The GEM Optimizing Compiler System

1 Abstract

The GEM compiler system is the technology Digital is using to build state-of-the-art compiler products for a variety of languages and hardware /software platforms. Portable, modular software components with carefully specified interfaces simplify the engineering of diverse compilers. A single optimizer, independent of the language and the target platform, transforms the intermediate language generated by the front end into a semantically equivalent form that executes faster on the target machine. The GEM system supports a range of languages and has been successfully retargeted and rehosted for the Alpha AXP and MIPS architectures and for several operating environments.

In the past, Digital has made major investments in optimizing compilers that were specifically directed at one hardware platform, namely VAX computers. When Digital began broadening its hardware offerings to include reduced instruction set computer (RISC) architectures, it became clear that new optimization technology was needed, as well as a new strategy for leveraging investments in compiler technology across an increasing number of hardware platforms.

This paper presents a technical description of the GEM compiler technology that Digital uses to generate compiler products for a wide range of hardware and software combinations. We begin with an explanation of the GEM strategy of leveraging investments by using portable, modular software components to build compiler products. The bulk of the paper describes the GEM optimizer and code generator technologies, with a focus on how they address challenges posed by the Alpha AXP architecture.[1] We then move to a discussion of compiler engineering and conclude with an overview of some planned enhancements to the software.

2 GEM Compiler Architecture

Because of the many hardware platforms available, often with multiple operating systems and a variety of languages offered on those platforms, building a compiler from scratch for each combination is no longer feasible. To simplify the engineering of diverse compilers, GEM compiler products share a basic architecture. The compiler is divided into several major components, in effect, the fundamental building blocks from which a compiler is constructed. The interfaces among these components are carefully specified. The major components of a GEM compiler are the front end, the optimizer, the code generator, and the compiler shell. The logical division of GEM components and the range of GEM support is shown in Figure 1. Note that the host is the computer on which the compiler runs, and the target is the computer on which the generated object runs.

The front end performs lexical analysis and parsing of the source program. The primary outputs are intermediate language (IL) graphs and symbol tables, which are both standardized. In an IL graph, each node, referred to as a tuple, represents an operation. Front ends for all source languages translate to the single standard IL. All language-specific code is encapsulated in the front end. All knowledge of the source language is communicated in the IL or through callbacks to the front end. Knowledge of the target hardware is represented in tables and in a minimal amount of conditional code.

The optimizer transforms the IL generated by the front end into a semantically equivalent form that will execute faster on the target machine. A significant technical achievement is that a single optimizer is used for all languages and target platforms.

The code generator translates the IL into a list of code cells, each of which represents one machine instruction for the target hardware. Virtually all the target machine instruction-specific code is encapsulated in the code generator.

The shell is a collection of common compiler functions such as listing generators, object file emitters, and command line processors. Basically, the shell is a portable interface to the external environment in which the compiler is used. This interface allows the other components to remain independent of the operating system.

There are numerous benefits to this modular approach:

o Adding a new feature to a common component enhances many products.

o Source language compatibility is ensured among all compilers that use the same front end.

o Standardized interfaces enable us to plug in a new front end to build a compiler for a new language, or a new shell to allow the compiler to run on a new host.

o When a new language is added, it can be offered quickly on many platforms.

o When a new target CPU or operating system is supported, many languages are immediately available to that target.

3 Order of Processing

When compiling a program, the overall order of processing must be carefully arranged so that each component of the compiler can see a large portion of the program at one time. When processing one portion of a program, certain information about other relevant parts of the source program can be useful.

Figure 2 illustrates the overall process of compiling a program. Since GEM compilers include interprocedural optimizations, as much of the program as possible should be presented to the optimizer at the same time. For this reason, GEM compilers allow the user to process multiple source files in a single compilation. The front end parses these source files and constructs the symbol table and a compact form of IL in memory before invoking the GEM back end. The portion of the user's program thus compiled is called a compilation unit.

The GEM back-end interprocedural optimization phase is the first to operate on the program. This phase analyzes the routines within a compilation unit to develop a call graph that shows which routines might call which other routines. Interprocedural optimizations are applied to the routines as a group.

Next, the global optimizer and the code generator process each routine in a bottom-up order, resulting in a translation of the program to code cells that represent operations at machine level. This bottom-up order is convenient for certain optimizations, as discussed in the Optimization section. The first action of the global optimizer is to translate the routine's IL from the compact form provided by the front end to an expanded form used by the optimizer and the code generator. Since only one routine at a time is stored in expanded form, a much larger data structure can be used to store the results of the optimizer analysis. The expansion from compact form also expands certain shorthand forms, which are convenient for a front end, into explicit operations in the expanded IL, much like a macro expansion facility in a source language.

Once all the routines have been processed by the global optimizer and the code generator, a complete description of the program is available at the machine instruction level. Certain machine-specific optimizations, such as peephole optimizations and instruction scheduling, are performed on this program description. Finally, the optimized machine instructions are converted to the appropriate object language for the target operating system.

