Debugging Optimized Code with Dynamic Deoptimization

Urs Hölzle
Computer Systems Laboratory
CIS 57
Stanford University
Stanford, CA 94305
urs@cs.stanford.edu

Craig Chambers
Dept. of Computer Science and Engineering
Sieg Hall, FR-35
University of Washington
Seattle, WA 98195
chambers@cs.washington.edu

David Ungar
Sun Microsystems Laboratories
MTV29-116
2500 Garcia St.
Mountain View, CA 94043
ungar@eng.sun.com

Abstract: SELF's debugging system provides complete source-level debugging (*expected behavior*) with globally optimized code. It shields the debugger from optimizations performed by the compiler by dynamically *deoptimizing* code on demand. Deoptimization only affects the procedure activations that are actively being debugged; all other code runs at full speed. Deoptimization requires the compiler to supply debugging information at discrete *interrupt points*; the compiler can still perform extensive optimizations between interrupt points without affecting debuggability. At the same time, the inability to interrupt between interrupt points is invisible to the user. Our debugging system also handles *programming changes* during debugging. Again, the system provides expected behavior: it is possible to change a running program and immediately observe the effects of the change. Dynamic deoptimization transforms old compiled code (which may contain inlined copies of the old version of the changed procedure) into new versions reflecting the current source-level state. To the best of our knowledge, SELF is the first practical system providing full expected behavior with globally optimized code.

1. Introduction

SELF is a pure object-oriented language designed for rapid prototyping, increasing programmer productivity by maximizing expressiveness and malleability [US87]. SELF's pure message-based model of computation requires extensive optimization to achieve good performance [CU91, HCU91]. Without aggressive procedure integration (inlining), for example, performance would be abysmal [Cha92]. But an interactive programming environment also demands rapid turnaround time and complete source-level debugging. To make SELF practical, the system must provide interpreter semantics at compiled-code speed, combining *expected behavior* [Zel84] with global optimization.

This work has been supported in part by the Swiss National Science Foundation (Nationalfonds), an IBM graduate student fellowship, NSF Presidential Young Investigator Award # CCR-8657631, and by Sun, IBM, Apple, Cray, Tandem, TI, and DEC.

Most existing systems do not support the debugging of optimized code. Programs can either be optimized for full speed, or they can be compiled without optimizations for full source-level debugging. Recently, techniques have been developed that strive to make it possible to debug optimized code [Hen82, Zel84, CMR88]. However, none of these systems is able to provide full source-level debugging. For example, it generally is not possible to obtain the values of all source-level variables, to single-step through the program, or to change the value of a variable. Optimization is given priority over debugging, and consequently these systems provide only restricted forms of debugging.

To the best of our knowledge, SELF is the first practical system providing full expected behavior with globally optimized code. Compared to previous techniques, our use of *dynamic deoptimization* and *interrupt points* permits us to place fewer restrictions on the kind of optimizations that can be performed while still preserving expected behavior.

The remainder of this paper is organized as follows. Section 2 discusses the optimizations performed by the SELF compiler and how they affect debugging. Section 3 describes how optimized code can be deoptimized, and section 4 explains how running programs can be changed. Section 5 discusses the implementation of common debugging operations. Section 6 discusses the benefits and limitations of our approach, and section 7 examines its run-time and space cost. Section 8 relates this paper to previous work, and section 9 contains our conclusions.

2. Optimization and debugging

This section briefly outlines the optimizations that the SELF compiler performs and discusses some of the problems they cause for the debugger.

ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, June 1992.

2.1 Optimizations performed by the SELF compiler

SELF uses dynamic compilation [DS84, CUL89]: instead of compiling whole programs prior to execution, code is generated incrementally at run-time and kept in a cache. Whenever a source method is invoked which hasn't already been compiled, a new compiled method is created and cached. (In the SELF system, source code is accessible at all times so that methods can be (re-)compiled at any time.)

In addition to standard optimizations such as global constant propagation, constant folding, and global register allocation, our compiler relies extensively on three optimizations which are important for pure object-oriented languages: inlining, customization, and splitting [CUL89, CU90, CU91]. *Inlining* reduces the call overhead and enables optimizations which span source method boundaries. *Customization* creates multiple compiled copies of source methods, each copy specialized for a particular receiver type. Customization allows many dynamically-dispatched calls to be statically bound and subsequently inlined. *Splitting* creates multiple compiled copies of a source-level expression and optimizes each copy for a particular set of types.

Additionally, the compiler performs dead code elimination, strength reduction, and global common subexpression elimination of arithmetic expressions, loads, and stores. Redundant computations are eliminated only if they cannot cause observable side effects such as arithmetic overflow. The compiler sometimes unrolls loops to avoid repeating type tests in every iteration of the loop, which frequently has the effect of hoisting invariant code out of loops. Low-level optimizations such as delay slot filling are also performed; more extensive instruction scheduling could easily be supported but has not been implemented. Induction variable elimination could be supported by extending the structure of our debugging information.

Since it must always provide full source-level debugging, the SELF compiler does not perform certain optimizations. In general, dead stores cannot be eliminated, and the registers of dead variables cannot be reused without spilling the variable to memory first. Both optimizations can be performed, however, if there is no interrupt point within the variable's scope (see section 3.4). Finally, the SELF compiler does not perform tail recursion elimination or tail call elimination because they cannot be supported transparently: in general, it is not possible to reconstruct the stack frames eliminated by these optimizations. Instead, iteration is supported through a primitive which restarts execution of the current scope. (The SELF language does not pre-define common control structures such as `if` and `while`; such control structures are user-defined).

