Reconciling Responsiveness with Performance in Pure Object-Oriented Languages

URS HÖLZLE
University of California, Santa Barbara

and

DAVID UNGAR
Sun Microsystems Laboratories

Dynamically-dispatched calls often limit the performance of object-oriented programs since object-oriented programming encourages factoring code into small, reusable units, thereby increasing the frequency of these expensive operations. Frequent calls not only slow down execution with the dispatch overhead per se, but more importantly they hinder optimization by limiting the range and effectiveness of standard global optimizations. In particular, dynamically-dispatched calls prevent standard interprocedural optimizations that depend on the availability of a static call graph.

The SELF implementation described here offers two novel approaches to optimization. *Type feedback* speculatively inlines dynamically-dispatched calls based on profile information that predicts likely receiver classes. *Adaptive optimization* reconciles optimizing compilation with interactive performance by incrementally optimizing only the frequently-executed parts of a program. When combined, these two techniques result in a system that can execute programs significantly faster than previous systems while retaining much of the interactiveness of an interpreted system.

Categories and Subject Descriptors: D.1.5 [Programming Languages]: Object-oriented programming—SELF; D.2.6 [Programming Languages]: Programming Environments—Interactive programming environments, exploratory programming environments; D.2.m [Programming Languages]: Miscellaneous—Rapid prototyping; D.3.4 [Programming Languages]: Processors—Compilers, Interpreters, Run-time environments; D.4.7 [Programming Languages]: Organization and Design—Interactive systems.

General terms: Compilers, Languages, Programming Environments

Additional Keywords and phrases: type feedback, profile-based optimization, adaptive optimization, pause clustering, run-time compilation

Earlier versions of parts of this article appeared at PLDI ‘94 and OOPSLA ‘94. This work was performed while the first author was with Stanford University and Sun Microsystems Laboratories.

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.

Author’s addresses: Urs Hölzle, Department of Computer Science, University of California, Santa Barbara, CA 93106, urs@cs.ucsb.edu; David Ungar, Sun Microsystems Laboratories, 2550 Garcia Ave., Mountain View, CA 94043-1100, ungar@eng.sun.com.

1. Introduction

Object-oriented programming is becoming increasingly popular because it makes programming easier. It allows the programmer to hide implementation details from the object's clients, turning each object into a member of an abstract data type whose concrete operations and state are encapsulated behind a message-passing interface. *Late binding* greatly enhances the power of abstract data types by allowing different implementations of the same abstract data type to be used interchangeably at run time. That is, an invocation of an access to or an operation on an object does not precisely specify which function is executed as a result of the invocation since late binding (*dynamic dispatch*) selects the appropriate implementation of the operation based on the object's exact type. As encapsulation and dynamic dispatch become pervasive, the resulting code gains flexibility and reusability. Ideally, programmers should use late binding even for very basic operations such instance variable access. Since late binding is so beneficial to object-oriented programming, compilers need to implement it as efficiently as possible.

But unfortunately, late binding creates efficiency problems: Object-oriented programs are harder to optimize than programs written in languages like C or Fortran, for two reasons. First, object-oriented programming encourages code factoring and differential programming; as a result, procedures are smaller and procedure calls more frequent. Second, it is hard to optimize calls because they use *dynamic dispatch*: the procedure invoked by the call is not known until run time because it depends on the dynamic type of the receiver. Therefore, a compiler usually cannot apply standard optimizations such as inline substitution or interprocedural analysis to these calls.

Consider the following example (written in pidgin C++):

```
class Point {
   virtual float get_x();                    // get x coordinate
   virtual float get_y();                    // ditto for y
   virtual float distance(Point p);          // compute distance between receiver and p
}
```

When the compiler encounters the expression `p->get_x()`, where `p`'s declared type is `Point`, it cannot optimize the call because it does not know `p`'s exact run-time type. For example, there could be two subclasses of `Point`, one for Cartesian points and one for polar points:

```
class CartesianPoint : Point {
   float x, y;
   virtual float get_x() { return x; }
   (other methods omitted)
}

class PolarPoint : Point {
   float rho, theta;
   virtual float get_x() { return rho * cos(theta); }
   (other methods omitted)
}
```

Since `p` could refer to either a `CartesianPoint` or a `PolarPoint` instance at run time, the compiler's type information is not precise enough to optimize the call: the compiler knows `p`'s *abstract type* (i.e., the set of operations that can be invoked and their signatures) but not its *concrete type* (i.e., the object's size, format, and the implementation of the operations). Therefore, a late-bound `get_x` operation must be compiled as a *dynamically-dispatched call* that selects the appropriate implementation at run time. What could have been a one-cycle instruction has become a ten-cycle call. The increased flexibility and reusability of the source code exacts a

significant run-time overhead; it seems that encapsulation and efficiency cannot coexist without aggressive optimization.

The first part of this paper describes *type feedback*, an optimization that uses profile information to predict and inline dynamically-dispatched calls. Type feedback allows any dynamically-dispatched call to be inlined and substantially mitigates the performance penalty of dynamic dispatch. In our example implementation for the dynamically-typed object-oriented language SELF, type feedback reduces the call frequency by a factor of four and improves performance by 70% compared to a system without type feedback. For two medium-sized benchmarks, SELF-93 ran two to three times faster than a leading commercial Smalltalk-80 implementation, 1 to 1.3 times slower than an optimized C++ translation in a similar style, and 2.5 times slower than hand-optimized C++ versions.

However, the simple application of optimizing compilation in interactive programming environments could destroy one of their key properties, responsiveness. The responsiveness of exploratory programming environments (such as the Smalltalk programming environment) allows the programmer to concentrate on the task at hand rather than being distracted by long pauses caused by compilation or linking. Unfortunately, this responsiveness is often achieved at the expense of performance, and vice versa. Most language implementations that offer a high level of responsiveness do so either by interpreting a near-source-level representation or by executing a straightforward translation to machine code. The overhead of interpretation or unoptimized execution, combined with the efficiency problems created by pervasive dynamic dispatch, limit execution speed; it seems that responsiveness and efficiency cannot coexist, either.

The second part of this paper describes how we have integrated our optimizing compiler into a responsive system by using *adaptive optimization* to discover and optimize the "hot spots" of a program while it is running. A method is compiled on-demand by a fast-and-dumb compiler, and the result is instrumented and cached. Only if a method is executed often is it recompiled with an optimizing compiler. Figure 1 shows an overview of the compilation process of the system.

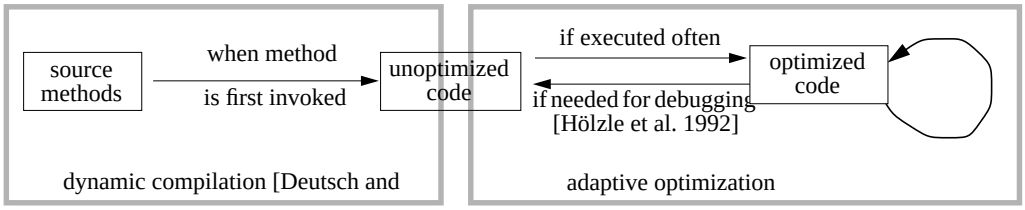

Figure 1. Compilation in the SELF-93 system

Adaptive optimization, when combined with type feedback, preserves most of the perceived responsiveness of an interpreter while achieving most of the efficiency of an optimizing compiler, and surpasses prior efforts to optimize the dynamic dispatch operation that is so characteristic of object-oriented programming.

2. Optimizing Dynamic Dispatch

Pure object-oriented languages exacerbate the performance problem caused by dynamic dispatch because *every* operation involves a dynamically-dispatched message send. The programming language used for this study, SELF, 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, so that 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. Consequently, a pure object-oriented language like SELF offers an ideal test case for optimization techniques tackling the problem of frequent dynamically-dispatched calls.

2.1 Type Feedback

The key idea of type feedback is to extract type information from executing programs and feed it back to the compiler (Figure 2). Specifically, an instrumented version of a program records the

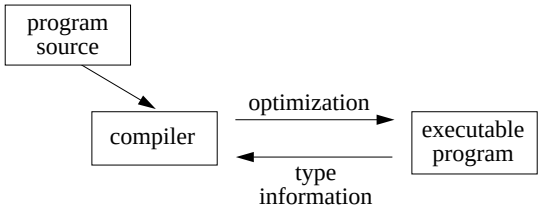

Figure 2. Overview of Type Feedback

program's *type profile*, i.e., a list of receiver types (and, optionally, their frequencies) for every single call site in the program. To obtain the type profile, the standard method dispatch mechanism is extended in some way to record the desired information, e.g., by keeping a table of receiver types per call site.

In the SELF system, no additional mechanism is needed to record receiver types since the system uses *polymorphic inline caches* to speed up dynamic dispatch. As observed in [Hölzle et al. 1991], these caches 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 any execution time overhead. However, the particular way in which type feedback information is collected is not important here; all that matters is that the information contains a list of receiver types (and, optionally, invocation counts) for each call site. (Section 3.7.1 describes how type feedback could be implemented in a more traditional compilation environment.)

The program's type profile is then fed back into the compiler to generate optimized code. 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. In the above example, the expression `x = p->get_x()` could be compiled as

```
if (p->class == CartesianPoint) {
    // inline CartesianPoint case
    x = p->x;
} else {
    // don't inline PolarPoint case because method is too big
    // this branch also covers all other receiver types
    x = p->get_x();    // dynamically-dispatched call
}
```

For `CartesianPoint` receivers, the above code sequence will execute significantly faster since the original virtual function call 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.

Some optimizations can enhance the benefits of inlining. *Splitting* [Chambers 1992] copies code following the `if` statement into the branches of the `if`, where it can profit from the more precise dataflow (or type) information that is specific to the branches of the `if`. However, splitting is limited to cases where the improved information can be used to optimize code immediately following (or very close to) the `if` statement. If the code that could benefit is further away, all code between it and the `if` statement must be duplicated, and the cost of the code increase may outweigh the benefits of the optimization.

Another optimization, *uncommon branch elimination*, is more aggressive and preserves the improved dataflow information throughout the caller. Uncommon branch elimination was first suggested to us by John Maloney and was implemented in Chambers' SELF-91 compiler [Chambers 1992] and (in a somewhat different and more aggressive form) in the SELF-93 compiler described in the next section. The main idea is that the optimized code handles *only* the predicted cases. Of course, the code still has to test for the uncommon cases, but upon encountering such a case, it branches to a separate (less optimized) copy of the code which does not merge back into the optimized version. Therefore, the optimized version's dataflow information is not "polluted" by the pessimistic *alias* and *kill* information caused by uncommon cases.

For example, if the type feedback information indicates that non-Cartesian points are almost never used, the expression `x = p->get_x()` could be compiled as

```
if (p->class != CartesianPoint) {
   goto uncommon_case;
   // branch to separate version of the code that handles
   // non-Cartesion points and never branches back
   // to this code
}
// inline CartesianPoint get_x()
x = p->x;
```

Now the code following this statement can be better optimized because the compiler knows `p`'s class, and that `get_x` has no side-effects.

Neither splitting nor uncommon branch elimination is necessary to implement type feedback; we have presented them here merely as examples of optimizations that profit from opportunities created by type feedback. The SELF-93 compiler described below implements both optimizations.

Predicting future receiver types based on past receiver types is only an educated guess. Similar guesses are made by optimizing compilers that base decisions on execution profiles taken from previous runs [Wall 1991]. However, in our experience, type profiles are more stable than time profiles—if a receiver type dominates a call site during one program execution, it also dominates during other executions. A recent study by Grove et al. [Grove et al. 1995] that measured the stability of type profiles in SELF, C++, and Cecil programs confirms our experience.

