generate for a particular operation (for example, **addsi3** means to do a three-operand addition of 4-byte integers). All patterns are used to specify what insns are valid and how to generate assembly code for the insn.

Each insn generated must match some pattern. The compiler ensures that all transformations applied to the insn chain as part of the optimization process continue to allow each insn to match some pattern. The **addsi3** pattern from the description file for the AMD 29000 (the pattern

for the RS/6000 is slightly more complex) is as follows:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "gen_reg_operand" "=r,r")
        (plus:SI (match_operand:SI 1 "gen_reg_operand" "%r,r")
                 (match_operand:SI 2 "add_operand" "rI,N")))]
  ""
  "@
   add %0,%1,%2
   sub %0,%1,%n2")
```

Since insns are represented as expression trees, the result of one insn can be substituted directly into its use in a second insn. If the insn that results from such a substitution matches a pattern in the machine description file, we are able to save an instruction in the compiled program. This transformation was originally done by the University of Arizona compiler and in GCC is done in a phase called the *combiner*. The combiner is also able to perform a wide variety of algebraic simplifications.

Typically, merging two or more insns together into one insn forms complex addressing modes on CISC machines. It is also used to form specialized instructions such as **`NOR`** or the various ‘‘multiply-accumulate’’ instructions that have become common.

Optimization phases of GCC which run before the combiner perform jump optimizations, common-subexpression elimination, and loop optimizations including strength reduction and loop unrolling. Following the combiner is instruction scheduling (to reduce data-dependent stalls on pipelined machines), register allocation, another jump optimization and instruction scheduling pass, and delay-slot filling.

4. Elimination of Conditional Jumps

In this section, we discuss the cost of conditional jumps, general methods used to eliminate them, and how we used the superoptimizer to produce efficient code on the RS/6000 that avoids jumps.

4.1. Costs of Jumps

Fundamentally, jumps interfere with the pipelined execution of instructions. Most modern processors have features that minimize this cost, for example by using delayed-branches, which require the compiler to find instructions to place in delay slots.

The RS/6000 uses a different, and quite novel, technique to streamline the execution of branch instructions. There are four execution units on the RS/6000: an integer unit, a floating-point unit, a condition-code unit, and a branch processor. Each cycle can fetch four instructions and dispatch them to the appropriate unit. Because of this structure, unconditional branches can be performed in parallel with other instruction execution at a net cost of zero cycles.

Conditional branches, however, often have a non-zero cost. The timing of conditional branches is quite complex and is described in more detail in [4]. Basically, the cost of a conditional branch depends on the number of cycles between the setting of the condition flag and the branch that tests it. If sufficient time has elapsed, the branch takes zero cycles. Otherwise, there is a delay of three cycles between the setting of the condition flag and the branch for taken branches and zero cycles for not taken branches (however, in the case of not taken branches, there must be a total of five cycles of instructions between the compare and the next branch after the not-taken branch to avoid additional delays).

On the RS/6000, instruction scheduling can sometimes be used to reduce the required delay, although this is usually not successful since most basic blocks are short and many scheduling algorithms only operate within one basic block. For machines with delay slots, delay slot scheduling can be used to reduce the cost of delay slots. GCC performs both optimizations.

Even when the cost of a branch can be reduced, it is nevertheless useful to eliminate as many jumps as possible. Not only has the branch cost not been reduced to zero by architectural features but it is desirable to increase the length of basic blocks. This is because optimizations are easier to perform within a single basic block; most of the optimizations in GCC, such as the combiner and instruction scheduling, only operate within a basic block.

The newly-announced DEC ‘‘Alpha’’ uses neither the technique used by the RS/6000 nor delay slots. Instead, it relies on branch-prediction techniques, and DEC warns that mis-predicted branches can cost ten cycles. Eliminating conditional jumps on the machine is therefore quite important.

4.2. Techniques for Elimination of Jumps

The techniques for eliminating jumps rely on the fact that jumps are often used to skip a relatively short section of code. These methods often involve what are sometimes called *set-condition code (scc)* instructions, which set a data register to one value if a comparison is true and to another value if the comparison is false. Machines that include instructions specifically for this purpose are the Motorola 68k and 88k families, the Intel i386 family, the AMD 29k family, the MIPS processors, and the newly-announced ‘‘Alpha’’ from DEC.

The ‘‘true’’ value is usually either plus or minus one but is sometimes a value consisting of just the sign bit. We sometimes want scc instructions that produce plus one for true and zero for false and sometimes want versions that yield minus one or zero. The following table shows how to obtain these values from all three versions of hardware instructions. Each entry shows the instruction required to convert the hardware ‘‘true’’ value into the desired representations.

Hardware Value	Desired Truth Value	
	+1	–1
+1	no-op	negate
–1	negate	no-op
sign bit	logical right shift	arithmetic right shift

In this section, we assume that all scc instructions return either plus one or zero, i.e., the value returned from a C relational expression.

The most straightforward application of scc instructions is to generate code for C relational operators assigned to a variable, for example, **`a = (b >= c)`**. To compile this statement, we simply apply the **`ge`** scc operation to **`b`** and **`c`** and place its output in **`a`**.

