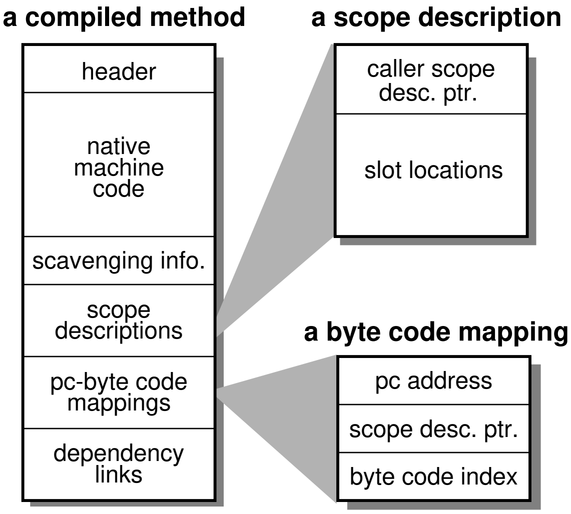
Chapter 13 Programming Environment Support

The SELF system is designed to be an interactive exploratory programming environment, and so the SELF implementation must support both rapid turn-around for programming changes and complete source-level debugging. These features are fairly easy to support in an interpretive environment but are much more difficult to achieve in a high-performance optimizing compiler environment, particularly one based on aggressive inlining. Other researchers have investigated the problem of enabling compilation and optimization to coexist gracefully with the programming environment [Hen82, Zel84, CMR88, ZJ91]. This chapter describes the techniques used in the SELF implementation to support the programming environment, focusing on the support provided by the compiler.

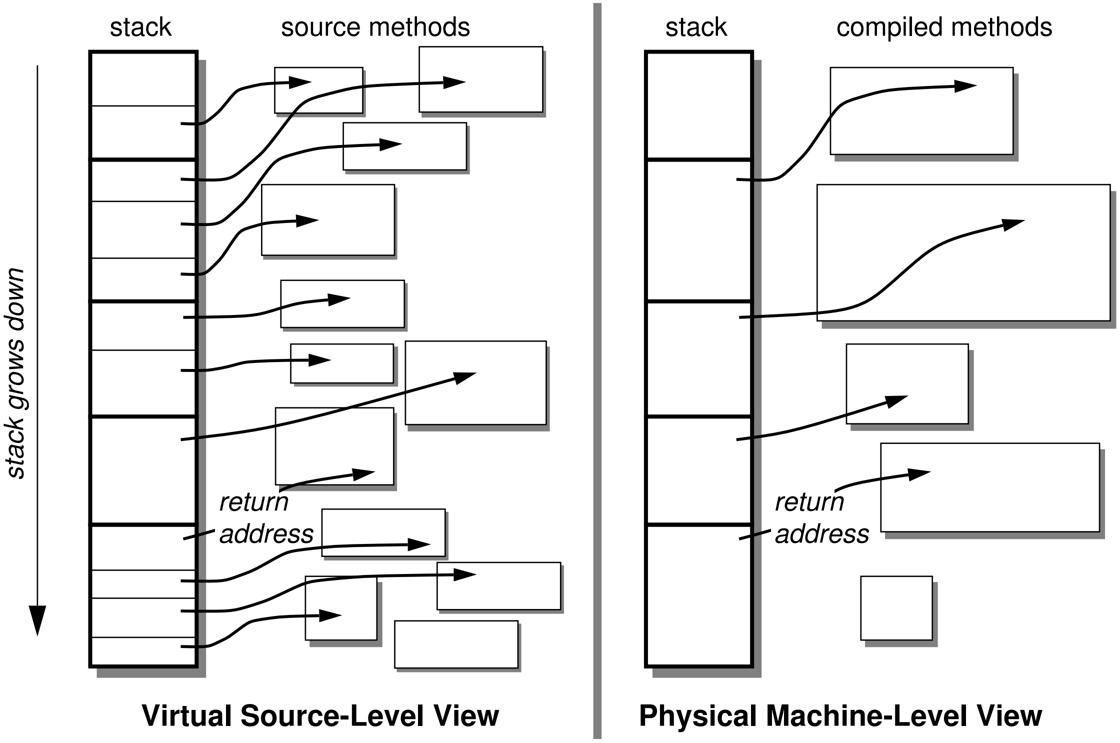
A compiled method contains more than just instructions. First, it includes a list of the offsets within the instructions of embedded object references, used by the garbage collector to modify the compiled code if a referenced object is moved. Second, a compiled method includes descriptions of the inlined methods, which are used to find the values of local slots of the method and to display source-level call stacks. Third, a compiled method contains a bidirectional mapping between source-level byte codes and actual program counter values. These two kinds of debugger-related information are described in section 13.1. Finally, a compiled method includes dependency links to support selective invalidation of methods after programming changes; these are described in detail in section 13.2.



