Debugging Optimized Code: Concepts and Implementation on DIGITAL Alpha Systems

Ronald F. Brender
Jeffrey E. Nelson
Mark E. Arsenault

Effective user debugging of optimized code has been a topic of theoretical and practical interest in the software development community for almost two decades, yet today the state of the art is still highly uneven. We present a brief survey of the literature and current practice that leads to the identification of three aspects of debugging optimized code that seem to be critical as well as tractable without extraordinary efforts. These aspects are (1) split lifetime support for variables whose allocation varies within a program combined with definition point reporting for currency determination, (2) stepping and setting breakpoints based on a semantic event characterization of program behavior, and (3) treatment of inlined routine calls in a manner that makes inlining largely transparent. We describe the realization of these capabilities as part of Compaq's GEM back-end compiler technology and the debugging component of the OpenVMS Alpha operating system.

Introduction

In software development, it is common practice to debug a program that has been compiled with little or no optimization applied. The generated code closely corresponds to the source and is readily described by a simple and straightforward debugging symbol table. A debugger can interpret and control execution of the code in a fashion close to the user's source-level view of the program.

Sometimes, however, developers find it necessary or desirable to debug an optimized version of the program. For instance, a bug—whether a compiler bug or incorrect source code—may only reveal itself when optimization is applied. In other cases, the resource constraints may not allow the unoptimized form to be used because the code is too big and/or too slow. Or, the developer may need to start analysis using the remains, such as a core file, of the failed program, whether or not this code has been optimized. Whatever the reason, debugging optimized code is harder than debugging unoptimized code—much harder—because optimization can greatly complicate the relationship between the source program and the generated code.

Zellweger[1] introduced the terms *expected behavior* and *truthful behavior* when referring to debugging optimized code. A debugger provides expected behavior if it provides the behavior a user would experience when debugging an unoptimized version of a program. Since achieving that behavior is often not possible, a secondary goal is to provide at least truthful behavior, that is, to never lie to or mislead a user. In our experience, even truthful behavior can be challenging to achieve, but it can be closely approached.

This paper describes three improvements made to Compaq's GEM back-end compiler system and to OpenVMS DEBUG, the debugging component of the OpenVMS Alpha operating system. These improvements address

1. Split lifetime variables and currency determination
2. Semantic events
3. Inlining

Before presenting the details of this work, we discuss the alternative approaches to debugging optimized code that we considered, the state of the art, and the operating strategies we adopted.

Alternative Approaches

Various approaches have been explored to improve the ability to debug optimized code. They include the following:

- Enhance debugger analysis
- Limit optimization
- Limit debugging to preplanned locations
- Dynamically deoptimize as needed
- Exploit an associated program database

We touch on these approaches in turn.

In probably the oldest theoretical analysis that supports debugging optimized code, Hennessy[2] studies whether the value displayed for a variable is current, that is, the expected value for that variable at a given point in the program. The value displayed might not be current because, for example, assignment of a later value has been moved forward or the relevant assignment has been delayed or omitted. Hennessy postulates that a flow graph description of a program is communicated to the debugger, which then solves certain flow analysis equations in response to debug commands to determine currency as needed. Copperman[3] takes a similar though much more general approach. Conversely, commercial implementations have favored more complete preprocessing of information in the compiler to enable simpler debugger mechanisms.[4–6]

If optimization is the "problem," then one approach to solving the problem is to limit optimization to only those kinds that are actually supported in an available debugger. Zurawski[7] develops the notion of a *recovery function* that matches each kind of optimization. As an optimization is applied during compilation, the compensating recovery function is also created and made available for later use by a debugger. If such a recovery function cannot be created, then the optimization is omitted. Unfortunately, code-motion-related optimizations generally lack recovery functions and so must be foregone. Taking this approach to the extreme converges with traditional practice, which is simply to disable all optimization and debug a completely unoptimized program.

If full debugger functionality need only be provided at some locations, then some debugger capabilities can be provided more easily. Zurawski[7] also employed this idea to make it easier to construct appropriate recovery functions. This approach builds on a language-dependent concept of *inspection points,* which generally must include all call sites and may correspond to most statement boundaries. His experience suggests, however, that even limiting inspection points to statement boundaries severely limits almost all kinds of optimization.

Hölzle et al.[8] describe techniques to dynamically deoptimize part of a program (replace optimized code with its unoptimized equivalent) during debugging to enable a debugger to perform requested actions. They make the technique more tractable, in part by delaying asynchronous events to well-defined *interruption points*, generally backward branches and calls. Optimization between interruption points is unrestricted. However, even this choice of interruption points severely limits most code motion and many other global optimizations.

Pollock and others[9,10] use a different kind of deoptimization, which might be called preplanned, incremental deoptimization. During a debugging session, any debugging requests that cannot be honored because of optimization effects are remembered so that a subsequent compilation can create an executable that can honor these requests. This scheme is supported by an incremental optimizer that uses a program database to provide rapid and smooth forward information flow to subsequent debugging sessions.

Feiler[11] uses a program database to achieve the benefits of interactive debugging while applying as much static compilation technology as possible. He describes techniques for maintaining consistency between the primary tree-based representation and a derivative compiled form of the program in the face of both debugging actions and program modifications on-the-fly. While he appears to demonstrate that more is possible than might be expected, substantial limitations still exist on debugging capability, optimization, or both.

A comprehensive introduction and overview to these and other approaches can be found in Copperman[3] and Adl-Tabatabi.[12] In addition, "An Annotated Bibliography on Debugging Optimized Code" is available separately on the *Digital Technical Journal* web site at http://www.digital.com/info/DTJ. This bibliography cites and summarizes the entire literature on debugging optimized code as best we know it.

State of the Art

When we began our work in early 1994, we assessed the level of support for debugging optimized code that was available with competitive compilers. Because we have not updated this assessment, it is not appropriate for us to report the results here in detail. We do however summarize the methodology used and the main results, which we believe remain generally valid.

We created a series of example programs that provide opportunities for optimization of a particular kind

or of related kinds, and which could lead a traditional debugger to deviate from expected behavior. We compiled and executed these programs under the control of each system's debugger and recorded how the system handled the various kinds of optimization. The range of observed behaviors was diverse.

At one extreme were compilers that automatically disable all optimization if a debugging symbol table is requested (or, equivalently for our purposes, give an error if both optimization and a debugging symbol table are requested). For these compilers, the whole exercise becomes moot; that is, attempting to debug optimized code is not allowed.

Some compiler/debugger combinations appeared to usefully support some of our test cases, although none handled all of them correctly. In particular, none seemed able to show a traceback of subroutine calls that compensated for inlining of routine calls and all seemed to produce a lot of jitter when stepping by line on systems where code is highly scheduled.

The worst example that we found allowed compilation using optimization but produced a debugging symbol table that did not reflect the results of that optimization. For example, local variables were described as allocated on the stack even though the generated code clearly used registers for these variables and never accessed any stack locations. At debug time, a request to examine such a variable resulted in the display of the irrelevant and never-accessed stack locations.

The bottom line from this analysis was very clear: the state of the art for support of debugging optimized code was generally quite poor. DIGITAL's debuggers, including OpenVMS DEBUG, were not unusual in this regard. The analysis did indicate some good examples, though. Both the CONVEX CXdb[4,5] and the HP 9000 DOC[6] systems provide many valuable capabilities.

Biases and Goals

Early in our work, we adopted the following strategies:

- Do not limit or compromise optimization in any way.
- Stay within the framework of the traditional edit-compile-link-debug cycle.
- Keep the burden of analysis within the compiler.

The prime directive for Compaq's GEM-based compilers is to achieve the highest possible performance from the Alpha architecture and chip technology. Any improvements in debugging such optimized code should be useful in the face of the best that a compiler has to offer. Conversely, if a programmer has the luxury of preparing a less optimized version for debugging purposes, there is little or no reason for that version to be anything other than completely unoptimized. There seems to be no particular benefit to creating a special intermediate level of combined debugger/optimization support.

