GEM Optimizing Compilers for Alpha AXP Systems

R. B. Grove, D. S. Blickstein, K. D. Glossop, W. B. Noyce
Digital Equipment Corporation
Nashua, NH 03062

Abstract

Digital's GEM compiler technology is used to generate state-of-the-art compilers for a variety of languages and hardware/software platforms. The Alpha AXP (tm) architecture provides a number of opportunities and challenges for compiler writers. This paper describes techniques used to optimize pipelining, multiple instruction issue, and memory references.

1 Introduction

This paper presents a technical description of the GEM compiler technology, a multi-language retargettable compiler system. We describe a number of optimization techniques that GEM compilers use to exploit the features of Alpha AXP systems [11].

Digital has developed GEM-based compilers for FORTRAN, C, C++, Ada, Pascal, COBOL, PL/I, and Bliss. The compilers operate on Alpha AXP systems under the OSF/1, OpenVMS, and Microsoft Windows NT operating systems. Digital's FORTRAN and Ada compilers on DECstations (MIPS R-series architecture) are also based on GEM. We expect that GEM will support additional languages, target architectures, and operating environments in the future.

In order to simplify the engineering of such a variety of compilers, there is an overall architecture for a "GEM-based compiler" product. The compiler is divided into several major components with carefully defined interfaces. The major components include the front ends, and the GEM optimizer, code generator, and compiler shell.

The front end does syntactic and semantic analysis of the source program and translates the program into the common GEM intermediate language (IL graphs and symbol tables). All source language specific code is encapsulated in the front end. The front ends used with GEM are based on Digital's VAX compilers. This provides a set of proven front ends with a high level of VAX compatibility and conformance to industry standards.

The optimizer transforms the IL generated by the front end into a semantically equivalent form that will execute faster on the target machine. The code generator translates the IL into a list of code cells, each of which represents one machine instruction for the target hardware. A final set of instruction optimizations and scheduling are done on the code cell representation. Most of the target architecture specific code is encapsulated in the code generator.

2 The optimizer

GEM's optimizer is a state-of-the-art, language independent, target platform independent optimizer. Optimizations include: inlining, loop unrolling, common subexpression elimination, code motion, value propagation, and strength reduction. Most of the optimizations are target independent, though some are controlled by target specific profitability tables.

2.1 Data access semantics

Since all source languages are translated to the common GEM IL and symbol table format, it is critical that their semantics be precisely specified. Many optimizations require a precise understanding of which symbols are being written or read by operations in the IL, and which operations might affect the results computed by other operations.

We developed a detailed specification known as the *Data Access Model*. It defines those operations that can write to memory, and those that can read from memory. Each of these memory accessing operations can explicitly designate the symbol being accessed when it is known. The model also requires the front end to specify when symbols may be aliased with parameters, and when they may be *pointer aliased*. A pointer aliased symbol is one that may be accessed through pointers or other operations that do not specify the symbol that they access.

The data access model provides a standard way for a front end to indicate how IL operations affect or depend upon symbols. However, some front ends can provide additional language specific discrimination of operations that cannot interfere with one another. For example, a strongly typed language like Pascal may stipulate that an assignment to a floating point target must be referring to different storage than a read of an integer. This property may hold even when the assignment target is accessed indirectly through a pointer, and cannot be associated with any particular symbol.

In order to represent language specific rules while still adhering to the philosophy that the back-end should not know about the source language, GEM employs a unique interface with the front end, known as the *Side Effects Interface*. The front end provides a set of procedures that can be called by GEM during optimization to ask questions about which IL operations have side effects, and which IL operations may depend upon those side effects.

2.2 Interprocedural optimization

GEM builds a call graph for all the routines in a compilation. The most significant interprocedural optimization that GEM performs is procedure inlining. Inlining is a well known method for reducing procedure call overhead, and increasing the effectiveness of global optimizations by increasing the scope of the operations that they can see at one time. Inlining has additional benefits on superscalar RISC architectures like Alpha because it allows the compiler to schedule the instructions of the two routines together. GEM uses heuristics to determine which calls should be inlined for maximum speed without unreasonable increases in code size or compilation time. The heuristics consider the number and kind of IL operations, the number of symbols referenced, and the kinds of optimization that would likely be enabled or disabled by inlining.