4 Optimization

The GEM compiler system's optimizer is state-of-the-art and independent of the language and the target platform. The input to the optimizer is the IL and symbol table for multiple routines; the output is the semantically equivalent IL and symbol table, both modified to run faster on the target platform.

GEM optimizations include interprocedural optimizations, modern optimizations for superscalar RISC architectures such as the Alpha AXP architecture, plus a robust implementation of the classical global optimizations. In addition, GEM's code generator includes a number of optimization features that help it produce extremely high local code quality.

Design Principles

Certain general design approaches or principles were applied throughout the optimizer. For instance, choices had to be made in the design of the IL; the front end could either provide a higher-level description of program features or rely on the back end to derive the higher-level description from an analysis of a lower-level description. In cases where accurate, well-defined algorithms for deriving those higher-level features exist, GEM chooses to derive the descriptions.

Describing source code loops is a key example of the implementation of this design principle. Most source languages have explicit syntax for writing loops, and the front end could translate these languages into a higher-level IL that designates those loops. Instead, GEM uses a lower-level IL with primitives such as conditional branch and label operators. The advantage of this approach is that GEM recognizes all loops, even those constructed with GOTO statements.

A general design approach that emerged from experience gained during the GEM project is the use of enabling or expanding transformations to support fundamental optimizations. Often, representing operations in the IL in a way that hides certain implicit operations is a compact and efficient approach. However at times, making these implicit operations explicit allows a particular optimization routine to operate on them. A good solution to this problem is to initially represent the operations in the IL in the compact form. Then, before applying optimizations that could benefit from seeing the implicit operations, apply expanding transformations to convert the IL into a longer form in which all operations are explicit.

Out of concern for the time required to compile large programs, GEM also established the design principle that the order of complexity as a function of the number of IL operations should be as close to linear as possible.

Data Access Model and Side Effects Interface

Since GEM compilers translate all source languages to a common IL and symbol table format, the semantics of these languages must be specified precisely. Many optimizations require an exact 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.

The GEM team developed a detailed specification known as the data access model, which 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

may be accessed through pointers or other operations that do not specify
the symbol that they access.

The model can indicate that the pointer-aliased property is derivable, i.e., a symbol is pointer aliased only if an operation that stores its address is present in the IL. A special IL operator marks such operations. When the derivation of this property is deferred, the optimizer can avoid marking symbols pointer aliased.

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 be allowed to interfere with one another. For example, a strongly typed language like Pascal may stipulate that an assignment to a floating-point target must refer to different storage than an integer read, even when the assignment target is accessed indirectly through a pointer.

To represent language-specific rules while adhering to the philosophy that the back end should have no knowledge of the source language, GEM compilers employ a unique interface with the front end, called the side effects interface. The front end provides a set of procedures that GEM can call during optimization to ask which IL operations have side effects and which IL operations depend upon those side effects.

Interprocedural Optimization

GEM's interprocedural optimization phase starts by walking over the IL for all routines to build the call graph. The call graph is a directed multigraph in which the nodes are routines, and the edges are calls from one routine to another. The graph is not a tree because recursion is allowed. A special virtual routine node represents all unknown routines that might call or be called by a routine in this compilation.

GEM walks the graph to determine which local symbols that are potential targets of up-level access are actually referenced in a called routine. When up-level references do occur, GEM can also determine the most efficient way to pass that context from the routine that declares the variable to the routine that references it.

On the same walk, GEM analyzes the use of symbols whose pointer-aliased property is derivable. If operations that store the address of such a symbol are present, then the symbol is marked as pointer aliased. The front end's indication is also retained so that this analysis can be repeated after address storing operations are eliminated.

The most significant interprocedural optimization that GEM performs is procedure inlining. Inlining is a well-known method for reducing procedure call overhead and for increasing the effectiveness of global optimizations by enlarging the scope of the operations seen at one time. Inlining has additional benefits on superscalar RISC architectures, like the Alpha AXP system, because the optimization allows the compiler to schedule the instructions of the two routines together.

GEM's inliner reviews all calls in the call graph and uses heuristic algorithms 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.

When callers pass constants as actual parameters, better optimization is likely to result from inlining because the corresponding formal parameter will have a known constant value. On the other hand, when two sections of the same array are passed as arguments, and the corresponding formals are described as not aliased with one another, eliminating the formal parameters through inlining discards valuable alias information.[2,3]

Also, the order in which inlining decisions are made can be important. In a chain of calls in which A calls B and B calls C, the call from A to B might be the most desirable inlining candidate. However, if the call from B to C is inlined first, the size of B may increase, making it a less attractive candidate for inlining into A. Consequently, GEM uses its heuristics to preevaluate all calls and then orders the calls by desirability. GEM inlines the most desirable candidate first, and then reevaluates the caller's properties, possibly adjusting its position in the ordered list.