The compiler can also do this transformation if the statement is coded as **`a = ((b >= c) ? 1 : 0)`** or by a series of statements such as

```
if (b >= c)
  a = 1;
else
  a = 0;
```

This transformation is actually a special case of a more general principle. If an expression is to have the value zero if some condition is true and some other, possibly its previous, value if the condition is false, an scc instruction can also be used.

For example, consider the expression **`a >= b ? c : 0`**. We can evaluate this expression by taking a value consisting of all one bits when the condition is true and zero when it is false and performing a logical **`and`** of this value with **`C`**. That is, the given expression is equivalent to **`c & –(a >= b)`**. The transformation can also be done if the equivalent **`if`** statement is used. Even more generally, code such as

```
if (cond)
  foo = 0;
```

is equivalent to **`foo &= –(! cond)`**.

The next section will show how this general principle along with the use of GSO-generated sequences can result in quite compact code for some conditional expressions. However, this principle can be used even in the absence of scc operations, because of the algebraic simplifications done by the GCC combine phase.

Consider what happens when the expression **`(a & 8) != 0 ? 4 : 0`** is compiled for the RS/6000 (the first operand of the conditional operator might be the result of testing a single-bit field). Using the transformation discussed above, this expression is equivalent to **`4 & –((a & 8) != 0)`**. The following transformations are then applied in sequence (we will represent logical right shifts as **`l>>`** and arithmetic right shifts as **`a>>`** and assume 32-bit words):

- To obtain a one if a single-bit value is not equal to zero and a zero if it is, we shift the value to the low-order bit. The expression is now **`4 & –((a & 8) l>> 3)`**.
- A single-bit value can be negated by shifting that bit to the sign bit and then arithmetically shifting it back (to the low-order bit in this case). The expression is now **`4 & ((((a & 8) l>> 3) << 31) a>> 31)`**.
- The two logical shifts are equivalent to a single right shift of 28 bits, yielding **`4 & (((a & 8) << 28) a>> 31)`**.
- Logically **`and`**'ing a sign-extended value with a single-bit constant in the range of the extended sign bits can be done by a logical shift of the sign bit to the desired position. The expression is now **`(((a & 8) << 28) l>> 29)`**
- Finally, the two logical shifts can be merged into one logical shift and one logical **`and`** operation. The original expression is thus equivalent to **`(a l>> 1) & 4`**.

This expression can be evaluated in one instruction on the RS/6000. The **`rlinm`** instruction rotates a register by a specified constant number of bits and logically **`and`**'s it with a mask, where a mask is either a single consecutive group of one bits in a word of zeros or vice versa. This single **`rlinm`** instruction is the code generated by GCC for the original expression. Not only does using these transformations convert what would otherwise be four instructions (which are generated by IBM's compiler) into a single instruction, it has eliminated an expensive branch, yielding an average-case savings of more than a factor of four.

4.3. ''scc'' Instructions for the RS/6000

Most of the transformations shown above assume the existence of short sequences of instructions that set a register to 1 or 0 depending on the result of a comparison. On the RS/6000, this can be done by doing a compare, possibly performing a logical operation on the condition register bits, copying the condition register into a general register (**`mfcr`**), and extracting the relevant bit.

Unfortunately, there is a two cycle delay between the compare and the **`mfcr`** and an additional single cycle delay before the result of the **`mfcr`** can be accessed to extract the desired bit. Thus, these operations take 6 or 7 cycles, which is more than code using branch instructions would take. (It might be possible to move other instructions to hide the latencies, but these instructions could also be used in the delay between compare and branch.) If we use these code sequences, the advantage of having longer basic blocks would be outweighed by the factor of two cost in cycles. For this reason, it was recommended by researchers

at IBM that this method not be used in GCC.

To see if better sequences exist, we had GSO search for the shortest sequences implementing the ten binary comparisons and the six unary comparisons with zero (the unsigned comparisons with zero are degenerate; for `a` unsigned, `a < 0` is always false, `a >= 0` is always true, `a > 0` is equivalent to `a != 0`, and `a <= 0` is equivalent to `a == 0`).

We found that all 16 operations can be done in no more than 3 instructions, three of the comparisons with zero (`==`, `!=`, and `>=`) can be done in two instructions, and one comparison with zero (`<`) can be done in one instruction (the sequences for `>= 0` and `== 0` were shown earlier and all of these sequences are shown in Section 8).

None of the instructions used in these sequences require any communication outside the integer execution unit (such as compare and the `mfcr` instruction discussed earlier); indeed only instructions executed by the integer unit were included in GSO. The fact that such sequences exist was an unexpected result, since discussions with some of the architects of the RS/6000 at IBM led us to believe that no such sequences existed for many of these operations (some of these sequences, such as those for `< 0` and `!=`, were well-known previously).

These sequences enable us to produce code for C statements of the form `a = b` *rel-op* `c;` for all 16 *rel-op*'s that contain at most three instructions and execute in no more than three cycles.

Recall from the previous section that we often need variants of these operations that produce a value of minus one for true and zero for false. One way to obtain such a result is to simply negate the result of the sequences that produce positive one for true and zero for false. However, a close look at the first example in Section 2 suggests that we might be able to find smaller sequences.