Pragmatically, we did not have the time or staffing to develop a new optimization framework, for example, based on some kind of program database. Nor were we interested in intruding into those parts of the GEM compiler that performed optimization to create more complicated options and variations, which might be needed for dynamic deoptimization or recovery function creation.

Finally, it seemed sensible to perform most analysis activities within the compiler, where the most complete information about the program is already available. It is conceivable that passing additional information from the compiler to the debugger using the object file debugging symbol table might eventually tip the balance toward performing more analysis in the debugger proper. The available size data (presented later in this paper in Table 3) do not indicate this.

We identified three areas in which we felt enhanced capabilities would significantly improve support for debugging optimized code. These areas are

1. The handling of split lifetime variables and currency determination
2. The process of stepping though the program
3. The handling of procedure inlining

In the following sections we present the capabilities we developed in each of these areas together with insight into the implementation techniques employed.

First, we review the GEM and OpenVMS DEBUG framework in which we worked. The next three sections address the new capabilities in turn. The last major section explores the resource costs (compile-time size and performance, and object and image sizes) needed to realize these capabilities.

Starting Framework

Compaq's GEM compiler system and the OpenVMS DEBUG component of the OpenVMS operating system provide the framework for our work. A brief description of each follows.

GEM

The GEM compiler system[13] is the technology Compaq is using to build state-of-the-art compiler products for a variety of languages and hardware and software platforms. The GEM system supports a range of languages (C, C++, FORTRAN including HPF, Pascal, Ada, COBOL, BLISS, and others) and has been successfully retargeted and rehosted for the Alpha, MIPS, and Intel IA-32 architectures and for the

OpenVMS, DIGITAL UNIX, Windows NT, and Windows 95 operating systems.

The major components of a GEM compiler are the front end, the optimizer, the code generator, the final code stream optimizer, and the compiler shell.

- The front end performs lexical analysis and parsing of the source program. The primary outputs are intermediate language (IL) graphs and symbol tables. Front ends for all source languages translate to the same common representation.
- 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 final phase performs pattern-based peephole optimizations followed by instruction scheduling.
- The shell is a portable interface to the external environment in which the compiler is used. It provides common compiler functions such as listing generators, object file emitters, and command line processors in a form that allows the other components to remain independent of the operating system.

The bulk of the GEM implementation work described in this paper occurs at the boundary between the final phase and the object file output portion of the shell. A new debugging optimized code analysis phase examines the generated code stream representation of the program, together with the compiler symbol table, to extract the information necessary to pass on to a debugger through the debugging symbol table. Most of the implementation is readily adapted to different target architectures by means of the same instruction property tables that are used in the code generator and final optimizer.

OpenVMS DEBUG

The OpenVMS Alpha debugger, originally developed for the OpenVMS VAX system,[14] is a full-function, source-level, symbolic debugger. It supports symbolic debugging of programs written in BLISS, MACRO-32, MACRO-64, FORTRAN, Ada, C, C++, Pascal, PL/1, BASIC, and COBOL. The debugger allows the user to control the execution and to examine the state of a program. Users can

- Set breakpoints to stop at certain points in the program
- Step through the execution of the program a line at a time
- Display the source-level view of the program's execution using either a graphical user interface or a character-based user interface
- Examine user variables and hardware registers
- Display a stack traceback showing the current call stack
- Set watch points
- Perform many other functions[15]

Split Lifetime Variables and Currency Determination

Displaying (printing) the value of a program variable is one of the most basic services that a debugger can provide. For unoptimized code and traditional debuggers, the mechanisms for doing this are generally based on several assumptions.

1. A variable has a single allocation that remains fixed throughout its lifetime. For a local or a stack-allocated variable that means throughout the lifetime of the scope in which the variable is declared.
2. Definitions and uses of the values of user variables occur in the same order in the generated code as they do in the original program source.
3. The set of instructions that belong to a given scope (which may be a routine body) can be described by a single contiguous range of addresses.

The first and second assumptions are of interest in this discussion because many GEM optimizations make them inappropriate. Split lifetime optimization (discussed later in this section) leads to violation of the first assumption. Code motion optimization leads to violation of the second assumption and thereby creates the so-called currency problem. We treat both of these problems together, and we refer to them collectively as *split lifetime support.* Statement and instruction scheduling optimization leads to violation of the third assumption. This topic is addressed later, in the section Inlining.

Split Lifetime Variable Definition

A variable is said to have split lifetimes if the set of fetches and stores of the variable can be partitioned such that none of the values stored in one subset are ever fetched in another subset. When such a partition exists, the variable can be "split" into several independent "child" variables, each corresponding to a partition. As independent variables, the child variables can be allocated independently. The effect is that the original variable can be thought to reside in different locations at different points in time—sometimes in a register, sometimes in memory, and sometimes nowhere at all. Indeed, it is even possible for the different child variables to be active simultaneously.

Split Lifetime Example A simple example of a split lifetime variable can be seen in the following straight-line code fragment:

```
A = ...;       ! Define (assign value to) A
...
B = ...A...;   ! Use definition (value of) A
A = ...;       ! Define A again
...
C = ...A...;   ! Use latter definition A
```

In this example, the first value assigned to variable *A* is used later in the assignment to variable *B* and then never used again. A new value is assigned to *A* and used in the assignment to variable *C*.

Without changing the meaning of this fragment, we can rewrite the code as

```
A1 = ...;      ! Define A1
...
B = ...A1...;  ! Use A1
A2 = ...;      ! Define A2
...
C = ...A2...;  ! Use A2
```

where variables *A*1 and *A*2 are split child variables of *A*.

Because *A*1 and *A*2 are independent, the following is also an equivalent fragment:

```
A1 = ...;      ! Define A1
...
A2 = ...;      ! Define A2
B = ...A1...;  ! Use A1
...
C = ...A2...;  ! Use A2
```

Here, we see that the value of *A*2 is assigned while the value of *A*1 is still alive. That is, the split children of a single variable have overlapping lifetimes.

This example illustrates that split lifetime optimization is possible even in simple straight-line code. Moreover, other optimizations can create opportunities for split lifetime optimization that may not be apparent from casual examination of the original source. In particular, loop unrolling (in which the body of a loop is replicated several times in a row) can create loop bodies for which split lifetime optimization is feasible and desirable.

Variables of Interest Our implementation deals only with scalar variables and parameters. This includes Alpha's extended precision floating-point (128-bit X_Floating) variables as well as variables of any of the complex types (see Sites[16]). These latter variables are referred to as two-part variables because each requires two registers to hold its value.

Currency Definition

The value of a variable in an optimized program is current with respect to a given position in the source program if the variable holds the value that would be expected in an unoptimized version of the program. Several kinds of optimization can lead to noncurrent variables. Consider the currency example in Figure 1.

As shown in Figure 1, the optimizing compiler has chosen to change the order of operations so that line 4 is executed prior to line 3. Now suppose that execution has stopped at the instruction in line 3 of the unoptimized code, the line that assigns a value to variable *C*.

Given a request to display (print) the value of *A*, a traditional debugger will display whatever value happens to be contained in the location of *A*, which here, in the optimized code, happens to be the result of the second assignment to *A*. This displayed value of *A* is a correct value, but it is not the expected value that should be displayed at line 3. This scenario might easily mislead a user into a frustrating and fruitless attempt to determine how the assignment in line 1 is computing and assigning the wrong value. The problem occurs because the compiler has moved the second assignment so that it is early relative to line 3.