After interprocedural analysis, the routines of the user's program pass through the optimizer and code generator one at a time. GEM's interprocedural phase chooses an order for the routines that is a bottom-up order in the call graph. Except for recursive cycles, this order causes the code to be generated for a called routine before its callers. GEM records which scratch registers were actually used in a called routine, and kills only those registers at its call sites [4]. GEM also determines whether the called routine requires an argument count.

2.3 Local tree transformations

GEM uses a peephole optimizer to do local optimization of the IL. In addition to obvious simplifications such as multiplying by one or adding zero, many more interesting transformations are included. Integer division by a constant is expanded into a multiply by a reciprocal. Integer multiplication by a constant is converted into an equivalent set of shift and add or subtract operations. String operations on short fixed length strings are converted into integer operations so that they can benefit from various optimizations that are only performed on scalars. These IL peepholes are applied multiple times during the optimization of a routine. This is useful for two reasons. IL peepholes sometimes expose new optimization opportunities by expanding complex operations into more explicit components. Also, other optimizations such as value propagation may create new opportunities to apply peepholes.

2.4 Data-flow analysis

GEM's implementation of data-flow analysis uses a technique developed by Reif for determining what variables are modified between basic blocks in the flow graph [8,9]. The variables modified between a basic block and its dominator are represented as a set called the *IDEF set*. The mapping from variables to set elements is done using the side-effects interface.

The process of detecting common subexpressions (CSEs) is divided into the following tasks:

- Knowing when two expressions would compute the same results given identical inputs. Within

GEM such expressions are said to be *formally equivalent*.

- Verifying that the inputs to formally equivalent subexpressions are always identical. Such expressions are then said to be *value equivalent*. This is done using the side-effects mechanism.

- Determining when an expression dominates a value equivalent expression [6]. This guarantees that the dominating expression will have been computed any time the dominated expression is needed.

Code motions introduce the additional task of finding those places in the flow graph to which an expression could be moved such that:

- The moved expression would be value equivalent to the original expression and,

- The moved expression would execute less often than the original expression.

In previous Digital compilers, the use of data-flow analysis was largely limited to the elimination of CSEs, value propagations and code motions. We found that the data-flow analysis technique could be generalized to perform a wider variety of optimizations including field merging, induction variable detection, dead store elimination, base binding and strength reduction.

2.5 Base-binding

On RISC machines, variables in memory are referenced by loading the address into a base register and then using indirect addressing through the base register. In order to reduce the number of address loads, we attempt to share base registers among sets of variables that are known to be allocated close to each other.

By substituting a new formal equivalence function such that two address expressions are formally equivalent if they differ by less than the amount of the hardware instruction offset field, the CSE detection algorithm determines which address expressions can share a base register, and the code motion algorithm moves the base register loads out of loops.

2.6 Induction variables

Some of GEM's most valuable optimizations require the identification of induction variables and inductive expressions which is done during data-flow analysis. An expression in a loop is inductive if its value on a particular iteration is a linear function of the trip count. The simplest forms of inductive expressions are the control variables of counted loops. Expressions that are linear functions of induction variables are also inductive.

The algorithm for detecting induction variables starts by presuming that all variables modified in the loop are induction variable candidates and then disqualifies variables that aren't redefined as a linear function of themselves with a coefficient of one.

The loops that GEM analyzes have a *loop top* which dominates all nodes within the loop. The IDEF set for

a loop top is exactly those variables that are modified within the loop and thus serves as the starting value for the *induction variable candidate set*, again using the side effects mapping of variables to set elements.

During the walk of the loop, whenever a disqualifying store is encountered, the *effects* of that store are subtracted from the induction variable candidate set. Thus, at the end of the walk the remaining variables in the set are known to be true induction variables.