In many C programs, the address of a variable (especially a struct) is passed to a called routine that refers to the variable through a pointer formal parameter. After inlining, a symbol's address is stored in a pointer and indirect references are made through the pointer. Later, while optimizing the routine, GEM's value propagation often eliminates the pointer variable. Finally, when one or more pointer-storing operations have been eliminated, GEM reevaluates the pointer-aliased property of derivable local symbols, and the variable that was once passed by address is no longer pointer aliased.

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 a bottom-up routine order in the call graph. Except for recursive cycles, this order causes GEM to generate the code for a called routine before generating the caller's code. GEM takes advantage of this property by recording the scratch registers that were actually used in a called routine and adjusting register usage at its call sites.[4] GEM also determines whether or not the called routine requires an argument count.

Intermediate Language Peepholes

GEM uses a peephole optimizer to improve the IL. In addition to performing the many obvious simplifications such as multiplying by one or adding zero, the optimizer performs other transformations. Integer division by a constant is expanded into a multiply by a reciprocal operation, which can be efficiently implemented with a UMULH instruction. String operations on short fixed-length strings are converted into integer operations, to benefit from various optimizations performed only on scalars. Also, integer multiply operations by a constant are converted into an equivalent set of shift and add or subtract operations.

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. To take advantage of these opportunities, GEM compilers apply these IL peepholes multiple times during the optimization of a routine.

Data-flow Analysis

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

The process of detecting CSEs is divided into the tasks of

- Knowing when two expressions would compute the same results given identical inputs. Within GEM compilers, such expressions are said to be formally equivalent.

- o Verifying that the inputs to formally equivalent subexpressions are always identical. Such expressions are said to be value equivalent. This verification is accomplished by using the side effects mechanism.

- o Determining when an expression dominates a value equivalent expression.[5] This information guarantees that GEM will have computed the dominating expression whenever 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 legally moved such that

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

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

The following sections describe how GEM detects base-binding and strength-reduction candidates by substituting slightly different equivalence functions.

Base Binding

On RISC machines, a variable in memory is referenced by loading the address into a base register and then using indirect addressing through the base register. To reduce the number of address loads, sets of variables that are closely allocated share base registers. GEM considers two address expressions formally equivalent if they differ by an amount less than the range of the hardware instruction offset field. The CSE detection algorithm determines which address expressions are formally equivalent and thus can share a base register, and the code motion algorithm moves the base register loads out of loops.

Induction Variables

Some of GEM's most valuable optimizations require the identification of inductive expressions and induction variables, 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.

GEM's implementation of data-flow analysis uses a technique for determining what variables are modified between basic blocks in the flow graph.[6,7] 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 algorithm for detecting induction variables starts by presuming that all variables modified in the loop are induction variable candidates. It then disqualifies variables not redefined as a linear function of themselves with a coefficient equal to one. The loops that GEM chooses to analyze have a loop top that 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 contents of the candidate set are updated. Thus, at the end of the walk, the remaining variables in the set are known to be true induction variables.

Strength Reduction of Induction Variables

Strength reduction is the process of replacing an expensive operation with a less expensive operation. The most basic example of strength reduction on induction is as follows:

If the original source program was

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

strength reduction would reduce the multiply to an add as follows:

```
     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 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:

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

- o 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.

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, i.e., split, into disjoint variables or lifetimes.

For example, consider the following program segment:

```
  1:   V = X * Y
  2:   Z = Z * V

  3:   V = R + S
  4:   T = T + V
```

The references to V can be divided into two disjoint lifetimes V' and V" without changing the semantics of the program as in:

```
  1:   V' = X * Y
  2:   Z  = Z * V'