2.2 Problems caused by optimization

The code transformations performed by global optimization make it hard to debug optimized code at the source level. Because optimizations delete, change, or rearrange parts of the original program, they become visible to the user who tries to debug the optimized program. This sections presents some of the problems that must be solved to provide source-level debugging of optimized code.

2.2.1 Displaying the stack

Optimizations such as inlining, register allocation, constant propagation, and copy propagation create methods whose activation records have no direct correspondence to the source-level activations. For example, a single physical stack frame may contain several source-level activations because message sends have been inlined. Variables may be in different locations at different times, and some variables may not have run-time locations at all.

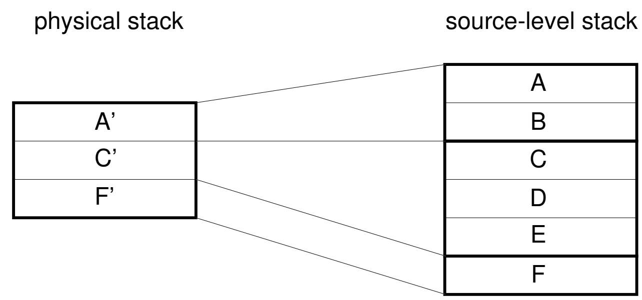

Figure 1. Displaying the stack

The example in Figure 1 shows the effects of inlining. The physical stack contains three activations A', C', and F'. In contrast, the source-level-stack contains additional activations which were inlined by the compiler. For example, the activation B was inlined into A', and so B does not appear in the physical stack trace.

2.2.2 Single-stepping

To single-step, the debugger has to find and execute the machine instructions belonging to the next source operation. Optimizations such as code motion or instruction scheduling make this a hard problem: the instructions for one statement may be interspersed with those of neighboring statements, and statements may have been reordered to execute out of source-level order. In contrast, single-stepping is simple with unoptimized code since the code for a statement is contiguous.

2.2.3 Changing the value of a variable

Consider the following code fragment:

```
i := 3;                              i := 3;
j := 4;        optimization          j := 4;
k := i + j;    ----------->          k := 7;
```

Since the expression `i + j` is a compile-time constant, the compiler has eliminated its computation from the generated code. But what if the program is suspended just before the assignment to `k` and the programmer changes `j` to be 10? Execution of the optimized code cannot be resumed since it would produce an unexpected value for `k`. With unoptimized code, of course, there would be no problem since the addition would still be performed by the compiled code.

2.2.4 Changing a procedure

A similar problem arises when an inlined procedure is changed during debugging. Suppose that the program is suspended just before executing the inlined copy of function `f` when the programmer changes `f` because she has found a bug. Obviously, execution cannot simply continue since `f`'s old definition is hard-wired into its caller. On the other hand, it would be easy to provide expected behavior with unoptimized code: `f`'s definition could simply be replaced, and the subsequent call to `f` would execute the correct code.

3. Deoptimization

None of the above problems would exist with unoptimized code. If optimized code could be converted to unoptimized code on demand, programs could be debugged easily while still running at full speed most of the time. SELF's debugging system is based on such a transformation. Compiled code exists in one of two states:

- *Optimized code*, which can be suspended only at relatively widely-spaced interrupt points; at every interrupt point, the source-level state can be reconstructed, and
- *Unoptimized code*, which can be suspended at any arbitrary source-level operation and thus supports all debugging operations (such as single-stepping).

Section 3.1 explains the data structures used to recover the source-level state from the optimized program state. Sections 3.2 and 3.3 describe how optimized code can be transformed into unoptimized code on demand, and section 3.4 discusses how interrupt points lessen the impact of debugging on optimization.

3.1 Recovering the unoptimized state

To display a source-level stack trace and to perform deoptimization, the system needs to reconstruct the source-level state from the optimized machine-level state. To support this reconstruction, the SELF compiler generates *scope descriptors* [CUL89] for each scope contained in a compiled method, i.e., for the initial source method and all methods inlined within it. A scope descriptor specifies the scope's place in the virtual call tree of the physical stack frame and records the locations or values of its arguments and locals (see Figure 2). The compiler also describes the location or value of each subexpression within the compiled method. This information is needed to reconstruct the stack of evaluated expressions that are waiting to be consumed by later message sends.

To find the correct scope for a given physical program counter, the debugger needs to know the *virtual program counter* (source position), i.e., the pair of a scope description and a source position within that scope. Therefore, the debugging information generated with each compiled method also includes a mapping between physical and virtual program counters.