Another currency example can be seen in the fragment (taken from Copperman[3]) that appears in Figure 2. In this case, the optimizing compiler has chosen to omit the second assignment to variable *A* and to assign that value directly into the actual parameter location used for the call of routine FOO. Suppose that the debugger is stopped at the call of routine FOO. Given a request to display *A*, a traditional debugger is likely to display the result of the first assignment to *A*. Again, this value is an actual value of *A*, but it is not the expected value.

Alternatively, it is possible that prior to reaching the call, the optimizing compiler has decided to reuse the

Line	Unoptimized		Optimized
1	`A = ...;`	`! Define A`	`A = ...;`
2	`B = ...A...;`	`! Use A`	`B = ...A...;`
3	`C = ...;`	`! C does not depend on A`	`A = ...;`
4	`A = ...;`	`! Define A`	`C = ...;`
5	`D = ...A...;`	`! Use second definition of A`	`D = ...A...;`

Figure 1
Currency Example 1

```
Line  Unoptimized                                  Optimized
1     A = expression1;                            A = expression1;
2     B = ...A...;        ! Use 1st def. of A     B = ...A...;
3     A = expression2;
4     FOO(A);             ! Use 2nd def. of A     FOO(expression2);
```

Figure 2
Currency Example 2

location that originally held the first value of *A* for another purpose. In this case, no value of *A* is available to display at the call of routine FOO.

Finally, consider the example shown in Figure 3, which illustrates that the currency of a variable is not a property that is invariant over time. Suppose that execution is stopped at line 5, inside the loop. In this case, *A* is not current during the first time through the loop body because the actual value comes from line 3 (moved from inside the loop); it should come from line 1. On subsequent times through the loop, the value from line 3 is the expected value, and the value of *A* is current.

As discussed earlier, most approaches to currency determination involve making certain kinds of flow graph and compiler optimization information available to the debugger so that it can report when a displayed value is not current. However, we wanted to avoid adding major new kinds of analysis capability to DIGITAL's debuggers.

More fundamentally, as the degree of optimization increases, the notion of *current position* in the program itself becomes increasingly ambiguous. Even when the particular instruction at which execution is pending can be clearly and unequivocally related to a particular source location, this location is not automatically the best one to use for currency determination. Nevertheless, the source location (or set of locations) where a displayed value was assigned can be reliably reported without needing to establish the current position.

Accordingly, we use an approach different than those considered in the literature. We use a straightforward flow analysis formulation to determine what locations hold values of user variables at any given point in the program and combine this with the set of definition locations that provide those values. Because there may be more than one source location, the user is given the basic information to determine where in the source the value of a variable may have originated. Consequently, the user can determine whether the value displayed is appropriate for his or her purpose.

Compiler Processing

A compiler performs most split lifetime analysis on a routine-by-routine basis. A preliminary walk over the entire symbol table identifies the variable symbols that are of interest for further analysis. Then, for each routine, the compiler performs the following steps:

- Code cell prepass
- Flow graph construction
- Basic block processing
- Parameter processing
- Backward propagation
- Forward propagation
- Information promotion and cleanup

After the compiler completes this processing for all routines, a symbol table postwalk performs final cleanup tasks. The following contains a brief discussion of these steps.

In this summary, we highlight only the main characteristics of general interest. In particular, we assume that each location, such as a register, is independent of all other locations. This assumption is not appropriate to locations on the stack because variables of different sizes

```
Line  Unoptimized                                  Optimized
1     A = ...;                                    A = ...;
2     ...A...;                                    ...A...;
3                                                 A = ...;
4     while (...) {                               while (...) {
5     ...;                                        ...;
6     A = ...;          // A is loop invariant
7     }                                           }
```

Figure 3
Currency Example 3

may overlay each other. The complexity of dealing with overlapping allocations is beyond the scope of this paper.

Of special importance in this processing is the fact that each operand of every instruction includes a *base symbol* field that refers to the compiler's symbol table entry for the entity that is involved.

Symbol Table Prewalk The symbol table prewalk identifies the variables of interest for analysis. As discussed, we are interested in scalars corresponding to user variables (not compiler-created temporaries), including Alpha's extended precision floating-point (128-bit X_Floating) and complex values.

DIGITAL's FORTRAN implementations pass parameters using a by-reference mechanism with bind (rather than copy-in/copy-out) semantics. GEM treats the hidden reference value as a variable that is subject to split lifetime optimization. Since the reference variable must be available to effect operations on the logical parameter variable, it follows that both the abstract parameter and its reference value must be treated as interesting variables.

Code Cell Prepass The code cell prepass performs a single walk over all code cells to determine

- The maximum and minimum offsets in the stack frame that hold any interesting variables
- The highest numbered register that is actually referenced by the code
- Whether the stack frame uses a frame pointer that is separate from the stack pointer

The compiler uses these characteristics to preallocate various working storage areas.

Flow Graph Construction A flow graph is built, in which each basic block is a node of the graph.

Basic Block Processing Basic block processing performs a kind of symbolic execution of the instructions of each block, keeping track of the effect on machine state as execution progresses.

When an instruction operand writes to a location with a base symbol that indicates an interesting variable, the compiler updates the location description to indicate that the variable is now known to reside in that location—this begins a lifetime segment. The instruction that assigned the value is also recorded with the lifetime segment.

If there was previously a known variable in that location, that lifetime segment is ended (even if it was for the same variable). The beginning and ending instructions for that segment are then recorded with the variable in the symbol table.

When an instruction reads an operand with a base symbol that indicates an interesting variable, some more unusual processing applies.

If the variable being read is already known to occupy that location, then no further processing is required. This is the most common case.

If the location already contains some other known variable, then the variable being read is added to the set of variables for that location. This situation can arise when there is an assignment of one variable to another and the register allocator arranges to allocate them both to the same location. As a result, the assignment happens implicitly.

If the location does not contain a known variable but there is a write operation to that location earlier in the same block (a fact that is available from the location description), the prior write is retroactively treated as though it did write that variable at the earlier instruction. This situation can arise when the result of a function call is assigned to a variable and the register allocator arranges to allocate that variable in the register where the call returns its value. The code cell representation for the call contains nothing that indicates a write to the variable; all that is known is that the return value location is written as a result of the call. Only when a later code cell indicates that it is using the value of a known variable from that location can we infer more of what actually happened.

If the location does not contain a known variable and there is no write to that same location earlier in this same basic block, then the defining instruction cannot be immediately determined. A location description is created for the beginning of the basic block indicating that the given variable or set of variables must have been defined in some predecessor block. Of course, the contents known as a result of the read operation can also propagate forward toward the end of the block, just as for any other read or write operation.

Special care is needed to deal with a two-part variable. Such a variable does not become defined until both instructions that assign the value have been encountered. Similarly, any reuse of either of the two locations ends the lifetime segment of the variable as a whole.

At the end of basic block processing, location descriptions specify what is known about the contents of each location as a result of read and write operations that occurred in the block. This description indicates the set of variables that occupy the location, or that the location was last written by some value that is not the value of a user variable, or that the location does not change during execution of the block.

Parameter Processing The compiler models parameters as locations that are defined with the contents of a known variable at the entry point of a routine.

Backward Propagation Backward propagation iterates over the flow graph and uses the locations with known contents at the beginning of a block to work backward to predecessor blocks looking for instructions that write to that location. For each variable in each input location, any such prior write instruction is retroactively made to look like a definition of the variable. Note that this propagation is not a flow algorithm because no convergence criteria is involved; it is simply a kind of spanning walk.

Forward Propagation Forward propagation iterates over the flow graph and uses the locations with known contents at the end of each block to work forward to successor blocks to provide known contents at the beginning of other blocks. This is a classic "reaching definitions" flow algorithm, in which the input state of a location for a block is the intersection of the known contents from the predecessors.

In our case, the compiler also propagates definition points, which are the addresses of the instructions that begin the lifetime segments. For those variables that are known to occupy a location, the set of definitions is the union of all the definitions that flow into that location.

