A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance

Urs Hölzle
Computer Science Department
University of California
Santa Barbara, CA 93106
urs@cs.ucsb.edu

David Ungar
Sun Microsystems Laboratories
2550 Garcia Avenue, Building 29
Mountain View, CA 94043-1100
ungar@eng.sun.com

Abstract: Programming systems should be both responsive (to support rapid development) and efficient (to complete computations quickly). Pure object-oriented languages are harder to implement efficiently since they need optimization to achieve good performance. Unfortunately, optimization conflicts with interactive responsiveness because it tends to produce long compilation pauses, leading to unresponsive programming environments. Therefore, to achieve good responsiveness, existing exploratory programming environments such as the Smalltalk-80 environment rely on interpretation or non-optimizing dynamic compilation. But such systems pay a price for their interactiveness, since they may execute programs several times slower than an optimizing system.

SELF-93 reconciles high performance with responsiveness by combining a fast, non-optimizing compiler with a slower, optimizing compiler. The resulting system achieves both excellent performance (two or three times faster than existing Smalltalk systems) and good responsiveness. Except for situations requiring large applications to be (re)compiled from scratch, the system allows for pleasant interactive use with few perceptible compilation pauses. To our knowledge, SELF-93 is the first implementation of a pure object-oriented language achieving both good performance and good responsiveness.

When measuring interactive pauses, it is imperative to treat multiple short pauses as one longer pause if the pauses occur in short succession, since they are perceived as one pause by the user. We propose a definition of *pause clustering* and show that clustering can make an order-of-magnitude difference in the pause time distribution.

In: OOPSLA ‘94 Conference Proceedings, Portland, OR, October 1994.

1. Introduction

Exploratory programming environments (such as the Smalltalk programming environment) increase programmer productivity by giving immediate feedback for all programming actions. The pause-free interaction allows the programmer to concentrate on the task at hand rather than being distracted by long pauses caused by compilation or linking. Traditionally, system designers have used interpreters or non-optimizing compilers in exploratory programming environments to achieve immediate feedback. For example, commercial Smalltalk implementations use either interpretation [Dig91] or non-optimizing dynamic compilation [DS84, PP92]. Unfortunately, the overhead of interpretation, combined with the efficiency problems created by the high call frequency and the heavy use of dynamic dispatch in pure object-oriented languages, slows down execution and can limit the usefulness of such systems. As a result, computationally intensive Smalltalk programs can be an order of magnitude slower [CU91] than programs written in hybrid object-oriented languages like C++ or conventional languages like C.

In response to this performance problem, previous SELF compilers have concentrated on optimization techniques aimed at reducing the overhead of message passing. The first-generation SELF compiler [CUL89] achieved a respectable speedup over standard Smalltalk implementations. The second-generation compiler [CU91] improved performance even more, bringing SELF’s performance to within a factor of less than two relative to C for a set of small integer benchmarks. However, as larger SELF programs were being written (for example, a graphical user interface [CU93] consisting of 15,000 lines of SELF code), it became increasingly clear that the existing SELF systems had neglected interactive performance. While many programs ultimately ran fast, programmers had to endure compile pauses lasting many seconds while their programs were being optimized. Although turnaround times were still better than in traditional batch-style compilation environments, the SELF system was noticeably more sluggish during program development than commercial Smalltalk systems running on the same hardware.

Language implementors (and thus, the programmers selecting a programming environment) are facing the old dilemma between throughput (execution speed) and latency (interactive responsiveness). Either they can use a very responsive interpreted system and accept inferior execution performance, or they can choose an optimizing system with good execution performance but sluggish interactive performance. Since an interactive programming environment is an important tool in understanding and developing object-oriented programs, programmers are not willing to give up interactive performance, and thus accept inferior execution performance as a given drawback of pure object-oriented languages.

SELF-93 is a step towards solving this dilemma. It provides both good interactive responsiveness and good performance by using a compilation system that dynamically *recompiles* the "hot spots" of an application. It uses a fast, non-optimizing compiler to generate the initial code, and then recompiles only the time-critical parts with a slower, optimizing compiler. Introducing dynamic recompilation dramatically improves interactive performance, making it possible to combine optimizing compilation with an exploratory programming environment.

As described elsewhere [HU94], SELF-93 provides excellent execution-time performance. This paper concentrates on the interactive behavior of the system and shows that it can provide good interactive performance on current workstations and should provide excellent interactive performance (i.e., virtually unnoticeable compile pauses) on future workstations. To the best of our knowledge, SELF-93 is the first implementation of any pure object-oriented language that simultaneously provides high execution performance and good interactive behavior.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the SELF-93 system and its compilation process. Section 3 introduces pause clustering and demonstrates its importance. Section 4 discusses the compilation pauses occurring during an interactive session, and section 6 the delays incurred when starting up new programs. (The appendix discussed the influence of system parameters on performance and shows that, by varying these parameters, one can trade off better pause behavior against better asymptotic performance.) All of the techniques described in this paper are fully implemented and stable enough to be part of the public SELF distribution.†

2. Background

SELF [US87] is a pure object-oriented language: all data are objects, and all computation is performed via dynamically-bound message sends (including accesses to all instance variables, even those in the receiver object). SELF merges state and behavior: syntactically, method invocation and variable access are indistinguishable—the sender of a message does not know whether the message is implemented as a simple data access or as a method. Consequently, all code is *representation independent* since the same code can be reused with objects of different structure, as long as those objects correctly implement the expected message protocol. SELF's pure semantics result in very frequent message sends; in this respect, it is even harder to implement efficiently than Smalltalk.

These implementation difficulties required some unusual compilation techniques [CUL89, CU91, HCU91, HU94]. The following sections briefly review the important aspects of SELF-93's implementation.‡

2.1 Adaptive optimization

The SELF-93 system uses dynamic compilation [DS84]. When a source method is invoked for the first time, it is compiled quickly by a very simple but completely non-optimizing compiler. Conversely, whenever the user changes a source method, all compiled code depending on the old definition is invalidated. To accomplish this, the system keeps *dependency links* between source and compiled methods [HCU92, Ch92]. Since there is no explicit compilation or linking step, the traditional edit-compile-link-run cycle is collapsed into an edit-continue cycle. Programs can be changed while they are running so that the application being debugged need not even be restarted.

† SELF is available via Mosaic URLs http://www.sun.com/smli and http://self.stanford.edu, or via ftp from self.stanford.edu. The system described here is largely identical to the current public release (3.0) but contains several performance improvements.

‡ This description is based on [HU94]; more details can be found in [Höl94].

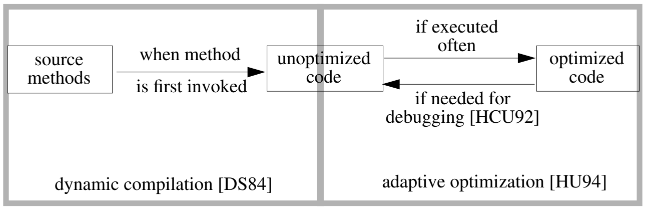


Figure 1. Compilation in the SELF-93 system

Figure 1 shows an overview of the compilation process of the system. In addition to using dynamic compilation to incrementally generate compiled code as needed, SELF-93 uses adaptive optimization to dynamically discover and optimize the "hot spots" of a program. If a method is executed often, it is recompiled with an optimizing compiler. The remainder of this section describes the adaptive optimization process in more detail, outlining how the system discovers methods to be optimized and how these methods are then optimized.