In that example, one of the ways we computed `(signed int) a >= 0` was by complementing `A` and shifting the sign bit to the low-order bit. If a logical shift is used, as in that code sequence, this produces a value of positive one for true and zero for false. On the other hand, if an *arithmetic* shift were used, it would produce negative one for true and zero for false, which is the required result. In this case, at least, the cost was the same whether a true value was represented by positive or negative one.

To see if this was the case in general, we used GSO to find the shortest sequences that compute C expressions of the form `-(a` *rel-op* `b)`. The result was that in 14 of the 16 cases, sequences were found of the same length as functions that computed the non-negated conditional value; in the remaining two cases, unsigned `>` and unsigned `<`, the sequences computing the negated result were one instruction *shorter* (two instructions instead of three).

Expressions of the form shown above are rarely encountered in C programs. However, as discussed above, code that conditionally clears a register can be transformed into expressions of the form `a & -(b` *rel-op* `c)`. These can be implemented by using the sequences for the previous set of operations and performing the indicated logical **and** operation. Inspection of some of these sequences showed that they often end with **nand** instructions, so we wondered if it might be possible to perform the required **and** operation without adding more instructions but instead modifying the sequence.

We therefore added code to GSO to search for the shortest sequences of the form `a & -(b` *rel-op* `c)` for all 16 comparison operations. The result was that in two cases, unsigned `>=` and `<=`, the sequences including the **and** had the same length as those without it, namely three instructions. In the other 14 cases, an additional instruction was required, as expected.

The most interesting result was obtained when we looked at the sequences for the 16 basic comparison operations and noticed that there was a sequence for almost all of them that ended with an add or subtract instruction. We were curious whether this instruction could also be used to add another value to the result of the comparison, i.e., how many instructions would be needed to compute `a + (b` *rel-op* `c)`. This operation is rather common, especially when written in its equivalent form as

```
if (b rel-op c)
  a++;
```

After adding code to GSO to search for the shortest sequences implementing these expressions, we discovered that in only two cases (`< 0` and `>= 0`) did the addition lengthen the sequence. Even more surprisingly, in two other cases (unsigned `<=` and unsigned `>=`), the code to compute the sum was *shorter* than that to compute the basic comparison. For example, to compute `a + ((unsigned) b <= (unsigned) c)`, the two instructions

```
sf     r0,r4,r5
aze    r6,r3
```

are used. (`a` is in r3, `b` in r4, `c` in r5, and the result is in r6.) The first instruction subtracts `b` from `c` and the second instructions adds the **carry** flag to zero plus `A` using an `aze` instruction.

The RS/6000 has no single instruction to copy the **carry** flag into a general register, so the sequence for computing the comparison must use an extra instruction to set a register to zero. That is why including the addition produces a shorter code sequence.

To take advantage of the surprising fact that the addition has no cost, we added code to GCC's jump

optimization routine to convert the **if** statement to the equivalent conditional expression shown above.

5. Adding GSO-produced Sequences to GCC

Getting GCC to produce the code sequences we presented is a fairly straightforward task. The first step is to choose the appropriate sequence to use, since many of these operations can be implemented by a number of different code sequences. We used three criteria to choose the best sequence:

(1) Some of the instructions on the RS/6000, particularly arithmetic instructions involving the **carry** bit, do not permit immediate operands. Some operands of these functions (such as the second operand being compared and the value added to the comparison result) will often be immediate constants. Sequences that only use these operands in contexts where immediate forms of the instruction exist are preferred to sequences that do not.

(2) The fewer scratch registers a sequence required, the more preferred the sequence.

(3) A sequence that could put its output in the same register as an input is preferred to one that requires the input and output to be in different registers.

The second step is to add the instruction patterns to generate the code sequences. Because of the way GCC's combiner phase operates, this is a straightforward, though sometimes tedious, task. For example, to generate the code for the case shown above, the following pattern was added to the RS/6000 definition file:

```
(define_insn ""
  [(set (match_operand:SI 0 "gen_reg_operand" "=r")
        (plus:SI
          (leu:SI
            (match_operand:SI 1 "gen_reg_operand" "r")
            (match_operand:SI 2 "reg_or_short_operand" "rI"))
        (match_operand:SI 3 "gen_reg_operand" "r")))
   (clobber (match_scratch:SI 4 "=&r"))]
  ""
  "sf%I2 %4,%1,%2;aze %0,%3")
```

This pattern matches the RTL resulting from compiling the expression **a + ((unsigned) b <= (unsigned) c**. Operand 0 (the result), operand 1 (**b**), and operand 3 (**a**) are all required to be in general-purpose registers. Operand 2 (**c**) can be either in a register or be a constant short enough to fit in the immediate field of an arithmetic instruction. One scratch register is needed for this instruction, specified by operand 4. The last line of the pattern lists the assembler instructions this pattern should generate. The string **%***n* means that the assembler text for operand *n* should be output. **%I2** is replaced by the single letter ''**i**'' if operand 2 is an immediate constant, otherwise nothing is output.