  3:   V" = R + S
  4:   T  = T + V"
```

V' and V" can be treated as two completely independent variables. This has several useful applications.

o V' and V" can be assigned to different registers, each with shorter lifetimes than the original variable V. The allocator can thus pack registers and memory more tightly.

o V' and V" can be scheduled independently. For example, the computation of Z in line 2 could be scheduled after the redefinition of V in line 3.

o A lifetime that begins with a fetch is an uninitialized variable. GEM issues a diagnostic in such cases.

o Any lifetime with only stores is effectively "dead," and thus, the stores can be eliminated.

o When a lifetime of an induction variable contains an equal number of stores and fetches, the variable is used only to compute itself. Thus, the whole lifetime can be eliminated. This is called induction variable elimination.

o GEM uses split lifetime information to optimize the flushing and reloading of register variables around routine calls.

o GEM uses split lifetime information to determine what variables are potentially referenced by exception handlers.

o Lifetimes often need to be extended around loop tops and loop bottoms. Split lifetime analysis has full information in many cases in which the code generator's lifetime computation must make pessimistic assumptions.

Thus, analyzed variables are allocated more efficiently inside loops.

The technique GEM uses for split lifetime analysis is based on the VAX Fortran SPLIT phase.[8] The technique includes several extensions in the areas of induction variables, unselected variables (the original algorithm analyzed only a fixed number of variables), and exception handling.

5 Code Generation

The GEM code generator matches code templates to sections of IL trees.[9] The code generator has a set of approximately 600 code patterns and 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 pattern specifies a set of interpretive code-generation actions to be applied if the template is selected. The code-generation actions create temporaries, determine their lifetimes, allocate registers and stack locations, and actually emit sequences of instructions. These actions are applied during the following four separate code-generation passes over the IL graph for a procedure:

o Context. During the context pass, the code generator creates data structures that describe each temporary variable. The information computed includes the lifetime, usage counts, and a weight scaled by loop depth.

o Register history. During the register history pass, the code generator does a dominator-order walk of the flow graph to identify potential redundant loads of values that could be available in registers.

o Temp name. During the temp name pass, the code generator performs register allocation using the lifetime and weight information computed during the context pass. The code generator also uses register history to allocate temporaries that hold the same value in the same register. If successful, this action eliminates load and move instructions.

o Code. During the code pass, the code generator emits instructions and code labels. The resulting code cells 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 in a symbolic form, pending further optimization and final offset assignment after instruction peephole optimization and instruction scheduling.

Template Matching and Result Modes

Code template enumeration and selection occurs during the context pass. The enumeration phase scans IL nodes in execution order (bottom-up) and labels each node with alternative patterns and costs. When a root node such as a store or branch tuple is reached, the lowest-cost template for that node is selected. The selection process is then applied recursively to the leaves for the entire tree.[10]

The IL tree pattern of a code-generation template consists of four pieces of information:

- A pattern tree that describes the arrangement of IL nodes that can be coded by this template. The interior nodes of the pattern tree are IL operators; the leaves are either result mode sets or IL operators with no operands.

- A predicate on the tree nodes of the pattern. The predicate must be true in order for the pattern to be applicable.

- A result mode that encodes the representation of a value computed by the template's generated code.

- An integer that represents the cost of the code generated by this template.

The result modes are an enumeration of the different ways the compiler can represent a value in the machine.[11] GEM compilers use the following result modes:

- Scalar, for a value, negated value, and complemented value

- Boolean, for low-bit, high-bit, and nonzero values

- Flow, for a Boolean represented as control flow

- Result modes for different sizes of integer literals

- Result modes for delayed generation of addressing calculations

- Result modes indicating that only a part of a value has been materialized, i.e., the low byte, or that the materialized value has used a lower-cost solution

As templates are matched to portions of the IL tree, each node is labeled with a vector of possible solutions. The vector is indexed by result mode, and the lowest-cost solution for each result mode is recorded on the forward bottom-up walk. When a root node is encountered, the lowest-cost template in its vector of solutions is chosen. This choice then determines the required result mode and solution for each leaf of the pattern, recursively.

GEM Code Generator Action Language

The GEM code generator uses and extends methods developed in the BLISS compilers, the Carnegie-Mellon University Production-Quality Compiler-Compiler Project, and Digital's VAX Pascal compiler.[12,13] One key GEM innovation is the use of a formalized action language to give a unified description of all actions performed in the four code-generation passes. The same formal action descriptions are interpreted by four different interpreters. For example, the Allocate_TN action is used to allocate long-lived temporaries that may be in a register or in memory. This action creates a data structure describing the temporary in the context pass, allocates a register during the temp name pass, and provides the actual temporary location for code emission.

Tree-matching code generators were originally developed for complex instruction set computer (CISC) machines, like the PDP-11 and VAX computers. The technique is also an effective way to build a retargetable compiler system for current RISC architectures. The overall code-generation structure and many of the actions are target independent. Some IL trees use simple, general code patterns, whereas special cases use more elaborate patterns and result modes.

Register Allocation

GEM compilers use a simple linear model to characterize register lifetimes. The context, temp name, and code passes process the basic blocks and the IL nodes of each block in execution order. Each code pattern has a certain number of lifetime ticks to represent points at which a temporary value is created or used. The lifetime of a temporary is then the interval defined by its starting lifetime tick and ending lifetime tick.

Simple expression temporaries have a linear lifetime contained within a basic block. User variables and CSEs may require that lifetimes be extended to cover loop tops and loop bottoms. The optimizer inserts special begin and end markers to delimit the precise lifetimes of variables created by the split lifetime phase.

The code generator uses a number of heuristics to allocate registers to avoid copying. If a new lifetime begins at exactly the same tick as another lifetime ends, this may indicate that they should share a register. Otherwise, the allocator uses a round-robin allocation to avoid packing registers too tightly, which would inhibit scheduling. The Move_Value action is used to copy one register to another and provides a hint that the source and destination should be allocated to the same register.

Actual allocation of registers and stack temporaries occurs in the temp name pass. The allocator uses a bin-packing technique to allocate each compiler and user variable to a register or to memory.[14] The allocator first attempts to assign variables to registers; lifetimes that conflict cannot be assigned to the same register. The allocator uses a density function to control the process. A new candidate can displace a previous variable that has a conflicting lifetime if this action increases the density measure. After the allocation of temporaries to registers is completed, any unallocated or spilled temporaries are allocated to stack locations.

Scheduling

To take advantage of high instruction-issue rates in 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 places artificial constraints on the ordering, as illustrated in the following code 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 were to use some other register, the code could be rescheduled to save three cycles on the DECchip 21064 processor, as shown in the following code:

```
  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 does not incorporate decisions made by the code generator. Therefore, instruction scheduling in GEM compilers occurs twice, before and after registers are allocated. This practice is fairly common in contemporary RISC compiler systems. In most other systems, scheduling is performed only on machine code. GEM has two different schedulers - one that schedules machine code and one that schedules IL.