2.7 Strength reduction

Strength reduction is the process of taking an expensive operation and replacing it with a less expensive operation. The most basic example of strength reduction on induction transforms a loop containing a multiply:

```
        DO 20 I = 1,10
20      PRINT I*4
```

Strength reduction replaces the multiply by an add:

```
        I' = 4
        DO 20 I = 1,10
        PRINT I'
20      I' = I' + 4
```

Note that the most common array references are of the form A[I] which implies a multiplication of I by a constant (the stride of the array). Thus strength reduction yields a significant performance improvement in array-intensive computations.

To detect strength reduction candidates, we redefine formal and value equivalence as follows:

- Two inductive expressions are formally equivalent if, given identical inputs they differ only by a constant.

- Two formally equivalent inductive expressions are value equivalent if their inputs are value equivalent or are direct references to induction variables.

Thus, strength reduction candidates appear loop-invariant, and two expressions that are value equivalent can share a single strength reduction. Code motion yields the initial value of the strength reduction.

2.8 Split lifetime analysis

The GEM optimizer analyzes the usage of certain variables to determine if the stores and fetches of a variable can be partitioned (split) into disjoint variables or lifetimes. For example, a program segment like:

```
1:    V   = X * Y
2:    Z   = Z * V
3:    V   = R + S
4:    T   = T + V
```

can be transformed to:

```
1:    V'  = X * Y
2:    Z   = Z * V'
3:    V"  = R + S
4:    T   = T + V"
```

Splitting lifetimes improves register allocation and scheduling. Split lifetime analysis also is used to cache static and external variables in registers.

3 Code generation

The GEM code generator is based on matching code templates to sections of IL trees [3,7]. The code generator has a set of approximately 600 code patterns, and it uses dynamic programming to guide the selection of a least-cost covering for each statement tree in the IL graph produced by the global optimizer.

Each code template consists of: an IL tree pattern to be used in the tree matching process; a set of predicates on the tree nodes; a cost; and a set of interpretive code generation actions to be applied if the template is selected.

The actions for each code template describe the code generation actions that are required for creating temporaries, determining their lifetimes, allocating registers and stack locations, and actually emitting sequences of instructions. These actions are applied during four separate code generation passes over the IL graph for a procedure:

- During the Context pass, lifetimes of variables and temporaries are computed. Usage counts and a weight scaled by loop depth are also computed.

- The Register History pass does a dominator-order walk of the flowgraph in order to identify potential redundant loads of values that could be available in registers.

- The Temp Name pass performs register allocation using the lifetime and weight information computed during the Context pass. It also uses register history to attempt to allocate temporaries that hold the same value in the same register. If successful, this eliminates load and move instructions.

- The Code pass actually emits instructions and code labels. The code cells that result are an internal representation at the assembly code level. Each code cell contains a single target machine instruction. The code cells have specific registers and bound offsets from base registers. References to labels in the code stream are still in a symbolic form, pending further optimization and final offset assignment after instruction peephole optimization and instruction scheduling.

Tree-matching code generators were originally developed for CISC machines like the PDP-11 and VAX. The technique is also a very effective way to build a retargettable compiler system for current RISC architectures. The overall code generation structure and many of the actions are target independent. Some IL trees can use simple general code patterns, while important special cases can use more elaborate patterns and result modes.

4 Scheduling

To take advantage of high instruction-issue rates on Alpha AXP systems, compilers must carefully schedule the object code, interleaving instructions from several parts of the program being compiled.

Performing instruction scheduling only once after registers have been allocated has the disadvantage that there are artificial constraints on the ordering, as in this example:

```
ldq     r0, a(sp)  ; Copy a to b
stq     r0, b(sp)
ldq     r0, c(sp)  ; Copy c to d
stq     r0, d(sp)
```

If the load of c and store of d used some other register, the code could be rescheduled like this, to save 3 cycles on the Alpha 21064:

```
ldq     r0, a(sp)  ; Copy a to b
ldq     r1, c(sp)  ; Copy c to d
stq     r0, b(sp)
stq     r1, d(sp)
```

On the other hand, scheduling only before register allocation would ignore decisions made by the code generator. Therefore, instruction scheduling in GEM compilers occurs twice, once before registers are allocated, and once after, which is becoming fairly common in contemporary RISC compiler systems. In most other systems scheduling is done only on machine code. GEM has two different schedulers: one that schedules machine code and one that schedules IL. Before register allocation, GEM schedules the IL using heuristics to avoid reusing a register "too soon", as long as that does not increase the register usage for a routine "too much".