As an example of the use of this pattern, we compiled the function

```
int
sub1 (a, b, c)
     int a;
     unsigned b, c;
{
  if (b <= c)
    a++;

  return a;
}
```

with GCC. This program produced a single insn in GCC's internal representation:

```
(insn 21 13 22 (parallel[
          (set (reg/i:SI 3)
               (plus:SI (leu:SI (reg:SI 4)
                                (reg:SI 5))
                        (reg:SI 3)))
          (clobber (reg:SI 0))
      ] ) 264 {sleu+30} (nil)
  (expr_list:REG_DEAD (reg:SI 4)
      (expr_list:REG_DEAD (reg:SI 5)
          (expr_list:REG_UNUSED (reg:SI 0)
              (nil)))))
```

The calling sequence on the RS/6000 has input parameters passed in successive registers, starting with register 3; the result is returned in register 3. In this case, register 0 (a normal register on the RS/6000 for all purposes except as a memory address) is used as a scratch register. The generated code for the entire **sub1** function consists of the following three instructions:

```
sf     r0,r4,r5
aze    r3,r3
br
```

The first two instructions were generated by the pattern shown above and the third instruction returns from the function.

6. The doz, abs, and nabs Instructions

One of the more interesting instructions on the RS/6000 is the Difference or Zero (**doz**) instruction. The instruction **doz rt,ra,rb** computes the C expression **rt = (ra > rb) ? 0 : rb – ra**.

This instruction has the potential to eliminate a number of branch instructions. The signed **max** of two values can be obtained by following this instruction with an add of **ra**, computing **rt = (ra > rb) ? ra : rb**. To compute the signed **min** of two values, the result of the **doz** instruction can be subtracted from **rb**, computing **rt = (ra > rb) ? rb : ra**. By having GCC detect the idiom of **a > b ? a : b** and convert it into a **max** operation (similarly for **min**), we can emit this two-instruction sequence, again avoiding branches.

For unsigned **min** and **max**, GSO produced a number of four-instruction sequences, some of which involve the **doz** instruction. These are also generated by GCC. The idiom **a > 0 ? a : –a** is also detected and the **abs** instruction is generated. Similarly **a > 0 ? –a : a** produces the **nabs** instruction, again eliminating potential branches.

We were recently informed that the upcoming RS/6000-based processor being designed by the new IBM/Apple/Motorola consortium will not support the **doz**, **abs**, and **nabs** instructions in hardware (they will trap and be emulated in software) and it was suggested that GCC should therefore avoid these instructions. To evaluate the impact of the absence of these instructions, we created a variant of GSO that did not have these instructions and reran GSO on all the expressions described above.

We reproduced the well-known results that **abs** can be done in three instructions and signed **min** and **max** can be done in four. More surprisingly, we found that all four of the signed comparisons (**>=**, **>**, **<=**, and **<**) use **doz**, **abs**, or **nabs** and require a fourth instruction if these are not available.

7. Operation of the GNU Superoptimizer

Massalin's superoptimizer accepts a sequence of assembler instructions describing a function and exhaustively generates all sequences of instructions of increasing length until one is found that performs the desired computation. Several heuristics prevent testing clearly impossible sequences. His superoptimizer is written in the assembler language for the target machine and operates by executing the sequences on the target. The GNU superoptimizer (GSO) has improvements in the areas of portability, configurability and search strategy.

7.1. Portability and Configurability of GSO

GSO is approximately 3000 lines of C code and is host-independent. Unlike Massalin's superoptimizer, GSO searches for sequences of instructions that compute one of several goal functions that have been compiled into GSO. The desired goal is selected with a command-line option with a mnemonic name.

GSO generates code sequences for many different target machines and is designed so that additional targets can easily be added. Currently, GSO supports the IBM RS/6000, the Sparc, the Motorola 68k and 88k, the AMD 29k, and the Intel 80386. Portability is accomplished by defining generic operations which include the union of all instructions supported by GSO for all supported machines. C code is written to simulate each of these operations.

7.1.1. Goal Functions in GSO

One example of a goal function is **eq0**, which returns the value 1 if its single argument is equal to zero and 0 otherwise. To implement this goal function, the routine **eval_goal_function** in GSO contains:

```
case EQ0:
  r = v0 == 0;
  break;
```

Another goal function discussed above was the result of performing a logical **and** between one value (in this case the third operand) and the negation of the comparison of the first two operands. This goal function for a signed **<=** comparison is called **nales** and is implemented as:

```
case NALES:
  r = (–((signed) v0 <= (signed) v1)) & v2;
  break;
```

Here is a list of some of the goal functions currently supported:

- **eq** and **ne**. Two-operand equality comparisons.
- **les**, **ges**, **lts**, and **gts**. Two-operand signed inequality comparisons.
- **leu**, **geu**, **ltu**, and **gtu**. Two-operand unsigned inequality comparisons.
- **eq0**, **ne0**, **les0**, **ges0**, **lts0**, and **gts0**. Single-operand comparisons against zero. As discussed above, the unsigned comparisons against zero are degenerate.
- **neq**, **nne**, **nles**, **nges**, **nlts**, **ngts**, **nleu**, **ngeu**, **nltu**, **ngtu**, **neq0**, **nne0**, **nles0**, **nges0**, **nlts0**, and **ngts0**. Same as above except that they return negative one for true and zero for false.