Information Promotion and Cleanup The final step of compiler processing is to combine information for adjacent blocks where possible. This action saves space in the debugging symbol table but does not affect the accuracy of the description. Descriptions for by-reference bind parameters are next merged with the descriptions for the associated reference variables. Finally, lifetime segment information not already associated with symbol table entries is copied back.

Object File Representation

The object file debugging symbol table representation for split lifetime variables is actually quite simple. Instead of a single address for a variable, there is a sequence of lifetime segment descriptions. Each lifetime segment consists of

- The range of addresses over which the child location applies
- The location (in a register, at a certain offset in the current stack frame, indirect through a register or stack location, etc.)
- The set of addresses that provide definitions for this lifetime segment

By convention, the last segment in the sequence can have the address range 0 to FFFFFFFF (hex). This address range is used for a static variable, for example in a FORTRAN `COMMON` block, that has a default allocation that applies whenever no active children exist.

Debugger Processing

Name resolution, that is, binding a textual name to the appropriate entry in the debug symbol table, is in no way affected by whether or not a variable has split lifetime segments. After the symbol table entry is found, any sequence of lifetime segments is searched for one that includes the current point of execution indicated by the program counter (PC). If found, the location of the value is taken from that segment. Otherwise, the value of the variable is not available.

Usage Example

To illustrate how a user sees the results of this processing, consider the small C program in Figure 4. Note that the numbers in the left column are listing line numbers.

When DOCT8 is compiled, linked, and executed under debugger control, the dialogue shown in Figure 5 appears. The figure also includes interpretive comments.

Known Limitations

The following limitations apply to the existing split lifetime support.

Multiple Active Split Children While the compiler analysis correctly determines multiple active split child variables and the debug symbol table correctly describes them, OpenVMS DEBUG does not currently support multiple active child variables. When searching a symbol's lifetime segments for one that includes the current PC, the first match is taken as the only match.

Two-part Variables Support for two-part variables (those occupying two registers) assumes that a complete definition will occur within a single basic block.

```
385  doct8 () {
386
387      int i, j, k;
388
389      i = 1;
390      j = 2;
391      k = 3;
392
393      if (foo(i)) {
394          j = 17;
395          }
396      else {
397          k = 18;
398          }
399
400      printf("%d, %d, %d\n", i, j, k);
401
402      }
```

Figure 4
C Example Routine DOCT8 (Source with Listing Line Numbers)

```
$ run doct8
        OpenVMS Alpha Debug64 Version T7.2-001
%I, language is C, module set to DOCT8
DBG> step/into
stepped to DOCT8\doct8\%LINE 391
   391:     k = 3;
DBG> examine i, j, k
%W, entity 'i' was not allocated in memory (was optimized away)
%W, entity 'j' does not have a value at the current PC
%W, entity 'k' does not have a value at the current PC
```

Note the difference in the message for variable *i* compared to the messages for variables *j* and *k*. We see that variable *i* was not allocated in memory (registers or otherwise), so there is no point in ever trying to examine its value again. Variables *j* and *k*, however, do not have a value "at the current PC." Somewhere later in the program they will have a value, but not here.

The dialogue continues as follows:

```
DBG> step 6
stepped to DOCT8\doct8\%LINE 391
   391:     k = 3;
DBG> step
stepped to DOCT8\doct8\%LINE 393
   393:     if (foo(i)) {
DBG> examine j, k
%W, entity 'j' does not have a value at the current PC
DOCT8\doct8\k:  3
    value defined at DOCT8\doct8\%LINE 391
```

Here we see that *j* is still undefined but *k* now has a value, namely 3, which was assigned at line 391. The source indicates that *j* was assigned a value at line 390, before the assignment to *k*, but *j*'s assignment has yet to occur.

Skipping ahead in the dialogue to the print statement at line 400, we see the following:

```
DBG> set break %line 400
DBG> go
break at DOCT8\doct8\%LINE 400
   400:     printf("%d, %d, %d\n", i, j, k);
DBG> examine j
DOCT8\doct8\j:  2
    value defined at DOCT8\doct8\%LINE 390
    value defined at DOCT8\doct8\%LINE 394
DBG> examine k
DOCT8\doct8\k:  18
    value defined at DOCT8\doct8\%LINE 397+4
    value defined at DOCT8\doct8\%LINE 391
```

This portion of the message shows that more than one definition location is given for both *j* and *k*. Which of each pair applies depends on which path was taken in the `if` statement. If a variable has an apparently inappropriate value, this mechanism provides a means to take a closer look at those places, and only those places, from which that value might have come.

Figure 5
Dialogue Resulting from Running DOCT8

That is, at the end of a basic block, if the second part of a definition is missing then the initial part is discarded and forgotten.

Consider the following FORTRAN fragment:

```
COMPLEX X, Y
...
X = ...
Y = X + (1.0, 0.0)
```

Suppose that the last use of variable *X* occurs in the assignment to variable *Y* so that *X* and *Y* can be and are allocated in the same location, in particular, the same register pair. In this case, the definition of *Y* requires only one instruction, which adds 1.0 to the real part of the location shared by *X* and *Y*. Because there is no second instruction to indicate completion of the definition, the definition will be lost by our implementation.

Semantic Stepping

A major problem with stepping by line though optimized code is that the apparent source program location "bounces" back and forth, with the same line often appearing again and again. In large part this bouncing is due to a compiler optimization called *code scheduling,* in which instructions that arise from the same source line are scheduled, that is, reordered and intermixed with other instructions, for better execution performance.

OpenVMS DEBUG, like most debuggers, interprets the `STEP/LINE` (step by line) command to mean that the program should execute until the line number changes. Line numbers change more frequently in scheduled code than in unoptimized code.

For example, in sample programs from the SPEC95 Benchmark Suite, the average number of instructions in sequence that share the same line number is typically between 2 and 3—and typically 50 to 70 percent of those sequences consist of just 1 instruction! In contrast, if only instruction-level scheduling is disabled, then the average number of instructions is between 4 and 6, with 20 to 30 percent consisting of one instruction. In a compilation with no optimization, there are 8 to 12 instructions in a sequence, with roughly 5 percent consisting of a single instruction.

A second problem with stepping by line through an optimized program is that, because of the behavior of revisiting the same line again and again, the user is never quite sure when the line has finished executing. It is unclear when an assignment actually occurs or a control flow decision is about to be made.

In unoptimized code, when a user requests a breakpoint on a certain line, the user expects execution to stop just before that line, hence before the line is carried out. In optimized code, however, there is no well-defined location that is "before the line is carried out," because the code for that line is typically scattered about, intermixed, and even combined with the code for various other lines. It is usually possible, however, to identify *the* instruction that actually carries out the effect of the line.

Semantic Event Concept

We introduce a new kind of stepping mode called semantic stepping to address these problems. Semantic stepping allows the program to execute up to, but not including, an instruction that causes a semantic effect. Instructions that cause semantic effects are instructions that

- Assign a value to a user variable
- Make a control flow decision
- Make a routine call

Not all such instructions are appropriate, however. We start with an initial set of candidate instructions and refine it. The following sections describe the heuristics that are currently in use.

Assignment The candidates for assignment events are the instructions that assign a value to a variable (or to one of its split children). The second instruction in an assignment to a two-part variable is excluded. Stopping between the two assignments is inadvisable because at that point the variable no longer has the complete old state and does not yet have the complete new state.

Branches There are two kinds of branch: unconditional and conditional. An unconditional branch may have a known destination or an unknown destination. Unconditional branches with known destinations most often arise as part of some larger semantic construct such as an if-then-else or a loop. For example, code for an if-then-else construct generally has an implicit join that occurs at the end of the statement. The join takes the form of a jump from the end of one alternative to the location just past the last instruction of the other (which has no explicit jump and falls through into the next statement). This jump turns the inherently symmetric join at the source level into an asymmetric construction at the code stream level.