2.2 Inlining Strategies

Although type feedback enables the compiler to inline any call in the program, not all calls should be inlined. Deciding whether to inline a particular send is difficult for several reasons. First, inlining one method may require other methods to be inlined as well (e.g., to reduce closure

creation overhead). Second, even if the compiler could accurately estimate the local impact of inlining a send, the overall performance impact may depend on the result of other inlining decisions. For example, inlining a send may be beneficial in one case but may hurt performance in another case because other inlined sends increase register pressure so much that important variables cannot be register-allocated.

The current SELF compiler uses a set of simple rules to guide the inlining process. Methods are inlined if they are small, and if the estimated size of the caller (including all methods inlined so far) is not too big. (The latter condition avoids excessive inlining that could arise when many small methods are called.) Although more sophisticated inlining strategies are possible [Dean et al. 1995], we did not consider them in this implementation.

Determining the "size" of an inlining candidate is harder in SELF than in more traditional languages: since SELF is a pure object-oriented language, it performs all computation via message sending, and thus virtually every source-code token represents a message send whose cost (both in terms of space and time) is highly variable. To improve its estimates, the SELF compiler examines previously-compiled optimized code where available [Hölzle 1994]. Besides being more accurate than source-level size estimates, this approach also has the advantage of considering a bigger picture: typically, the compiled method for a source method includes not only code for the method itself but also that of inlined calls. By examining previously-compiled code, the compiler can obtain a better estimate of the ultimate space cost of an inlining decision.

2.3 Structure of the SELF-93 Compiler

This section briefly describes the optimizing SELF-93 compiler which combines simplicity with good compilation speed and good code quality. The front end of the compiler performs a variety of optimizations that are necessary to achieve good performance with pure object-oriented languages—inlining (based on type feedback), customization, and splitting—and generates a graph of intermediate code nodes. The back end performs only few optimizations on the intermediate code before generating machine code. In particular, the compiler does not perform full-fledged dataflow analysis or coloring register allocation because we considered these techniques to be too expensive in terms of compilation speed.

After computing the definitions and uses of each pseudo register, the compiler performs the following optimizations:

- *Closure analysis* determines which closures can be eliminated because they are not needed as actual run-time objects.
- *Copy propagation* propagates pseudo registers within basic blocks, and singly-assigned pseudo registers globally. (These propagations can be performed without computing full dataflow information.)
- *Dead code elimination* discards nodes whose results are no longer needed.

A simple usage-count based register allocator computes the register assignments, and the final machine code is generated in a single pass over the intermediate graph.

Later sections will compare SELF-93 with a previous implementation, SELF-91 [Chambers 1992]. SELF-93 mainly differs from SELF-91 in substituting type feedback for iterative type analysis, and in a less ambitious back end. As a result, SELF-93 is considerably simpler (11,000 vs. 26,000 lines of C++). However, compared to SELF-91, SELF-93 has several shortcomings:

- *Inferior local code quality.* The compiler does not fill delay slots except within fixed code patterns. Also, code often contains branches that branch to other (unconditional) branch instructions instead of directly branching to the final target. Finally, values may be repeatedly loaded from memory, even within the same basic block. This is especially inefficient if the loaded value is an uplevel-accessed variable since an entire sequence of loads (following the lexical chain) is repeated in this case.
- *Inferior register allocation*. The register allocator is very simple and can cause unnecessary register moves or spills.
- *Redundant type tests*. Since the compiler does not perform type analysis or full dataflow analysis, a value may be tested repeatedly for its type even though only the first test is necessary.

It is hard to estimate the performance impact of these shortcomings. However, based on Chambers' analysis of the SELF-91 compiler [Chambers 1992] and an inspection of the compiled code of several programs, we believe that they slow down the large object-oriented programs measured in this study by at least 10%. For programs with small integer loops, the overhead can be much higher. Therefore, the performance of type feedback as reported in Section 3 is a conservative indication of what a fully optimizing SELF compiler with type feedback could achieve.

3. Performance of Type Feedback

To evaluate the performance of the SELF-93 compiler and the contribution of type feedback, we measured the run-time performance of several large SELF programs (see Table A-2 in the appendix for a short description of the benchmarks). With the exception of the Richards benchmark, all programs are real applications that were not written for benchmarking purposes. Table 3 lists the systems used in our study.

System	Description
SELF-93	The current SELF system using dynamic recompilation and type feedback; methods are compiled by a fast non-optimizing compiler first, then recompiled with the optimizing compiler if necessary.
SELF-93 nofeedback	Same as SELF-93, but without type feedback and recompilation; all methods are always optimized from the beginning.
SELF-91	Chambers' SELF compiler [Chambers 1992] using iterative type analysis; all methods are always optimized from the beginning. This compiler has been shown to achieve excellent performance for smaller programs.
Smalltalk-80	ParcPlace Smalltalk-80™ release 4.0, generally regarded as the fastest commercial Smalltalk system (based on techniques described in [Deutsch and Schiffman 1984])
C/C++	GNU C and C++ compilers, version 2.4.5, using -O2 optimization
Lisp	Sun CommonLisp 4.0™ using full optimization

Table 1: Systems used for benchmarking

3.1 Methodology

To measure accurate execution times, the programs were run under a SPARC simulator based on the SPA [Irlam 1991] and Shade [Cmelik and Keppel 1993] tracing tools and the Dinero cache simulator [Hill 1987]. The simulator models the Cypress CY7C601 implementation of the SPARC™ architecture, i.e., the chip used in the SPARCstation-2™ workstation.

The simulator also accurately models the memory system of a SPARCstation-2, with the exception of the cache organization. Instead of the unified direct-mapped 64K cache of the SPARCstation-2, we simulate a machine with a 32K 2-way associative instruction cache and a 32K 2-way associative data cache using write-allocate with subblock placement. "Write-allocate with subblock placement" caches allocate a cache line when a store instruction references a location not currently residing in the cache. This organization is used in current workstations (e.g., the DECstation 5000™ series) and has been shown to be effective for programs with intensive heap allocation [Koopman et al. 1992], [Reinhold 1993], [Diwan et al. 1995].

We do not use the original SPARCstation-2 cache configuration because it suffers from large variations in cache miss ratios caused by small differences in code and data positioning (we have observed variations of up to 15% of total execution time). With the changed cache configuration, these variations become much smaller (on the order of 2% of execution time) so that the performance of two systems can be more accurately compared. To ensure that our choice of cache organization did not distort the results, we measured different cache organizations, including 32K and 64K direct-mapped caches. While absolute execution times varied, the resulting performance ratios (e.g., SELF-93 vs. SELF-93-nofeedback) were within 10% of the ratios presented here.

Table 2 shows the benchmarks used. With the exception of the Richards benchmark, all programs

	Benchmark	Size[a]	Description
small benchmarks	DeltaBlue	500	two-way constraint solver [Wilson and Moher 1989] developed at the University of Washington
	PrimMaker	1100	program generating "glue" stubs for external primitives callable from SELF
	Richards	400	simple operating system simulator originally written in BCPL by Martin Richards
large benchmarks	CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function (the compiler shares about 80% of its code with the interpreter, CecilInt)
	CecilInt	9,000	interpreter for the Cecil language [Chambers 1993] running a short Cecil test program
	Mango	7,000	automatically generated lexer/parser for ANSI C, parsing a 700-line C file
	Typeinf	8,600	type inferencer for SELF [Agesen et al. 1993]
	UI1	15,200	prototype user interface using animation techniques [Chang and Ungar 1993][b]
	UI3	4,000	experimental 3D user interface[b]

Table 2: Benchmark programs

[a] Lines of code (excluding blank lines and comments).
[b] Time for both UI1 and UI3 excludes the time spent in graphics primitives

are real applications that were not written for benchmarking purposes. The applications represent several different coding styles; Mango is not handwritten but was generated by a parser generator.

The execution times for the SELF programs reflect the performance of (re-)optimized code, i.e., they do not include compile time. For the recompiling system, the programs were run until performance stabilized, and the next run not involving compilations was used. (The impact of

dynamic recompilation on interactive performance is explored later in Section 6.) SELF-91 and SELF-93-nofeedback do not use recompilation, so we used the second run for our measurements.

3.2 Impact of Type Feedback on Execution Time

To evaluate the performance impact of type feedback, we compared the three versions of the SELF system mentioned in Table 3. Figure 3 shows the results (Table A-1 in the appendix contains detailed data).[1] Comparing SELF-93 with SELF-93-nofeedback shows that type feedback significantly improves the quality of the generated code, resulting in a speedup of 1.7 (geometric mean) even though SELF-93-nofeedback always optimizes all code whereas SELF-93 optimizes only parts of the code. (Sections 3.4 and 3.5 will analyze the reasons for the increased performance of SELF-93 in more detail.) SELF-93 also outperforms SELF-91 by a considerable

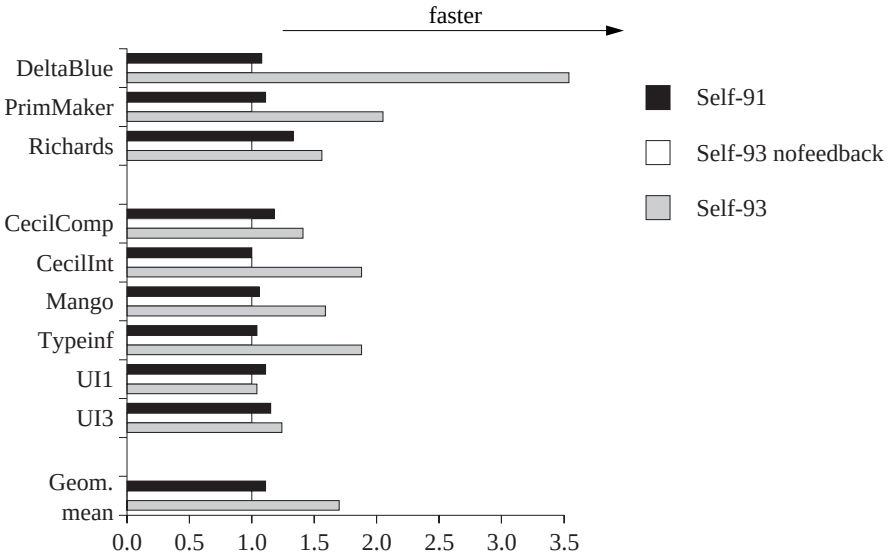

Figure 3. Performance impact of type feedback (all speeds relative to SELF-93-nofeedback)

margin, with a speedup of 1.5. Apparently, the better back end and iterative type analysis are not enough for SELF-91 to compensate for the wealth of type information provided by type feedback. In fact, SELF-91 is only marginally faster than SELF-93-nofeedback which does not use any type analysis. In other words, SELF-91's type analysis appears to be largely ineffective for the programs we measured.

3.3 Impact of Type Feedback on Call Frequency

Type feedback drastically reduces the number of calls executed by the benchmark programs. Figure 4 shows the number of calls relative to unoptimized SELF, where each message send is implemented as a dynamically-dispatched call (with the exception of accesses to instance variables in the receiver). Both SELF-91 and SELF-93 run many times faster than unoptimized programs.

[1] The execution times of the benchmarks were kept relatively short to allow cycle-level simulation. To ensure that the small inputs do not distort the performance figures, we measured three of the benchmarks with larger inputs. Table A-2 in the appendix shows that the speedups achieved by type feedback are very similar to the speedups with smaller inputs.