- **naeq**, **nane**, **nales**, **nages**, **nalts**, **nagts**, **naleu**, **nageu**, **naltu**, **nagtu**, **naeq0**, **nane0**, **nales0**, **nages0**, **nalts0**, and **nagts0**. Three-operand functions that compute the logical **and** of one operand with the negated result of the comparison of the other two operands. An example was shown at the start of this section.
- **peq**, **pne**, **ples**, **pges**, **plts**, **pgts**, **pleu**, **pgeu**, **pltu**, **pgtu**, **peq0**, **pne0**, **ples0**, **pges0**, **plts0**, and **pgts0**. These functions add one operand to the result of comparing the other two operands. E.g, **plts0(a, b, c)** would be written **((signed) a < (signed) b) + c** in C.
- **mins**, **maxs**, **minu**, and **maxu**. Signed and unsigned **min** and **max** operations.
- **sgn**. Returns positive one if the single operand is positive, zero if the operand is zero, and negative one if the operand is negative.

- **abs**. Absolute value function.
- **nabs**. Negated absolute value function.

New goal functions are included by adding a tag to an **enum**, adding the name and tag to a table used to parse command line options, and adding a **case** to the **switch** on goal tags to compute the desired function.

7.1.2. Target Machine Operations in GSO

This version of GSO was written to find instruction sequences for code involving conditional operations. The parts of the machine that are known to GSO are the **carry** bit and a set of registers. All operations manipulate these objects. We do not support operations referencing memory since these will not be part of an optimal sequence of instructions that compute functions on registers. If an input or an output is actually in memory, the sequence produced will normally have to be augmented with load or store instructions on RISC machines. On CISC machines, it may be possible to perform an operation directly on operands in memory. In both cases, the essence of the instruction sequence is not effected.

The operations that GSO uses to create sequences are expressed in terms of generic operations, each of which is available on some subset of the supported machines.

Twelve addition operations are provided, divided into three groups of four. We believe that these operations are general enough to describe the addition operations on all current processors. Within each group, an operation can either use or not use the value of the **carry** bit, and either update the **carry** bit or leave it unchanged. The three groups are addition (**ADD**), subtraction (**SUB**), and the addition of one value to the complement of the other (**ADC**). A typical operation is denoted by **ADC_CIO**, which computes the sum of the **carry** bit, one operand, and the complement of the other and sets the **carry** bit corresponding to the result of the addition.

Each machine has either **ADD** and **SUB** operations or **ADD** and **ADC** operations. The distinction between the **SUB** and **ADC** operations is in the meaning of the **carry** bit. On machines with an **ADC** operation, the **carry** bit is the carry out of the high-order bit when the ALU performs the subtraction, which it does by adding the first operand, the one's complement (bit-wise inverse) of the second operand, and the constant 1. An **ADC_CI** operation computes the sum of the first operand, the one's complement of the second operand, and the **carry** bit.

On machines with a **SUB** operation, the **carry** bit is negated during subtractions so that it represents ''borrow''. A **SUB_CI** operation subtracts the **carry** (really ''borrow'') from the difference of the two operands.

Of the machines supported by GSO, the Sparc, 68k, and i386 have **SUB** operations and the remainder have **ADC** operations. It is interesting to note that this difference in behavior of the i386 and RS/6000 was a surprise to at least one quite proficient assembler language programmer who has written compilers for both machines.

These operations are executed by a block of C code. Here is the code for the **ADC_CIO** operation, which is the same as the **ADC_CI** operation mentioned above, except that it also sets the **carry** bit as an output:

```
#define PERFORM_ADC_CIO(d, co, r1, r2, ci)
{ word __d = (r1) + ~(r2) + (ci);
  (co) = (ci) ? __d <= (r1) : __d < (r1);
  (d) = __d;
}
```

The following is the complete list of operations in the current version of GSO:

- **COPY**. Copy one register to another, or move an immediate value into a register.
- **EXCHANGE**. Exchange two registers. Used only on two-operand machines.
- **ADD**, **ADD_CI**, **ADD_CO**, **ADD_CIO**, **ADC**, **ADC_CI**, **ADC_CO**, **ADC_CIO**, **SUB**, **SUB_CI**, **SUB_CO**, and **SUB_CIO**. These are the addition and subtraction operations discussed above.
- **CMP**. Set **carry** bit true if the first operand is less than the second, when interpreted as unsigned numbers.
- **COMCY**. Complement the **carry** bit.
- **AND**, **IOR**, **XOR**, **ANDC**, **IORC**, **EQV**, **NAND**, and **NOR**. Various logical operations. The **carry** bit is not affected.
- **AND_RC**, **IOR_RC**, **XOR_RC**, **ANDC_RC**, **IORC_RC**, **EQV_RC**, **NAND_RC**, and **NOR_RC**. Similar, but reset the **carry** bit.
- **LSHIFTR**. Logical right shift.
- **ASHIFTR**. Arithmetic right shift.
- **LSHIFTR_CO** and **ASHIFTR_CO**. Logical and arithmetic right shift, but copy last bit shifted out into the **carry** bit.
- **ASHIFTR_CON.** Arithmetic right shift of the RS/6000. Set carry bit if the shifted value is negative and if any bits shifted out were non-zero, otherwise clear the **carry** bit.