With the help of this information, the debugger can hide the effects of inlining, splitting, register allocation, constant propagation, and constant folding from the user. For example, if the compiler eliminates a variable because its value is a compile-time constant, the variable's descriptor would contain that constant. A straightforward extension of the descriptor structure could be used to handle variables whose values can be computed from other values (such as eliminated induction variables).

```
struct ScopeDesc {
    oop method;                       // pointer to the method object
    ScopeDesc* caller;                // scope into which this scope was inlined (if any)
    int posWithinCaller;              // source position within caller
    ScopeDesc* enclosingScope;        // lexically enclosing scope (if any)
    NameDesc args[];                  // descriptors for receiver and arguments
    NameDesc locals[];                // descriptors for locals
    NameDesc expressionStack[];       // descriptors for all subexpressions
};

struct NameDesc {
    enum { const, loc } tag;          // compile-time constant or run-time value
    union {
        oop value;                    // constant value
        Location location;            // run-time location
    };
};
```

Figure 2. Pseudo-code declarations for scope data structures

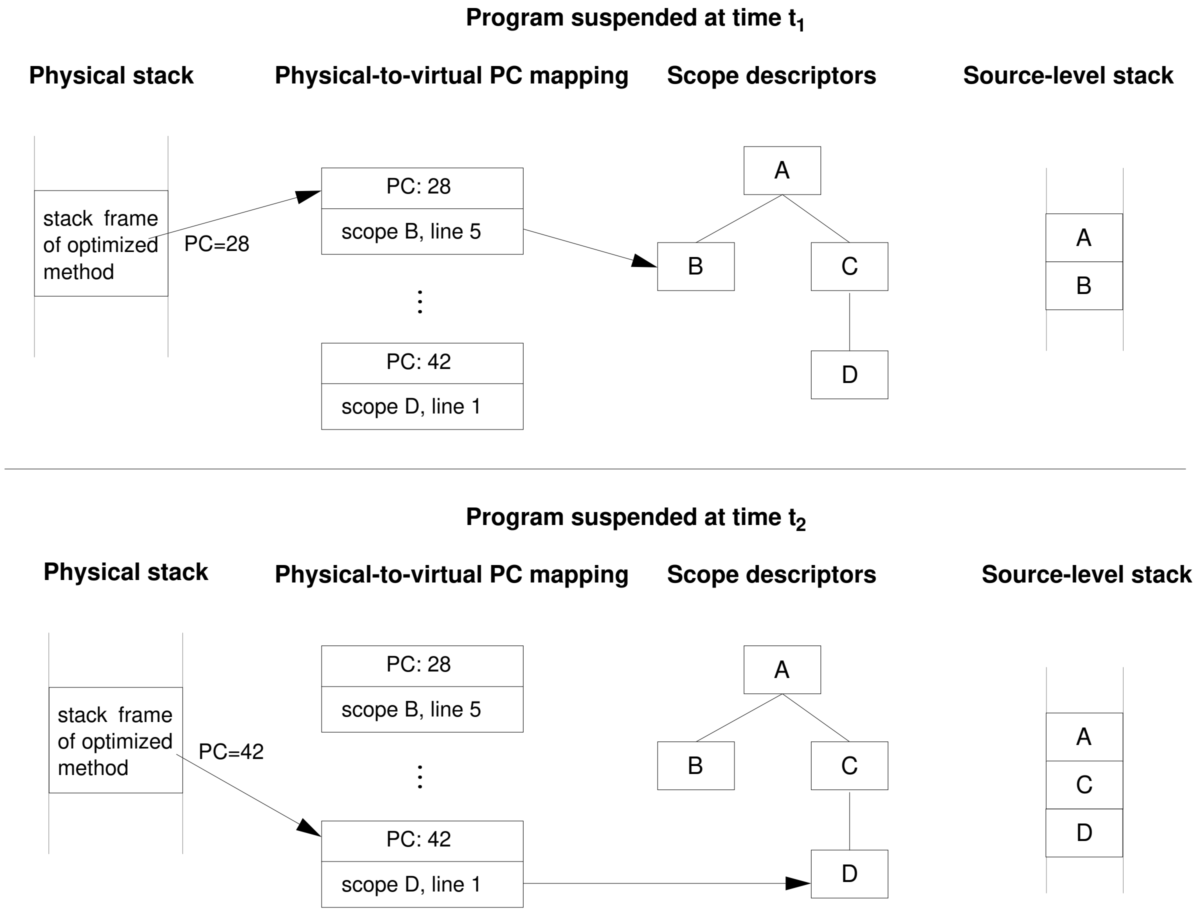

Figure 3. Recovering the source-level state

Figure 3 shows a method suspended at two different times. When the method is suspended at time t_1, the physical PC is 28 and the corresponding source position is line 5 of method B. A stack trace would therefore display B being called by A, hiding the fact that B has been inlined into A by the compiler. Similarly, at time t_2 the source-level view would show D being called by C being called by A, displaying three virtual stack frames instead of the single physical stack frame. To display a complete stack trace, this process is simply repeated for each physical stack frame

3.2 The transformation function

Our approach transforms an optimized method into one or more equivalent unoptimized methods. For the moment, we assume that only the topmost stack activation needs to be transformed so that stack frames can easily be removed or added; section 3.3 explains how to remove this restriction. The deoptimizing transformation can then be performed as follows:

1. Save the contents of the physical activation (stack frame) which is to be transformed, and remove it from the run-time stack.

2. Using the mechanisms described in the previous section, determine the source-level (*virtual*) activations contained in the physical activation, the values of their locals, and their virtual PC.

3. For each virtual activation, create a new compiled method and a corresponding physical activation. To simplify the transformation function and subsequent debugging activities, the new methods (the *target* methods) are completely unoptimized: every message send corresponds to a call, and no optimizations such as constant folding or common subexpression elimination are performed.

4. For each virtual activation, find the new physical PC in the corresponding compiled method. Since the target method is unoptimized, there will be exactly one physical PC for the given virtual PC. (This would not necessarily be the case if the target methods were optimized.) Initialize the stack frames created in the previous step by filling in the return PC and other fields needed by the run-time system, such as the frame pointer.