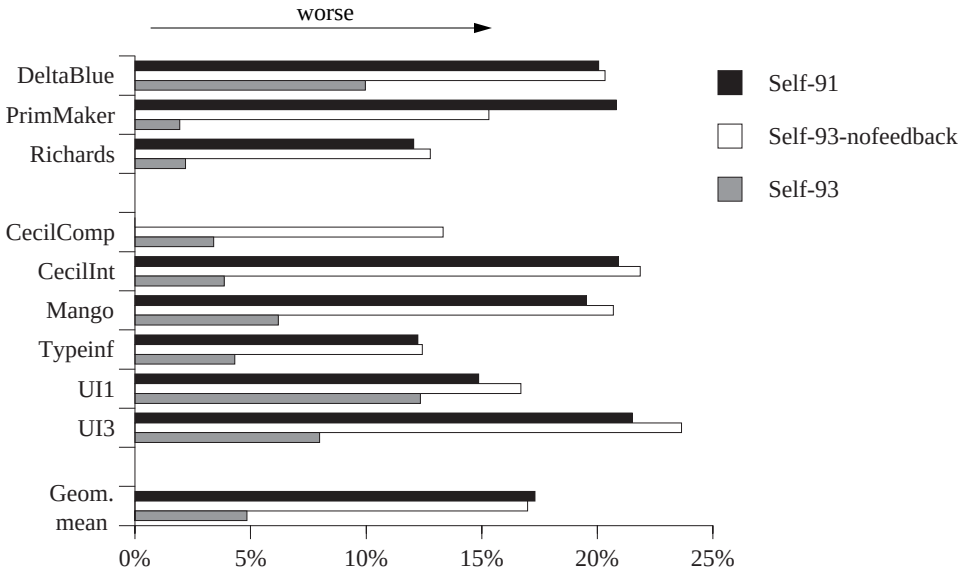

Figure 4. Impact of type feedback on number of calls (all numbers are relative to unoptimized SELF)

Whereas 10-25% of the original calls remain in SELF-91 and SELF-93-nofeedback, SELF-93 reduces the call frequency to about 5% of the unoptimized system. Compared to the SELF systems without type feedback, calls are reduced by a factor of 3.6. Since SELF-93-nofeedback performs about the same number of calls as SELF-91, we can also assume that comparing SELF-93 to SELF-91 is fair, i.e., that the reduction in call frequency and execution time is entirely due to type feedback and cannot be attributed to other differences (such as more aggressive inlining). As with performance, the sophisticated type analysis in SELF-91 fails to give it an advantage over SELF-93-nofeedback when it comes to eliminating calls.

3.4 Type Testing Overhead

Since type feedback transforms dynamically-dispatched calls into type tests followed by inlined methods, it is interesting to look at the characteristics of these type tests. In SELF-93, type tests are used in two situations: for sends inlined by type feedback (*inlined tests*), and for the dispatch of non-inlined sends (*dispatch tests*). SELF uses *Polymorphic Inline Caches* (PICs [Hölzle et al. 1991]) to implement dynamically-dispatched calls as a sequence of comparisons testing for the expected types, followed by a direct call. For example, if a non-inlined send has had Point and rectangle receivers in the past, its PIC would first test the receiver type against Point and then against Rectangle; if neither of these match, a lookup routine is invoked which will either extend or change the PIC to include the new case [Hölzle et al. 1991].

The average number of type tests executed per send (i.e., the number of comparisons testing for the expected types) is very small. Figure 5 shows the distribution of the per-benchmark averages for SELF-93-nofeedback (left boxes) and SELF-93 (right boxes). Since we are interested in the work done per type test sequence, the data excludes sends requiring no type test, i.e. sends whose receiver type was known with certainty.

SELF-93-nofeedback executes some inlined type tests because it uses static type prediction [Deutsch and Schiffman 1984] to predict the receiver type of certain very frequent messages. Static type prediction always predicts for a single type, except for sends to boolean receivers (true

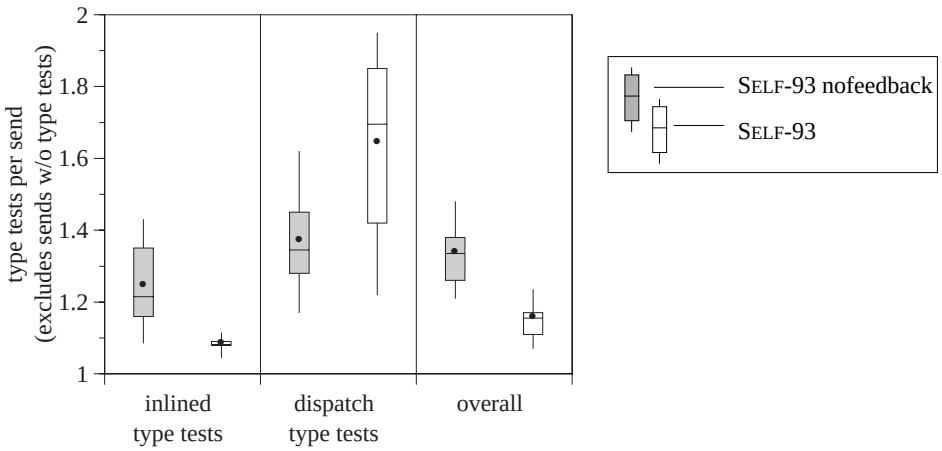

Figure 5. Number of type tests per dispatched send

Box charts show the range of data (vertical lines) as well as the 25% and 75% percentiles (end of the boxes) and the median (horizontal lines). The dots (•) indicate the mean.

and false are two different types in SELF). Thus, the low average of 1.2 tests per send in SELF-93-nofeedback is not surprising. What is surprising, however, is that type feedback reduces this average even more, to 1.08 tests per send. In other words, the vast majority of inlined type tests need only one comparison to find their targets. Apparently, most sends optimized with type feedback have only one receiver type or are dominated by a single receiver type.

For non-inlined sends, type feedback pushes up the median number of type tests per send from 1.35 to 1.7 tests per send. Type feedback does not actually increase the degree of polymorphism of sends; however, since the compiler does not inline highly polymorphic sends (with 5 or more receiver types) but at the same time eliminates many of the other sends, the distribution of the remaining sends is skewed towards higher polymorphism, and thus the average number of type tests per send increases.

Finally, the last category of Figure 5 shows that the overall number of type tests per send is reduced by type feedback. Does this mean that programs optimized by type feedback perform *fewer* type tests? Figure 6 shows that this is indeed the case: on average, SELF-93 programs execute 27% fewer type tests. At first sight, such a reduction seems impossible. Dispatch is implemented as a type test followed by a call, and type feedback just transforms this sequence into a type test followed by inlined code. Thus, it would seem that the total number of type tests should remain exactly the same since type feedback merely turns dispatch tests into inlined tests. (Figure 6 confirms that many dispatch tests are indeed transformed into inlined tests.)

However, type feedback can reduce the number of type tests because the compiler may statically know the types of the arguments of a send inlined via type feedback. For example, suppose that a method *m* is called with a constant argument. If this send is not inlined, each send in *m* to the argument will require a type test since the argument's type is not known statically. However, after *m* has been inlined using type feedback, constant propagation can reach all uses of the constant argument and eliminate the type tests. Thus, by inserting one type feedback test, the compiler has eliminated other type tests and has reduced the overall number of type tests. In the benchmarks we measured, each type feedback test removed 0.8 other type tests on average, even though the compiler performs only rudimentary dataflow analysis. With a more sophisticated analysis, this "bonus" might be even higher.

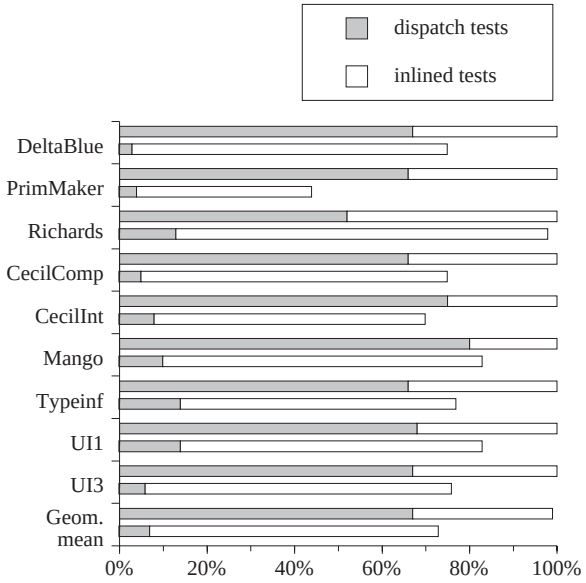

Figure 6. Number of type tests relative to SELF-93-nofeedback (upper bars: SELF-93-nofeedback, lower bars: SELF-93)

3.5 Analysis of Speedup

Why does type feedback speed up programs? One reason for the increased speed is the reduced call overhead, but how much of the speedup is obtained by just eliminating call overhead, and how much is due to other factors? Figure 7 shows that the sources of improved performance can

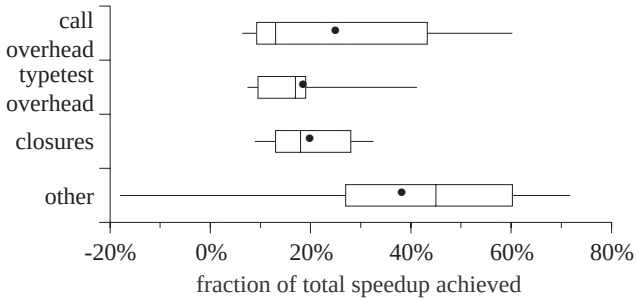

Figure 7. Reasons for SELF-93's improved performance

vary widely from benchmark to benchmark. (The data assumes a savings of 10 cycles per eliminated call since we could not measure the exact savings per call.) Depending on the benchmark, the reduced call overhead represents between 6% and 63% of the total savings in execution time, with a median of 13% and an arithmetic mean of 25% (geometric mean: 18%). The reduced number of type tests contributes almost as much to the speedup, with a median contribution of 17% and a mean of 19%, as does the reduced number of closure creations.

Other effects (such as standard optimizations that perform better with the increased size of compiled methods) make the greatest contribution to the speedup (with a median of 45% and a mean of 38%) but also show the largest variation. For one benchmark, the contribution is actually negative, i.e., slows down execution. Some of the possible reasons for the slowdown are inferior register allocation due to increased register pressure, or higher instruction cache misses. All of the above measurements include cache effects.

To summarize, the measurements in Figure 7 show that the performance improvement obtained by using type feedback is by no means dominated by the decreased call overhead. In most benchmarks, factors other than call overhead dominate the savings in execution time. Inlining based on type feedback is an enabling optimization that allows other optimizations to work better, thus creating indirect performance benefits in addition to the direct benefits obtained by eliminating calls.

Exponential code growth is a well-known potential problem of procedure inlining. However, the additional inlining performed by SELF-93 does not increase code size much over the systems not using type feedback (Figure 8). On average, compiled code is only 25% larger in SELF-93 than in

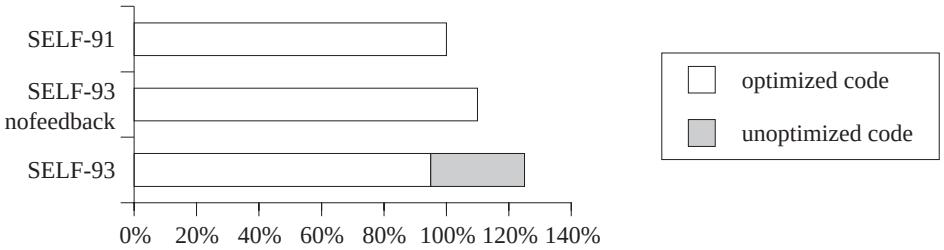

Figure 8. Size of compiled code relative to SELF-91

SELF-91; comparing SELF-93-nofeedback to SELF-91 shows that part of the code size increase may be caused by the inferior SELF-93 back end. For some programs, the resulting code actually becomes smaller. This behavior suggests that previous SELF systems could not inline many attractive inlining candidates (i.e., very small methods), so that type feedback can reduce the call frequency by a factor of 3.6 with a code growth of only 15-25%.