Parts of a Compiled Method

13.1 Support for Source-Level Debugging

A good programming environment must include a source-level debugger. The SELF debugger presents the program execution state in terms of the programmer's execution model: the state of the source code interpreter, with *no* optimizations. This requires that the debugger be able to examine the state of the compiled, optimized SELF program and construct a view of that state (the *virtual* state) in terms of the byte-coded execution model. Examining the execution state is complicated by having activation records in the *virtual call stack* actually be inlined within other activation records in the *physical call stack*, and by allocating the slots of virtual methods to registers and/or stack locations in the compiled methods.



13.1.1 Compiler-Generated Debugging Information

To allow the debugger to reconstruct the virtual call stack from the physical call stack, the SELF compiler appends debugging information to each compiled method.

- For each scope compiled (the initial method plus any methods or block methods inlined within it), the compiler outputs information describing that scope's place in the virtual call tree within the compiled method's single physical stack frame.
- For each argument and local slot in the scope, the compiler outputs either the value of the slot (if it is a constant known at compile-time, as many slots are) or the register or stack location allocated to hold the value of the slot at run-time.
- For each subexpression within the compiled method, the compiler describes either the compile-time constant value of the subexpression or the register or stack location allocated for the subexpression. This information is used to reconstruct the stack of evaluated expressions that are waiting to be consumed by later message sends.

For example, consider a simple method to compute the minimum of two values:

```
min: arg = (
  < arg ifTrue: [self] False: [arg] ).
```

If **`min:`** is sent to an integer, the compiler will generate a compiled version of the **`min:`** source method customized for integers. (Customization was the subject of Chapter 8.) The **<** method for integers will be looked up at compile-time, locating the following method:

```
< x = ( _IntLT: x IfFail: [...] ).
```

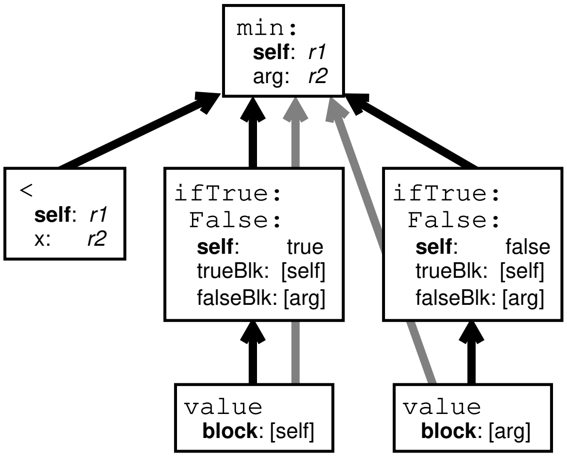
This method and the contained call to the **intLT** primitive will be inlined. The result of the **<** message will be either the **true** object or the **false** object (in the common case), leading the compiler to split the succeeding **ifTrue:False:** message along these two possible outcomes. (Splitting was the subject of Chapter 10.). The compiler looks up the definition of **ifTrue:False:** for **true**:

```
  ifTrue: trueBlk False: falseBlk = ( trueBlk value ).
```

and for **false**:

```
  ifTrue: trueBlk False: falseBlk = ( falseBlk value ).
```

These two methods will be inlined, as will the nested **value** messages within each method. When generating code for this compiler method, the compiler outputs debugging information to represent this tree of inlined methods:



Each scope description refers to its calling scope description (black arrows in the diagram); a block scope also references its lexically-enclosing scope description (gray arrows in the diagram). For each slot within a scope, the debugging information identifies either the slot's compile-time value or its run-time location. For the **min:** example, only the initial arguments have run-time locations (registers **r1** and **r2** in this case); all other slot contents are known statically at compile-time. The expression stack debugging information is omitted from this illustration.