4.1 IL scheduling

IL scheduling is performed one basic block at a time. First a forward pass over the block gathers information needed to control the scheduling, and then a backward pass builds the new ordered list of tuples. During the forward pass, the compiler builds dependence edges to represent the necessary ordering relationships between pairs of tuples. Tuples that would require an excessive number of edges, such as CALL tuples, are instead considered markers, and no tuples at all can be reordered across a marker.

The compiler uses the data access model to determine whether two memory-access tuples conflict. It also recognizes that if two tuples have address operands with the same value (using data-flow information), but different offset attributes, the tuples must access different memory, so no dependence edge is needed. This allows more rescheduling than would otherwise be possible.

The general code for an expression tuple places its result into a compiler-generated temporary, and the general code for a STORE into a register variable involves moving the value from a compiler-generated temporary into the variable. Many GEM code patterns for expression tuples allow targetting, where the expression is computed directly into the variable instead of into a temporary. These code patterns are valid only if there are no FETCHes of the variable between the expression tuple and the STORE. In a similar way, a FETCH tuple need not generate any code (called *virtual*) if there are no STOREs between the FETCH and its consumer. For example,

T = A-1; A = B+1; C = T;

might generate the GEM IL:

```
1$: FETCH(A)
2$: SUB(1$, [1])
3$: FETCH(B)
4$: ADD(3$, [1])
5$: STORE(A, 4$)
6$: STORE(C, 2$)
```

Here the SUB can operate directly on the register allocated for A, and the ADD can target its result to the register allocated for A. The obvious dependence edge is from the FETCH(A) to the STORE(A,...). But IL scheduling must be careful not to invalidate the code patterns, as would happen if it moved the FETCH(A) between the ADD and the STORE(A), or if it moved the STORE(A) between the FETCH(A) and the SUB. To ensure this, the first pass moves the head of dependence edges backwards from targeted STOREs to the expression tuple that does the targeting. Similarly, it moves the tail of dependence edges forward from virtual FETCHes to their consumers. In this example, the edge runs from 2$ to 4$, and prevents either of the illegal reorderings.

In addition to building dependence edges, the first pass computes some heuristics for each tuple, to be used by the second, scheduling, pass. One heuristic is the *anticipated execution time* or *AET*. This estimates the earliest time at which the tuple could execute. The AET for tuple T is either the maximum AET of any tuple that must precede T, or the maximum AET + latency of T's operands. If some of the tuples that must precede T require the same hardware resources, the AET can be over-optimistic. Nevertheless, it is a useful guide to the scheduling pass.

The first pass computes another heuristic for each tuple that measures the minimum number of registers (separately for integer and floating-point registers) that would be needed to evaluate the subexpression rooted at that tuple. This is just the Sethi-Ullman number [10]: the number of registers needed to evaluate the subexpressions in the optimal order, keeping their intermediate values, plus the additional registers to evaluate the tuple itself. If the second pass schedules tuples with a lower count later in the program, the register usage will be kept low. Without some such mechanism, scheduling before register allocation tends to cause excessive register pressure.

CSEs can be treated similarly to subexpressions in this computation, with two complications. The first pass cannot predict which will be the last use of the CSE, so it treats each use as the last one. The scheduler ignores any register usage associated with CSEs that are not both created and used within the block being scheduled, allowing the register allocator to place them in memory if the scheduled code has better uses for registers.

468

The second pass of the IL scheduler works backwards over the basic block. The scheduler removes all the tuples up to the last marker, and makes available those that have no dependence edges that must follow them. It selects an available tuple and places it in the scheduled output, updates the state of each modeled functional unit, and makes available new tuples whose dependences are now satisfied. When the marker is scheduled, it removes the next group of tuples from the block, until the entire block has been scheduled.