- **ABS** and **NABS**. Unary absolute value and negative absolute value, respectively.
- **DOZ**. The RS/6000 **doz** instruction, described above.
- **CPEQ**, **CPGE**, **CPGEU**, **CPGT**, **CPLE**, **CPLEU**, **CPLT**, **CPLTU**, and **CPNEQ**. If the specified comparison of two operands is true, set the result to a value that contains just the sign bit; otherwise set the result to zero. These instructions are present in the AMD 29k family

processors.

The search routines know which machines have which operations and what operands are supported for each operation. A target-specific (via **`ifdef`**) function in GSO produces assembler-language output of the sequences for the selected target machine. To add support for a new machine, the following must be done:

- Define generic operations for any instructions present on the new machine but on no previous machine and write C code to emulate them.
- Modify the conditionalization of the search routines to reflect which operations are present in the new machine and add any new operations.
- Write code to output the operations in the assembly language of the target machine.

7.2. Search strategy of GSO

Both Massalin's superoptimizer and GSO search for the shortest sequence by first searching for sequences of length 1, then sequences of length 2 and continuing until a sequence is found, or a specified bound is exceeded. The strategies used when searching for a sequence of a given length differ. Massalin's superoptimizer generates every possible sequence of the desired length. For each sequence it applies a pruning procedure (described in his paper) to eliminate obviously incorrect sequences and then tests the sequence for correctness.

Like Massalin's superoptimizer, GSO initially generates random arguments for all input operands of the goal function. All computations are performed on these values and values computed by instructions GSO has generated for a putative sequence. No symbolic manipulations are performed. GSO searches for instruction sequences using a recursive iterative-deepening method, and avoids generating clearly useless instructions that would otherwise have to be pruned later. Only operands that are either inputs to the sequence or have been generated by previous instructions are tried. Similarly, if the **carry** bit has not yet been set, an instruction that uses it will not be generated. If the iterative-deepening search has reached a leaf node, the ultimately computed value is compared to the value returned by the goal function for the same input operands. If the values are not equal, the generated sequence is immediately discarded.

Initially the sequence consists of no instructions, the **carry** flag is undefined, and the only available operands are the inputs to the sequence. At each level of recursion, the search function, **`synth`**, scans the list of all available operations for the target machine, adds an operation to the sequence, and makes a recursive call to add more operations if a leaf node is not reached. The operation is also simulated and the resulting value is saved. Two versions of **`synth`** are provided, one for 2-operand machines (the 68k and i386, of those supported) and one for 3-operand machines. On 3-operand machines, register move instructions are never present in an optimal instruction sequence, so they need never be generated.

Goal	Generated Code	Goal	Generated Code

`eq`	`sf r5,r3,r4`
	`sfe r6,r4,r3`
	`ae r7,r6,r5`
		`eq0`	`sfi r4,r3,0`
			`ae r5,r4,3`

`ges`	`doz r5,r3,r4`
	`sfi r6,r5,0`
	`ae r7,r6,r5`
		`ges0`	`sri r4,r3,31`
			`xoril r5,r4,1`

`geu`	`sf r5,r4,r3`
	`rlinm r6,r5,0,0,0`
	`ae r7,r6,r6`
		`gtu`	`sf r5,r3,r4`
			`sfe r6,r6,r6`
			`neg r7,r6`

`gts`	`doz r5,r4,r3`
	`nabs r6,r5`
	`sri r7,r6,31`
		`gts0`	`sfi r4,r3,0`
			`ame r5,r4`
			`sri r6,r5,31`

`les`	`doz r5,r4,r3`
	`sfi r6,r5,0`
	`ae r7,r6,r5`
		`les0`	`ai r4,r3,-1`
			`aze r5,r4`
			`sri r6,r5,31`

`leu`	`sf r5,r3,r4`
	`rlinm r6,r5,0,0,0`
	`ae r7,r6,r6`
		`ltu`	`sf r5,r4,r3`
			`sfe r6,r6,r6`
			`neg r7,r6`

`lts`	`doz r5,r3,r4`
	`nabs r6,r5`
	`sri r7,r6,31`
		`lts0`	`sri r4,r3,31`

`ne`	`xor r5,r4,r3`
	`nabs r6,r5`
	`sri r7,r6,31`
		`ne0`	`nabs r4,r3`
			`sri r5,r4,31`

Table 1: Sequences for Simple Comparisons, **`(A`** *rel-op* **`B)`**

At each leaf of the search tree, the result of the last operation is compared with the result of the goal function applied to the random input values. If they agree, a potentially valid sequence has been found and a more exhaustive test will be performed to see if the sequence is most likely to be correct. This test is similar to that used by Massalin: first some typical operand values are tried and then random values are tested. If the sequence passes, it is printed.

To reduce significantly the amount of searching required, the choices of operands are limited to values generated by previous instructions. Operations that accept the **carry** bit as an input will not be generated unless a previous operation set the **carry** bit. For commutative operations, only one ordering of the operands will be tried. The destination of all operations is always the next available register for a 3-operand machine, and one of the input operands for a 2-operand machine. When generating the last instruction even more restrictions are made.