Intermediate Language 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 considered markers. No tuples can be reordered across a marker.

The compiler uses the data access model to determine whether two memory-access tuples conflict. Also, if two tuples have address operands with the same value (using data-flow information) but different offset attributes, the tuples must access different memory. Thus, no dependence edge is needed, and more rescheduling is possible.

The general code for an expression tuple places the result into a compiler-generated temporary, and the general code for a store into a register variable moves the value from a temporary into the variable. Many GEM code patterns for expression tuples allow targeting, 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 operation. Similarly, a fetch tuple need not generate any code (called virtual), if no stores exist 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$: STORE(C, 2$)
```

In this example, SUB operates directly on the register allocated for A, and ADD targets its result to the register allocated for A. The obvious dependence edge is from FETCH(A) to STORE(A,...). However, IL scheduling must be careful not to invalidate the code patterns, which would happen if it moved FETCH(A) between ADD and STORE(A) or STORE(A) between FETCH(A) and SUB. To ensure valid code patterns, the first pass moves the head of dependence edges backward from targeted stores to the expression tuple that does the targeting. Similarly, the first pass 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 heuristics for each tuple, to be used by the second, i.e., scheduling, pass. One heuristic, the anticipated execution time (AET), 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 plus the latency of T's operands. If some of the tuples that must precede T require the same hardware resources, the AET may be optimistic. Nevertheless, the AET is a useful guide to the scheduling pass.

The first pass also computes the minimum number of registers (separately for integer and floating-point registers) needed to evaluate the subexpression rooted at a particular tuple. The value of this heuristic is the Sethi-Ullman number, i.e., 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.[15] If the second pass schedules tuples with a lower count later in the program, the register usage will be kept low. Without such a mechanism, scheduling before register allocation tends to cause excessive register pressure.

CSEs can be treated similarly to subexpressions in this computation, but with two complications. The first pass cannot predict the last use of the CSE and therefore 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. This action allows the register allocator to place the CSEs in memory, if the scheduled code has better uses for registers.

The second pass of the IL scheduler works backward over the basic block. The scheduler removes all the tuples up to the last marker and makes available those that have no dependence edges to tuples that must follow. The scheduler then 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, the scheduler continues to remove the preceding group of tuples from the block until the entire block has been scheduled.

The scheduler keeps track of the number of scheduled cycles and the estimated 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, the scheduler chooses the one with the greatest AET.

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.

Instruction Peepholing

After code has been generated or code cells have been created directly, the instruction processing phases are run as a group. Instruction peepholing performs a variety of localized transformations, typically by matching patterns of adjacent instructions and replacing them with better patterns. From the perspective of instruction scheduling, the most interesting function of the instruction peepholer is to perform a set of branch reductions. The peepholer also replicates short sequences of code to facilitate instruction scheduling and to eliminate the instruction pipeline effects of branches.

A control flow processing phase follows the instruction peepholing phase. Currently, this phase determines labels that are backward branch targets for alignment purposes. This action occurs before instruction scheduling, because instruction alignment is important for the DECchip 21064 Alpha AXP processor, in which instructions must be aligned on quadword boundaries to exploit dual instruction issue. In the near future, the control flow processing phase will collect register information for each basic block to allow additional scheduling transformations.

Instruction Scheduling

The instruction scheduler is the next phase. At this point, all register binding and code modifications other than branch/jump resolution have occurred. The scheduler does a forward walk over the basic blocks in each code section to determine the alignment of the first instruction in each block.

For each basic block, the instruction scheduler does two passes that are effectively the inverse of the passes that the IL scheduler performs, namely 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.

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

The forward scheduling pass does a cycle-by-cycle model of the machine, including modeling multiple issue. The reasons for choosing this approach rather than an approach that just selects an ordering of the instructions are as follows:

- For machines with significant issue limitations, e.g., nonpipelined functional units or multiple issue pairing rules, packing the limiting resource well is often preferable to obtaining a good schedule. A cycle model allows other instructions to "float" into the no-issue slots, while allowing the critical resource to be scheduled well.

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

- Modeling the machine in a forward direction captures the fact that processors are typically "greedy" and issue all the instructions that they can issue at a given time.

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

The forward pass does a topological sort of the instructions. The scheduler moves instructions that have either a direct dependence or an antidependence (e.g., register reuse) to a data structure called the issuing ring for future issue.

The scheduler represents the instructions available for issuing as a list of data structures known as heaps, which are priority queues. Each heap on the list contains instructions with a similar "signature." For example, a heap might contain all store instructions. When looking for the next instruction to issue, the scheduler examines the top instruction in each heap. Within each heap, instructions are typically ordered by their AET values, with occasional small biases for different instruction properties, such as loads that may have a variable execution time longer than the projected time.

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

Instructions that use multiple resources are represented in the heap ordering. For example, floating-point multiplies on the MIPS R3000 machine use both the multiplier and some of the same resources as additions. As a result, the heap that holds multiplies is always kept ahead of the heap that holds adds. This ordering scheme works well for both machines with a significant number of nonpipelined units, such as the MIPS processors, and machines that have largely pipelined functional units with only particular combinations of multiple issue allowed, like the DECchip 21064 processors.

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

6 Field Merging Example

Generating efficient code for the extraction and insertion of fields within records is particularly challenging on RISC architectures, like Alpha AXP, that provide only 32-bit (longword) or 64-bit (quadword) memory operations.

Often, a program will fetch or store several fields that are contained in the same longword. Without optimization, each fetch would load the longword from memory, and each store would both load and store the longword. However, 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 longword 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. Thus, the real task of optimizing field accesses is to identify IL operations that can be combined.

In the initial IL, a field fetch or store is represented as an IL operator. 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. The IL generated by the front end for two field extractions as shown in (a) of Figure 3 is expanded into the IL shown in (b) of Figure 3. With the loads exposed as fetches, data-flow analysis is now capable of finding the common subexpressions of 1$ and 3$.

Similarly, each field store expands into a fetch of the background longword, 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.

In some cases, a program operates on the field and thus eliminates the extract and insert operations. For example, the following example generates the machine code shown in Figure 4.

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

The unoptimized code would contain a load and an extract for each reference to n_kind or n_flags, plus an insert and a store for the latter two references. The optimizer has eliminated two of the three loads, two of the three extracts, both inserts, and one of the two stores.

7 Branch Optimization Examples

Branch instructions can hurt the performance of high-performance systems in several ways. In addition to consuming space and causing time to be expended while issuing the instruction, branches can disrupt the hardware pipeline. Also, branches can inhibit optimizations such as code scheduling. Therefore, the GEM compiler system uses several strategies to avoid branches in the IL and generated code or to eliminate some bad effects of branch instructions.

Some branches appear as part of a well-defined pattern that need not inhibit optimizations. GEM uses special operators for these cases. A simple example is the MAX function. For Alpha AXP systems, MAX can be implemented using the CMOVxx instructions, avoiding branch instructions entirely. For other architectures, the main benefit is that the branch does not appear in the IL. A more complicated example involves the so-called "flow-Boolean" operators. Consider the C code 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 programs to represent branching code 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 and then converts the operators to an unconditional form, allowing the generated code to be free of branches.

GEM performs other transformations on the IL to eliminate branches and thus enable further optimizations. For example, GEM transforms

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

into

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

Alpha AXP 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 differ from the order in the source program. For each basic block that ends with an unconditional branch, GEM places the target block 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;
}
```