3.6 Performance Relative to Other Systems

To provide some context about SELF's performance, we measured versions of the DeltaBlue and Richards benchmarks written in C++ and Smalltalk, as well as a Lisp version of Richards. (See Table 3 for details about the C++ and Smalltalk systems, and Table A-4 in the Appendix for detailed performance data; none of the other benchmarks are available in other languages.) Since it was not possible to run Smalltalk or Lisp with the simulator, we could only measure SPARCstation-2 CPU times. Simulated times of SELF programs usually are between 5 and 25% lower than measured execution times on a SPARCstation-2 since the simulation models a better cache organization and does not include OS overhead. Therefore, for comparison with SELF and C++, we reduced the measured Smalltalk and Lisp execution times by a conservative 25%. Figure 9 shows the results.

For DeltaBlue and Richards, SELF-93 runs 2.3 and 3.5 times faster than ParcPlace Smalltalk (generally regarded as the fastest commercially available Smalltalk system) even though SELF's language model is purer and thus harder to implement efficiently.[2] For Richards, SELF-93 runs 2.6 times faster than an equivalent CommonLisp program compiled with maximum optimization and

[2] SELF does have one feature, constant slots, that could improve performance relative to Smalltalk. However, constant slots are mainly used for inheritance (parent slots) and to hold methods; in both cases, the Smalltalk equivalents (superclass link and method dictionary) are constant as well. Replacing all other constant slots by assignable slots in the SELF versions of DeltaBlue and Richards decreases performance by less than 1% and 15%, respectively, resulting in speedups of 2.3 and 3.0 over Smalltalk.

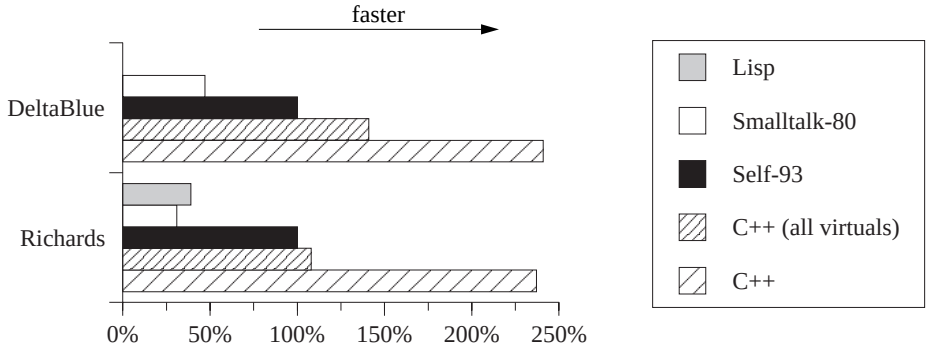

Figure 9. Execution speed (SELF-93 = 100%)

minimum safety (i.e., the Lisp code would not detect some run-time errors). In conclusion, for these two programs SELF-93 runs two to three times faster than languages with roughly comparable semantics.

Comparing SELF and C++ is harder since the two languages have very different language models. SELF provides code reuse and safety by basing the language on extensible control structures, pointer safety, bounds and overflow checking, generic and extensible arithmetic, and pure message passing. On the other hand, C++ omits these features (with the exception of virtual functions) in its quest for high performance. Consequently, C++ programmers has a choice of programming style: either they use virtual functions liberally to get more flexibility, reusability, and maintainability, or minimize virtual function usage to get maximum performance.

We have measured both extremes in order to compare SELF-93's performance against C++. If the two C++ programs are hand-optimized to make minimal usage of virtual calls, C++ is 2.3 times faster than SELF-93. If all C++ functions are declared "virtual," however, C++ is only 10% to 40% faster than SELF-93 despite SELF's inferior back end.

We have also measured the size of compiled code relative to C++. This comparison should be taken *cum grano salis* since our measurements are somewhat imprecise. First, the SELF numbers include some code in the measurement loop calling the actual benchmarks; since the two

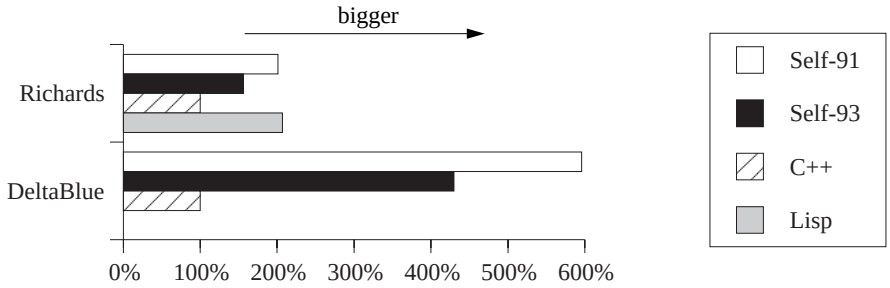

Figure 10. Code size relative to GNU C++

benchmarks are fairly small (10-40 Kbytes), this code may inflate the numbers for SELF. Second, all numbers include only the actual code generated by the compilers and exclude any library code needed by the programs (for both C++ programs the library code is an order of magnitude larger than the actual compiled code). Third, as we have mentioned above, SELF's execution semantics are very different from C++'s, and additional code is sometimes needed to preserve them (e.g., overflow checks).

Figure 10 shows that for Richards and DeltaBlue, the additional inlining performed by SELF-93 actually decreases code size relative to SELF-91 (see Table A-5 in the appendix for absolute data). But compared to GNU C++ the code is larger, especially for DeltaBlue where several methods defined for constraints are customized to the three constraint types. In this particular case, the compiler actually overcustomizes—not all of the customization is necessary to get good performance. Thus, the code increase is not a result of type feedback but of overcustomization (type feedback actually decreases DeltaBlue's code size).[3]

3.7 Applicability of Type Feedback to Other Systems

As demonstrated by the measurements in Section 3, type feedback works very well for SELF. How well would it work with more conventional implementation techniques (i.e., static compilation), and how does it apply to other languages?

3.7.1 Type Feedback and Static Compilation

Type feedback is not dependent on the "exotic" implementation techniques used in SELF-93 (e.g., dynamic compilation or dynamic recompilation). If anything, these techniques make it harder to optimize programs: using dynamic compilation in an interactive system places high demands on compile speed and space efficiency. For these reasons, the SELF-93 implementation of type feedback has to cope with incomplete information (i.e., partial type profiles and inexact invocation counts) and must refrain from performing some optimizations to achieve good compilation speed.

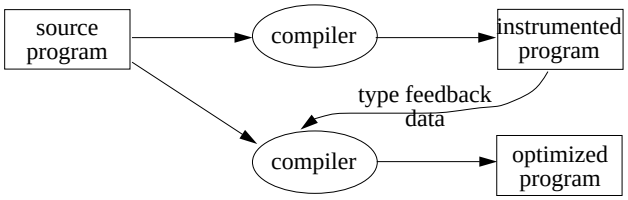

Figure 11. Type feedback in a statically compiled system

Thus, we believe that type feedback is probably easier to add to a conventional batch-style compilation system. In such a system, optimization would proceed in three phases (Figure 11). First, the executable is instrumented to record receiver types, for example with a `gprof`-like profiler [Graham et al. 1983]. (The standard `gprof` profiler already collects almost all information needed by type feedback, except that its data are caller-specific rather than call-site specific, i.e., it does not separate two calls of `foo` if both come from the same function.) Then, the application is run with one or more test inputs that are representative of the expected inputs for production use. Finally, the collected type and profiling information is fed back to the compiler to produce the final optimized code.

Static compilation has the advantage that the compiler has complete information (i.e., a complete call graph and type profile) since optimization starts after a complete program execution. In contrast, a dynamic recompilation system has to make decisions based on incomplete information. For example, it cannot afford to keep a complete call graph, and the first recompilations may be necessary while the program is still in the initialization phases so that the type profile is not yet

[3] With type feedback, it would be possible to customize less aggressively (thus reducing code size) since customization is no longer needed to enable inlining (i.e., with type feedback the main benefit of customization is that it can reduce the number of type tests required).

representative. On the other hand, a dynamic recompilation system can dynamically adapt to unforeseen changes in the program's behavior.

3.7.2 Applicability to Other Languages

Type feedback could be applied to other object-oriented languages (e.g., Smalltalk or C++), or for languages with generic operators that could be optimized with the type feedback information (e.g., APL or Lisp). But how effective would it be?

In general, the effectiveness of type feedback depends on the frequency of dynamically-dispatched calls, their predictability, and on the optimization opportunities exposed by inlining such calls. For a pure object-oriented language like Smalltalk which shares SELF's high call frequency and small methods, we expect the resulting speedups to be similar to those achieved for SELF. Some language differences (e.g., prototype- vs. class-based inheritance, hardwired control structures and full closures in Smalltalk) will alter the results somewhat, but overall the similar execution characteristics should result in similar speedups.

Even a hybrid language like C++ whose execution behavior (and language philosophy) differs much more from SELF will nevertheless benefit from type feedback if it displays similar execution characteristics, although the benefits are reduced by inherently lower call frequencies. A recent prototype C++ compiler employing type feedback confirms this assumption, improving the performance of a suite of large C++ programs by a median of 20% while reducing the frequency of dynamically-dispatched calls by a factor of five [Aigner and Hölzle 1995].

4. Adaptive Optimization

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. Originally, system designers used interpreters in exploratory object-oriented programming environments in order to achieve the immediate feedback that allows a programmer to concentrate on the task at hand. 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. Initial Smalltalk interpreters on stock hardware ran twenty to thirty times slower than programs written in conventional languages like C [Krasner 1983], and for a long time the only Smalltalk implementation running at acceptable speed was microcoded onto fast, expensive ECL machines [Deutsch 1983].

Dynamic compilation [Deutsch and Schiffman 1984] improved the performance of Smalltalk on off-the-shelf microprocessors. Inspired by the observation that a straightforward translation to machine code could eliminate part of the interpretation overhead, its inventors built a system that translated bytecodes to machine code on demand. When a bytecoded method was invoked for the first time, it was compiled quickly by a fast, compiler that performed only peephole optimizations. The compiled code was then stored in a code cache for later use. The compiler was fast enough to escape the user's notice, and compilation was implicit (i.e., the user never needed to invoke the compiler explicitly). Usually, dynamically compiled programs ran about twice as fast as interpreted ones because they did not have to decode pseudo-instructions at run-time [Deutsch and Schiffman 1984]. However, this level of performance was still up to an order of magnitude

below optimized C and C++ [Chambers at al. 1989]. Despite this limitation, several commercial Smalltalk implementations use dynamic compilation today (e.g., [PP92]).

In theory, the performance of a dynamically-compiled programs could be improved by adding more aggressive optimizations to the compiler. However, optimizing compilers usually run slowly and thus create distracting compilation pauses. In an interactive programming environment with dynamic compilation (where compilation occurs at run-time), such pauses are even more distracting than in a conventional setting because users have higher expectations. Even a pause as short as one second might be considered distracting by most users of an interactive system, whereas the same one-second pause would be considered negligible in a batch-compilation system. If compilation is implicit, minimizing compilation pauses is even more important since compilation (rightfully) vanishes from the user's mental model of the system. A compiler that runs implicitly and dynamically cannot afford to spend time on elaborate optimizations, no matter how effective they may be.

This incompatibility between execution speed and compilation speed surfaced in early SELF implementations. In response to the performance problems of pure object-oriented languages, previous SELF compilers concentrated on optimization techniques aimed at reducing the overhead of message passing. The first-generation SELF compiler achieved a respectable speedup over standard Smalltalk implementations. The second-generation compiler improved performance even more, bringing SELF's performance to within a factor of 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 [Chang and Ungar 1993] 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.