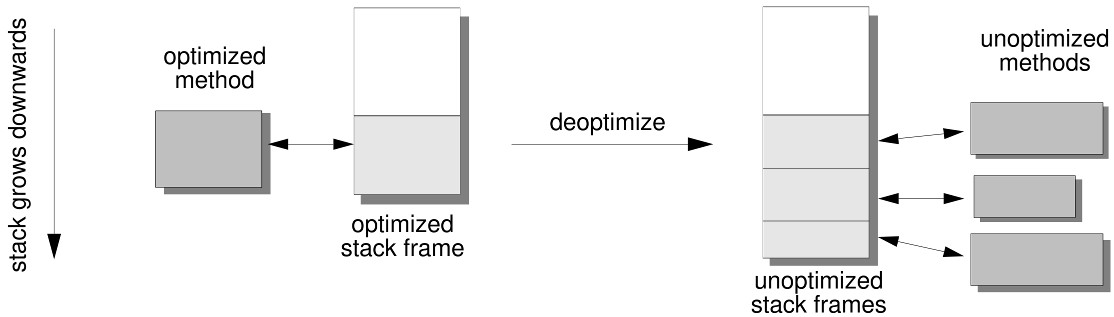

Figure 4. Transforming an optimized stack frame into unoptimized form

5. For each virtual activation, copy the values of all parameters, locals, and expression stack entries from the optimized to the unoptimized activation. Since the unoptimized method is a straightforward one-to-one translation of the source method, all variables will be mapped to locations in the target activation, and thus all copied values will have an unambiguous destination. (This would not necessarily be the case if the target methods were optimized.) Furthermore, since the target method is unoptimized, it does not contain any hidden state which would need to be initialized (such as the value of a common subexpression). Thus, together with step 4, we have completely initialized the new stack frames for all virtual activations, and the transformation is complete.

Figure 4 illustrates the process. The transformation expands an optimized stack frame containing three virtual activations into a sequence of three unoptimized stack frames, thus creating a one-to-one correspondence between virtual and physical frames.

Only those parts of the program which are actively being debugged (e.g., by stepping through them) need to be transformed. These parts will be the only parts of the program running unoptimized code; all other parts can run at full speed. No transformations are necessary just to inspect the program state, as described in section 3.1.

3.3 Lazy deoptimization

But how can a stack frame be deoptimized when it is in the *middle* of the stack, where new stack frames cannot be inserted easily? To solve this problem, our current implementation always transforms stack frames *lazily*: deoptimization is deferred until control is about to return into the frame (see Figure 5). For example, if the virtual activation vf_2 to be deoptimized is inlined in frame f in the middle of the stack, f is not immediately deoptimized. Instead, the return address of g (the stack frame called by f) is changed to point to a routine which will transform f when g returns. At that point, f is the topmost frame and is deoptimized. Transforming only the most recent activation simplifies the transformation process because no other stack frames need to be adjusted even if deoptimization causes the stack frames to grow in size.

Lazy deoptimization can simplify a system considerably, but it may also restrict the debugging functionality. Our system currently does not allow the contents of local variables to be changed during debugging because a variable might not have a run-time location. In order to create a run-time location for the variable, it might be necessary to transform an activation in the middle of the stack, which our system currently cannot do. However, this is not a fundamental problem; for example, the transformed stack frames could be heap-allocated as described in [DS84]. An even simpler solution would be to always allocate stack locations for eliminated variables. These locations would be unused during normal program execution but would spring into life when the programmer manually changes the value of an eliminated variable. Since the compiled code depended on the old (supposedly constant) value, it would be invalidated as if the programmer had changed the method's source code (see section 4).

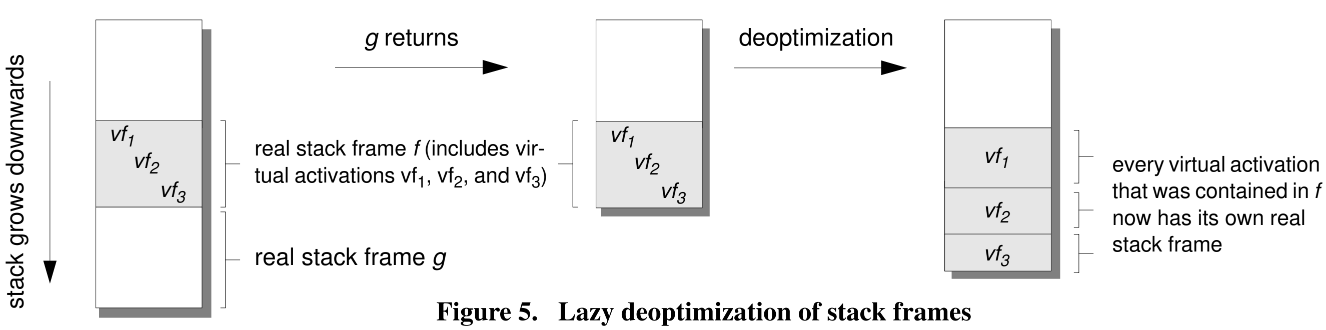

Figure 5. Lazy deoptimization of stack frames

3.4 Interrupt points

If optimized programs could be interrupted at any instruction boundary, debugging optimized code would be hard, since the source-level state would have to be recoverable at every single point in the program. To ease the restrictions that this would impose on optimization, an optimized SELF program can be interrupted only at certain *interrupt points*† where its state is guaranteed to be consistent. Notification of any asynchronous event occurring between two interrupt points is delayed until the next interrupt point is reached. Currently, the SELF system defines two kinds of interrupt points: method prologues (including some process control primitives) and the end of loop bodies ("backward branches"). This definition implies that the maximum interrupt latency is bounded by the length of the longest code sequence containing neither a call nor a loop end, typically only a few dozen instructions. Because the latency is so short, the use of interrupt points is not noticed by the programmer. (If only sends were interrupt points, loops without calls could not be interrupted.)