2.2 When to recompile

A dynamic recompilation system needs to decide when to interrupt a program in order to optimize it by recompiling some methods. To be successful, the system needs to strike a balance between compilation and execution. If the system recompiles too eagerly, it will waste time in compilations; if it recompiles too lazily, it will also waste time because programs spend too much time in unoptimized code.

SELF-93 uses *invocation counts* to drive recompilation. Each unoptimized method has its own counter that is incremented in the method prologue. When the counter exceeds a certain limit, the recompilation driver is invoked to decide which method (if any) should be recompiled. If the method overflowing its counter isn't recompiled, its counter is reset to zero. Counter values decay exponentially with time.

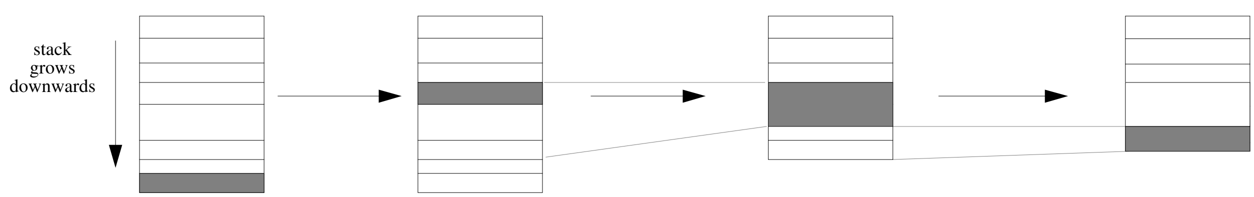
A simple recompilation strategy would always recompile the method whose counter overflowed, since it obviously was invoked often. But suppose that the method just returns a constant. Optimizing this method would not gain much; rather, the method should be inlined into its caller. The next section describes how the system chooses the method(s) to be recompiled.

2.3 What to recompile

To find a "good" candidate for recompilation, the recompilation driver walks up the call chain, inspecting the callers of the method triggering the recompilation. A caller that performs many calls to unoptimized or small methods is recompiled in the hope that these calls will be eliminated. Similarly, a method creating many closure objects (blocks) is recompiled in the hope of eliminating these closure creations.

If a recompilee is found, it is (re)optimized, and the old code is discarded. Then, the reoptimized method replaces the corresponding unoptimized methods on the stack, possibly replacing several unoptimized stack frames with a single optimized stack frame (see Figure 2).† Since the system tries to optimize an entire call chain from the top recompilee down to the current execution point, recompilation continues until all unoptimized stack frames below the original recompilee have been optimized. Usually, the recompiled call chain is only one or two compiled methods deep, so that the program's execution resumes after one or two compilations. In this way, a program's execution speed will gradually improve as more and more of its hot spots are optimized.

† This process is the reverse of dynamic deoptimization as described in [HCU92]; that paper also describes how the compiler represents the source-level state of optimized code. SELF-93 cannot always replace unoptimized with optimized frames (see [HU94]); in such cases, the unoptimized frames are left on the stack until they return.



A method overflows its invocation counter and triggers a recompilation

The system inspects the stack to determine which method to recompile. Then, it calls the compiler to generate new code.

The system replaces the old (unoptimized) stack frames with the frame of the newly compiled method. In the example, it replaces three unoptimized frames with one optimized frame.

The system continues until all of the remaining stack is optimized. Here, it performs one more optimization which replaces the bottom two frames.

Figure 2. Optimization process

2.4 Type Feedback

When recompiling a method, the system extracts type information from the previous version of compiled code and feeds it back to the compiler. (This technique is called *Type Feedback* [HU94].) Specifically, the SELF-93 system uses *Polymorphic Inline Caches* (PICs) [HCU91] to record the program's *type profile*, i.e., a list of receiver types (and, optionally, their frequencies) for every single call site in the program. PICs were originally conceived to speed up dynamic dispatch, but as observed in [HCU91] they record receiver types as a side-effect. Therefore, a program's type profile is readily available, and collecting the type feedback data does not incur an execution time overhead.

Using type feedback, the compiler can optimize any dynamically-dispatched call (if desired) by *predicting* likely receiver types and inlining the call for these types. For example, suppose a method contains the expression `p x` (i.e., send the `x` message to `p`) where `p` is a graphical object. If the type feedback information shows that `p` was a `CartesianPoint` most of the time and only infrequently a `PolarPoint`, the expression could be compiled as

```
if (p->type == CartesianPoint) {
    // inline CartesianPoint case
    <load x instance variable>
} else {
    // don't inline PolarPoint case because method
    // is too big
    // this branch also covers all other receiver types
    <send x to p>
}
```

For `CartesianPoint` receivers, the above code sequence will execute significantly faster since the original message send is reduced to a comparison and a simple load instruction. Inlining not only eliminates the calling overhead but also enables the compiler to optimize the inlined code using dataflow information particular to this call site.

2.5 Performance

Laziness wins in SELF-93: delaying optimization until needed not only saves compilation time, it also allows the optimizing compiler to generate better code than if it had tried to optimize the method right away. With type feedback, the compiler can inline more message sends and thus to achieve better performance than previous compilers. On average, SELF-93 executes a suite of six large (4,000–15,000 lines) and three medium-sized (400–1,100 lines) programs 1.5 times faster than the SELF-91 compiler [HU94]. For the two medium-sized programs that are also available in Smalltalk, SELF-93 is about three times faster than ParcPlace Smalltalk.†

But raw execution performance is not the focus of this paper. Rather, if focuses on how optimizing compilation influences the interactive behavior of a system. Since SELF implementations use runtime compilation because interpretation would be too slow, compile pauses may impact the interactiveness of the system. For example, the first time a menu pops up, the code to draw the menu must be compiled. Runtime compilation can create distracting pauses in such situations. Similarly, dynamic recompilation (as used in SELF-93) may introduce pauses during later executions as code is optimized. The remainder of this paper explores the severity of such compilation pauses with a variety of measurements, such as the pauses experienced in an actual interactive session, the distribution of compile pauses, and compilation speed.

2.6 Measurement methodology

Unless otherwise mentioned, CPU times were measured on an otherwise idle SPARCstation-2. Due to cache-related performance fluctuations, measurements are probably only accurate to within 10-15%. Because the SPARCstation-2 is considered “low end” today (Fall 1994), some data is also given for faster machines. The data in section 4 was obtained using PC sampling, i.e., by interrupting the program 100 times per second and inspecting the program counter to determine whether the system was compiling at the time of the interrupt. The results of these samples were written to a log file. Instrumentation slowed down the system by less than 10%.

3. What is a pause?

One of the main goals of this paper is to evaluate SELF-93's interactive performance by measuring compile pauses. But what constitutes a compile pause? It is tempting to measure the duration of *individual* compilations; however, such measurements would lead to an overly optimistic picture since compilations tend to

† These measurements represent final performance and do not include compilation. In the SELF-93 system, programs like these usually spend less than 20% of their time in unoptimized code [Höl94]; in contrast, the SELF-91 system optimizes all code. The system described here uses slightly different optimization parameters that the system measured in [HU94], reducing performance by about 10%.

occur in clusters. When two compilations occur back-to-back, they are perceived as one pause by the user and thus should be counted as a single long pause rather than two shorter pauses. That is, even if individual pauses are short, the user may notice distracting pauses if many compilations occur in quick succession. Since the goal is to characterize the pause behavior *as experienced by the user*, “pause” must be defined in a way that correctly handles the non-uniform distribution of pauses in time.