To eliminate the unconditional branch when the loop iterates, GEM transforms the pretested loop into a posttested loop. Since the return statement is outside the loop, the generated code looks 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 and thus reduce the number of times the branch back must be executed. More important, GEM often allows operations from different iterations to be scheduled together. Unrolling by four transforms the above loop into a cleanup loop and the main loop into code that resembles

```
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 four fall-through branches and one taken branch, whereas the original code executed four fall-through branches and four taken branches.

Certain code patterns generate code that is likely not to be executed. For example, when the compiler believes that a 16-bit value in memory is apt to be naturally aligned, but may be unaligned, it generates the instructions shown in Figure 5 to load the value, given the address in r0. The code runs quickly for the aligned case, because the branch is correctly predicted to fall through, but gets the correct value for unaligned data, as well. A similar code pattern handles stores.

8 Compiler Engineering

Engineering compilers for a large combination of languages and platforms required a considerable number of innovations in the area of project engineering. In this section we describe some of the project methods and tools GEM uses.

Opal Intermediate Language Compiler

The task of a GEM compiler is to translate a program presented by the front end in the form of an IL graph and symbol table into machine code. In the early stages of GEM development, no front ends existed to generate IL graphs and symbol tables. To fill this requirement, a syntactic specification of the IL and symbol table was designed and an IL assembler called Opal was built to compile this syntax. Opal uses GEM components such as the shell and thus supports a robust set of features including listing generation, object files, include files, debug support, and language editor diagnostics.

Even with the availability of front ends, Opal remains a vital project tool: it allows GEM developers to exercise new features before front-end support is available; front-end developers use Opal to experiment with different IL alternatives; and the Opal syntax serves as the output format of the IL dumper.

Attribute and Operator Signature Tables

GEM tables give a complete description of all GEM data structures, including IL operators and symbol table nodes. The operator signature table contains the operator type, result type, number of operands, and legal operand types for IL operators. The attribute tables describe each component in a node including location, abstract GEM data type, legal values, node type for pointers, and special print formats. When a new attribute is added to the GEM specification, the attribute is described once in the tables and automatically the Opal compiler understands the syntax and semantics, the GEM dump utility is able to dump the attribute, and the GEM integrity checker is able to verify the structure.

Automatic KFOLD Builder

The GEM compiler needs to evaluate constant expressions at compile time, which is referred to as constant folding. GEM's intermediate language has many IL operators and data types. A constant folder is thus a complicated routine with many cases, and the compile-time and run-time results must be identical.

After writing our first, incomplete, handcrafted constant folder, we searched for a method to automate the process. No source language supported all the operators and data types of the GEM IL. The key insight was that there is one language in which IL programs can be written precisely and tersely: the GEM IL itself. Since GEM already embodies knowledge of the code sequences to evaluate every IL operator, no other encoding is needed.

The automatic KFOLD builder is a specialized GEM compiler that uses the standard GEM back end but has a front end that compiles only one program. The KFOLD builder scans the GEM operator signature table and constructs a procedure that contains a many-way conditional branch to select a case based on the IL operator specified in the argument list. Each case fetches operand values from the argument list, applies the operator, and returns the result. Since most GEM IL tuples operate on several data types, additional subcases may be based on the operator type or result type. We have already recovered the investment in developing the automatic KFOLD builder, and it significantly eases the task of retargeting GEM.

9 Conclusion

This paper describes the current GEM compiler system. However, a portable, optimizing compiler provides many opportunities that we have not yet exploited. Some enhancements planned for future versions are:

o Additional IL operators and data types, to support more languages

o Support for additional architecture and operating system combinations

o Dependence analysis, to enable some of the following enhancements

o Loop transformations, to improve the use of the memory hierarchy

o Software pipelining, to increase parallelism in vectorizable loops

o Better reordering of memory references during instruction scheduling

o The scheduling of instructions into different basic blocks

o The relaxing of the linear restriction on the lifetime model, i.e., allowing holes in register lifetimes

The GEM compiler system has met demanding technical and time-to-market goals. The system has been successfully retargeted and rehosted for the Alpha AXP and MIPS architectures and several operating environments. GEM supports a wide range of languages and provides high levels of optimization for each. The current version of GEM generates efficient code for Alpha AXP systems, and the implementation is robust and flexible enough to support future improvements.

10 Acknowledgments

The authors wish to acknowledge the contributions of the following individuals to the design and implementation of the GEM compilers: Ron Brender, Patsy Griffin, Lucy Hamnett, Brian Koblenz, Dennis Murphy, Bob

Peterson, Paul Winalski, Stan Whitlock (Fortran), Bevin Brett (Ada), and Farokh Morshed (C).

11 References

1. R. Sites, ed., Alpha Architecture Reference Manual (Burlington, MA: Digital Press, 1992).

2. K. Cooper, M. Hall, and L. Torczon, "The Perils of Interprocedural Knowledge," Rice COMP TR90-132 (1990).

3. K. Cooper, M. Hall, and L. Torczon, "Unexpected Side Effects of Inline Substitution: A Case Study," TOPLAS (March 1992): 22-32.

4. F. Chow, "Minimizing Register Usage Penalty at Procedure Calls," SIGPLAN '88 Conference on Programming Language Design and Implementation (June 1988): 85-94.