Unconditional jumps almost never define interesting semantic events—some related instruction usually provides a more useful event point, such as the termination test in the case of a loop. One exception is a simple goto statement, but these are very often optimized away in any case. Consequently, unconditional branches with known destinations are not treated as semantic events.

Unconditional branches with unknown destinations are really conditional branches: they arise from constructs such as a C `switch` statement implemented as a table dispatch or a FORTRAN assigned `GO TO` statement. These branches definitely are interesting points at which to allow user interaction before the new direction is taken. Thus, the compiler retains unconditional branches as semantic events.

Similarly, in general, conditional branches to known destinations are important semantic event points. Often more than one branch instruction is generated for a single high-level source construct, for example, a decision tree of tests and branches used to implement a small C `switch` statement. In this case, only the first in the execution sequence is used as the semantic event point.

Calls Most calls are visible to a user and constitute semantically interesting events. However, calls to some run-time library routines are usually not interest-

ing because these calls are perceived to be merely software implementations of primitive operations, such as integer division in the case of the Alpha architecture. GEM internally marks calls to all its own run-time support routines as not semantically interesting. Compiler front ends accomplish this where appropriate for their own set of run-time support routines by setting a flag on the associated entry symbol node.

Compiler Processing

In most cases, the compiler can identify semantic event locations by simple predicates on each instruction. The exceptions are

- The second of the two instructions that assign values to a two-part variable is identified during split lifetime analysis.
- Conditional branches that are part of a larger construct are identified during a simple pass over the flow graph.

Object Module Representation

The object module debugging semantic event representation contains a sequence of address and event kind pairs, in ascending address order.

Debugger Processing

Semantic stepping in the debugger involves a new algorithm for determining the range of instructions to execute. This algorithm is built on a debugger primitive mechanism that supports full-speed execution of user instructions within a given range of addresses but traps any transfer out of that range, whether by reaching the end or by executing any kind of branch or call instruction.

Semantic stepping works as follows. Starting with the current program counter address, OpenVMS DEBUG finds the next higher address that is a semantic event point; this is the target event point. OpenVMS DEBUG executes instructions in the address range that starts at the address of the current instruction and ends at the instruction that precedes the target event point. The range execution terminates in the following two cases:

1. If the next instruction to execute is the target event point, then execution reached the end of target range and the step operation is complete.
2. If the next instruction to execute is not the target event point, then the next address becomes the current address and the process repeats (silently).

Note that, unlike the algorithm that determines the range for stepping by line, the new algorithm does not require an explicit test for the kind of instruction, in particular, to test if it is a kind of branch. The compiler already marks branches with the semantic event attribute, if appropriate. Also unlike the traditional stepping-by-line algorithm, the new algorithm does not consider the source line number.

Visible Effect

With semantic stepping, a user's perception of forward progress through the code is no longer dominated by the side effects of code scheduling, that is, stopping every few instructions regardless of what is happening. Rather, this perception is much more closely related to the actual semantic behavior, that is, stopping every statement or so, independent of how many instructions from disparate statements may have executed.

Note that jumping forward and backward in the source may still occur, for example, when code motions have changed the order in which semantic actions are performed. Nothing about semantic event handling attempts to hide such reordering.

Inlining

Procedure call inlining can be confusing when using a traditional debugger. For example, if routine INNER is inlined into routine CALLER and the current point of execution is within INNER, should the debugger report the current source location as at a location in the caller routine or in the called routine? Neither is completely satisfactory by itself. If the current line is reported as at the location within INNER, then that information will appear to conflict with information from a call stack traceback, which would not show routine INNER. If the current line is reported as though in CALLER, then relevant location information from the callee will be obscured or suppressed. Worse yet, in the case of nested inlining, potentially crucial information about the intermediate call path may not be available in any form.

The problem of dealing with inlining was solved long ago by Zellweger[1]—at least the topic has not been treated again since. Zellweger's approach adds additional information to an otherwise traditional table that maps from instruction addresses to the corresponding source line numbers. Our approach is different: it includes additional information in the scope description of the debugging symbol table.

A key underpinning for inline support is the ability to accurately describe scopes that consist of multiple discontiguous ranges of instruction addresses, rather than the traditional single range. This capability is quite independent of inlining as such. However, because code from an inlined routine is freely scheduled with other code from the calling context, dealing accurately with the resulting disjoint scopes is an essential building block for effective support.

Goals for Debugger Support

Our overall goal is to support debugging of inlined code with expected behavior, that is, as though the inlining has not occurred. More specifically, we seek to provide the ability to

- Report the source location corresponding to the current position in the code
- Display parameters and local variables of an inlined routine
- Show a traceback that includes call frames corresponding to inlined routines
- Set a breakpoint at a given routine entry
- Set a breakpoint at a given line number (from within an inlined routine)
- Call an inlined routine

We have achieved these goals to a substantial extent.

GEM Locators

Before describing the mechanisms for inlining, we introduce the GEM notion of a *locator.* A locator describes a place in the source text. The simplest kinds of locator describe a point in the source, including the name of the file, the line within that file, and the column within that line; they even describe the point at which that file was included by another file (as for a C or C++ `#include` directive), if applicable.

A crucial characteristic of locators is that they are all of a uniform fixed size that is no larger than an integer or pointer. (How this is achieved is beyond the scope of this paper.) In particular, locators are small enough that every tuple node in the intermediate language (IL) and every code cell in the generated code stream contains one. Moreover, GEM as a whole is quite meticulous about maintaining and propagating high-quality locator information throughout its optimization and code generation.

An additional kind of locator was introduced for inlining support. This *inline locator* encodes a pair that consists of a locator (which may also be an inline locator) and the address of an associated scope node in the GEM symbol table.

Compiler Processing

Debugging optimized code support for inlining generally builds on and is a minor enhancement of the GEM inlining mechanism. Inlining occurs during an early part of the GEM optimizer phase.

Inlining is implemented in GEM as follows:

- Within the scope that contains the call site, an *inline scope* block is introduced. This scope represents the result of the inlining operation. It is populated with local variable declarations that correspond one-to-one with the formal parameters of the inlined routine.
- The actual arguments of the call are transformed into assignments that initialize the values of the surrogate parameter variables.
- The inline scope is also made to contain a *body scope,* which is a copy of the body of the inlined routine, including a copy of its local variables.
- The original call is replaced with a jump to a copy of the IL for the body of the routine, in which references to declarations or parameters of the routine are replaced with references to their corresponding copied declarations. In addition, returns from the routine are replaced with jumps back to the tuple following the original call.
- Similar "boundary adjustments" are made to deal with function results, output parameters, choice of entry point (when there is more than one, as might occur for FORTRAN alternate entry statements), etc. (The bookkeeping is a bit intricate, but it is conceptually straightforward.)

The calling routine, which now incorporates a copy of the inlined routine, is then further processed as a normal (though larger) routine.

Inlining Annotations for Debugging The main changes introduced for debugging optimized code support are as follows.

- The newly created inline scope block is annotated with additional information, namely,
  - A pointer to the routine declaration being inlined.
  - The locator from the call that is replaced. In a simple call with no arguments, there may be nothing left in the IL from the original call after inlining is completed; this locator captures the original call location for possible later use, for example, as a supplement to the information that maps instruction addresses to source line numbers.
- As the code list of the original inlined routine is copied, each locator from the original is replaced by a new inline locator that records
  - The original locator.
  - The newly created inline scope into which it is being copied.

As a result of these steps, every inlined instruction can be related back to the scope into which it was inlined and hence to the routine from which it was inlined, regardless of how it may be modified or moved as a result of subsequent optimization.