In an interactive system using dynamic compilation, optimizing less may *improve* overall efficiency if doing so reduces compile time more than it increases execution time. The system described here, SELF-93, reconciles optimizing compilation with interactive environments by optimizing only the performance-critical parts of a program. Consequently, compilation overhead is reduced since not all code needs to be optimized, and compilation pauses are further reduced since optimizing compilations are better distributed over time. In addition to using dynamic compilation to generate compiled code as needed, SELF-93 uses *adaptive optimization* to discover and optimize the "hot spots" of a program while it is running. Only if a method is executed often is it recompiled with an optimizing compiler. The remainder of this section describes the adaptive optimization process in more detail, outlining how the system discovers methods needing optimization.

4.1 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.

The ideal recompilation policy is simple to state: recompile a method only if the recompilation reduces total execution time (i.e., the sum of compilation and execution time), and recompile as early as possible to accrue the maximum execution time savings. Unfortunately, this policy is impossible to implement:

- The system cannot know how often a method will be executed in the future and thus cannot determine the savings from running better-optimized code. In addition, it is hard to estimate how much faster the recompiled version will be without first performing (most of) the compilation.
- Methods need to execute for some time in order to accumulate representative type feedback information before being recompiled. Thus, determining the earliest possible time to recompile requires another perfect prediction of future program behavior.
- Finally, the ideal recompilation policy described above ignores interactive aspects. In addition to minimizing overall execution time, an interactive system must also try to minimize compilation pauses that are visible to the user and thus may have to delay compilations in order not to disrupt a user interaction. (Sections 5 and 6 will discuss this issue in more detail.)

SELF-93 approximates the ideal policy by using *invocation counts* to drive recompilation. Assuming that past behavior predicts future behavior, methods are recompiled if their invocation counter exceeds a certain limit. Each unoptimized method has its own counter that is incremented in the method prologue. When the counter exceeds the 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.

If nothing further were done, most methods would eventually reach the invocation limit and would be recompiled even though they might not execute more often than a few times per second, so that optimization would hardly bring any benefits. Therefore, invocation counters decay exponentially over time. The decay rate is given as the half-life time, i.e., the time after which a counter loses half of its value. The decay process is approximated by periodically dividing 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, counters are divided by $p = 1.2$ (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 be recompiled.[4]

Originally, counters were envisioned as a first step, to be used only until a better solution was found. Since the simple counter-based approach worked so well, we did not extensively

[4] 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.

investigate other mechanisms.[5] However, there are some interesting questions relating to invocation counter decay:

- Is exponential decay the right model? Ideally, the system would recompile only those methods where the optimization cost is smaller than the benefits that accrue over future invocations of the optimized method.[6] Of course, the system does not know how often a method will be executed in the future, but a rate-based measure also ignores the past: a method that executes less often than the minimum execution rate will never trigger a recompilation, even if it is executed many times.
- The invocation limit N should not be a constant; rather, it should depend on the particular method. Counters are and approximation of the execution time wasted by running unoptimized code. Thus, a method that would benefit greatly from optimization should count faster (or have a lower limit) than a method that barely benefits from optimization. Of course, it may be hard to estimate the performance impact of optimization on a particular method.
- How should half-life times be adapted when executing on a faster (or slower) machine? Suppose that the original half-life parameter was 10 seconds, but that the system now executes on a new machine that is twice as fast. Should the half-life parameter be changed, and if so, how? One could view that faster machine as a system where real time runs half as fast (since twice as many operations are completed per second), and thus reduce the half-life to 5 seconds. However, one could also argue that the invocation rate limit is absolute: if a method executes less than n times per second, it is not worth optimizing.
- Similarly, should the half-life time be measured in real time, CPU time, or some machine-specific unit (e.g., number of instructions executed)? Intuitively, using real time seems wrong, since the user's think pauses (or coffee pauses) would influence recompilation. Using CPU time has its problems, too: for example, if most of the time is spent in the VM (e.g., in garbage collection, compilation, or graphics primitives), the half-life time is effectively shortened since compiled methods get less time to execute and increase their invocation counters. On the other hand, this effect may be desirable: if not much time is spent in compiled SELF code, optimizing that code may not increase performance by much (except, of course, if the optimizations reduce the VM overhead, e.g., by reducing the number of block closures created, thus reducing allocation costs and garbage collections).

Given these limitations, it may seem surprising that the simplest counter-based approach works at all. However, as long as programs have "hot" and stable distributions of execution, even this simple approach works well. In the course of our experiments we discovered that the trigger mechanism ("when") is much less important for good recompilation results than the selection mechanism ("what").

[5] We did consider two alternatives. The first placed counters on the edges of the call graph rather than on the nodes, providing more information to the recompilation system. Unfortunately, the space cost of edge counts was prohibitive in the SELF system because the call graph of unoptimized code is extremely large. The second approach would use PC sampling to discover time-consuming methods (similar to some profilers). Unoptimized SELF methods are very short and numerous, and even control structures like `if` are implemented via message sending, so that the coarse timer-based profile information would not be very helpful in making good recompilation decisions. In more conventional systems, however, both edge counters and timer-based profile information might be useful mechanisms for driving dynamic recompilation.

[6] This statement is actually not quite accurate—while such a system would theoretically minimize total execution time, it ignores the fact that optimization cannot begin until type feedback information has accumulated. Also, it ignores interactive behavior (clustering of compilations).

4.2 What to Recompile

When a counter overflows, the recompilation system is invoked to decide which method to recompile (if any). A simple strategy would always recompile the method whose counter overflowed, since it obviously was invoked often. However, this strategy would not work well. For example, suppose that the method overflowing its counter just returns a constant. Optimizing this method would not gain much; rather, the method should be inlined into its caller. So, in general, to find a "good" candidate for recompilation, the system walks up the call chain and inspects the callers of the method triggering the recompilation.

4.2.1 Overview of the Recompilation Process

Figure 12 shows an overview of the recompilation process. Starting with the method that overflowed its counter, the recompilation system walks up the stack to find a "good" candidate for recompilation (the next section will explain what "good" means). Once a recompilee is found, the compiler is invoked to reoptimize the method, and the old version is discarded (if no recompilee is

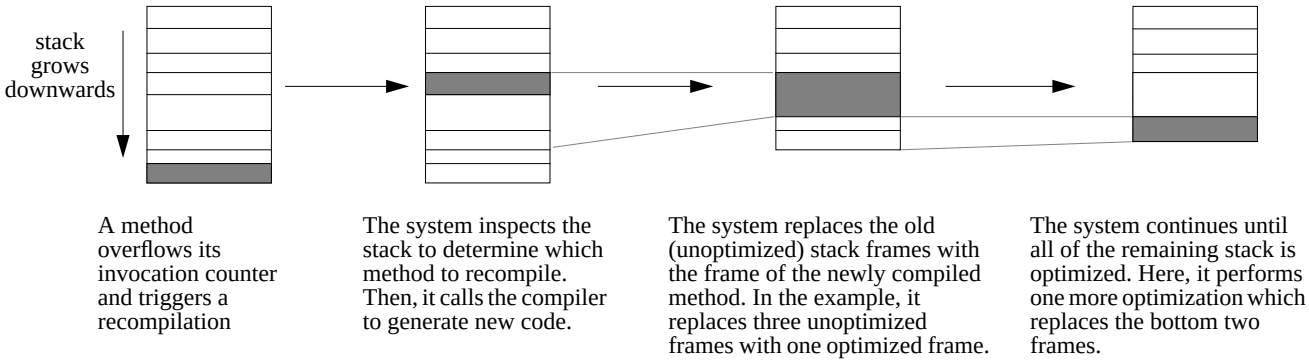

Figure 12. Optimization process

found, execution continues normally). During the optimizing compilation, the compiler marks the restart point (i.e., the point where execution will be resumed) and computes the contents of all live registers at that point. If this computation is successful,[7] the reoptimized method replaces the corresponding unoptimized methods on the stack, possibly replacing several unoptimized activation records with a single optimized activation record. Then, if the newly optimized method isn't at the top of the stack, recompilation continues with the newly optimized method's callee. In this way, the system optimizes an entire call chain from the top recompilee down to the current execution point. (Usually, the recompiled call chain is only one or two compiled methods deep.)

If the unoptimized methods cannot be replaced on the stack, they are left to finish their current activations, but subsequent invocations will use the new, optimized method. The main effect of failing to replace the unoptimized methods is that additional recompilations may occur if the unoptimized code continues to execute for a while. For example, if the optimized method contains a loop but cannot be placed on the stack immediately, the reoptimization system may later try to replace just the loop body with optimized code.[8]

[7] The compiler cannot always describe the register contents in source-level terms since it does not track the effects of all optimizations in order to keep the compiler simple. However, it can always detect such a situation and signal it to the recompilation system.

[8] Recall the SELF implements control structures using blocks (closures) and message sends, and so the body of a loop is a method invoked via a message send and thus can be optimized like any other send.

4.2.2 Selecting the Method to be Recompiled

The SELF-93 system uses several heuristics to select the method to be recompiled. The assumptions underlying these rules are that frequently executed methods are worth optimizing, and that inlining small methods and eliminating closures will lead to faster execution. Although the rules are simple, they result in good performance as seen in Section 3.

For any compiled method *m*, the following values are defined:

- *m.size* is the size of *m*'s instructions.
- *m.count* is the number of times *m* was invoked.
- *m.sends* is the dynamic number of calls directly made from *m*.[9]
- *m.version* records how many times *m* has been recompiled.

The search for a recompilee can be outlined as follows. Let *trip* be the method tripping its counter, and *recompilee* be the current candidate for recompilation.

1. Start with *recompilee* = *trip*.
2. If *recompilee* has closure arguments, choose the closure's lexically enclosing method if it meets the conditions described below. This rule eliminates closures by inlining the closure's use into the closure's home; if inlining succeeds, the closure can usually be optimized away completely.
3. Otherwise, choose *recompilee*'s caller if it meets the conditions below. This rule will walk up the stack until encountering a method that is either too large or does not appear to cause many message sends to be executed.
4. Repeat steps 2 and 3 until *recompilee* reaches a fixpoint.

Whenever the recompilation system considers a new recompilee *m* (in steps 2 and 3 above), it will only accept the new recompilee if it meets both of the following conditions:[10]

- *m.count* > *MinInvocations* and *m.version* < *MaxVersion*. The first clause ensures that the method has been executed enough times to consider its type information representative. The second clause prevents endless recompilation of the same method.
- *m.sends* > *MinSends* or *m.size* < *TinySizeLimit* or *m* is unoptimized. The first clause accepts methods sending many messages, and the other two accept methods that are likely to be combined with the caller through inlining.

The rules used by the recompilation system for finding a "good" recompilation candidate in many aspects mirror the rules used by the compiler for choosing "good" inlining opportunities. For example, the rule skipping "tiny" methods has an equivalent rule in the compiler that causes "tiny" methods to be inlined. Ideally, the recompilation system should consult the compiler before every decision to walk upwards on the stack (i.e., towards a caller) to make sure the compiler would inline that send. However, such a system is probably unrealistic: to make its inlining decisions, the compiler needs much more context, such as the overall size of the caller when combined with other inlining candidates (see [Hölzle 1994]). Computing this context would be

[9] Note that *m.count* and *m.sends* are based on incomplete data since our system does not count the invocations of optimized methods, nor does it use edge counts.

[10] The values of the parameters that our current system uses are MinInvocations = MinSends = 10,000, TinySizeLimit = 50 instructions (200 bytes), and MaxVersion = 7.

very expensive, and this approach was therefore rejected. However, the recompilation system and the compiler do share a common structure in the SELF-93 system; essentially, the recompiler's criteria for walking up the stack are a subset of the compiler's criteria for inlining.

After a recompilation, the system also checks to see if recompilation was effective, i.e., if it actually improved the code. If the previous and new compiled methods have exactly the same non-inlined calls, recompilation did not really gain anything, and thus the new method is marked so it won't be considered for future recompilations.