Pause clustering attempts to define pauses in such a way. A pause cluster is any time period satisfying the following three criteria:

1. A cluster starts and ends with a pause.
2. Individual pauses consume at least 50% of the cluster’s time. Thus, if many small pauses occur in short succession, they are lumped together into one long pause cluster as long as the pauses consumes more than half of CPU time during the interval. We believe that a limit of 50% is conservative since the system is still making progress at half the normal speed, so that the user may not even notice the temporary slowdown.
3. A cluster contains no pause-free interval longer than 0.5 seconds. If two groups of pauses that would be grouped together by the first two rules are separated by more than half a second, we assume that they are perceived as two distinct pauses and therefore do not lump them together. (It seemed clear to us that two events separated by half a second could be distinguished.)

Figure 3 shows an example. The first four pauses are clustered together because together they use more than 50% of total execution time during that time period (rule 2). Similarly, the next three short pauses are grouped with the next (long) pause, forming a long pause cluster of more than a second. The two clusters won’t be fused into one big 2.5-second cluster (even if the resulting cluster still satisfied rule 2, which it does not in the example) because they are separated by a pause-free period of more than 0.5 seconds (rule 3).

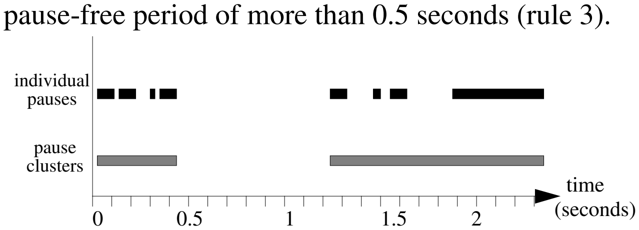


Figure 3. Individual pauses and the resulting pause clusters

This example illustrates that pause clustering is quite conservative and may overestimate the true pauses experienced by the user.† However, we believe that pause clustering is more realistic than measuring individual pauses. Furthermore, we hope that this approach will strengthen our results since the measured pause behavior is still good despite the conservative methodology. We also hope that this work will inspire others (for example, implementors of incremental garbage collectors) to use similar approaches when characterizing pause times.

Figure 4 shows the effect of pause clustering when measuring compile pauses. The graph shows the number of compile pauses that exceed a certain length on a SPARCstation-2. By ignoring pause clustering we could have reported that only 5% of the pauses in SELF-93 exceed 10 milliseconds, and that less than 2% exceed 0.1 seconds. However, with pause clustering 37% of the combined pauses exceed 0.1 seconds. *Clustering pauses makes an order-of-magnitude difference.* Reporting only individual pauses would result in a distorted picture.

Of course, the parameter values of pause clustering (CPU percentage and intergroup time) will affect the results. For example, increasing the pause percentage towards 100% will make the results more optimistic. However, our results are fairly insensitive to changes in the parameter values. In particular, varying the pause percentage between 35% and 70% does not

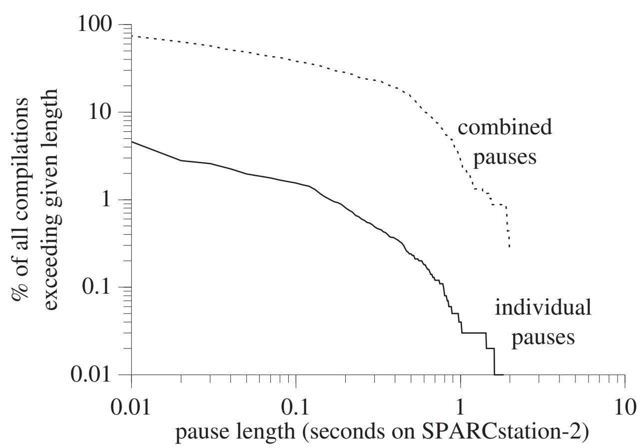


Figure 4. Distribution of individual compile pauses vs. distribution of combined pauses

† Pause clustering may also be too conservative for compilation pauses because it ignores execution speed; a SELF interpreter could be so slow that it causes distracting interaction pauses. For the sake of simplicity, we assume that an interpreter would be fast enough for interactive use.

qualitatively change the results, nor does doubling the intergroup time to one second.

4. Compile pauses during an interactive session

We measured the (clustered) compilation pauses occurring during a 50-minute session of the SELF user interface [CU93]. The session involved completing a SELF tutorial, which includes browsing, editing, and making small programming changes. During the tutorial, a bug in the tutorial's cut-and-paste code was discovered, so that the session also includes some "real-life" debugging. Figure 5 shows the distribution of compile pauses during the experiment in absolute terms. Assuming 200 ms as a lower threshold for perceptible pauses, 195 pauses were perceptible on a SPARCstation-2. Similarly, using one second as the lower threshold for distracting pauses, there were 21 such pauses during the 50-minute run. Almost two thirds of the measurable pauses† are below a tenth of a second, and 97% are below one second.

Pause clustering addresses the short-term clustering of compile pauses. However, pauses are also non-uniformly distributed on a larger time scale. Figure 6 (on the next page) shows how the same pauses are distributed over the 50-minute interaction. Each pause is represented as a spike whose height corresponds to the (clustered) pause length; the x axis shows elapsed time. Note that the x axis' range is much larger than the y axis' range (by three orders of magnitude) so that the graph visually exaggerates both the spikes' height and proximity.

† Since we obtained the data by sampling the system at 100 Hz, very short compilations were either omitted or counted as a pause of 1/100 second.

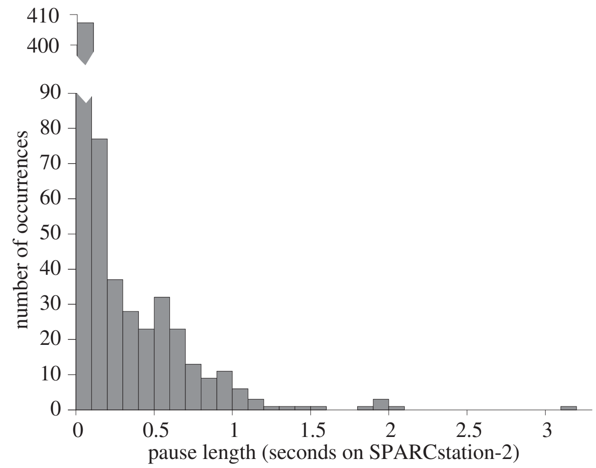


Figure 5. Compile pauses during a 50-minute interaction

During the run, several substantial programs were started from scratch (i.e., without precompiled code). The initial peak that includes the highest pause corresponds to starting up the user interface. The next cluster represents the first phase of the tutorial, where much of the user interface code is exercised for the first time. The last two clusters correspond to invoking the SELF debugger after discovering a bug, and inspecting the state of the tutorial process to find the cause of the error. The the entire session contains few substantial "think pauses"; thus, periods with no compilation pauses are not just idle periods.