Interrupt points need to cover all possible points where a program could be suspended; that is, they also need to handle synchronous events such as arithmetic overflow. In our system, all possible run-time errors are interrupt points because all SELF primitives are safe: if the requested operation cannot be performed, the primitive calls a user-defined error handler which usually invokes the debugger.

Once an optimized program is suspended, the current activation can be deoptimized if necessary to carry out debugging requests. In an unoptimized method, every source point is an interrupt point, and the program can therefore stop at any point.

Since the debugger can be invoked only at interrupt points, debugging information need be generated only for those points. This reduces the space used by the debugging information, but more importantly it allows extensive optimizations between interrupt points. Essentially, the compiler may perform any optimization whose effects either do not reach an interrupt point or can be undone at that point. For example, the compiler can reuse a dead variable's register as long as there are no subsequent interrupt points within the variable's scope. The more widely-spaced interrupt points are, the fewer restrictions source-level debugging imposes on optimization.

Interrupt points also lessen the impact of garbage collection on compiler optimization. Garbage collections can only occur at interrupt points, and so the compiler can generate code between interrupt points that temporarily violates the invariants needed by the garbage collector [Cha87].

† Interrupt points have been used in other systems before; see section 8 for a discussion of the Deutsch-Schiffman Smalltalk-80 system.

4. Updating active methods

During debugging, a programmer might not only change the value of a variable but also the definition of a method. To invalidate the compiled code affected by such a change, the SELF system maintains *dependency links* between compiled code and the objects representing source code methods [CUL89]. For example, if a compiled method contains inlined copies of a method that was changed, the compiled method is discarded.

However, if a compiled method containing an inlined copy of a changed method is active (i.e., has at least one activation), it cannot simply be discarded. Instead, the compiled method must be replaced by a new compiled method before execution can continue. Fortunately, deoptimization can be used for this purpose. After the active compiled method has been deoptimized, it will no longer contain any inlined methods. When its execution continues, all subsequent calls to the changed method will correctly invoke the new definition.

If the changed method itself is currently active, updating its activation is hard. Fortunately, in SELF we don't have to solve this problem because in SELF's language model, activations are created by cloning the method object. Once created, the clone is independent from its original, so changes to the original do not affect the clone.

Lazy transformation elegantly solves the problem of invalidated compiled methods in the middle of the stack: we simply wait until the invalid method is on top of the stack, then transform it. Lazy transformation is very desirable in an interactive system since it spreads out the repair effort over time, avoiding distracting pauses. Furthermore, it handles sequences of changes well, for example when reading in a file containing new definitions for a group of related objects. With eager transformation, every new definition would cause all affected compiled methods to be recompiled, and many methods would be recompiled several times since they are likely to be affected by several of the changes. With lazy transformation, these compiled methods will be invalidated repeatedly (which is no problem since invalidation is very cheap) but only transformed once.

In conclusion, with our debugging mechanisms it is almost trivial to support changing running programs. Our current implementation consists of less than a hundred lines of C++ code on top of the previously described debugging functionality and the code maintaining the dependencies.

5. Common debugging operations

This section describes the debugging operations implemented in the SELF system and outlines possible implementations of additional operations. With deoptimization, it is relatively easy to implement common debugging operations such as *single-step* and *finish* because these operations are simple to perform in unoptimized code, and deoptimization can supply unoptimized code for every program piece on demand. In contrast, neither *single-step* nor *finish* could generally be provided by previous systems for debugging optimized code [Zel84, CMR88].

5.1 Single-step

Because every source point has an interrupt point associated with it in a deoptimized method, the implementation of single-stepping becomes trivial. The system deoptimizes the current activation and restarts the process with the interrupt flag already set. The process will relinquish control upon reaching the next interrupt point, i.e. after executing a single step.

5.2 Finish

The *finish* operation continues program execution until the selected activation returns. It is implemented by changing the return address of the selected activation's stack frame to a special routine that will suspend execution when the activation returns. Thus, the program is not slowed down during the *finish* operation because it can run optimized code.

If the selected activation does not have its own physical stack frame (because it was inlined into another method), its stack frame is deoptimized using lazy deoptimization. In this case, the program can still run optimized code most of the time; only at the very end (when lazy deoptimization is performed) does it run unoptimized code.

5.3 Next

The *next* operation (also called "step over") executes the next source operation without stepping into calls. That is, the program will stop after the next source operation has completed. *Next* can be synthesized by performing a *single-step*, possibly followed by a *finish* (if the operation was a call). Consequently, *next* is implemented by a few lines of SELF code in our system.

5.4 Breakpoints and watchpoints

SELF currently supports breakpoints through source transformation: the programmer inserts a breakpoint by simply inserting a send of `halt` into the source method (`halt` explicitly invokes the debugger). To implement breakpoints without explicit changes by the programmer, the debugger could perform this source transformation transparently.