Note that these additional steps are an exception to the general assertion that debugging optimized code support occurs after code generation and just prior to object code emission. These steps in no way influence the generated code—only the debugging symbol table that is output.

Prologue and Epilogue Sets The prologue of a routine generally consists of those instructions at the beginning of the routine that establish the routine stack frame (for example, allocate stack and save the return address and other preserved registers) and that must be executed before a debugger can usefully interpret the state of the routine. For this reason, setting a breakpoint at the beginning of a routine is usually (transparently) implemented by setting a breakpoint after the prologue of that routine is completed.

Conversely, the epilogue of a routine consists of those instructions at the end of a routine that tear down the stack frame, reestablish the caller's context, and make the return value, if any, available to the caller. For this reason, stopping at the end of a routine is usually (transparently) implemented by setting a breakpoint before the epilogue of that routine begins.

One benefit of inlining is that most prologue and epilogue code is avoided; however, there may still be some scope management associated with scope entry and exit. Also, some programming language–related environment management associated with the scope may exist and should be treated in a manner analogous to traditional prologue and epilogue code. The problem is how to identify it, because most of the traditional compiler code generation hooks do not apply.

The model we chose takes advantage of the semantic event information that we describe in the section Semantic Stepping. In particular, we define the first semantic event that can be executed within the inlined routine to be the end of the prologue. For reasons discussed later, we define the last instruction (not the last semantic event) of the inlined code as the beginning of the epilogue. As a result of unrelated optimization effects, each of these may turn out to be a set of instructions. Determination of inline prologue and epilogue sets occurs after split lifetime and semantic event determination is completed so that the results of those analyses can be used.

To determine the set of prologue instructions, for each inline instance, GEM starts with every possible entry block and scans forward through the flow graph looking for the first semantic event instruction that can be reached from that entry. The set of such instructions constitutes the prologue set for that instance of the inlined routine.

This is a spanning walk forward from the routine entry (or entries) that stops either when a block is found to contain an instruction from the given inline instance or when the block has already been encountered (each block is considered at most once). Note that there may be execution paths that include one or more instructions from an inlining, none of which is a semantic event instruction.

The set of epilogue instructions is determined using an inverse of the prologue algorithm. The process starts with each possible exit block and scans backward through the flow graph looking for the last instruction (that is, the instruction closest to the routine exit) of an inline instance that can reach an exit.

Note that prologue and epilogue sets are not strictly symmetric: prologue sets consist of only instructions that are also semantic events, whereas epilogue sets include instructions that may or may not be semantic events.

Object Module Representation

To describe any inlining that may have occurred during compilation, we include three new kinds of information in the debugging symbol table.

If the instructions contained in a scope do not form a single contiguous range, then the description of the scope is augmented with a discontiguous range description. This description consists of a sequence of ranges. (The scope itself indicates the traditional approximate range description to provide backward compatibility with older versions of OpenVMS DEBUG). This augmented description applies to all scopes, whether or not they are the result of inlining.

For a scope that results from inlining a call, the description of the scope is augmented with a record that refers to the routine that was inlined as well as the line number of the call. Each scope also contains two entries that consist of the sequence of prologue and epilogue addresses, respectively.

Backward compatibility is fully maintained. An older version of OpenVMS DEBUG that does not recognize the new kinds of information will simply ignore it.

Debugger Processing

As the debugger reads the debugging symbol table of a module, it constructs a list of the inlined instances for each routine. This process makes it possible to find all instances of a given routine. Note, however, that if every call of the routine is expanded inline and the routine cannot otherwise be called from outside that module, then GEM does not create a noninlined (closed-form) version of the routine.

Report Source Location It is a simple process to report the source location that corresponds to the current code address. When stopped inside the code resulting from an inlined routine, the program counter maps directly to a source line within the inlined routine.

Display Parameters and Local Variables As is the case for a noninlined routine, the scope description for an inlined routine contains copies of the parameters and the local variables. No special processing is required to perform name binding for such entities.

Include Inlined Calls in Traceback The debugger presents inlined routines as if they are real routine calls. A stack frame whose current code address corresponds

to an inlined routine instance is described with two or more virtual stack frames: one or more for the inlined instance(s) and one for the ultimate caller. (An example is shown later in Figure 7.)

Set Breakpoints at Inlined Routine Instances The strategy for setting breakpoints at inlined routines is based on a generalization of processing that previously existed for C++ member functions. Compilation of C++ modules can result in code for a given member function being compiled every time the class or template definition that contains the member function is compiled. We refer to all these compilations as *clones.* (It is not necessary to distinguish which of them is the original.) In our generalization, an inlined routine call instance is treated like a clone. To set a breakpoint at a routine, the debugger sets breakpoints at all the end-of-prologue addresses of every clone of the given routine in all the currently active modules.

Set Breakpoints at Inlined Line Number Instances The strategy for setting breakpoints on line numbers shares some features of setting breakpoints on routines, with additional complications. Compiler-reported line numbers on OpenVMS systems are unique across all the files included in a compilation. It follows that the same file included in more than one compilation may have different associated line numbers.

To set a breakpoint at a particular line number, that line number needs to be first normalized relative to the containing file. This normalized line number value is then compared to normalized line numbers for that same file that are included in other compilations. (If different versions of the same named file occur in different compilations, the versions are treated as unrelated.) The original line number is converted into the set of address ranges that correspond to it in all modules, taking into account inlining and cloning.

Call a Routine That Is Inlined If the compiler creates a closed-form version of a routine, then the debugger can call that routine independent of whether there may also be inlined instances of the routine. If no such version of the routine exists, then the debugger cannot call the routine.

Usage Example

Inlining support has many aspects, but we will illustrate only one—a call traceback that includes inlined calls. Consider the sample program shown in Figure 6. This program has four routines: three are combined in a single file (enabling the GEM FORTRAN compiler to perform inline optimizations), and the last is in a separate file. To help correlate the lines of code in these two files with those in Figure 7, we added line numbers to the left of the code. Note that these numbers are not part of the program.

```
Line +++++ File DOCFJ-INLINE-2.FOR
---
1    C     Main routine
2    C
3          INTEGER A, C
4          TYPE *, A(3, C(0))
5          END
6    C
7          FUNCTION A(I, L)
8          INTEGER A, B
9          A = B(5, I) + 2*L
10         RETURN
11         END
12   C
13         FUNCTION B(J, K)
14         INTEGER B, C
15         B = C(9) + J + K
16         END

     +++++ File DOCFJ-INLINE-2A.FOR
1    C
2          FUNCTION C(I)
3          INTEGER C
4          C = 2*I
5          RETURN
6          END
```

Figure 6
Program to Illustrate Inlining Support

If we compile, link, and run this program using the OpenVMS DEBUG option, we can step to a place in routine B that is just before the call to routine C and then request a traceback of the call stack. This dialogue is shown in Figure 7.

Figure 7 shows that pseudo stack frames are reported for routines A and B, even though the call of routine B has been inlined into routine A and the call of routine A has been inlined into the main program. The main difference from a real stack frame is the extra line that reports that the "above routine is inlined."

Limitations

In a real stack frame, it is possible to examine (and even deposit into) the real machine registers, rather than examine the variables that happen to be allocated in machine registers. In an inlined stack frame, this operation is not well defined and consequently not supported. In a noninlined stack frame, these operations are still allowed.