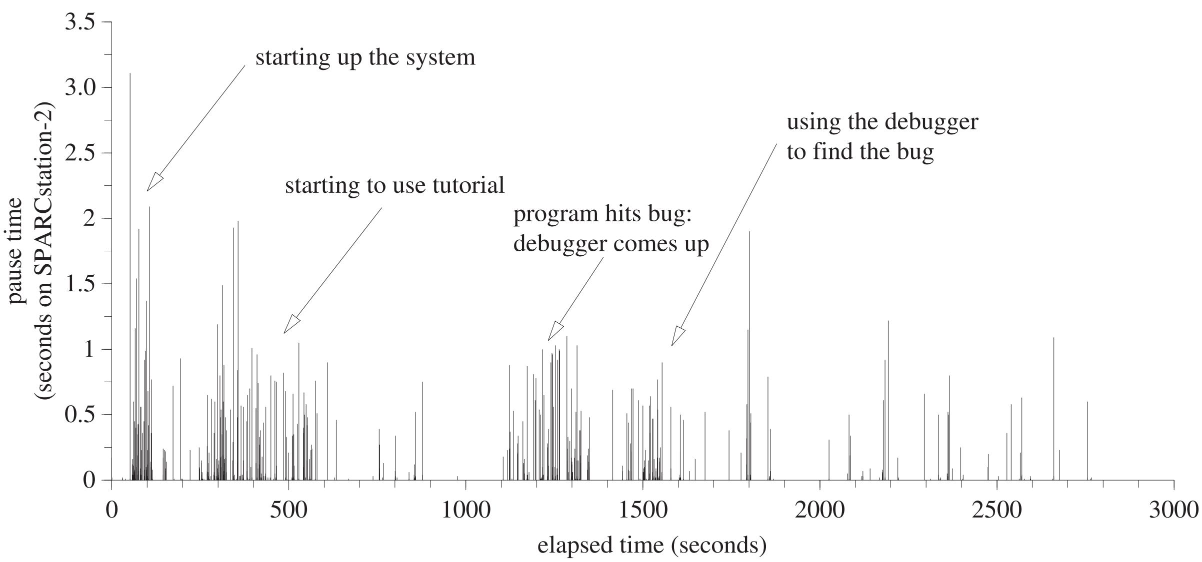


Figure 6. Distribution of compilation pauses over time

The system's pause behavior on the SPARCstation-2 seems adequate. Pauses are rarely distracting and much shorter than in traditional programming environments using batch-style compilation. Furthermore, the measured interaction represents a worst-case scenario since it starts a large system (with an estimated 20,000 lines of SELF code) from scratch, without any precompiled code. During normal usage of the system, most of the standard system (user interface, debugger, etc.) is already optimized, and only the application that the programmer is actively changing needs to be (re)compiled.

5. Pauses on faster systems

The practicality of optimizing compilation in an interactive system is strongly dependent on CPU speed. A system significantly slower than the 20-SPECInt92 SPARCstation-2 would probably make pause times too distracting. That is, our system would have been impractical on the machines commonly in use when the Deutsch-Schiffman Smalltalk compiler was developed, since they were at least an order of magnitude slower.

On the other hand, the system's interactive behavior will improve with faster CPUs. Today's workstations and high-end PCs are already significantly faster than the SPARCstation-2 used for our measurements (see Table 1) To investigate the effect of faster CPUs, we reanalyzed our trace with parameters chosen to represent a current-generation workstation or PC (three times faster than a SPARCstation-2) and a future workstation† (ten times faster). Figure 7 compares the SPARCstation-2 pauses with those on the two simulated systems. For each pause length, the graph shows the number of pauses exceeding that length. Note that the graph for a faster machine is not just the original graph shifted to the left, because it may combine compilations that were not combined in the slower system. For example, if two groups of compilations are 1 second apart on the SPARCstation-2, they are only 0.33 seconds apart on the "current" workstation and thus must be combined into a single pause (see the rules in section 3). However, the overall impact of this effect is quite small, which confirms our earlier observation that pause clustering is fairly insensitive to value of the interpause time parameter.

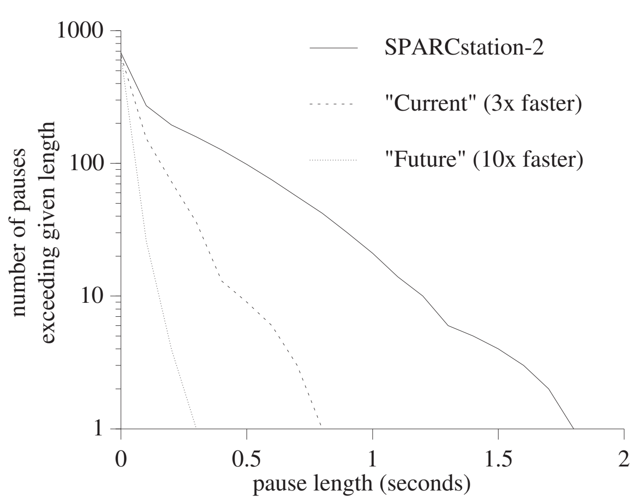


Figure 7. Compilation pauses on faster CPUs

System	SPECInt92 (higher is faster)	
	absolute	relative to SS-2
SPARCstation-2	22	1.0
66MHz Pentium PC	65	3.0
80 MHz PowerPC Macintosh	63	2.9
SPARCstation-20/61	89	4.0
1994 high-end workstation	135	6.1
expected in 1995	>200	>9.1

Table 1: Speed of workstation and PCs

On the current-generation workstation, only 13 pauses exceed 0.4 seconds. These numbers confirm our informal experience of SELF running on current-generation SPARCstation-10 machines: pauses sometimes are still noticeable, but they are rarely distracting. The even faster next-generation workstation will eliminate virtually all noticeable pauses: only 4 pauses will be longer than 0.2 seconds. On such a machine, the current SELF system should feel like an "optimizing interpreter." (However, pauses may still not be short enough for real-time animation or video, since the human eye can spot pauses of less than 0.1 seconds in such situations.)

Arguments of the kind "with a faster machine, everything will be better" are often misleading, especially if they assume that everything else (e.g., the problem size or program size) remains constant. We believe that our argument escapes this fallacy because the length of individual compilations does not depend on program size or program input but on the size of a compilation unit, i.e., the size of a method (and any methods inlined into it during optimization). Unless people's programming styles change, the average size

† Workstation vendors are expected to announce 200-SPECint workstations by the end of this year, so that even these "future" workstations should be available in 1995.

of methods will remain the same, and thus individual pauses will become shorter with faster processors. Furthermore, as already noted above, we did not simply divide pause times by the speedup factor but reanalyzed the trace, combining individual pauses into longer pauses by correctly accounting for the shorter inter-pause times. Larger programs may prolong the time needed for recompilation to settle down (see section 6.2 below), but our experience is that program size does not influence the clustering of individual compilations. In other words, while larger programs may cause more pauses, they do not lengthen the pauses themselves. Therefore, we believe that it is safe to predict the interactiveness of the system as perceived by the user will improve with faster processors, as shown in Figure 7.

To summarize, one could characterize the compilation pauses of the SELF-93 system as noticeable but only occasionally distracting on previous-generation systems, sometimes noticeable but almost never distracting on current-generation systems, and virtually unnoticeable on next-generation systems.

6. Starting new code

A responsive system should be able to quickly start up new (as yet uncompiled) programs or program parts. For example, the first time the user positions the cursor in an editor, the corresponding editor code has to be compiled. Starting without precompiled code is similar to continuing after a programming change, since such a change invalidates previously-compiled code. For example, if the programmer changes some methods related to pop-up menus and then tries to test the change, the corresponding compiled code must be generated first. Thus, by measuring the time needed to complete small program parts (e.g., user interface interactions) without precompiled code, one can characterize the behavior of the system during a typical debugging session where the programmer changes source code and then tests the changed code.