Watchpoints ("stop when the value of this variable changes") are also easy to provide because SELF is a pure object-oriented language, and all accesses are performed through message sends (at least conceptually; the compiler will usually optimize away such sends). To monitor all accesses to an object's instance variable `x`, we can rename the variable to `private_x` and install two new methods `x` and `x:` which monitor accesses and assignments, respectively, and return or change `private_x`. The dependency system will invalidate all code that inlined the old definition of `x` or `x:` (i.e., that directly accessed or changed `x`).

6. Discussion

In this section, we discuss some of the strengths and weaknesses of our approach and assess its generality.

6.1 Benefits

Our debugging technique has several important advantages. First, it is simple: the current implementation of the transformation process consists of less than 400 lines of C++ code on top of the code implementing the debugging information described in section 3.1. Second, it allows a loose coupling between debugger and compiler—neither has to know very much about the other. Third, it places no additional restrictions beyond those described in section 2 on the kind of optimizations which can be performed by the compiler. Thus, many common optimizations such as inlining, loop unrolling, common subexpression elimination, and instruction scheduling can be used without affecting debuggability. Finally, our method is well suited for an interactive system since it is incremental: usually, at most one stack frame needs to be converted as a result of a user command.

6.2 Current limitations

Using unoptimized code during debugging introduces a potential performance problem when the user decides to continue execution. Execution should proceed at full speed, but some of the stack frames may be unoptimized. However, this problem usually is not severe: only a few frames are running unoptimized code, and the unoptimized code will be discarded as soon as these frames return. All other parts of the system can run at full speed.

Methods containing loops could still pose a problem since they could remain on the stack in unoptimized form indefinitely. However, we currently are working to solve the more general problem of *adaptive compilation* [HCU91]. With adaptive compilation, methods are created in unoptimized form first to minimize compile pauses. Later, the frequently-used parts are automatically recompiled with optimization. Therefore, a system with adaptive compilation would automatically reoptimize any unoptimized loops created by debugging. The current SELF system already contains a primitive form of adaptive compilation.

6.3 Generality

The debugging approach presented here is not specific to SELF and could be exploited in other languages as well. Our system appears to require run-time compilation for deoptimization, but systems without run-time compilation could include an unoptimized copy of every procedure in an executable or dynamically link these in as needed.

For pointer-safe languages like Lisp, Smalltalk, or a pointer-safe subset of C++, our approach seems directly applicable. In pointer-unsafe languages[†] like C which allow pointer errors, interrupt points might be more closely spaced. The debugger could potentially be invoked at every load or store where the compiler could not prove that no address fault would occur. But even if interrupt points caused by unsafe loads or stores were indeed very frequent, our approach would still allow at least as many optimizations as other approaches for source-level debugging.

Pointers into the stack require special care during deoptimization if the locations of such pointers are unknown. In this case, the address of a stack variable potentially referenced by a pointer may not be changed. However, this problem could probably be solved at the expense of some stack space by requiring the layout of optimized and unoptimized stack frames to be identical.

6.4 Implementation status

The first implementation of the debugging system was completed in the spring of 1991 (recovering the source-level state of optimized programs was implemented in 1989). Today's system implements all functions described in section 5 and is in daily use at several research institutions. A source-level debugger (written in SELF by Lars Bak) is also part of the system. The SELF implementation is available free of charge via ftp from self.stanford.edu.

7. Cost

Providing full source-level debugging in the presence of an optimizing compiler does not come for free. In this section, we examine the impact of our techniques on responsiveness, run-time performance, and memory usage.

[†] In a way, true source-level debugging of unsafe languages is something of an oxymoron: since programs can overwrite arbitrary memory regions, they can always produce behavior which cannot be explained at the language (source) level. For example, if an integer is erroneously stored into the location of a floating-point variable, the resulting behavior cannot be explained without referring to the particular integer and floating-point representations used by the system.

7.1 Impact on responsiveness

Neither the deoptimization process nor the use of interrupt points are perceptible to users. The compiler typically creates the unoptimized methods in about one millisecond on a SPARCStation 1, and thus the pauses introduced by dynamic deoptimization are negligible. Interrupt points increase the latency for user and system interrupts by only a few microseconds because an interrupt point is usually reached within a few dozen instructions after the run-time system has set the interrupt flag. In summary, providing full source-level debugging in the SELF system has not reduced its responsiveness.

7.2 Impact on run-time performance

Ideally, the performance impact of full source-level debugging could be measured by completely disabling it and re-measuring the system. However, this is not possible because source-level debugging was a fundamental design goal of the SELF system. Disabling debugging support would require a major redesign of the compiler and run-time system if any better performance is to be achieved. Furthermore, the garbage collector already imposes some constraints on the optimizer, such as the requirement that live registers may not contain derived pointers (pointers into the middle of objects). In many cases, the optimizations inhibited by garbage collection are very similar to those inhibited by debugging requirements, such as dead store elimination and some forms of common subexpression elimination [Cha87]. Thus, it would be difficult to separate the impact of garbage collection on optimization from the impact of full source-level debugging.