An attractive feature that would round out the expected behavior of inlined routine calls would be to support stepping into or over the inlined call in the same way that is possible for noninlined calls. This feature is not currently supported—execution always steps into the call.

```
GEMEVN$ run DOCFJ-INLINE-2
         OpenVMS Alpha Debug64 Version T7.2-001
%I, Language: FORTRAN, Module: DOCFJ-INLINE-2$MAIN
...
DBG> step/semantic
stepped to DOCFJ-INLINE-2$MAIN\A\B\%LINE 15+8
    15:       B = C(9) + J + K
DBG> show calls
 module name  routine name line      rel PC           abs PC
*DOCFJ-INLINE-2$MAIN
              B            15  000000000000001C 000000000002006C
----- above routine is inlined
*DOCFJ-INLINE-2$MAIN
              A             9  0000000000000004 0000000000020054
----- above routine is inlined
*DOCFJ-INLINE-2$MAIN
              DOCFJ-INLINE-2$MAIN
                            4  0000000000000038 0000000000020038
                               0000000000000000 FFFFFFFF8590716C
```

Figure 7
OpenVMS DEBUG Dialogue to Illustrate Inlining Support

Performance and Resource Usage

We gathered a number of statistics to determine typical resource requirements for using the enhanced debugging optimized code capability compared to the traditional practice of debugging unoptimized code. A short summary of the findings follows.

- All metrics tend to show wide variance from program to program, especially small ones.
- Generating traditional debugging symbol information increases the size of object modules typically by 50 to 100 percent on the OpenVMS system. Executable image sizes show similar but smaller size increases.
- Generating enhanced symbol table information adds about 2 to 5 percent to the typical compilation time, although higher percentages have been seen for unusually large programs.
- Generating enhanced symbol table information uses significant memory during compilation but does not affect the peak memory requirement of a compilation.
- Generating enhanced symbol table information further increases the size of the symbol table information compared to that for an unoptimized compilation. On the OpenVMS system, this adds 100 to 200 percent to the debugging symbol table of object modules and perhaps 50 to 100 percent for executable images.
- Compiling with full optimization reduces the resulting image size. Total net image size increases typically by 50 to 80 percent.

A more detailed presentation of findings follows. Tables 1 through 3 present data collected using production OpenVMS Alpha native compilers built in December 1996. In developing these results, we used five combinations of compilation options as follows:

S1: no optimization (noopt), no debugging information (nodebug, nodbgopt)

S2: no optimization (noopt), normal debugging information (debug, nodbgopt)

S4: full (default) optimization (opt), no debugging information (nodebug, nodbgopt)

S5: full optimization (opt), normal debugging information only (debug, nodbgopt)

S8: full optimization (opt), enhanced debugging information (debug, dbgopt)

Note that the option combination numbering system is historical; we retained the system to help keep data logs consistent over time.

Compile-time Speed

The incremental compile-time cost of creating enhanced symbol table information is presented in Table 1 for a sampling of BLISS, C, and FORTRAN modules. The data in this table can be summarized as follows:

- Traditional debugging (column 1) increases the total compilation time by about 1 percent.
- Enhanced debugging (column 2) increases the compilation time by about 4 percent. The largest component of that time, approximately 3 percent, is attributed to the flow analysis involved in handling split lifetime variables (column 3).
- Debugging tends to increase as a percentage of time in larger modules, which suggests that processing time is slightly nonlinear in program size; however, this increase does not seem to be excessive even in very large modules.

Compile-time Space

The compile-time memory usage during the creation of enhanced symbol information is presented in Table 2.

Table 1
Percent of Compilation Time Used to Create/Output Debugging Information

Module	S2 (noopt, debug, nodbgopt)	S8 (opt, debug, dbgopt)	(Split Lifetime Analysis Only)
		BLISS CODE	
GEM_AN	0.3%	1.1%	0.7%
GEM_DB	0.9	1.8	1.3
GEM_DF	0.8	5.2	4.4
GEM_FB	0.7	3.5	2.7
GEM_IL_PEEP	0.6	14.4	13.9
		C CODE	
C_METRIC	1.5	5.2	4.1
GRAM	0.5	2.9	2.2
INTERP	1.2	4.5	3.2
		FORTRAN CODE	
MATRIX300X	nm	nm	nm
NAGL	1.4	13.0	11.9
SPICE_V07	3.0	6.4	4.7
WAVEX	2.5	6.3	4.8
	----	----	----
Average	1.2%	4.3%	3.2%
Typical range	(0.5%–1.5%)	(3.0%–7.0%)	(2.0%–5.0%)

Note: "nm" represents "not meaningful," that is, too small to be accurately measured.

Table 2
Key Dynamic Memory Zone Sizes during BLISS GEM Compilations

File	Peak Total	SYMBOL ZONE	EIL ZONE	CODE ZONE	OM ZONE	% Peak	% Larg	% EIL
				BLISS CODE				
GEM_AN	2,507	130	85	184	15	6%	8%	18%
GEM_DF	11,305	836	1,672	2,056	1,180	10	57	71
GEM_FB	4,694	316	522	457	304	6	58	58
GEM_IL_PEEP	40,419	1,606	17,666	4,411	14,143	34	80	80
				C CODE				
C_METRIC	7,381	1,115	494	2,563	167	2	6	34
GRAM	3,031	82	815	211	267	9	33	33
INTERP	3,563	354	308	688	131	4	20	43
				FORTRAN CODE				
MATRIX300X	934	143	227	101	58	6	26	26
NAGL	6,267	1,520	1,791	1,742	68	11	38	38
SPICE_V07	6,234	1,051	3,256	885	459	7	14	14
WAVEX	12,812	4,676	3,119	3,482	68	5	14	22
						---	---	---
Average						9%	32%	40%

Note: All numbers to the left of the vertical bar are thousands of bytes, not multiples of 1,024.

Column Key: Column	Description
Peak Total	The peak dynamic memory allocated in all zones during the compilation
SYMBOL ZONE	The zone that holds the GEM symbol table
EIL ZONE	The zone that holds the largest EIL ZONE (used for the expanded intermediate representation)
CODE ZONE	The zone that holds the GEM generated code list
OM ZONE	The zone that holds split lifetime and other working data
%Peak	The OM ZONE size as a percentage of the Peak Total size
%Larg	The OM ZONE size as a percentage of the largest single zone in the compilation
%EIL	The OM ZONE size as a percentage of the EIL ZONE size

The following is a summary of the data, where OM ZONE refers to the temporary working virtual memory zone used for split lifetime analysis:

- The OM ZONE size averages about 10 percent of the peak compilation size.
- The OM ZONE size is one-quarter to one-half of the EIL ZONE size. (The latter is well known for typically being the largest zone in a GEM compilation.)
- Since the OM ZONE is created and destroyed after all EIL ZONEs are destroyed, the OM ZONE does not contribute to establishing the peak total size.

Object Module Size

The increased size of enhanced symbol table information for both object files and executable image files is shown in Table 3.

In Table 3, the application or group of modules is identified in the first column. The columns labeled S1, S2, etc. give the resulting size for the combination of compilation options described earlier. Object module and executable image data is presented in successive rows.

Three ratios of particular interest are computed.

S2/S1: This ratio shows the object or image size with traditional debugging information compared to a base compilation without any debugging information. This ratio indicates the additional cost, in terms of increased object and image file size, associated with doing traditional symbolic debugging.

(S8-S5)/(S2-S1): This ratio shows the increase in debugging symbol table size (exclusive of base object, image text, etc.) due to the inclusion of enhanced information compared to the traditional symbol table size.

S8/S2: This ratio shows the object or image size with enhanced debugging information with optimization compared to the traditional debugging size without optimization.

The last ratio, S8/S2, is especially interesting because it combines two effects: (1) the reduction in size as a result of compiler optimization, and (2) the increase in size because the larger debugging symbol table needed to describe the result of the optimization. The resulting net increase is reasonably modest.