As it places tuples in the output, the scheduler keeps track of the number of scheduled cycles, and an estimate of the number of live registers. When choosing among tuples, the scheduler prefers one whose subtree can be evaluated within the available registers, or, failing that, one whose subtree can be evaluated with the fewest registers. When several tuples qualify, it prefers one with the greatest AET, especially if the sum of AET and scheduled cycles is as large as any such sum yet seen (these tuples are on the critical path for the block).

Limiting register pressure, while not important for all programs, is important in blocks with a lot of available parallelism. With this feature, IL scheduling is a significant contributor to the high performance of GEM-compiled programs.

4.2 Instruction peepholing

After code cells have been created by the code generator, the instruction processing phases are run as a group. The first instruction processing phase is instruction peepholing, which performs a variety of very localized transformations, typically by matching patterns of adjacent instructions and replacing them with ones that are better. From the perspective of instruction scheduling, the most interesting thing the instruction peepholer does is a set of branch reductions. It also replicates short sequences of code to allow instruction scheduling to have more opportunities and to eliminate the instruction pipeline effects of branches.

Following the instruction peepholer, a control flow processing phase is run. This phase collects and propagates register information for each basic block in order to allow additional scheduling transformations, including detailed information from earlier phases about which registers are utilized by routine calls. It also determines labels that are backward branch targets for heuristic purposes, such as label alignment and avoiding motion of instructions utilizing non-pipelined resources into loops. Label alignment determination occurs before instruction scheduling because instruction alignment is important for the Alpha 21064 where pairs of instructions must be aligned on quadword boundaries to exploit dual instruction issue.

4.3 Instruction scheduling

The next phase is the instruction scheduler. At this point, all register binding has occurred, and all code modifications other than branch/jump resolution have occurred.

The scheduler does a forward walk over the basic blocks in each code section, so that the alignment of the first instruction in each block is known.

For each scheduling unit, which is a single-entry multi-exit block, the instruction scheduler performs two passes which are effectively the inverse of the passes that the IL scheduler performs. It does a backward walk to determine instruction ordering requirements and path length to the end of the block, and a forward pass that actually schedules the code.

Since the scheduler has knowledge of register utilization at branches and routine calls, instructions that write registers that are not exposed at a given branch are not constrained by that branch, unless they have side effects, including potential exceptions. GEM can determine that some memory references cannot cause exceptions and that loads can be moved back across branches, reducing their effective latency.

The backward ordering pass uses an AET computation similar to the one used by the IL scheduler, though somewhat more detailed. The instruction scheduler knows the actual instructions to be scheduled, and it has a more detailed machine model. For the Alpha 21064, the detailed information includes asymmetric bypassing information and information about multiple issue. For architectures that have branch delay slots, the AET computation is also biased so that instructions that are likely to be able to fill branch delay slots will occur immediately before branches.

The forward scheduling pass actually does a cycle by cycle model of the machine, including modelling multiple issue. There were several reasons for chosing this approach, rather than an approach that just selects an ordering of the instructions:

- For machines with significant issue limitations (e.g. non-pipelined functional units or multiple issue *pairing rules*), packing the limiting resource well often dominates getting a good schedule. A cycle model effectively allows other instructions to *float* into the no-issue slots while allowing the critical resource to be scheduled well.

- Modelling the machine allows easy determination of where stalls are occurring, which will allow instructions from the current block or from successor blocks to be moved into no-issue slots.

- Modelling the machine in a forward direction models the fact that processors are typically *greedy* and normally issue all of the instructions that they can at a given point in time.

- The cycle model allows a variety of dumps, which can be useful both to people using the compiler system, and to developers trying to improve the performance of generated code.

The forward pass does a topological sort of the instructions. As each instruction is scheduled, any instructions which have either a direct dependence or an anti-dependence (e.g. register re-use) are moved to a data structure called the issuing ring, which is the set of instructions that will become available at a future cycle. As the modelled machine time advances, these instructions become available for issuing.