However, we have measured some effects of source-level debugging in the SELF system. To determine the impact of debugger-visible names, the compiler was changed to release registers allocated to dead variables even if they were visible at an interrupt point. The performance improvement with the changed compiler was insignificant (less than 2%) for a wide range of programs [Cha92]. That is, the extension of variable lifetimes needed to support debugging seems to incur virtually no cost in our system. (One reason for this might be that SELF methods are typically very short, so that few variables are unused in significant portions of their scope.)

The system currently detects interrupts by testing a special register; each test takes two cycles on a SPARC. This polling slows down typical programs by about 4%; some numerical programs with very tight loops are slowed down by up to 13% [Cha92]. With a more complicated run-time system using conditional traps, the overhead could be reduced to one cycle per check, and loop unrolling could further reduce the problem for tight loops. Alternatively, we could switch to a non-polling system where the interrupt handler would patch the code of the currently executing procedure to cause a process switch at the next interrupt point.

While we could not measure the full performance impact of our debugging scheme, inspection of the generated code indicated no obvious debugging-related inefficiencies. In fact, the current SELF system has attained excellent performance, executing a set of benchmarks four to six times faster than ParcPlace Smalltalk-80[†] and about half the speed of optimized C [CU91].

7.3 Memory usage

The debugging and dependency information for compiled methods is kept in virtual memory in the current SELF system. Table 1 shows the memory usage of the various parts of compiled methods relative to the space used by the machine instructions. For example, the physical-to-virtual PC mapping is about 17% the size of the actual machine code. The column labelled "adaptive" represents the default configuration where only often-used methods are optimized, while the "optimized only" column represents a system which always optimizes. The data were obtained from two interactive sessions using the prototype SELF user interface (written in SELF). Both runs represent more than 7 Mbytes of compiler-generated data.

The space consumption can be split into three main groups. The first group contains the method headers and the machine instructions; together, these represent all the information needed to actually execute programs. The second group contains the dependency links needed to invalidate compiled code after programming changes (see section 4). The third group contains all debugging-related information: the scope descriptors and the PC mapping (see section 3.1), relocation information for the garbage collector (to update object pointers contained in the debugging information), and the various objects representing the methods, message name strings, and object literals corresponding to the compiled code. This includes all information needed to recompile methods but not the source code itself.[‡]

The space consumed by debugging information varies with the degree of optimization. Optimized methods show a higher relative space overhead than unoptimized methods because the debugging information for an inlined method is typically larger than the inline-expanded code. Therefore, the debugging information grows faster with more aggressive inlining than the compiled code.

The total space overhead for debugging is reasonable. In the standard system, debugging information uses slightly more space (122%) than the instructions themselves; in the system that optimizes everything, the overhead is 233%. In other words, adding the debugging information increases space usage (excluding the dependencies) by a factor of between 2.2 and 3.3.

In order to be conservative, we have left out the space used by the method headers even though they would also be needed in a system without debugging. (The headers contain the lookup key and various control information.) If we include the headers, debugging increases space usage by a factor of between 1.8 and 2.6.

The cost of supporting changes to running programs is smaller. The dependency information occupies between 0.9 and 1.6 times the size of the instructions. Furthermore, the

[†] Smalltalk-80 is a trademark of ParcPlace Systems.

[‡] This grouping is a slight simplification. For example, the compiler occasionally generates instructions just to support debugging. Also, a small portion of the relocation information can be attributed to the code rather than the debugging information. However, these simplifications do not significantly distort the numbers presented here.

Category		adaptive (default)	optimized only
Machine instructions	actual machine instructions and control information	1.00	1.00
Method headers		0.56	0.43
Dependency links	to invalidate code after programming changes	0.92	1.69
Scope descriptors	to recreate the source-level state of optimized code and to recompile methods	0.42	1.09
Physical-to-virtual PC mapping		0.17	0.17
Relocation information for GC		0.24	0.39
Method objects, strings, etc.		0.39	0.69

Table 1: Space cost of debugging information (relative to instructions)

current representation of the dependencies contains significant redundancies. An alternate implementation could probably reduce the space usage significantly [Cha92].

As a rough comparison, when compiling the SELF virtual machine, a standard commercial C++ 2.1 compiler generated 55 Mbytes of debugging information on top of an executable of 2.4 Mbytes, incurring an overhead of a factor of 24. Apparently, the compiler produced multiple copies of identical debugging information, one copy per object file, and the linker included them all in the executable file. Using the GNU C++ compiler and GNU-specific pragmas, we were able to reduce the space overhead of debugging information to 11.2 Mbytes, for a factor of 5.7.† While this comparison should be taken with a grain of salt, it indicates that despite the increased functionality, the space overhead of debugging in our system is probably not higher than in other systems.

During the design and implementation of our current data structures, simplicity was considered more important than space efficiency, and we did not optimize our representation much. For example, our method headers and scope representations use 32-bit words in many places where 16 or even 8 bits would suffice. A reorganization of these data structures could therefore result in significant savings. Other techniques such as generating the debugging information on demand by re-executing the optimizing compilation could save even more space at the expense of longer pauses during debugging interactions.

Furthermore, little of the debugging information needs to remain in main memory at all times, and much of it can be paged out. Ultimately, only the machine instructions need be in main memory for working, debugged programs, thus keeping real memory costs down to a fraction of the total virtual memory costs.

8. Related work

The Smalltalk-80 system described by Deutsch and Schiffman [DS84] pioneered the use of dynamic compilation and interrupt points. To hide the effects of compilation to native code, compiled methods included a mapping from compiled code to source position. Activations normally were created on the stack for run-time efficiency but were converted on demand to the full-fledged activation objects required by the language definition, and converted back when needed for execution. As in our system, interrupts were delayed until the next call or backward branch. Since the compiler performed no global optimizations, the system could provide expected behavior without deoptimization.