To measure the effect of adaptive optimization, several systems were used (Table 2). Comparing the standard SELF-93 system to a version which always optimizes all code (SELF-93-norecomp) shows the direct effect of adaptive optimization. The previous SELF system (SELF-91) was included as a reference point.

System	Description
SELF-91	Chambers' SELF compiler [Cha92]; all methods are always optimized from the beginning.
SELF-93	The current SELF system using dynamic recompilation; methods are compiled by a fast non-optimizing compiler first, then recompiled with the optimizing compiler if necessary.
SELF-93-norecomp	Same as SELF-93, but without recompilation; all methods are always optimized from the beginning.

Table 2: Systems used in the study of start-up times

6.1 Starting small programs

In order to evaluate the behavior of start-up situations, we measured the time taken by a sequence of common user interactions such as displaying an object or opening an editor. At the beginning of the interaction sequence, all compiled code was removed from the code cache. Table 3 shows the individual interactions of the sequence. For the measurements, the interactions were executed in the sequence shown in the table, with no

	Description	execution time (compile + run) (seconds on SPARCstation-2)		
		S-91	S-93-NR	S-93
1	start user interface, display initial objects	91.9	42.2	**26.3**
2	dismiss standard editor window	11.5	4.7	**1.5**
3	dismiss lobby object	0.8	0.7	**0.5**
4	show slot "thisObjectPrints" of a point object	2.3	0.7	**0.6**
5	open editor on slot's comment	8.1	3.2	**2.4**
6	dismiss editor	0.2	0.4	**0.5**
7	sprout x coordinate (the integer 3)	11.4	4.4	**2.0**
8	sprout 3's parent object (traits integer)	2.5	1.2	**2.4**
9	display "+" slot	7.5	2.6	**1.5**
10	sprout "+" method	11.8	4.2	**2.9**
11	dismiss "+" method	3.0	1.5	**1.0**
12	open editor on "+" method	4.6	2.2	**1.7**
13	change "+" method (changing the definition of integer addition)	82.4	31.0	**13.9**
14	reopen editor on "+" method[a]	28.2	12.0	**6.9**
15	undo the previous change (changing the definition of integer addition back to the original definition)	82.0	30.7	**15.0**
16	dismiss traits smallInteger object	9.8	4.0	**1.0**
	geometric mean of ratios to SELF-93	340%	160%	100%
	median	400%	160%	100%

Table 3: UI interaction sequence

[a] same action as no. 12, but slower because of the preceding change (see text)

other activities in-between except for trivial actions such as placing the cursor in the text editor. Although the sequence starts with an empty code cache, an interaction may reuse some code compiled for previous interactions; for example, the first action (starting up the user interface) will generate code for drawing methods used by most other interactions. All times are total execution times, i.e., the sum of execution time and compile time.

SELF-93 usually executes the interactions fastest; on average, it is 3.4 times faster than SELF-91 and 1.6 times faster than SELF-93-norecomp (geometric means; the medians are 4.0 and 1.6). However, there are fairly large variations: some tests run much faster with SELF-93 (e.g., number 16 is 9.8 and 4.0 times faster, respectively), but a few tests (e.g., number 6) run slower than in the other systems.

Several factors contribute to these results. SELF-93 is usually fastest because the non-optimizing compiler saves compilation time. However, dynamic recompilation introduces a certain variability in running times, slowing down some interactions. This can happen when recompilation is too timid (so that too much time is spent in unoptimized code) or when it is too aggressive (so that some methods are recompiled too early and then have to be recompiled again). SELF-93-norecomp is faster than SELF-91 because type feedback allowed its design to be kept simpler without compromising the performance of the compiled code.

Rows 13 to 15 show how quickly the system can recover from a massive change: in both cases, a large amount (300 Kbytes) of compiled user interface code needed to be regenerated after the definition of integer addition had changed. Since the integer addition method is small and frequently used, it was inlined into many compiled methods, and all such compiled methods were discarded after the change. Adaptive recompilation allowed the system to recover in less than 15 seconds (the user interface consists of about 15,000 lines of code, not counting general system code such as collections, lists, etc.). This time included the time to accept (parse) the changed method, dismiss the editor, update the screen, and react to the next mouse click. Compared to SELF-93-norecomp, dynamic optimization buys a speedup of 2 in this case; compared to SELF-91, the speedup is a factor of 5 to 6. Of course, subsequent interactions may also be slower as a result of the change because code needs to be recreated. For example, opening an editor (row 14) takes four to six times longer than before the change (row 12).

With adaptive optimization, small pieces of code are compiled quickly, and thus small changes to a program can be handled quickly. Compared to the previous SELF system, SELF-93 incurs significantly shorter pauses; on average, the above interactions run three to four times faster.

6.2 Starting large programs

The previous section characterized the pauses caused by (re-)compiling small pieces of code. This section investigates what happens when large programs must be compiled from scratch. Table 4 lists the large applications used for the study. The programs were started

Benchmark	Size (lines)[a]	Description
CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function
CecilInt	9,000	interpreter for the Cecil language [Cha93] running a short test program
Typeinf	8,600	type inferencer for SELF [APS93]
UI1	15,200	prototype user interface using animation techniques [CU93][b]

Table 4: Large SELF applications

[a] Excluding blank lines.
[b] Time excludes the time spent in graphics primitives

with an empty code cache and then repeatedly executed 100 times. Although the programs are fairly large, the test runs were kept short, at about 2 seconds for optimized code. Thus, the first few runs are dominated by compilation time since a large body of code is compiled and optimized (Figure 8). For example, Typeinf's first run takes more than a minute, whereas the tenth run takes less than three seconds. After a few runs, the compilations die down and execution time

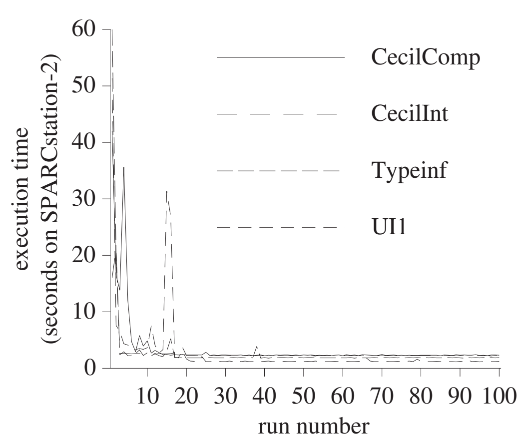


Figure 8. Start-up behavior of large applications

becomes stable for all programs except `UI1` which experiences another flurry of compilation around run 15.

Note that the shape of the graph (i.e., the height of the initial peak) is strongly influenced by the (asymptotic) length of the test runs; had we chosen to use ten-second test runs, the picture would be quite different. Figure 9 shows the same data as Figure 8, except that five successive runs were treated as one, simulating test runs of about 15 seconds duration. Suddenly, the initial peak in execution time looks much smaller even though the system's behavior has not changed.