This additional debugging information is fairly space-consuming. As described in section B.6 of Appendix B, scope descriptions take up between 1.5 and 5 times as much space as do the compiled machine instructions, depending on the degree of inlining performed when compiling the routine. Fortunately, this information can be paged out when the associated routine is not being debugged. Also, it might be possible to avoid storing the debugging information, instead regenerating the debugging information upon demand by re-executing the compiler.

13.1.2 Virtual/Physical Program Counter Translation

The SELF compiler also outputs debugging information to support translation between the source-level *virtual program counter* within a virtual stack frame (the pair of a scope description and a byte code index within the scope description) and the machine-level *physical program counter* within a physical stack frame. This information is used to translate a hardware physical return address of a stack frame into a byte code index within a virtual stack frame of the physical frame, such as when displaying the current virtual execution stack. The mapping also is used to locate the physical program counter corresponding to a particular virtual program counter, such as when setting breakpoints at particular source positions.

The p.c./byte code mapping is not one-to-one but instead is many-to-many. Several virtual program counter addresses can map to the same physical program counter address: a sequence of source-level messages can get inlined and optimized away completely (such as most of the messages sent during the execution of a user-defined control structure), generating no machine code for any of the eliminated messages, and so each of the messages will end up mapping to the same physical machine address. Additionally, several physical program counter addresses may map to the same virtual program counter address: a single source-level message can get split and compiled in more than one place, thus leading several physical program counter addresses to map to the same source-level message. Consequently, the compiler treats this mapping as a simple relation and generates a long list of three-word tuples, each tuple consisting of a physical program counter address (the physical view) and the pair of a pointer to a virtual scope description and a byte code index within the scope (the virtual source-level view).

As reported in section B.6 of Appendix B, the p.c./byte code mapping is fairly concise, only requiring space that is about 25% of that taken up by compiled instructions. One reason for its comparatively small size is that the compiler only generates tuples that correspond to call sites in the compiled code, since those are the only places that the system might suspend the method and examine its p.c./byte code mapping.

13.1.3 Current Debugging Primitives

The current SELF implementation includes partial support for interactive debugging. The system supports displaying the virtual execution call stack, complete with the current values of all local variables of all virtual activation records; all optimizations including inlining are completely invisible. The system also supports manipulating individual activation records directly as SELF objects, querying the contents of their local slots, examining their expression stack, and navigating around the dynamic and static call chains. The current implementation does not yet support modifying the contents of local variables in activation records, but we do not think that adding this facility would be too difficult. The system supports breakpoints and single-stepping through process control primitives. The programmer can set a breakpoint by editing in a call to user-defined SELF code which eventually invokes the process suspend primitive. A suspended process can be single-stepped by invoking other process control primitives.*

13.1.4 Interactions between Debugging and Optimization

Most optimizing compilers do not support complete source-level debugging because the optimizations they perform prevent the virtual source-level state from being completely reconstructed. For example, tail call optimizations prevent the programmer from examining the elided stack frames, and dead variable elimination and dead assignment elimination prevent the programmer from examining the contents of a variable that is in scope but no longer needed by the compiled code. The SELF compiler performs no optimization that would prevent the debugger from completely reconstructing the virtual source-level execution state as if no optimizations had been performed. Even so, the SELF compiler still can perform many effective optimizations including inlining, splitting, and common subexpression elimination, since these optimizations can be "undone" at debug-time given the appropriate debugging information.

The SELF compiler's job of balancing debugging support against various optimizations is eased by only requiring debugger support at those places in which a debugging primitive might be invoked, such as message sends and **`_Restart`** loop tails.** The compiler is not required to support debugging at arbitrary instruction boundaries (as would be required if an interrupt could occur at any point in the program) or even at source-level byte code boundaries (as would be required if the user could single-step through optimized code; single stepping is implemented by recompiling methods with no optimization and then stopping at every call site). Since the debugger can be invoked only at well-defined locations in the compiled code, the compiler can perform optimizations between these potential interruption points that would be difficult or impossible to perform if instruction-level or byte-code-level debugging information were required. For example, the compiler can reuse the register of a dead variable as long as there are no subsequent call sites or interruption points in which the variable is still in scope.