By deferring recompilation to run time, we hope to achieve high performance on dynamically-typed, pure object-oriented programs without burdening the user with training the system, selecting test cases, or enduring long pauses at run time. But before evaluating its effectiveness, we must pause for a moment to describe the methodology used to characterize the pauses.

5. Pause Clustering: Measuring Pauses in Interactive Systems

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 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 compilation.

2. Individual compilations 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 compilations 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 compilation-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. (Two events separated by half a second clearly can be distinguished.)

Figure 13 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).

This example illustrates that pause clustering is quite conservative and may overestimate the true pauses experienced by the user.[11] However, we believe that pause clustering is more realistic than measuring individual pauses. Furthermore, we hope that this approach will strengthen our results

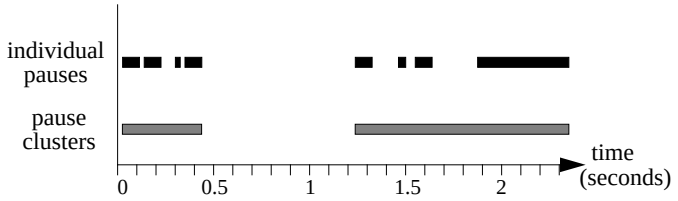

Figure 13. Individual pauses and the resulting pause clusters

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.

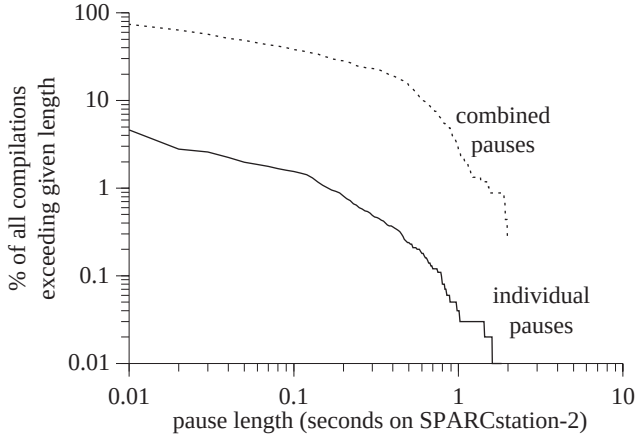

Figure 14. Distribution of individual compile pauses vs. distribution of combined pauses

Figure 14 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 individual pauses would therefore result in a distorted (overly optimistic) pause characterization that would not reflect the user's experience.

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, the results presented in this paper are fairly insensitive to changes in the parameter values. In particular, varying the pause percentage between 35% and 70% does not qualitatively change the results, nor does doubling the intergroup time to one second.

6. Evaluation of Interactive Behavior

In this section, we examine how optimizing compilation influences the interactive behavior of the SELF-93 system. Run-time compilation can create distracting pauses; for example, the first time a

[11]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.

menu pops up, the code to draw the menu must be compiled. Similarly, dynamic recompilation may introduce additional pauses during later executions as code is optimized. The remainder of this section explores the severity of such compilation pauses with a variety of measurements, such as the pauses experienced in an actual interactive session or when starting up large, uncompiled applications.

Ideally, the impact of optimizing compilation on interactive behavior should be measured in terms of the end user's experience—does compilation create additional pauses? Unfortunately, such measurements are hard to obtain since the pauses the end user observes are caused by many factors such as application design (e.g., an inefficient way of drawing menus), garbage collection, and compilation. For this study, we ignore all factors except compilation, and report compilation pauses independently of the context they occur. That is, all compilation pauses are treated equally, whether they occur during a graphical animation (where they are highly visible) or during a lengthy non-interactive computation where the user is unlikely to notice it.

Unless otherwise mentioned, CPU times were measured on an otherwise idle SPARCstation-2 with 64 Mbytes of memory. Compared to current workstations and personal computers, the SPARCstation-2 is slow (with a SPECInt92 of about 20). No significant disk or paging activity occurred during the run. SELF's compiled-code cache was kept large enough to hold the tutorial code so that no optimized code was discarded. The data 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%.

6.1 Compile Pauses During an Interactive Session

We measured the (clustered) compilation pauses occurring during a 50-minute session of the SELF user interface [Chang and Ungar 1993]. The session involved completing a SELF tutorial, which included browsing, editing, and making small programming changes. Although input was not scripted, the pace of the interaction was kept fairly high, so that the session includes few "think pauses". During the interaction a bug in the tutorial's cut-and-paste code was discovered, so that the session also included some "real-life" debugging. Figure 15 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[12] 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 16 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.

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

[12] 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.

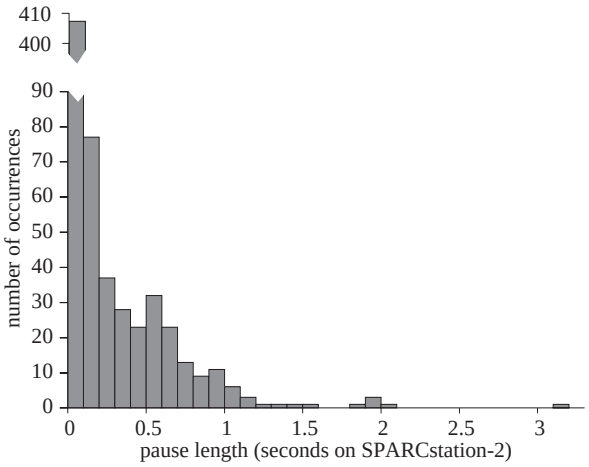

Figure 15. Compile pauses during a 50-minute interaction

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.

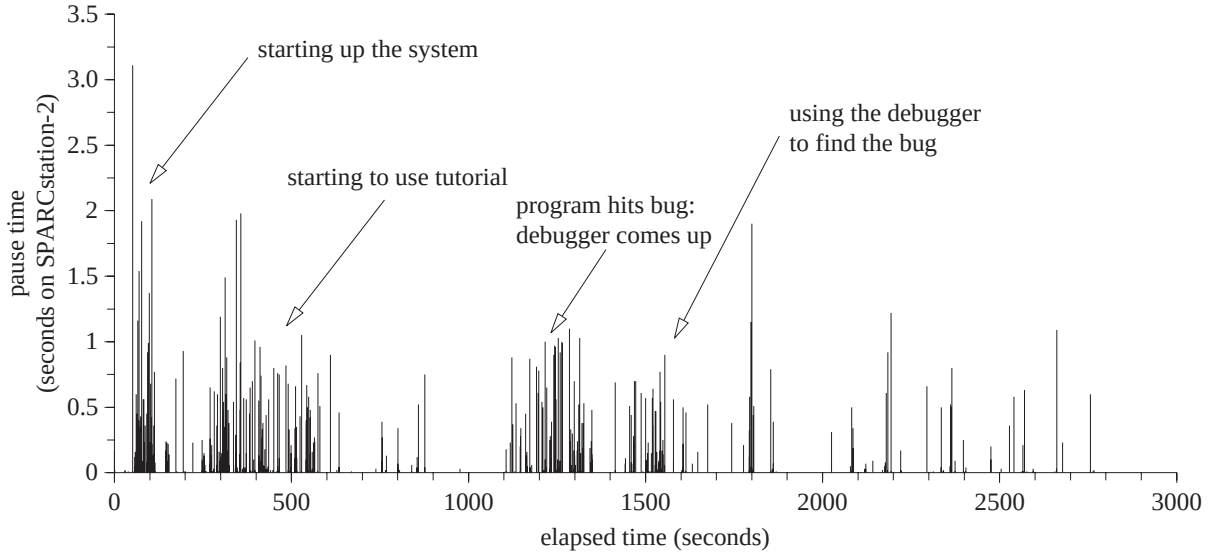

Figure 16. Distribution of compilation pauses over time

The system's pause behavior on the SPARCstation-2 is adequate. Pauses are rarely distracting and much shorter than in traditional programming environments using batch-style compilation. The measured interaction represents a worst-case scenario since it starts a moderately 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.) would already be optimized, and only the application that the programmer is actively changing would need to be (re)compiled.

The system's interactive behavior will improve with faster CPUs. Today's mid-range workstations and PCs are already several times faster than the SPARCstation-2 used for our measurements. Although arguments of the kind "with a faster machine, everything will be better" are often misleading, we believe that our argument escapes this fallacy. 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 of methods will remain the same, and thus individual pauses will become shorter with faster processors even if the programs per se become larger. Larger programs may prolong the time needed for recompilation to settle down (see Section 6.4 below), but in our experience 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.

6.2 Starting New Code

A responsive system should be able to start up new (as yet uncompiled) programs or program parts without introducing long pauses. 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 3). 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 [Chambers 1992]; 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 3: Systems used in the study of start-up times

6.3 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 4 shows the individual interactions of the sequence. For the measurements, the interactions were executed in the sequence shown in the table, with no 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

	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 4: UI interaction sequence

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

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 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 substantial 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 additional 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.

6.4 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 5 lists the large applications used for the study. The programs were started with an empty code

Benchmark	Size (lines)[a]	Description
CecilComp	11,500	Cecil-to-C compiler compiling the Fibonacci function
CecilInt	9,000	interpreter for the Cecil language [Chambers 1993] running a short test program
Typeinf	8,600	type inferencer for SELF [Agesen et al. 1993]
UI1	15,200	prototype user interface using animation techniques [Chang and Ungar 1993][b]

Table 5: Large SELF applications

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

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

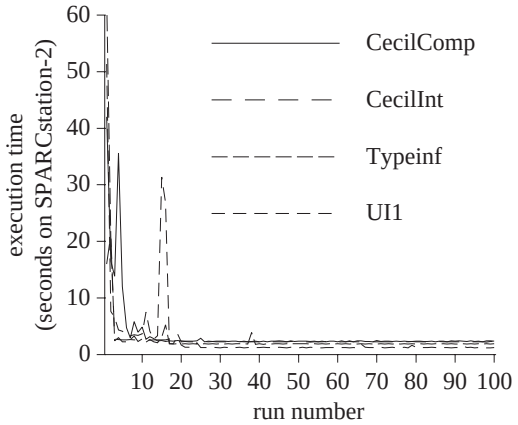

Figure 17. Start-up behavior of large applications

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 18 shows the same data as Figure 17, 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.

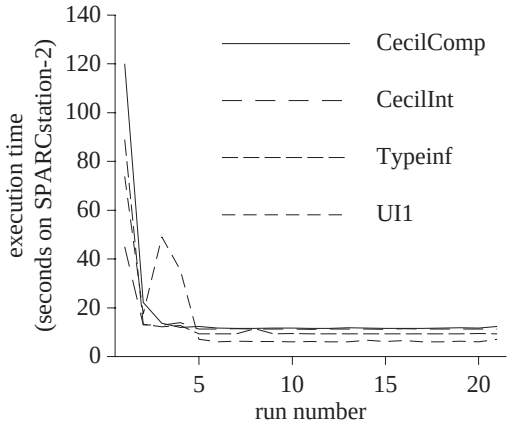

Figure 18. Alternate view of Figure 17

Table 6 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

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

Table 6: Start-up behavior of dynamic compilation

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.[13] In real life, one might expect large programs to run longer, and thus start-up behavior would be better than with our benchmarks.

A program's start-up time should roughly correlate with program size: in general, one would expect larger programs to take longer to reach stable performance since more code has to be (re)compiled. Figure 19 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.[14]) 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.

What exactly causes the first few runs to be so slow? Figure 20 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.[15] To reduce the initial peak in execution time for `UI1`, the non-

[13]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.

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

[15]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.