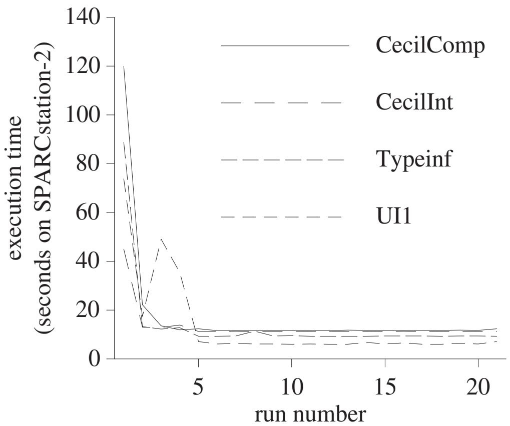


Figure 9. Alternate view of figure 8

Table 5 characterizes the start-up behavior of the system depending on program size and execution time. If programs are small, so is the start-up time, and thus the start-up behavior is good. If programs run for a long time, start-up behavior is good as well, since the initial compilations are hidden by the long execution time. However, if large programs execute only for a short time, the start-up costs of dynamic compilation cannot be hidden in the current SELF system. Our benchmarks all fall in this category because their inputs were chosen to keep execution times short.† In real life, one might expect large programs to run longer, and thus start-up behavior would be better than with our benchmarks.

		program size	
		small	big
execution time	short	good	not good
	long	good	good

Table 5: Start-up behavior of dynamic compilation

† The benchmarks (and their inputs) were originally chosen to measure execution performance using an instruction-level simulator, and most of them run for only one or two seconds on a SPARCstation-2.

The programs' start-up time should correlate with program size: one would expect larger programs to take longer to reach stable performance since more code has to be (re)compiled. Figure 10 shows the "stabilization time" of several large SELF programs, plotted against the programs' sizes. (The stabilization time is the compilation time incurred by a program until it reaches the "knee" of the initial start-up peak.‡) As expected, some correlation does exist: in general, larger programs take longer to start. However, the correlation is not perfect, nor can it expected to be. For example, a large program that spends most of its time in a small inner loop will quickly reach good performance since only a small portion needs to be optimized.

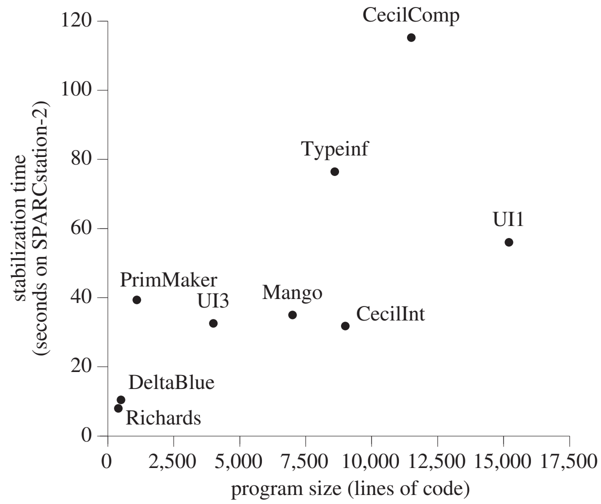


Figure 10. Correlation between program size and time to stable performance

What exactly causes the first few runs to be so slow? Figure 11 (on the next page) breaks down the start-up phase of the programs into compilation and execution. Most of the initial runs of `UI1` consists of non-optimizing compilations and slow-running unoptimized code; in `UI1`, optimizing compilation never dominates execution time.†† To reduce the initial peak in execution time for `UI1`, the non-optimizing compiler would have to be substantially faster and generate better code. In contrast, optimizing compilation dominates the start-up phase of `Typeinf` and (to a lesser extent) `CecilComp`. `CecilInt` lies somewhere in-between—optimizing compilation consumes only a mi-

‡ The knee was determined informally since we were interested in a qualitative picture and not in precise quantitative values.

††UI1 spends a relatively high percentage of time in unoptimized code because it uses dynamic inheritance, a unique SELF feature that allows programs to change their inheritance structure at run-time. Dynamic inheritance is currently not handled well by the recompilation system.

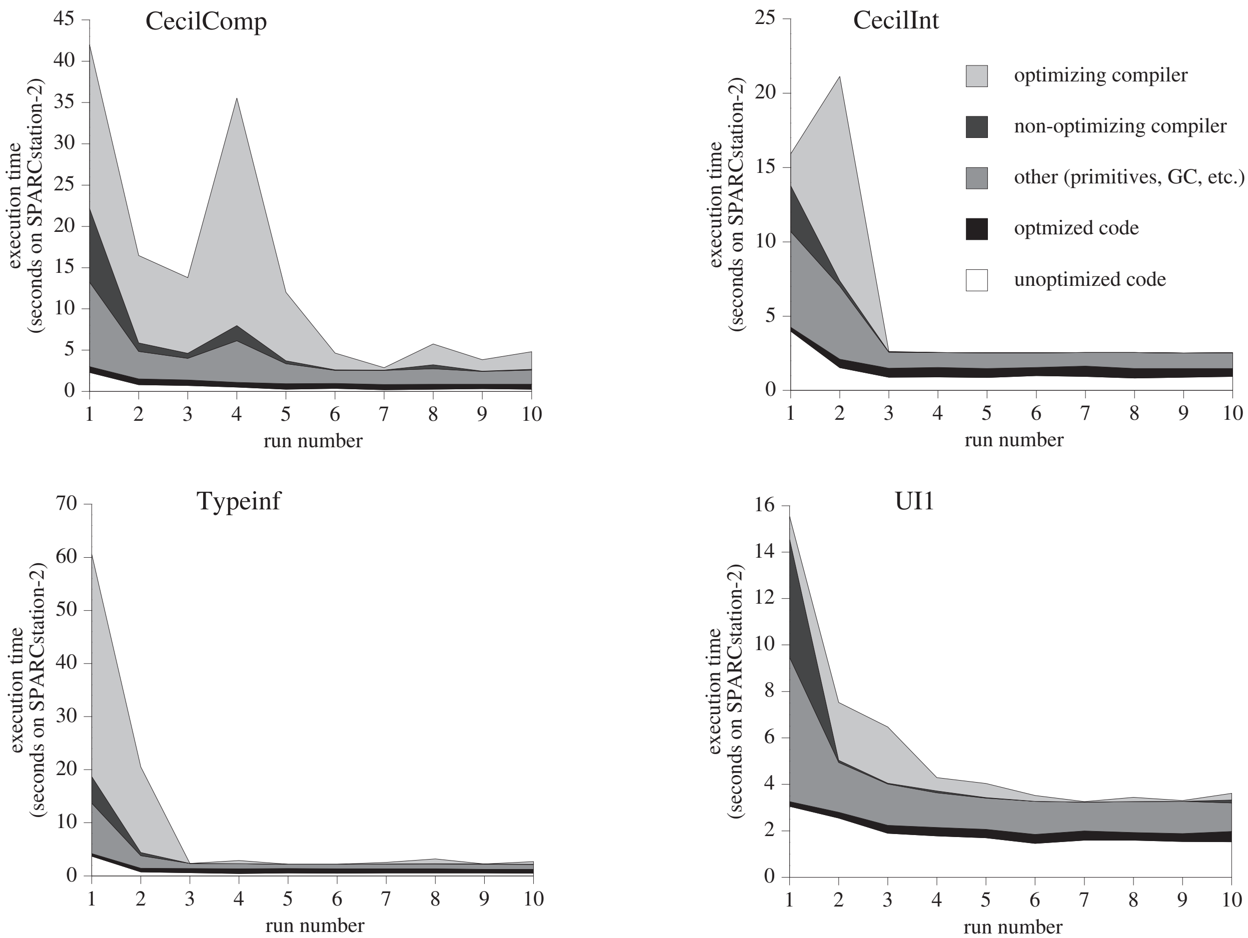


Figure 11. Start-up phase of selected benchmarks

nor portion of the first run but a major portion of the second run. In summary, the reasons for the high initial execution time vary from benchmark to benchmark, and there is no single bottleneck.