Unfortunately, the current representation of debugging information places restrictions on the compiler that can hurt performance. As mentioned in section 12.1, the current SELF register allocator either can allocate a particular name to a single location for its entire lifetime or can mark the name as bound to a particular compile-time constant for its entire lifetime. The restriction that the allocation be constant over the name's entire lifetime primarily is caused by the limited abilities of the debugging information to describe the allocation; there is no easy way to have different allocations for different subranges of the name's lifetime. The system might be able to get added flexibility within the constraints of the current representation by making copies of virtual scope descriptions whenever a name had different allocations for different parts of its lifetime, and using the physical/virtual program counter mapping to select the appropriate virtual scope description for the physical program counter. This approach would support some form of position-varying allocation, but could lead to a lot of duplicated debugging information. A better approach would be to redesign the debugging information representation from scratch to efficiently support names with position-varying allocations.

* Urs Hölzle implemented most of the process control and activation record manipulating primitives, and Bay-Wei Chang integrated these primitives into the graphical user interface.

** The debugger might run at **`_Restart`** points since these points check for interrupts (as described in section 6.3.1), and the user-defined interrupt handler might call the debugger.

13.2 Support for Programming Changes

As described in section 7.1, the SELF compiler assumes that certain hard-to-change parts of objects will remain constant, and the compiler performs optimizations based on these assumptions. For example, the compiler assumes that the set of slots of a particular object, such as an integer or the **`true`** object, will remain the same, and this allows the compiler to perform message lookup at compile-time. Similarly, the compiler assumes that the contents of a non-assignable data slot will never change and that the offset of an assignable data slot will never change. These assumptions enable the compiler to inline the bodies of methods and replace data slot access methods with load and store instructions, thereby generating much faster code.

These assumptions will always be correct if the only object mutations available to programs and programmers are normal assignments to assignable data slots (the compiler explicitly avoids depending on the contents of an assignable slot). However, additional operations are available in a programming environment to mutate objects in other ways, such as adding and removing slots or changing the contents of non-assignable data slots. These modifications may invalidate the assumptions made by the compiler when compiling and optimizing methods. If executed, such out-of-date compiled code can lead to incorrect behavior or even system crashes.

13.2.1 Ways of Supporting Programming Changes

Traditional batch compiling systems support programming changes by requiring the programmer to recompile manually those files that are out-of-date, relink the program, and restart the application. At best, some of this process can be automated by using utilities to determine which files need to be recompiled after a set of programming changes. Turn-around time for a single programming change can be quite long, at least tens of seconds and more typically minutes or tens of minutes. Programmer productivity suffers greatly with turn-around times of this length for simple programming changes.

Interactive systems are designed to support rapid programming turn-around times, on the order of a few seconds or less. They usually achieve this level of interactive performance by limiting the dependencies among components of a system, so individual components can be replaced as simply and easily as possible without requiring complex time-consuming system relinking or recompilation of other components not directly altered by a programming change. However, execution performance tends to be much lower than in the traditional optimizing environment, since inter-component information is not used for optimizations. The SELF compiler clearly violates the basic assumptions of this style of system, since the SELF compiler delights in performing optimizations such as inlining that create many inter-component dependencies.

SELF's run-time compilation architecture offers one possible solution to this dilemma. After a programming change that might have invalidated the assumptions used to compile some method, the system could simply *flush* all the compiled code from the compiled code cache. New code will be compiled as needed from the (possibly changed) source methods, using the new correct assumptions about the relatively unchanging parts of the object structure. Since compiled code cache flushing is fast, this would seem to solve the programming turn-around time problem.

Unfortunately, this approach simply shifts the cost of the programming change from the flushing operation to immediately after the flushing. Since after the flush no methods are compiled, nearly every message send will require new compiled code to be generated, leading to a long sequence of compiler pauses immediately after the flush. While these compiler pauses will be spread out somewhat over the next chunk of program execution, and SELF's compiler architecture will allow code to be generated and "relinked" much faster than a traditional file-based environment, turn-around time usually will still be much longer than the second or two supported by the unoptimizing interactive system.