Specifically, the last instruction must use the output or **carry** bit produced by the penultimate instruction, otherwise the penultimate instruction would not have been needed.

These techniques allow for quite rapid generation of sequences. All the sequences shown in this paper were generated in only a few seconds on a modern workstation, although searching for sequences of five instructions for goal functions with three inputs can take a good part of an hour.

7.3. Limitations of Superoptimizers

A fundamental problem with the technique of superoptimization is that the search space is approximately exponential in the sequence length, and that the branching factor, i.e., the number of instruction choices at each point, is relatively large. The approximate number of tested sequences is b^k, where b is the branching factor and k is the number of insns generated in an output sequence. If b is large, the number of tested sequences becomes huge, and the search space will be too large for any longer sequences.

Keeping the branching factor as small as possible is the most important way to make a superoptimizer able to generate longer sequences. GSO's approach is to avoid instructions with constants, with the exceptions of –1, 0, +1, 2^{31}, and 2^{31}–1. As a result, the branching factor lies between 100 and 1000, depending on the target instruction set and the current search depth. If all possible 32-bit constants were used blindly, the branching factor would exceed 2^{32} — and superoptimization would become impractical.

For the kind of goal functions currently defined in GSO, the limited set of constants is not believed to be a problem. For other goal functions, we would have to be smarter. To adequately choose the right set of constants for a particular goal function is an interesting area for future work.

Aside from the intentional omission of constants, it is possible that instructions have been omitted unintentionally, and hence some of the sequences produced may not be optimal.

For example in Massalin's superoptimizer, which tries a significant number of machine instructions, non-optimal sequences have been produced. Massalin shows sequences on the 68020 for computing **`d0 *= 29`**, **`d0 *= 39`**, **`d0 *= 156`**, and **`d0 *= 625`**. The first three use 5 instructions, while the last uses 7. What was not noticed is that **`lea`** (load effective address) instructions can be used to perform multiplications by 3, 5, and (sometimes) 9. Since 625 is the fourth power of 5, it can be computed with four consecutive **`lea`** instructions. Because **`lea`** only operates on address registers, another two instructions are required to copy between an address register and **`d0`**.

Thus, the shortest sequence for multiplication by 625 has 6 instructions. However, if the statement of the problem is weakened to the problem of taking a value in an arbitrary register and putting the result into any register, multiplication by 29, 39, and 625 can all be done in four

Goal	Generated Code	Goal	Generated Code
neq	xor r5,r4,r3 ai r6,r5,–1 sfe r7,r7,r7	neq0	ai r4,r3,–1 sfe r5,r5,r5
nges	doz r5,r3,r4 ai r6,r5,–1 sfe r7,r7,r7	nges0	sfi r4,r3,–1 srai r5,r4,31
ngeu	sf r5,r4,r3 sfe r6,r6,r6 nand r7,r6,r6	ngtu	sf r5,r3,r4 sfe r6,r6,r6
ngts	doz r5,r4,r3 nabs r6,r5 srai r7,r6,31	ngts0	sfi r4,r3,0 ame r5,r4 srai r6,r5,31
nles	doz r5,r4,r3 ai r6,r5,–1 sfe r7,r7,r7	nles0	ai r4,r3,–1 aze r5,r4 srai r6,r5,31
nleu	sf r5,r3,r4 sfe r6,r6,r6 nand r7,r6,r6	nltu	sf r5,r4,r3 sfe r6,r6,r6
nlts	doz r5,r3,r4 nabs r6,r5 srai r7,r6,31	nlts0	srai r4,r3,31
nne	xor r5,r4,r3 nabs r6,r5 srai r7,r6,31	nne0	nabs r4,r3 srai r5,r4,31

Table 2: Negated Comparisons, **`–(A`** *rel-op* **`B)`**

Goal	Generated Code	Goal	Generated Code
naeq0	ai r5,r3,–1 sfe r6,r6,r6 and r7,r6,r4	nages0	srai r5,r3,31 andc r6,r4,r5
nageu	sf r6,r4,r3 sfe r7,r7,r7 andc r8,r5,r7	nagtu	sf r6,r3,r4 sfe r7,r7,r7 and r8,r7,r5
naleu	sf r6,r3,r4 sfe r7,r7,r7 andc r8,r5,r7	nalts0	srai r5,r3,31 and r6,r5,r4
naltu	sf r6,r4,r3 sfe r7,r7,r7 and r8,r7,r5	nane0	nabs r5,r3 srai r6,r5,31 and r7,r6,r4

Table 3: Negated Comparisons with **`and`**.

instructions.

If one wishes to include all machine instructions in a superoptimizer, the GSO approach makes it slightly harder to ensure that all instructions are included, because of the introduction of the abstract operations. If actual machine instructions were used, it would be possible to check them off against a list of all valid machine instructions. With GSO, there is a level of indirection that must also be checked.

An early version of GSO demonstrated an interesting case of an omitted operation. That version of GSO did not try operations that set a register to a constant value but it did try logical operations between previous values and the constants listed above. In several sequences, an **and** that cleared everything but the sign bit was generated. However, in the sequences in question, the subsequent use of the result was in a context where the sign bit was irrelevant. A more straightforward sequence would have been to set a register to 0, but this was not one of the operations **synth** tried in that version.