469

The set of instructions that are available for issuing is represented as a list of data structures known as heaps, which are a priority queue. Each heap on the list contains instructions with a similar *signature* - for example, all available instructions that need access to the integer instruction pipeline, or all store instructions. The reason for this is that the top instruction in each heap is examined when looking for the next instruction to issue. Inside each heap, instructions are typically ordered by their AET values, sometimes with occasional small biases for different instruction properties, such as loads which may have a variable execution time longer than the projected time.

The heaps themselves are, in turn, ordered in the list according to how desireable it is that a particular heap's top instruction be issued. All non-pipelined instruction heaps are first on the list, followed by all semi-pipelined ones, then all fully pipelined ones. (A semi-pipelined resource is one that may prevent certain instructions from issuing in certain future cycles, but that can issue every cycle. For example, stores on some machines interact with later loads.)

If there are instructions which use multiple resources, this is typically represented in the heap ordering. (For example, for the MIPS R3000, floating multiplies use both the multiplier and some of the same resources as adds. As a result, the heap holding multiplies is always kept ahead of the heap holding adds in the list.) This combination of orderings works very well for both machines with a significant number of non-pipelined units, such as the MIPS processors, and on machines like the Alpha 21064 that have largely pipelined functional units, but also have multiple issue with only particular combinations of multiple issue allowed.

It is worth noting that other than the architecture specific computation for AET, and per-processor implementation data tables, the scheduler is completely target-independent. For example, there are currently processor implementation tables for the MIPS R3000 and R4000 processors, the Alpha 21064 processor, and Alpha processors that are under development.

5 Optimizing memory references

The Alpha AXP architecture does not have 8-bit and 16-bit load and store instructions. Not having byte store instructions allows simpler and faster cache and memory implementations; the cache and memory subsystems are optimized for aligned 32-bit and 64-bit operations.

Alpha provides 64-bit register-to-register Extract, Mask, and Insert operations to support byte manipulation. The compiler must use these primitives to generate efficient code for char and short data in C. Figure 1 shows code to load a 16-bit zero-extended value. GEM uses 2 instructions for aligned data, 5 instructions if unaligned, and a faster 7-instruction sequence if "probably aligned". The "probably aligned" code runs quickly for the aligned case, because the branch is correctly predicted to fall through, while still getting the correct value for unaligned data. Similar code patterns handle stores.

GEM has a large set of load and store code patterns

```
Aligned 16-bit load
      ldq_u   r1, (r0)
      extwl   r1, r0, r1

Unaligned 16-bit load
      ldq_u   r1, (r0)
      ldq_u   r2, 1(r0)
      extwl   r1, r0, r1
      extwh   r2, r0, r2
      or      r2, r1, r1

Probably aligned 16-bit load
      ldq_u   r1, (r0)
      extwl   r1, r0, r1
      blbs    r0, 20$
10$:  ...

20$:  ldq_u   r28, 1(r0)
      extwh   r28, r0, r28
      or      r1, r28, r1
      br      10$
```

Figure 1: Code to load 16 bits

to deal with combinations of data size, alignment, and sign-extension. This is a domain where RISC machines are not simple and a sophisticated code selection mechanism is required.

Generating efficient code for the extraction and insertion of fields within records is particularly challenging on RISC architectures. Often a program will fetch or store several fields that are contained in the same word. Without optimization, each fetch would load the word from memory, and each store would both load and store the word. But it is possible to perform a collection of field fetches and stores with a single load and store to memory. As another example, two bit tests within the same word could be done in parallel as a mask operation.

In the IL generated by the front end, each field operation is generated as a separate IL operation. The real task of optimizing field accesses is to identify IL operations that can be combined. The underlying problem is that the redundant loads and stores are not visible in this representation. The first part of the solution involves expanding the field fetch or store into lower-level operators (Fetch, Extract, Insert, Store). With the loads exposed (as Fetches) data-flow is capable of eliminating redundant loads. Similarly, each field store expands into a fetch of the *background* word, an insertion of the new data into the proper position, and a store back. Given two field stores, value propagation can eliminate the second fetch, and then dead-store elimination can eliminate the first store.