To avoid these lengthy recompilation pauses, the SELF system maintains enough inter-component dependency information to *selectively invalidate* only those compiled methods that are affected by a programming change. If this set is small, then the number of pauses to recompile invalidated methods can be kept small and the overall perceived turn-around time can be kept short. In these situations, selective invalidation enables our SELF system to support both fast turn-around on programming changes and fast run-time execution between changes.

Selective invalidation is not a cure-all, however. If some extremely common method that is inlined in many places is changed, such as the definition of **+** for integers, then the selective invalidation approach reduces to the expensive total flush approach, producing the same lengthy compilation pauses as the whole system gets recompiled with the new definition. Fortunately, this has not been a problem in practice, since the common methods used throughout the system are changed very rarely; in other systems such sweeping changes could not be made at all. The current trade-off between run-time performance and programming turn-around time favors run-time performance over turn-around time for these kinds of short, commonly-used “system” methods.

13.2.2 Dependency Links

To support selective invalidation, the compiler maintains two-way *change dependency links* between each compiled method in the cache and the information that the compiler assumed would remain constant. This information used to compile code—the set of slots in objects, the offsets of assignable data slots, and the contents of non-assignable slots—is precisely the information stored in maps. (Maps were described in section 6.1.1 and in Appendix A.) Therefore, the system only needs to maintain dependency links between maps and compiled methods. Since many compiled methods may depend on a particular map, and each compiled method may depend on many maps, these dependency links must support a many-to-many mapping between maps and compiled methods.

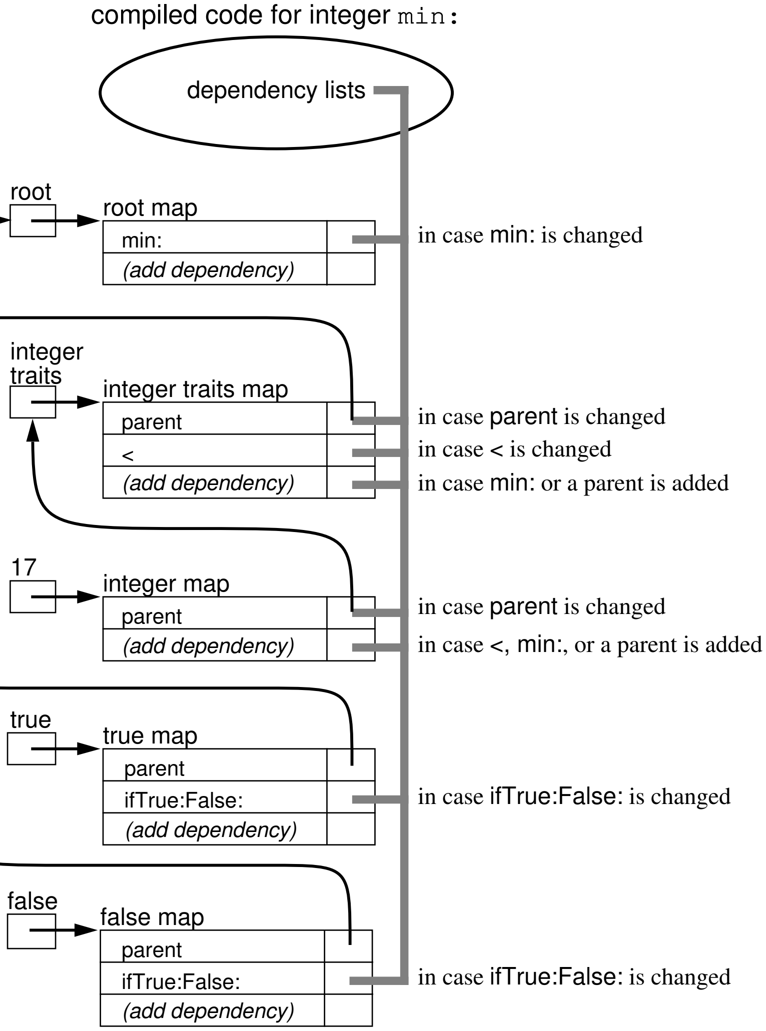
Dependency links are created as a result of message lookup to record those aspects of objects traversed during lookup that if they were modified potentially could change the result of the lookup and consequently the correctness of the compiled code. Clearly the compiled code depends on the result of the message lookup, and so the system leaves behind a dependency link between the matching slot found at the end of the lookup and the method being compiled. If the matching slot is later changed (either by changing the contents of a constant slot such as a method or parent slot or by changing the offset in the object of the contents of an assignable data slot) or removed altogether, all linked compiled methods are flushed from the compiled code cache.