The initial slowdown experienced when starting large applications need not be a problem when delivering finished applications to users, since precompiled code could be stored on disk and loaded on demand. (Only optimized code needs to be stored since unoptimized code can be created quickly enough on the fly.) However, during program development, most of an application's code may be discarded after a significant change, and thus it is still important that the system can start up large programs reasonably quickly. On a SPARCstation-2, a 10,000-line program approaches stable performance after about one to two minutes in SELF-93 (see Figure 10), which is adequate.

7. Related work

One of the first people concerned with the implementation of efficient but flexible programming systems was Mitchell [Mit70]. His design (the system was not actually implemented) mixed interpretation and compilation: code was first interpreted, but subsequent executions used compiled code that was generated as a side-effect of interpretation.

Hansen [Han74] describes an adaptive compiler for Fortran. His compiler optimized the inner loops of Fortran programs at runtime. The main goal was to minimize the total cost of running a program (which presumably was executed only once), and programs were run batch-style. The system tried to allocate compile time wisely in order to minimize total execution time, i.e. the sum of compile and runtime; being a batch system, interactive pauses were not an issue.

The Deutsch-Schiffman Smalltalk system [DS84] (and its commercial successor, ParcPlace Smalltalk [PP92]) were the first object-oriented systems to use dynamic compilation. Using a very fast non-optimizing compiler, the system achieves excellent interactive performance; compilation pauses are virtually unnoticeable on current hardware. Smalltalk is easier to compile with a non-optimizing compiler than SELF since Smalltalk "hardwires" important control structures (such as `if` and `while`) whereas SELF doesn't [Höl94]. However, compared to SELF-93, unoptimized Smalltalk code runs roughly three times slower on the programs measured in [HU94], demonstrating the limits of non-optimizing compilation.

Lisp systems have long used a mixture of interpreted and compiled code (or optimized and unoptimized compiled code). However, the user usually has to manually compile programs. Since the transition from unoptimized (interpreted) code to optimized (compiled) code is not automatic, such systems cannot directly be compared to systems using dynamic recompilation.

Some commercial implementations of Eiffel [ISE93] and C++ [SGI93] can handle the reverse transition (from compiled to interpreted) automatically. That is, after a source method is changed, the compiled code is no longer executed and the method is interpreted until the programmer recompiles that part of the program.† Of course, the affected code will run much more slowly when interpreted, so that this approach is only practical for code that is not executed too often.

8. Conclusions

Like other languages, object-oriented languages need both good runtime performance and good interactive performance. Pure object-oriented languages make this task harder since they need aggressive optimization to run at acceptable speed, thus compromising interactive performance. With adaptive recompilation, a system can provide both good runtime performance and good interactive performance, even for a pure object-oriented language like SELF. On a SPARCstation-2, fewer than 200 pauses exceeded 200 ms during a 50-minute interaction with the system, and 21 pauses exceeded one second. With faster CPUs, compilation pauses should start becoming unnoticeable: on a next-generation workstation (likely to be available in 1995), no pause would exceed 400 ms, and only four pauses would exceed 200 ms.

When discussing pause times, it is imperative to measure pauses as they would be experienced by a user. Pause clustering achieves this by combining consecutive short pauses into one longer pause, rather than just measuring individual pauses. Applying pause clustering to the compilation pauses of the SELF-93 system changes the pause distribution by an order of magnitude, emphasizing the importance of pause clustering. We believe that pause clustering should be used whenever pause length is important, for example, when evaluating incremental garbage collectors.

Adaptive recompilation also helps to improve the system's responsiveness to programming changes. For example, it takes less than 15 seconds on a SPARCstation-2 for the SELF user interface to start responding to user events again after the radical change of redefining the integer addition method (which invalidates all compiled code that has inlined integer addition).

In the future, it should be possible to hide compilation pauses even better than the current SELF-93 system does. With dynamic recompilation, optimization is "optional" in the sense that the optimized code is not needed immediately. Thus, if the system decides that a certain method should be optimized, the actual optimizing compilation could be deferred if desired. For example, the system could enter the optimization requests into a queue and process them during the user's "think pauses" (similar to opportunistic garbage collection [WM89]). Alternatively, optimizing compilations could be performed in parallel with program execution on a multiprocessor machine.

Adaptive recompilation gives implementors of pure object-oriented languages such as Smalltalk a new option for implementing their systems. With the increasing speed of hardware, interpreters or unoptimizing dynamic compilers (as used in current Smalltalk systems) may no longer represent the optimal compromise between performance and responsiveness. Today, it is practical to push for better performance—thus widening the applicability of such systems—without forfeiting responsiveness. We hope that this paper will encourage implementors of object-oriented languages to explore a new region in the design space, resulting in new high-performance exploratory programming environments for object-oriented languages.

† The SELF system uses a similar mechanism to provide source-level debugging of optimized code and to allow the programmer to change programs while they are running [HCU92].

Acknowledgments: The authors would like to thank Lars Bak, Bay-Wei Chang, Brian Lewis, John Maloney, and the anonymous OOPSLA referees for their comments on earlier versions of this paper.

The first author has been generously supported by Sun Microsystems Laboratories. The SELF project has been supported by Sun Microsystems, National Science Foundation PYI Grant #CCR-8657631, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

References

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993.

[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*, p. 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.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Conference Proceedings*, Phoenix, AZ, October 1991.

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

[Cha93] Craig Chambers. The Cecil Language - Specification and Rationale. University of Washington, Technical Report UW CS TR 93-03-05, 1993.

[CU93] Bay-Wei Chang and David Ungar. Animation: From Cartoons to the User Interface. *User Interface Software and Technology Conference Proceedings*, Atlanta, GA, November 1993.

[Dig91] Digitalk Inc. Smalltalk/V system, 1991.

[DS84] L. Peter Deutsch and Alan Schiffman, "Efficient Implementation of the Smalltalk-80 System." *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[Han74] Gilbert J. Hansen, *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. Ph.D. Thesis, Carnegie-Mellon University, 1974.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin, 1991.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, p. 32-43. Published as *SIGPLAN Notices 27(7)*, July 1992.

[Höl94] Urs Hölzle. Adaptive Optimization in SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Department of Computer Science, Stanford University, 1994. (Available via ftp/http from self.stanford.edu.)

[HU94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, p. 326-336. Published as *SIGPLAN Notices 29(6)*, June 1994.

[ISE93] ISE Inc. ISE Eiffel 3.0, 1993.

[Mit70] J. G. Mitchell, *Design and Construction of Flexible and Efficient Interactive Programming Systems*. Ph.D. Thesis, Carnegie-Mellon University, 1970.

[PP92] ParcPlace Systems. VisualWorks 1.0, 1992.

[SM+93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience 23 (5)*: 529-566, 1993.

[SGI93] Silicon Graphics, Inc. SGI C++, 1993.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, p. 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.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *OOPSLA '89 Conference Proceedings*, pp. 23-35, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

Appendix. Influence of recompilation parameters on performance

SELF-93's recompilation system is governed by several configuration parameters that influence how aggressively methods are recompiled and optimized. This appendix describes two of those parameters (recompilation limit and half-life time) and shows how they influence the behavior of the system. By varying these parameters, it is possible to trade better interactive behavior for execution speed and vice versa.

As mentioned before, unoptimized methods have invocation counters. If a counter exceeds the *recompilation limit*, the recompilation system is invoked to determine if optimization is necessary. Thus, lower limits will lead to more aggressive recompilation (since more methods will reach the limit); the current system uses a limit of 10,000.