Unoptimized code for the example in Figure 2 would contain a load and extract for each reference to n_kind or n_flags, plus an insert and store for the latter two references. In the optimized code in Figure 3, GEM eliminated two of three loads, two of three extracts, both inserts, and one of the two stores.

```
typedef struct node {
    char n_kind, n_flags;
    struct node *xl_car, *xl_cdr;
} NODE;

#define MARK  1
#define LEFT  2

void demo(ptr)
  NODE *ptr;
{
    while (ptr) {
        if (ptr->n_kind == 0) {
            ptr->n_flags |= MARK;
            ptr->n_flags &= ~LEFT;
        }
        ptr = ptr->xl_cdr;
    }
}
```

Figure 2: Field-merging C code

```
demo::
    beq   ptr, 3$
    nop
1$: ldl   r0, (ptr)      ; Load
    and   r0, 255, r1
    bne   r1, 2$         ; Test n_kind
    mov   256, r17
    or    r0, r17, r17   ; Set MARK
    mov   -513, r1
    and   r17, r1, r17   ; Clear LEFT
    stl   r17, (ptr)     ; Store
2$: ldl   ptr, 8(ptr)
    bne   ptr, 1$
3$: ret   r26
```

Figure 3: Field-merging optimized code

6 Branch optimization examples

Branch instructions can hurt the performance of
high performance systems in several ways. In addition
to the instruction space and the time to issue the in-
struction, branches can disrupt the hardware pipeline.
Also, branches can inhibit optimizations such as code
scheduling. Therefore, GEM uses several strategies to
avoid branches in the IL and generated code, or to
eliminate some of their bad effects.

Some branches appear as part of a well-defined pat-
tern that need not inhibit optimizations. GEM uses
special operators for these cases. A simple example
is the MAX function. For Alpha, MAX can be im-
plemented using the CMOVxx instructions, avoiding
branch instructions entirely; for other architectures
the main benefit is that the branch doesn't appear in
the IL. A more complicated example is the so-called
"flow-Boolean" operators. Consider the following C
example:

$$x = (p\ \&\&\ *p)\ ?\ *y\ :\ *z;$$

which generates the following GEM IL:

```
 1$: FETCH(P)
 2$: NONZERO(1$)
 3$: ANDSKIP(2$)
 4$: FETCH(1$)
 5$: NONZERO(4$)
 6$: LANDC(3$, 5$)
 7$: SELTHEN(6$)
 8$: FETCH(Y)
 9$: FETCH(8$)
10$: SELELSE(9$)
11$: FETCH(Z)
12$: FETCH(11$)
13$: SELC(7$, 10$, 12$)
14$: STORE(X, 13$)
```

The ANDSKIP and LANDC tuples implement the
conditional-and operator: if tuple 2$ is false, tuples
4$ and 5$ are skipped, and the result of the LANDC
is false; otherwise the LANDC uses the result of tuple
5$. Similarly, the SELTHEN, SELELSE, and SELC
tuples implement the select operator. If tuple 6$ is
true then tuples 8$ and 9$ compute the result, and
tuples 11$ and 12$ are skipped. If tuple 6$ is false
then tuples 8$ and 9$ are skipped, and tuples 11$ and
12$ compute the result.

These operators allow branching code to be rep-
resented within the standard basic-block framework,
but require branches in the generated code, to avoid
undesired side-effects of the skipped tuples. In some
cases, though, GEM can determine that the skipped
tuples have no side-effects. In these cases, GEM con-
verts the operators to an unconditional form, allowing
the generated code to be free of branches.

GEM performs some other transformations on the
IL to remove branches. For example, it transforms

```
if (expr) var = 1; else var = 0;
```

into

```
var = ((expr) != 0);
```

and

```
if (expr) var = const;
```

into

```
var = (expr) ? const : var;
```

Both of these transformations enable further optimizations by eliminating branches.

Alpha implementations typically include a branch prediction mechanism. Correctly predicted branches take several cycles less time than mispredicted branches. The fastest conditional branch is one that is correctly predicted not to be taken. GEM uses several strategies to arrange branches for best performance.