The lookup system scans other parts of objects which affect the outcome of the lookup and so need dependencies. The message lookup system fetches the contents of a parent slot when searching an object’s parents. If the parent slot is later changed or removed, the outcome of the message send could change. To record this fact, the system creates a dependency link between the parent slot and the compiled code for the method eventually found as the result of the lookup. Then if the parent slot is modified or removed, all linked compiled methods will be flushed appropriately.

A more subtle kind of link handles the problem that a slot may be added to an object that affects the outcome of a message send. The message lookup system frequently searches an object for a matching slot and is unsuccessful; the object’s parents are searched in turn for a matching slot. If either a matching slot or a parent slot that inherits a matching slot is later added to the object, then the results of the earlier message would likely be changed, possibly invalidating some compiled code. To handle this problem, the compiler creates a special *add dependency link* between compiled code and the maps of objects unsuccessfully searched for a particular slot; this dependency is not associated with any slot in the map but instead with the map as a whole. If a slot is ever added to the map, then all compiled methods linked by the add dependency are flushed, since the added slot might affect their lookup results.

Unlike slot-specific dependency links, add dependency links are imprecise. Since they do not record exactly which message names were unsuccessfully scanned previously, methods may be flushed that do not need to be flushed. This could significantly reduce the selectiveness of the flushing, possibly leading to long compile pauses after a programming change in which slots were added. On the other hand, recording exactly which message names have been searched unsuccessfully for each map would consume a lot of space, and many maps would have similar long lists of unsuccessful matches. We are currently exploring alternative mechanisms that would support selective invalidation even for slot additions.

The following diagram illustrates the dependency links that are created when compiling the **min:** method described earlier in this chapter. The gray line represents eight separate dependency links, each link connecting a slot in a map (or the map as a whole in the case of the add dependency links) to the compiled code for **min:**.



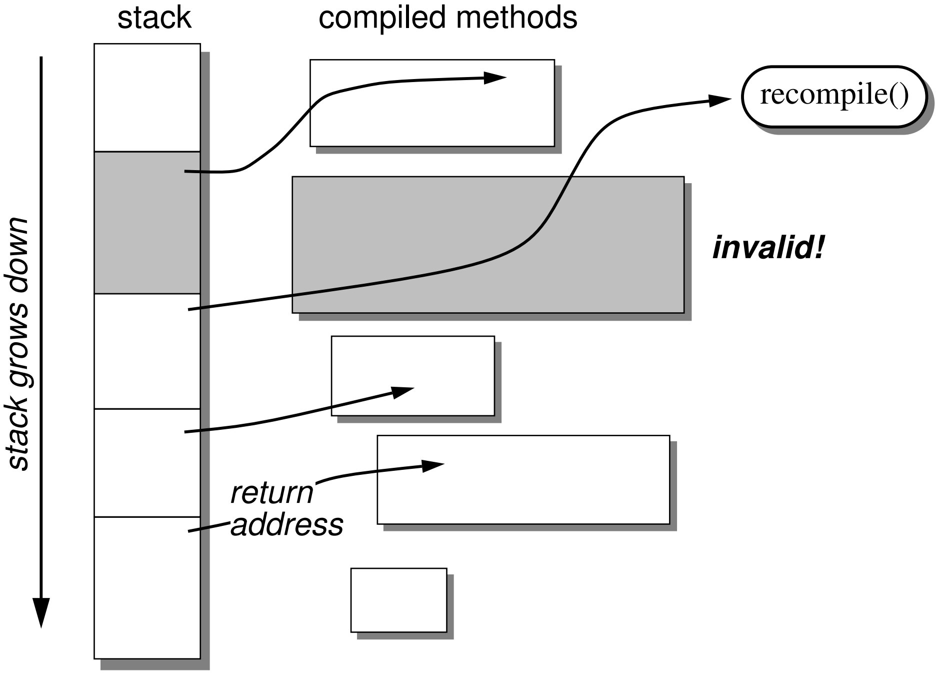
13.2.3 Invalidation

After a programming change, the compiler traverses dependency links to invalidate compiled methods linked to the updated information. Invalidation is normally quite straightforward, simply requiring the invalid compiled method to be thrown out of the compiled code cache. However, if a compiled method is currently running (i.e., if there is a stack frame suspended within the compiled method), then this invalidation becomes complicated. These compiled methods cannot just be flushed, because they are still executing and will be returned to. Nor can they remain untouched, since they have been optimized based on information that is no longer correct. The approach taken in the SELF system is to recompile the out-of-date compiled method and rebuild its stack frame based on the data stored in the old stack frame.*