5. T. Lengauer and R. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," TOPLAS, vol. 1, no. 1 (July 1979): 121-141.

6. J. Reif, "Symbolic Interpretation in Almost Linear Time," Conference Records of the Fifth ACM Symposium on Principles of Programming Languages (1978): 76-83.

7. J. Reif and R. Tarjan, "Symbolic Program Analysis in Almost-Linear Time," SIAM Journal of Computing, vol. 11, no. 1 (February 1981): 81-93.

8. K. Harris and S. Hobbs, "VAX Fortran," Optimization in Compilers, ed., F. Allen, B. Rosen, and F. Zadek (New York, NY: ACM Press, forthcoming).

9. R. Cattell, "Formalization and Automatic Derivation of Code Generators," Ph.D. thesis, CMU-CS-78-115, Carnegie-Mellon University, April 1978.

10. A. Aho and S. Johnson, "Optimal Code Generation for Expression Trees," Journal of the ACM, vol. 23, no. 3 (July 1976): 488-501.

11. B. Leverett, "Register Allocation in Optimizing Compilers," Ph.D. thesis, CMU-CS-81-103, Carnegie-Mellon University, February 1981.

12. W. Wulf et al., The Design of an Optimizing Compiler (New York, NY: American Elsevier Publishing Co., 1975).

13. B. Leverett et al., "An Overview of the Production-Quality Compiler-Compiler Project," Computer, vol. 13, no. 8 (August 1980): 38-49.

14. R. Johnsson, "An Approach to Global Register Allocation," Ph.D. thesis, Carnegie-Mellon University, December 1975.

15. R. Sethi and J. Ullman, "The Generation of Optimal Code for Arithmetic

Expressions," Journal of the ACM, vol. 17, no. 4 (October, 1970): 715-728.

12 General Reference

P. Anklam et al., Engineering a Compiler (Bedford, MA: Digital Press, 1982).

13 Trademarks

The following are trademarks of Digital Equipment Corporation:

Alpha AXP, AXP, DECchip 21064, Digital, OpenVMS, ULTRIX, VAX, VAX Fortran, and VAX Pascal

The following are third-party trademarks:

MIPS is a trademark of MIPS Computer Systems, Inc.

OSF/1 is a registered trademark of Open Software Foundation, Inc.

Windows is a trademark of Microsoft Corporation.

14 Biographies

David S. Blickstein Principal software engineer David Blickstein has worked on optimizations for the GEM compiler system since the project began in 1985. During that time, he designed various optimization techniques, including induction variables, loop unrolling, code motions, common subexpressions, base binding, and binary shadowing. Prior to this, David worked on Digital's PDP-11 and VAX APL implementations and led the VAX-11 PL/I project. He received a B.A. (1980) in mathematics from Rutgers College, Rutgers University, and holds one patent on side effects analysis and another on induction variable analysis.

Peter W. Craig Peter Craig is a principal software engineer in the Software Development Technologies Group. He is currently responsible for the design and implementation of a dependence analyzer for use in future compiler products. Peter was a project leader for the VAX Code Generator used in the VAX C and VAX PL/I compilers, and prior to this, he developed CPU performance simulation software in the VAX Architecture Group. He received a B.S.E.E. (magna cum laude, 1982) from the University of Connecticut and joined Digital in 1983.

Caroline S. Davidson Since joining Digital in 1981, Caroline Davidson has contributed to several software projects, primarily related to code generation. Currently a principal software engineer, she is working on the GEM compiler generator project and is responsible for the areas of lifetimes, storage allocation, and entry-exit calls. Caroline is also a project leader for the Intel code generation effort. Her prior work involved the VAX FORTRAN for ULTRIX, VAX Code Generator, and FORTRAN IV software products. Caroline has a B.S.C.S. from the State University of New York at Stony Brook.

R. Neil Faiman, Jr. Neil Faiman is a consultant software engineer in the

Software Development Technologies Group. He was the primary architect of the GEM intermediate language and a project leader for the GEM compiler optimizer. Prior to this work, he led the BLISS compiler project. Neil came to Digital in 1983 from MDSI (now Schlumberger/Applicon). He has B.S. (1974) and M.S. (1975) degrees in computer science, both from Michigan

State University. Neil is a member of Tau Beta Pi and ACM, and an affiliate member of the IEEE Computer Society.

Kent D. Glossop Kent Glossop is a principal engineer in the Software Development Technologies Group. Since 1987 he has worked on the GEM compiler system, focusing on code generation and instruction-level transformations. Prior to this, Kent was the project leader for a release of the VAX PL/I compiler and contributed to version 1 of the VAX Performance and Coverage Analyzer. Kent joined Digital in 1983 after receiving a B.S. in computer science from the University of Michigan. He is a member of IEEE.

Richard B. Grove Senior consultant software engineer Rich Grove joined Digital in 1971 and is currently in the Software Development Technologies Group. He has led the GEM compiler project since the effort began in 1985, contributing to the code generation phases. Prior to this work, Rich was the project leader for the PDP-11 and VAX FORTRAN compilers, worked on VAX Ada V1, and was a member of the ANSI X3J3 FORTRAN Committee. He is presently a member of the design team for Alpha AXP calling standards and architecture. Rich has B.S. and M.S. degrees in mathematics from Carnegie-Mellon University.

Steven O. Hobbs A member of the Software Development Technologies Group, Steven Hobbs is working on the GEM compiler project. In prior contributions at Digital, he was the project leader for VAX Pascal, the lead designer for the global optimizer in VAX FORTRAN, and a member of the Alpha AXP architecture design team. Steve received his A.B. (1969) in mathematics at Dartmouth College and while there, helped develop the original BASIC time-sharing system. He has an M.A. (1972) in mathematics from the University of Michigan and has done additional graduate work in computer science at Carnegie-Mellon University.

William B. Noyce Senior consultant software engineer William Noyce is a member of the Software Development Technologies Group. He has developed several GEM compiler optimizations, including those that eliminate branches. In prior positions at Digital, Bill implemented support for new disks and processors on the RSTS/E project, led the development of VAX DBMS V1 and VAX Rdb/VMS V1, and designed and implemented automatic parallel processing for VAX FORTRAN/HPO. Bill received a B.A. (1976) in mathematics from Dartmouth College, where he implemented enhancements to the time-sharing system.