GEM selects an order for the basic blocks of a program that may be different from the order in the source program. For each basic block that ends with an unconditional branch, the target block is placed next unless that block has already been placed. Similarly, if a basic block within a loop ends with an unconditional branch, a target block within that loop is placed next if possible. For example,

```
while (--i > 0) {
  if (a[i] != b[i]) return a[i]-b[i];
  a[i] = 0;
}
```

First, GEM transforms the pre-tested loop into a post-tested loop to eliminate the unconditional branch when the loop iterates. Since the return statement is outside of the loop, the generated code looks more like:

```
if (--i > 0)
  do {
    if (a[i] != b[i]) goto label;
    a[i] = 0;
  } while (--i > 0);
  ...

label:  return a[i]-b[i];
```

GEM can also unroll loops. This reduces the number of times the branch back must be executed. More important, it often allows operations from different iterations to be scheduled together. Unrolling by 4 transforms the above loop into a cleanup loop, and a main loop that looks approximately like:

```
do {
  if (a[i] != b[i]) goto label;
  a[i] = 0;
  if (a[i-1] != b[i-1]) goto label;
  a[i-1] = 0;
  if (a[i-2] != b[i-2]) goto label;
  a[i-2] = 0;
  if (a[i-3] != b[i-3]) goto label;
  a[i-3] = 0;
} while (i -= 4);
```

This code executes 4 fall-through branches and 1 taken branch where the original code executed 4 fall-through branches and 4 taken branches.

7 Conclusion

The GEM compiler system has met demanding technical and time-to-market goals. It has been retargetted and rehosted for the Alpha and MIPS architectures and several operating environments. GEM supports an extremely wide range of languages and provides high levels of optimization for each.

This paper describes the current version of the GEM compiler system. A portable, optimizing compiler provides many opportunities for extensions and improvements. Some of the enhancements we plan for future versions include:

- Dependence analysis and loop transformations to optimize use of the memory hierarchy

- Software pipelining to increase parallelism in vector loops

- Scheduling improvements - better reordering of memory references, more scheduling across basic blocks.

- Support for additional host/target combinations

Acknowledgments

References

[1] A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees", *JACM*, 23,3 (July 1976), pp. 488-501.

[2] D. S. Blickstein, P. A. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. B. Grove, S. O. Hobbs, W. B. Noyce, "The GEM Optimizing Compiler System", *Digital Technical Journal*, 1993.

[3] R. G. G. Cattell, *Formalization and Automatic Derivation of Code Generators*, Ph.D. thesis, CMU-CS-78-115, Carnegie-Mellon Univ., Pittsburgh, PA, April 1978.

[4] F. C. Chow, "Minimizing Register Usage Penalty at Procedure Calls", *SIGPLAN 88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, pp. 85-94.

[5] K. W. Harris and S. O. Hobbs, "VAX Fortran", in *Optimization in Compilers*, edited by Fran Allen, Barry Rosen, and F. Kenneth Zadek, ACM Press, New York, in press.

[6] Thomas Lengauer and Robert E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph", *TOPLAS*, 1,1 (July 1979), pp. 121-141.

[7] B. W. Leverett et al., "An Overview of the Production Quality Compiler Compiler Project", *Computer*, 13,8, (August 1980), pp. 38-49.

[8] J. H. Reif, "Symbolic Interpretation in Almost Linear Time", *Fifth ACM Symposium on Principles of Programming Languages*, pp. 76-83, 1978.

[9] J. H. Reif and R. E. Tarjan, "Symbolic Program Analysis in Almost-Linear Time", *SIAM J. Computing*, 11,1 (February 1981), pp. 81-93.

[10] R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions", *JACM* 17:4, pp. 715-728, 1970.

[11] Richard L. Sites (ed.), *Alpha Architecture Reference Manual*, Digital Press, Bedford, MA, 1992.

[12] William Wulf et al., *The Design of an Optimizing Compiler*, American Elsevier Publishing Co., New York, 1975.