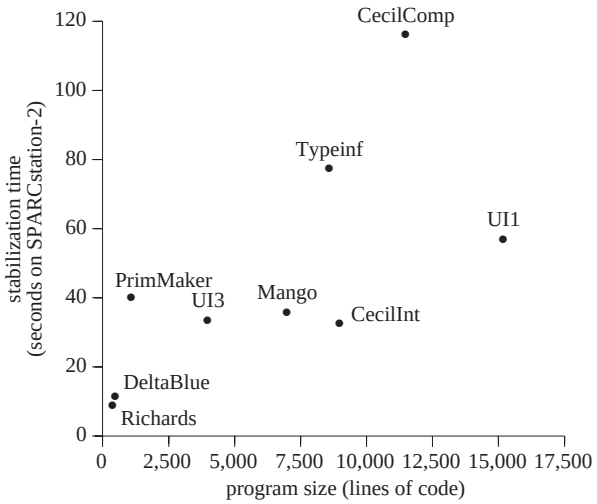

Figure 19. Correlation between program size and time to stable performance

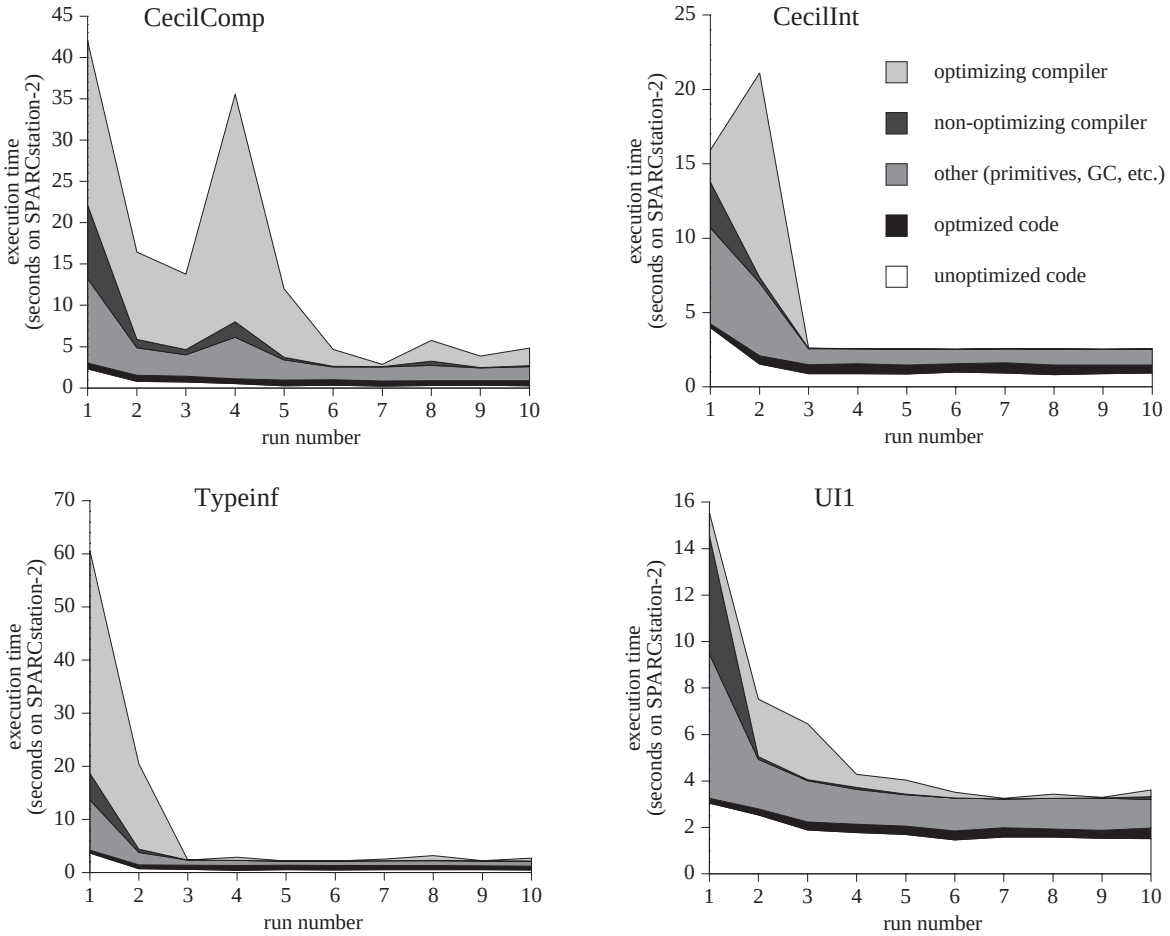

Figure 20. Start-up phase of selected benchmarks

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 minor 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 19), which we consider adequate.

6.5 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 section describes how two of those parameters (recompilation limit and half-life time) 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 in Section 4.1, unoptimized methods have invocation counters; if a counter exceeds the *recompilation limit*, the recompilation system is invoked to determine if optimization is necessary. 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.

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 21 shows the execution times of 100 repetitions of the programs shown in Figure 20, 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 21 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

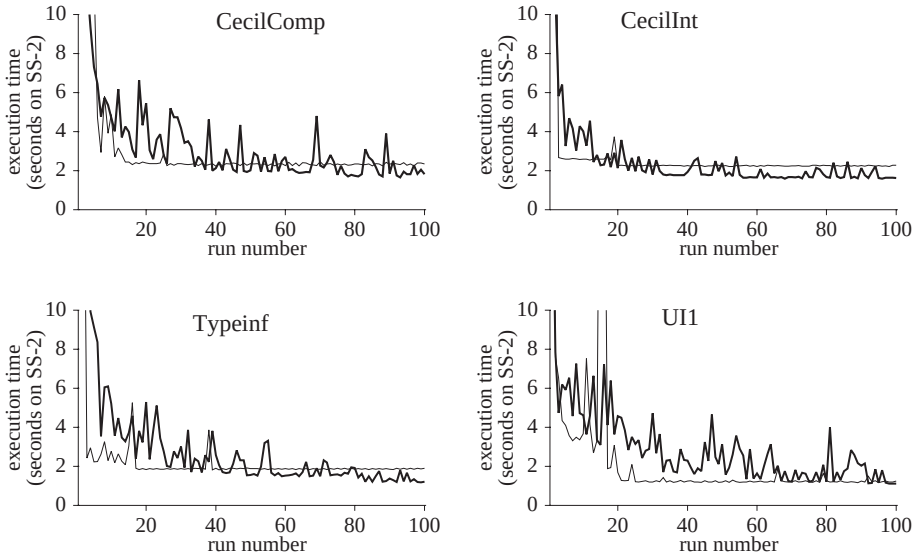

Figure 21. Performance variations if invocation counters don't decay (thin lines: standard system (15 second half-life), thick lines: system without decay)

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.

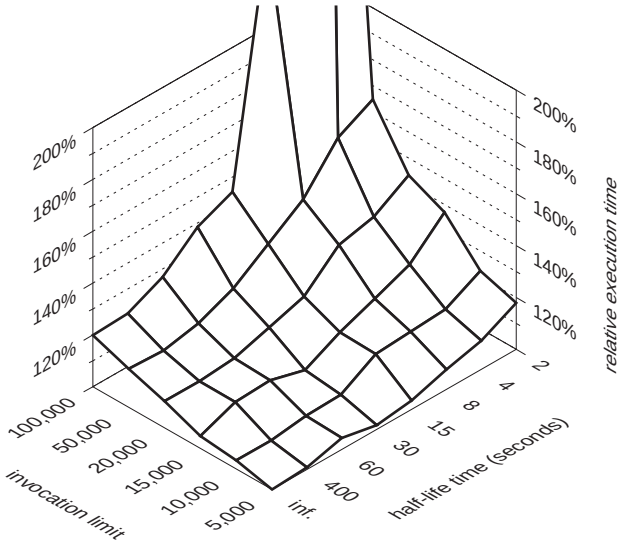

Figure 22. Influence of recompilation parameters on performance

Figure 22 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 their exact location and height 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 21). 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.

7. Related Work

7.1 Type Feedback

Previous systems have used static type prediction to inline operations that depend on the run-time type of their operands. For example, Lisp systems usually inline the integer case of generic arithmetic and handle all other type combinations with a call to a routine in the run-time system. The Deutsch-Schiffman Smalltalk compiler was the first object-oriented system to predict integer receivers for common message names such as "+" [Deutsch and Schiffman 1984]. However, none of these systems predicted types adaptively as does SELF-93.

Other systems have used some form of run-time type information for optimization, although not to the same extent as SELF-93 and not in combination with recompilation. For example, Mitchell's system [Mitchell 1970] specialized arithmetic operations to the run-time types of the operands (similar to SELF-89's customization [Chambers at al. 1989]). Similarly, several APL compilers created specialized code for certain expressions (e.g., [Johnston 1979], [Dyke 1977]). Of these systems, the HP APL compiler [Dyke 1977] came closest to customization and type feedback. The system compiled code on a statement-by-statement basis. In addition to performing APL-specific optimizations, compiled code was specialized according to the specific operand types (number of dimensions, size of each dimension, element type, etc.). This so-called "hard" code could execute much more efficiently than more general versions since the cost of an APL operator varies greatly depending on the actual argument types. If the code was invoked with incompatible types, a new version with less restrictive assumptions was generated (so-called "soft" code). Since the system never used type information to *re*optimize code, the technique is more akin to customization than to type feedback.

The system described in this paper was inspired by the experimental proof-of-concept system described in [Hölzle et al. 1991]. That system was the first to use type feedback (then called "PIC-based inlining") for optimization purposes. However, being an experimental system, its structure and performance was very different. It did not use dynamic recompilation; methods had to be recompiled "by hand," and the system lacked any mechanism for determining "good" recompilation candidates (i.e., it never looked at the callers). As a result, its speedup over a system without type feedback was modest (about 11%). Recently, type feedback combined with static compilation (i.e., based on off-line profiles) has been shown to be effective for the pure object-oriented language Cecil [Grove et al. 1995] and for C++ [Aigner and Hölzle 1995].

The Apple Object Pascal linker [App88] turned dynamically-dispatched calls into statically-bound calls if a type had exactly one implementation (e.g., the system contained only a `CartesianPoint` class and no `PolarPoint` class). Fernandez [Fernandez 1995] performs a similar optimization for Modula-3. Such systems do not exploit the optimization opportunities created by static binding, do not optimize polymorphic calls, and preclude extensibility through dynamic linking. Srivastava and Wall [Srivastava and Wall 1992] perform more extensive link-time optimization but do not optimize calls.

Some type inference systems (e.g., [Agesen et al. 1993], [Pande and Ryder 1994], [Plevyak and Chien 1994], [Agesen and Hölzle 1995]) or simpler static analyses [Dean et al. 1995] can determine the concrete receiver types of message sends. Compared to type feedback, a type inferencer may provide more precise information since it may be able to prove that only a single receiver type is possible at a given call site. However, its information may also be less precise since it may include types that could occur in theory but never happen in practice. (In other words, the information lacks frequency data.) Like link-time optimizations, type inference requires knowledge of the entire program, thus precluding run-time extensibility.

Studies of inlining for more conventional languages like C or Fortran have found that it often does not increase execution speed but tends to increase code size significantly (e.g., [Davidson and Holler 1988], [Cooper et al. 1991], [Chang et al. 1992], [Hall 1991]). In contrast, inlining in SELF results in both significant speedups and only moderate code growth. The main reason for this striking difference is that SELF methods are much smaller on average than C or Fortran procedures, so that inlining can actually reduce code size. (Because of dynamic dispatch, calls in object-oriented languages take up more instructions than conventional procedure calls.) Furthermore, the additional inlining provided by type feedback enables some optimizations to be more effective, reducing code size as well. Finally, inlining is more important for object-oriented languages because calls are more frequent. While this is particularly true for pure object-oriented languages, it is also true for hybrid languages like C++, as shown in [Aigner and Hölzle 1995].