Invocation counters decay exponentially; the decay rate is given as the *half-life time*, i.e., the time after which a counter loses half of its value. Without decay, every method would eventually reach the invocation limit and would be recompiled even though it might not execute more often than a few times per second, so that optimizing it would hardly bring any benefits. Exponential counter decay is implemented by periodically dividing the counters by a constant p; for example, if the process adjusting the counters wakes up every 4 seconds and the half-life time is 15 seconds, the constant factor is $p = 1.2$ (since $1.2^{15/4} = 2$). The decay process converts the counters from invocation counts to invocation rates: given invocation limit N and decay factor p, a method has to execute more often than $N * (1 - 1/p)$ times per decay interval to trigger a recompilation.†

While searching for a good configuration for our standard system, we experimented with a wide range of parameter values. Although there appeared to be no hard rules (i.e., rules without exceptions) to predict the influence of parameter changes, two clear trends emerged:

- Letting invocation counts decay significantly reduces compile pauses by reducing the number of recompilations. The closer the half-life time is to infinity (i.e., no decay), the more variable the execution times become, and the longer it takes for performance to stabilize.
- Increasing the recompilation limit (i.e., the invocation count value at which a recompilation is triggered) lengthens the start-up phase because more time is spent in unoptimized code. However, it does not always reduce compile pauses.

Counter decay has the most influence on compile pauses; in particular, the difference between some decay and no decay is striking. Figure 12 shows the exe-

† Assume a method's count is C just before the decaying process wakes up. Its decayed value is C/p, and thus it has to execute $C * (1 - 1/p)$ times to reach the same count of C before the decay process wakes up again. Since the method eventually needs to reach $C = N$ to be recompiled, it must execute at least $N * (1 - 1/p) + 1$ times during a decay interval. p can be derived from half-life L and decay interval D (4 seconds) as $p = 2^{D/L}$.

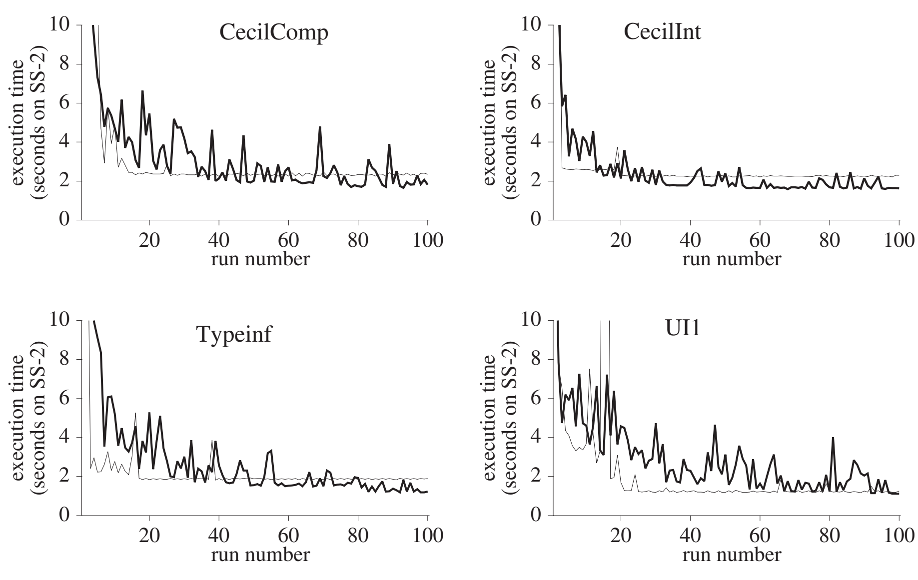


Figure 12. Performance variations if invocation counters don't decay (thin lines: standard system (15 second half-life), thick lines: system with no decay)

cution times of 100 repetitions of the programs shown in Figure 11, comparing the standard system (with a half-life time of 15 seconds) to a system with no counter decay. All programs show significantly higher performance variations if counters do not decay; these variations are caused by recompilations. Also, several of the programs do not seem to converge towards a stable performance level within 100 iterations. Intuitively, the reason for this behavior is clear: without decaying invocation counts, every single method will eventually exceed the recompilation threshold, and thus will be optimized. That is, performance only stabilizes when there are few methods left to recompile.

Figure 12 shows another interesting effect: for three of the four programs, asymptotic performance improves when counters are not decayed, presumably because more methods are optimized. To measure this effect, we varied invocation limit and half-life time and measured the resulting execution time. For each combination of parameters, the lowest execution time out of 100 repetitions of a benchmark was chosen and normalized to the best time achieved with any parameter configuration for that benchmark. That is, the parameter configuration resulting in the best performance for a particular benchmark receives a value of 100%; a value of 150% for another parameter combination would mean that this combination results in an execution time that is 1.5 times longer than that of the best parameter combination.

Figure 13 shows the resulting performance profile, averaged over all benchmarks (the data was clipped at z = 200%; the true value for the worst parameter combination is over 1100%). Overall, the two parameters behave as expected: increasing the invocation limit and decreasing half-life both increase execution time because a smaller part of the application is optimized since fewer methods are recompiled. However, the performance profile is "bumpy," showing that performance does not vary monotonically as one parameter is varied. The bumps are partly a result of measurement errors (recall that variations caused by cache effects on the SPARCstation-2 can be as high as 15%) and partly a result of an element of randomness introduced by using timer-based decaying. Since the timer interrupts governing the counter-decaying routine do not always arrive at exactly the same points during the program's execution, a method may be optimized in one run (e.g., with a half-life time of 4 seconds) but not in another run (with half-life time of 8 seconds) because there the counters are always decayed in time before they can trigger a recompilation. This explanation is consistent with the fact that the bumps are not exactly reproducible (i.e., the bumpiness is reproducible, but the exact location and height of the bumps is not).

These measurements show that one can vary the recompilation parameters within certain bounds without affecting performance too much. Recall that the point with the best asymptotic performance (e.g., no counter decay and an invocation limit of 5,000) is not be the best overall choice since interactive performance suffers with such parameters (see Figure 12). Since the performance profile is fairly flat near the optimal asymptotic performance point, it is possible to trade around 10% execution speed for much better interactive behavior.

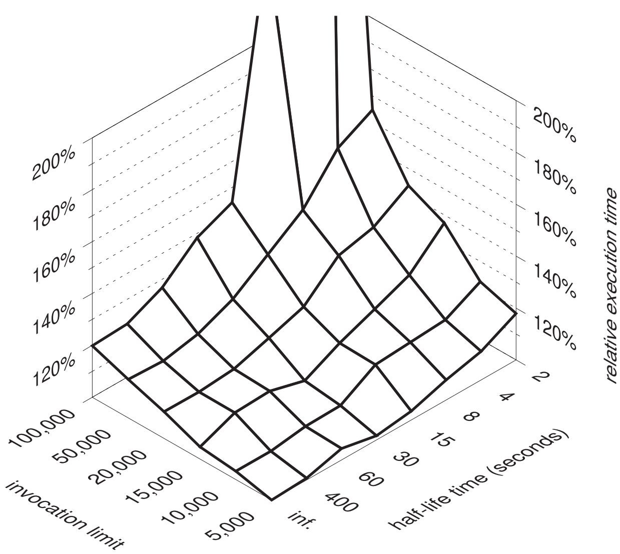


Figure 13. Influence of recompilation parameters on performance