Goal	Generated Code	Goal	Generated Code
peq	xor r6,r4,r3 sfi r7,r6,0 aze r8,r5	peq0	sfi r5,r3,0 aze r6,r4
pges	doz r6,r3,r4 sfi r7,r6,0 aze r8,r5	pges0	dozi r5,r3,0 sfi r6,r5,0 aze r7,r4
pgeu	sf r6,r4,r3 aze r7,r5	pgtu	sf r6,r3,r4 sfe r7,r5,r6 sf r8,r7,r6
pgts	doz r6,r4,r3 ai r7,r6,–1 aze r8,r5	pgts0	a r5,r3,r3 sfe r6,r3,r5 aze r7,r4
ples	doz r6,r4,r3 sfi r7,r6,0 aze r8,r5	ples0	srai r5,r3,31 sf r6,r3,r5 aze r7,r4
pleu	sf r6,r3,r4 aze r7,r5	pltu	sf r6,r4,r3 sfe r7,r5,r6 sf r8,r7,r6
plts	doz r6,r3,r4 ai r7,r6,–1 aze r8,r5	plts0	a r5,r3,r3 aze r6,r4
pne	xor r6,r4,r3 ai r7,r6,–1 aze r8,r5	pne0	ai r5,r3,–1 aze r6,r4

Table 4: Operations of the form **(A** *rel-op* **B) + C**.

7.4. Correctness of the Sequences

One must be careful in using the results of a superoptimizer, either GSO or Massalin's original work. Neither program exhaustively tests the resulting code sequences. Although it is highly likely that they are correct, and an incorrect sequence has never been found, each sequence should be checked manually to ensure its correctness.

8. Complete Superoptimizer Results for the RS/6000

This section contains tables showing the shortest code sequence for every goal function described in Section 7.1.1. We list only one of multiple sequences, usually the one that GCC will generate.

Table 1 shows the code generated for the 16 simple comparisons. Table 2 shows the same comparisons, but returning a negative one for true and zero for false. In some cases, we can logically **and** the result of the previous operations with another value at no extra cost; Table 3 shows those operations. Entries that are missing cannot be done any cheaper than the code in Table 2 followed by an **and** instruction. Table 4 shows the operations of the form **(A** *rel-op* **B) + C**. Finally, Table 5 shows the code sequence for miscellaneous operations.

9. Conclusions and Future Work

We have described a variation of Massalin's Superoptimizer and shown how it can be used to produce extremely efficient code sequences that do not contain jumps from

Goal	Generated Code
abs	abs 4,3
nabs	nabs 4,3
maxs	doz r5,r3,r4 a r6,r5,r3
maxu	xoriu r5,r4,0x8000 xoriu r6,r3,0x8000 doz r7,r5,r6 a r8,r7,r4
mins	doz r5,r3,r4 sf r6,r5,r4
minu	xoriu r5,r4,0x8000 xoriu r6,r3,0x8000 doz r7,r5,r6 sf r8,r7,r3
sgn	a r4,r3,r3 sfe r5,r3,r4 sfe r6,r5,r3

Table 5: Code Sequences for Miscellaneous Operations

code that initially has jumps. We have shown that instruction simulation techniques can be used to make a superoptimizer both efficient and easily portable.

There are many possible areas for future work. Clearly, more operations and goal functions can be added to GSO, and this work is currently in progress. Similarly, more machines can be supported.

Another interesting area for future work is the possibility of integrating GSO into the configuration process of GCC. This would allow the compiler to produce these optimal sequences without the manual process of converting GSO-produced code sequences into instruction patterns for the GCC configuration file.

Acknowledgments

Richard Kenner was supported by the U.S. Department of Energy under grant number DE-FG02-88ER25052. Torbjörn Granlund was supported by Sics' research program sponsored by the Swedish National Board for Technical Development (NUTEK), Swedish Telecom, Ericsson Group, ASEA Brown Boveri, IBM Sweden, Nobel Tech System AB, and the Swedish Defence Material Administration (FMV).

This work would not have been possible without the original idea of a superoptimizer suggested by Henry Massalin. We also wish to thank Robert Dewar of New York University for his urging that these results be published. Christopher Fraser, the Program Chair of this conference provided considerable advice on how to improve this paper from the initial extended abstract.

The GNU C compiler was written mostly by Richard Stallman and is being developed and distributed by the Free Software Foundation, to which numerous people and organizations have contributed. The League for Programming Freedom (LPF) is an organization that is fighting user interface copyrights and software patents, both of which would prevent the development of software like GCC. Contact the LPF for more information at **`league@prep.ai.mit.edu`** or at 1 Kendall Square #143, Cambridge, MA 02139.

References

[1] Henry Massalin, Superoptimizer: A Look at the Smallest Program, in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS II), pages 122-126, 1987.

[2] Richard Stallman, *Using and Porting GNU CC*, Free Software Foundation, 1992.

[3] Jack W. Davidson and Christopher W. Fraser, Register Allocation and Exhaustive Peephole Optimization, *Software – Practice and Experience* 14(9):857-865, September 1984.

[4] Henry Warren, Predicting Execution Time on the IBM RISC System/6000, Preliminary Version, IBM, 1991.