Summary and Conclusions

There exists a small but significant literature regarding the debugging of optimized code, yet very few debuggers take advantage of what is known. In this paper we describe the new capabilities for debugging optimized code that are now supported in the GEM compiler system and the OpenVMS DEBUG component of the OpenVMS Alpha operating system. These capabilities deal with split lifetime variables and currency determination, semantic stepping, and procedure inlining. For each case, we describe the problem addressed and then present an overview of GEM compiler and OpenVMS DEBUG processing and the object module representation that mediates between them. All but the inlining support are included in OpenVMS DEBUG V7.0 and in GEM-based compilers for Alpha systems that have been shipping since 1996. The inlining support is

Table 3
Object/Executable (.OBJ/.EXE) File Sizes (in Number of Blocks) for Various OpenVMS Components

File	S1 noopt nodebug nodbgopt	S2 noopt debug nodbgopt	S2/S1 Ratio	S4 opt nodebug nodbdopt	S5 opt debug nodbgopt	S8 opt debug dbgopt	(S8-S5)/(S2-S1) Ratio	S8/S2 Ratio
				BLISS CODE				
GEM_*.OBJ	31,477	51,069	1.62	27,483	47,031	68,728	1.11	1.35
GEM_*.EXE	12,160	29,543	2.43	10,373	27,755	32,288	0.26	1.09
				C CODE				
C_METRIC.OBJ	436	653	1.50	478	733	1,680	4.36	2.57
C_METRIC.EXE	250	348	1.39	250	385	581	2.00	1.67
GRAM.OBJ	102	120	1.19	100	117	224	5.94	1.87
GRAM.EXE	60	70	1.17	58	69	91	2.20	1.30
INTERP.OBJ	140	207	1.48	134	205	450	3.66	2.17
INTERP.EXE	80	113	1.41	75	113	167	1.64	1.47
				FORTRAN CODE				
MATRIX300X.OBJ	20	34	1.70	16	29	71	3.00	2.08
MATRIX300X.EXE	19	29	1.53	15	25	34	0.90	1.17
NAGL.OBJ	42	63	1.51	288	509	1,178	3.11	1.84
NAGL.EXE	289	388	1.34	187	333	469	1.37	1.21
SPICE.OBJ	1,652	3,117	1.89	1,073	2,571	4,916	1.60	1.58
SPICE.EXE	1,031	1,660	1.61	549	1,318	1,803	0.77	1.09
WAVEX.OBJ	555	1,639	2.95	393	1,556	2,949	1.29	1.80
WAVEX.EXE	634	1,190	1.88	490	1,167	1,437	0.49	1.21

currently in field test. Work is under way to provide similar capabilities in the ladebug debugger[17,18] component of the DIGITAL UNIX operating system.

There are and will always be more opportunities and new challenges to improve the ability to debug optimized code. Perhaps the biggest problem of all is to figure out where best to focus future attention. It is easy to see how the capabilities described in this paper provide major benefits. We find it much harder to see what capability could provide the next major increment in debugging effectiveness when working with optimized code.

References

1. P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Ph.D. Dissertation, University of California, Xerox PARC CSL-84-5 (May 1984).
2. J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3 (July 1982): 323–344.
3. M. Copperman, "Debugging Optimized Code Without Being Misled," Ph.D. Dissertation, University of California at Santa Cruz, UCSC Technical Report UCSC-CRL-93-21 (June 11, 1993).
4. G. Brooks, G. Hansen, and S. Simmons, "A New Approach to Debugging Optimized Code," *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices,* vol. 27, no. 7 (July 1992): 1–11.
5. Convex Computer Corporation, *CONVEX CXdb Concepts* (Richardson, Tex.: Convex Press, Order No. DSW–471, May 1991).
6. D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* Atlanta, Ga. (June 22–24, 1988): 125–134.
7. L. Zurawski, "Source-Level Debugging of Globally Optimized Code with Expected Behavior," Ph.D. Dissertation, University of Illinois at Urbana-Champaign (1989).
8. U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* San Francisco, Calif. (June 17–19, 1992) and SIGPLAN Notices, vol. 27, no. 7 (July 1992): 32–43.
9. L. Pollock and M. Soffa, "High-level Debugging with the Aid of an Incremental Optimizer," *Proceedings of the 21st Hawaii International Conference on System Sciences* (January 1988): 524–532.
10. L. Pollock, M. Bivens, and M. Soffa, "Debugging Optimized Code via Tailoring," *International Symposium on Software Testing and Analysis* (August 1994).
11. P. Feiler, "A Language-Oriented Interactive Programming Environment Based on Compilation Technology," Ph.D. Dissertation, Carnegie-Mellon University, CMU-CS-82-117 (May 1982).
12. A. Adl-Tabatabi, "Source-Level Debugging of Globally Optimized Code," Ph.D. Dissertation, Carnegie Mellon University, CMU-CS-96-133 (June 1996).
13. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 121–136.
14. B. Beander, "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger," *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, ACM SIGPLAN Notices,* vol. 18, no. 8 (August 1983): 173–179.
15. *OpenVMS Debugger Manual,* Order No. AA-QSBJB-TE (Maynard, Mass.: Digital Equipment Corporation, November 1996).
16. R. Sites, ed., *Alpha Architecture Reference Manual,* 3d ed. (Woburn, Mass.: Digital Press, 1998).
17. T. Bingham, N. Hobbs, and D. Husson, "Experiences Developing and Using an Object-Oriented Library for Program Manipulation," *OOPSLA Conference Proceedings, ACM SIGPLAN Notices,* vol. 12, no. 10 (October 1993): 83–89.
18. *Digital UNIX Ladebug Debugger Manual,* Order No. AA-PZ7EE-T1TE (Maynard, Mass.: Digital Equipment Corporation, March 1996).

Biographies

Ronald F. Brender

Ronald F. Brender is a senior consultant software engineer in Compaq's Core Technology Group, where he is working on both the GEM compiler and the UNIX ladebug projects. During his career, Ron has worked in advanced development and product development roles for BLISS, FORTRAN, Ada, and multilanguage debugging on DIGITAL's DECsystem-10, PDP-11, VAX, and Alpha computer systems. He served as a representative on the ANSI and ISO standards committees for FORTRAN 77 and later for Ada 83, also serving as a U.S. Department of Defense invited Distinguished Reviewer and a member of the Ada Board and the Ada Language Maintenance Committee for more than eight years. Ron joined Digital Equipment Corporation in 1970, after earning the degrees of B.S.E. (engineering sciences), M.S. (applied mathematics), and Ph.D. (computer and communication sciences) in 1965, 1968, and 1969, respectively, all from the University of Michigan. He is a member of the Association for Computing Machinery and the IEEE Computer Society. Ron holds seven patents and has published several papers in the area of programming language design and implementation.

Jeffrey E. Nelson
Jeffrey E. Nelson is a senior software developer at Candle Corporation in Minneapolis, Minnesota. He currently develops message broker software for Roma BSP, Candle's middleware framework for integrating business applications. Previously at DIGITAL, Jeff was a principal software engineer on the OpenVMS and ladebug debugger projects. He specialized in debug symbol table formats, run-time language support, and computer architecture support. He contributed to porting the OpenVMS debugger from the VAX to the Alpha platform. He represented DIGITAL on the industry-wide PLSIG committee that developed the DWARF debugging symbol table format. Jeff holds an M.S. degree in computer science and applications from Virginia Polytechnic Institute and State University and a B.S. degree in computer science from the University of Wisconsin–LaCrosse. Jeff is an alumnus of the Graduate Engineering Education Program (GEEP), has been awarded one patent, and has previously published and presented work in the area of real-time, object-oriented garbage collection.

Mark E. Arsenault
Mark E. Arsenault is a principal software engineer in Compaq's OpenVMS Engineering Group working on the OpenVMS debugger. Mark has implemented support in the debugger for 64-bit addressing, C++, and inlining. He joined DIGITAL in 1981 and has worked on several software development tools, including the BLISS compiler and the Source Code Analyzer. Mark holds two patents, one each for the Heap Analyzer and for the Correlation Facility. He received a B.A. in physics from Boston University in 1981.