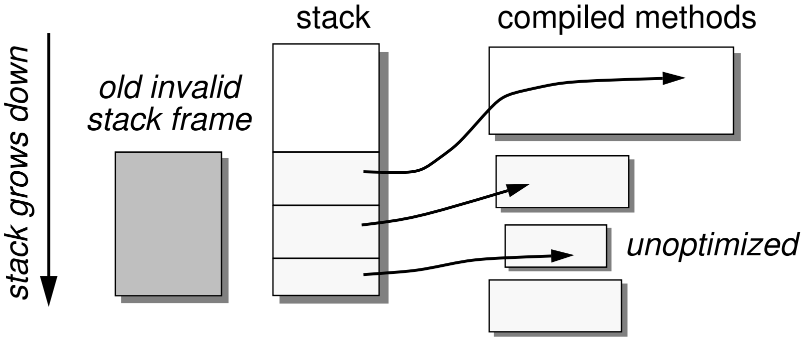
The SELF system performs this conversion *lazily* to make this recompilation easier and less intrusive [HCU92]. When a compiled method suspended on the stack is first invalidated as part of the execution of some programming primitive, the system marks the method as invalid and removes it from the lookup cache (so that future message sends will not be bound to the invalid compiled method), but the system does not yet flush the compiled method from the compiled code cache. Instead the system adjusts the return address of the stack frame that would have returned to the invalid

* The data in the stack frame is still valid. Only the compiled code of the invalidated method is suspect.

method's stack frame to instead "return" to a special support routine in the run-time system. The system then returns control to the programming primitive which will eventually return to the SELF process that invoked it.



Eventually the stack frame below the one for the invalid compiled method will "return," calling the special run-time support routine. This routine recompiles the invalid compiled method and builds new stack frames that represent the same abstract state as did the invalidated compiled method's stack frame, which because of the lazy recompilation is now on the bottom of the stack of SELF activation records. To make it easy to fill in the state for the new stack frame and to keep recompilation pauses short, the new method is compiled without optimization. Since the invalidated compiled method was probably compiled with optimization, including inlining, the system may need to compile several unoptimized methods to represent the same abstract state as the invalidated compiled method, one unoptimized method and physical stack frame for each virtual stack frame inlined into the single physical stack frame of the invalidated compiled method at the point of call.



To complete the conversion process, the recompiling routine returns into the appropriate point in the new compiled method for the topmost stack frame. The invalidated compiled method can be flushed from the compiled code cache if the old invalid stack frame is the last activation of this method.*

Lazy conversion spreads the load of recompilation out across a longer period of time, reducing the perceived pauses after a programming change. If several programming changes occur before returning to an invalid method, then less overall work may be performed since the method will not be recompiled after each programming change. Lazy

* Urs Hölzle implemented the mechanisms to lazily recompile invalid methods on the stack.

conversion also simplifies and speeds the conversion process by limiting recompilation and stack frame creation to the top of the stack. This eliminates the need to copy whole stacks and adjust interior addresses when recompiling and rebuilding some stack frame buried in the middle of the stack.

13.3 Summary

The SELF compiler is designed to coexist with an interactive exploratory programming environment. This kind of environment requires complete source-level debugging to be available at all times and “down time” caused by programming changes to be limited to a few seconds at most. The SELF compiler supports complete source-level debugging in the face of optimizations such as inlining and splitting by generating additional information that allows the debugger to view a single physical stack frame as several source-level virtual stack frames. Fast turn-around time for programming changes is supported by a selective invalidation mechanism based on dependency links that flushes out-of-date compiled methods from the compiled code cache. This invalidation is performed lazily for compiled methods currently executing on the stack.