Zurawski and Johnson [ZJ91] describe a model for a debugger (developed concurrently with this work) which closely resembles ours, using "inspection points" and dynamic deoptimization to provide expected behavior for optimized code. However, the system does not use lazy conversion. Furthermore, their definition of inspection points allows asynchronous events such as user interrupts to be delayed arbitrarily. Some of their ideas were implemented for the Typed Smalltalk compiler [JGZ88], but the system apparently could only run very small programs and was not used in practice, unlike our system which is in daily use.

Most of the other work on debugging optimized code places the priority on optimization; the goal was to get as much debugging as possible while preserving code efficiency [CMR88, SW78]. Hennessy [Hen82] addresses the problem of recovering the values of variables in the presence of selected local and global code reordering optimizations. His algorithms can usually detect when a variable has an incorrect value (in terms of the source program) and can sometimes reconstruct the source-level value. In contrast, we are not willing to accept any debugging failures and therefore do not perform optimizations which would create such situations at an interrupt point.

Zellweger [Zel83, Zel84] describes an interactive source-level debugger for Cedar which handles two optimizations, procedure inlining and cross-jumping, to provide expected behavior in most cases. While her techniques can always recover the source-level values of variables, they cannot hide certain code location problems; for example, single-stepping through optimized code would be difficult. Since our system uses unoptimized code in these situations, we are able to circumvent these problems.

LOIPE [Fei83] uses transparent incremental recompilation for debugging purposes. For example, when the user sets a breakpoint in some procedure, this procedure is converted to unoptimized form to make debugging easier. However, LOIPE cannot perform such transformations on active procedures. Thus, if the program is suspended in an optimized procedure, it is generally not possible to set a breakpoint in this procedure or to continue execution by single-stepping. To mitigate this problem, users were able to specify the amount of optimization to be performed (possibly impacting debuggability) and the amount of debugging transparency needed (possibly affecting code quality). As far as we know, most of the support for optimized code in LOIPE was not actually implemented.

† GNU C++ allows the source-level debugging of optimized code but offers only restricted functionality. Many optimizations are not transparent to the programmer. The compiler version used for the measurements was GCC 1.94.7; version 2.0 generates a significantly larger executable.

Tolmach and Appel [TA90] describe a debugger for ML where the compiler always performs optimizations, but where the program is automatically annotated with debugging statements before compilation. To debug an optimized program, the programmer has to manually recompile and re-execute the program. Like unoptimized programs, annotated programs run significantly slower than fully optimized programs.

9. Conclusions

Global optimization need not impair source-level debugging. The SELF system increases programmer productivity by providing full source-level debugging of globally optimized code. To the best of our knowledge, SELF is the first system to do so; other systems either compromise on debugging functionality or severely restrict the kinds of optimizations that can be performed. In SELF, the compiler can perform optimizations such as constant folding, common subexpression elimination, dead code elimination, procedure integration, code motion, and instruction scheduling without affecting debuggability.

Two techniques make this possible: *lazy deoptimization* and *interrupt points*. The optimizations performed by the compiler are hidden from the debugger by deoptimizing code whenever necessary. Deoptimization supports single-stepping, running a method to completion, replacing an inlined method, and other operations, but only affects the procedure activations which are actively being debugged; all other code runs at full speed. Debugging information is only needed at relatively widely-spaced interrupt points, so that the compiler can perform extensive optimizations between interrupt points without affecting debuggability. Our debugging techniques is not specific to SELF and could be applied to other programming languages as well.

Acknowledgments: We wish to thank Ole Agesen, Lars Bak, Bay-Wei Chang, Peter Deutsch, David Keppel, and John Maloney for their valuable comments on earlier drafts of this paper.

References

[Cha87] David Chase. *Garbage Collection and Other Optimizations*. Ph.D. dissertation, Computer Science Department, Rice University, 1987.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph.D. dissertation, Computer Science Department, Stanford University. March 1992.

[CMR88] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 150-164, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(11),* November, 1991.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 297-302, Salt Lake City, UT, January, 1984.

[Fei83] Peter H. Feiler. *A Language-Oriented Interactive Programming Environment Based on Compilation Technology*. Ph.D. dissertation, Carnegie-Mellon University, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991. Published as Springer Verlag LNCS 512, 1991.

[Hen82] John L. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions of Programming Languages and Systems 4(3)*, July 1982.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*, pp. 18-26, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[SW78] H. Schlaeppi and H. Warren. Design of the FDS Interactive Debugging System. IBM Research Report RC7214, IBM Yorktown Heights, July 1978.

[TA90] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML Without Reverse Engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990, pp. 1-12.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Zel83] Polle T. Zellweger. *An Interactive High-Level Debugger for Control-Flow Optimized Programs.* Xerox PARC Technical Report CSL-83-1, January 1983.

[Zel84] Polle T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Ph.D. dissertation, Computer Science Department, University of California, Berkeley, 1984. Also published as Xerox PARC Technical Report CSL-84-5, May 1984.

[ZJ91] Lawrence W. Zurawski and Ralph E. Johnson. *Debugging Optimized Code With Expected Behavior.* Unpublished manuscript, 1991.