7.2 Adaptive Recompilation

One of the first people concerned with the implementation of efficient but flexible programming systems was Mitchell [Mitchell 1970]. 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 [Hansen 1974] describes an adaptive compiler for Fortran. His compiler optimized the inner loops of Fortran programs at run time. 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 run time; being a batch system, interactive pauses were not an issue.

The Deutsch-Schiffman Smalltalk system [Deutsch and Schiffman 1984] (and its commercial successor, ParcPlace Smalltalk [ParcPlace 1992]) 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. As discussed in Section 3.6, unoptimized Smalltalk code runs roughly three times slower compared to SELF-93, 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 compile programs manually. 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 [ISE 1993] and C++ [SGI 1993] 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.[16] 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

Any language implementation needs (or, at least, desires) both good run-time performance and good interactive performance. A purely object-oriented computational metaphor makes it harder to satisfy these needs since it requires such aggressive optimization that interactive performance suffers. However, by adopting *adaptive recompilation*, a system can provide both good run-time 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. Adaptive recompilation also helps to improve the system's responsiveness to programming changes. For example, it takes less than 15 seconds on a 1991 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).

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. When the pauses of the SELF-93 system were analyzed in this way, the distribution of pauses as perceived by a user was an order of magnitude different from the raw pause distribution. The magnitude of this difference suggests that some sort of perception-based analysis must be used when evaluating interactive performance, for example, when evaluating incremental compilers or garbage collectors.

In future systems, it should be possible to hide compilation pauses even better than in the current SELF-93 system. 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 [Wilson and Moher 1989]). Alternatively, optimizing compilations could be performed in parallel with program execution on a multiprocessor machine.

Adaptive recompilation gives implementors of high-level languages such as Smalltalk a new option for implementing their systems. With the increasing speed of hardware, interpreters or non-optimizing 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 sacrificing responsiveness. We hope that this paper will encourage implementors of high-level languages to

[16] 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 [Hölzle et al. 1992].

explore a new region in the design space, resulting in new high-performance exploratory programming environments for such languages.

Adaptive recompilation also gives implementors of object-oriented languages new sources of information for guiding optimization. In particular, by using type information collected during previous execution of calls (*type feedback*), an optimizing compiler can replace dynamically-dispatched calls with faster inline-substituted code sequences guarded by type tests for the common case(s). Both the process of collecting type information and the inlining transformations based on that information are straightforward and do not pose significant implementation difficulties. With type feedback, a suite of large SELF applications runs 1.7 times faster than without type feedback, and performs 3.6 times fewer calls. On the two medium-sized programs also available in Smalltalk, our new system outperforms a commercial Smalltalk implementation by factors of 2.3 and 3.5, respectively.

We believe that type feedback is an attractive optimization for situations where the exact (implementation-level) type of the arguments to a relatively costly operation is unknown at compile time, and where knowing the types would allow the compiler to generate more efficient code. With the advent of object-oriented languages and their use of late-bound operations, such optimizations are likely to become more important even for statically-typed languages. We believe that type feedback is applicable to both statically-typed and dynamically-typed object-oriented languages (e.g., CLOS, C++, Smalltalk) and to languages with type-dependent generic operators (e.g., APL and Lisp).

Acknowledgments: We are very grateful to Bob Cmelik for making it possible to run SELF under Shade, to Mark D. Hill for Dinero, and to Gordon Irlam for Spanner. We would also like to thank Lars Bak, Bay-Wei Chang, Roger Hayes, Peter Kessler, Brian Lewis, John Maloney, Eliot Moss, Mario Wolczko, and the anonymous referees of OOPSLA, PLDI, and TOPLAS for their comments on earlier versions of this paper and related papers.

References

APPLE COMPUTER, INC., 1988. *Object Pascal User's Manual*.

AGESEN, O., PALSBERG, J., AND SCHWARTZBACH, M., 1993. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. *ECOOP '93 Conference Proceedings*. Lecture Notes in Computer Science, vol. 707, Springer Verlag, Berlin, 247-267.

AGESEN, O. AND HÖLZLE, U., 1995. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. *OOPSLA'95 Conference Proceedings*, 91-107.

AIGNER, G. AND HÖLZLE, U., 1995. *Eliminating Virtual Function Calls in C++ Programs*. Technical Report TRCS 95-22, Department of Computer Science, UCSB.

CHAMBERS, C., UNGAR, D., AND LEE, E., 1989. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *OOPSLA '89 Conference Proceedings*, 29-70.

CHAMBERS, C. AND UNGAR, D., 1991. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Conference Proceedings*, 1-15.

CHAMBERS, C., 1992. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. Thesis, Stanford University, April 1992

CHAMBERS, C., 1993. The Cecil Language—Specification and Rationale. University of Washington, Technical Report UW CS TR 93-03-05, Department of Computer Science, University of Washington.

CHANG, B. AND UNGAR, D., 1993. Animation: From Cartoons to the User Interface. *User Interface Software and Technology Conference Proceedings*.

CHANG, P., MAHLKE, S., CHEN, W., AND HWU, W.-M., 1992. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience* 22 (5), 349-369.

COOPER, K., HALL, M., AND TORCZON, L., 1991. An experiment with inline substitution. *Software—Practice and Experience* 21 (6), 581-601.

CMELIK, R. AND KEPPEL, D., 1993. *Shade: A Fast Instruction-Set Simulator for Execution Profiling*. Technical Report SMLI TR-93-12, Sun Microsystems Laboratories.

DAVIDSON, J. AND HOLLER, A., 1998. A study of a C function inliner. *Software—Practice and Experience* 18(8), 775-90.

DEAN, J. AND CHAMBERS, C., 1994. Towards better inlining decisions using inlining trials. *1994 Conference on Lisp and Functional Programming*, 273-282.

DEAN, J., GROVE, G., AND CHAMBERS, C., 1995. *Optimization of object-oriented programs using static class hierarchy analysis*. *ECOOP '95 Conference Proceedings*. Lecture Notes in Computer Science, vol. 952, Springer Verlag, Berlin.

DEUTSCH, L. P., 1983. *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*. In KRASNER, G., ed., *Smalltalk-80: Bits of History and Words of Advice.* Addison-Wesley, Reading, MA.

DEUTSCH, L. P. AND SCHIFFMAN, A., 1984. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, 297-302.

DIWAN, A., TARDITI, D., AND MOSS, E. B., 1995. Memory Subsystem Performance of Programs with Intensive Heap Allocation. *ACM Trans. on Computer Systems* 13(3), 244-273, August 1995.

VAN DYKE, E., 1977. A dynamic incremental compiler for an interpretative language. *HP Journal*, July 1977, 17-24.

FERNANDEZ, M. Simple and effective link-time optimization of Modula-3 programs. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, 103-115.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C., 1995. Profile-Guided Receiver Class Prediction. *OOPSLA'95 Conference Proceedings*, 108-123.

GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K., 1983. An Execution Profiler for Modular Programs. *Software—Practice and Experience* 13, 671-685.

GUIBAS, L., AND WYATT, D., 1978. Compilation and Delayed Evaluation in APL. *Fifth Annual ACM Symposium on Principles of Programming Languages*, 1-8.

HALL, M. W., 1991. *Managing Interprocedural Optimization.* Ph.D. Thesis, Technical Report COMP TR91-157, Computer Science Department, Rice University, April 1991.

HANSEN, G. 1974, *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. Ph.D. Thesis, Carnegie-Mellon University, 1974.

HILL, M.D. 1987. *Aspects of Cache Memory and Instruction Buffer Performance.* Technical Report UCB/CSD 87/381, Computer Science Division, University of California, Berkeley, November 1987.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1991. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. *ECOOP'91 Conference Proceedings*. *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin.

HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. 1992. Debugging Optimized Code with Dynamic Deoptimization. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, 32-43.

HÖLZLE, U. 1994. Adaptive Optimization in SELF: *Reconciling High Performance with Exploratory Programming*. Ph.D. Thesis, Department of Computer Science, Stanford University, 1994. Also published as Sun Microsystems Laboratories Technical Report 95-35, 1995.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 326-336.

HÖLZLE, U. AND AGESEN, O. 1994. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. *Theory and Practice of Object Systems 1(3)*, 1995.

HWU, W. W. AND CHANG, P. P. 1989. Inline function expansion for compiling C programs. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 246-57.

IRLAM, G. *SPA—SPARC analyzer tool set.* Available via ftp from cs.adelaide.edu.au, 1991.

ISE Inc. ISE Eiffel 3.0, 1993.

JOHNSTON, R. L. 1979. The Dynamic Incremental Compiler of APL\3000. *Proceedings of the APL '79 Conference*. Published as *APL Quote Quad 9(4*), 82-87.

KOOPMAN, P., LEE, P., AND SIEWIOREK, D. 1992. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems 14 (2)*, 265-297,.

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

PANDE, H. AND RYDER, B. 1994. *Static Type Determination for C++*. Technical Report LCSR-TR-197a, Rutgers University, 1994.

PARCPLACE SYSTEMS, 1992. VisualWorks 1.0 Smalltalk System.

PLEVYAK, J. AND CHIEN, A. 1994. Precise concrete type inference for object-oriented languages. *OOPSLA '94 Conference Proceedings*, 324-340.

REINHOLD, M. *Cache Performance of Garbage-Collected Programming Languages*. Technical Report MIT/LCS/TR-581 (Ph.D. Thesis), Massachusetts Institute of Technology, September 1993.

SILICON GRAPHICS, INC., 1993. SGI Delta C++ Programming Environment.

SANNELLA, M., MALONEY, J., FREEMAN-BENSON, B., AND BORNING, A. 1993. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience 23 (5)*, 529-566.

SRIVASTAVA, A. AND WALL, D. 1992. *A Practical System for Intermodule Code Optimization at Link- Time*. DEC WRL Research Report 92/6, December 1992.

UNGAR, D. AND SMITH, R. SELF: The Power of Simplicity. *OOPSLA '87 Conference Proceedings*, 227-241.

WILSON, P. AND MOHER, T. 1989. Design of the Opportunistic Garbage Collector. *OOPSLA '89 Conference Proceedings*, 23-35.

WALL, D. 1991. Predicting Program Behavior Using Real or Estimated Profiles. *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 59-70.

Appendix: Type Feedback Detailed Data

Benchmark	execution time (ms)		
	SELF-93 nofeedback	SELF-93	SELF-91
CecilComp	1,348	953	1,144
CecilInt	2,035	1,085	2,026
DeltaBlue	744	210	687
Mango	2,423	1,526	2,292
PrimMaker	2,520	1,227	2,279
Richards	922	591	693
Typeinf	1,448	769	1,388
UI1	716	686	645
UI3	656	528	571

Table A-1: Execution times

The execution times of the above benchmarks were kept relatively short to allow easy simulation. To make sure that the small inputs do not distort the performance figures, we measured three of the benchmarks with larger inputs. Table A-2 shows that the speedups achieved by type feedback are very similar to the speedups with smaller inputs.

Benchmark	execution time (seconds)		speedup	
	SELF-93 nofeedback	SELF-93	large input	small input[a]
CecilComp-2	97.2	71.5	1.36	1.41
CecilInt-2	38.5	21.9	1.76	1.88
Mango-2	18.5	11.6	1.59	1.59

Table A-2: Performance of long-running benchmarks

[a] computed from the data in Table A-1

Benchmark	unoptimized	SELF-91	SELF-93	SELF-93 nofeedback
CecilComp	3,542,858	N/A	120,418	472,422
CecilInt	1,254,244	262,424	48,383	274,166
DeltaBlue	2,030,319	407,283	202,241	413,024
Mango	3,290,836	642,545	204,048	681,070

Table A-3: Number of dynamically-dispatched calls

Benchmark	unoptimized	SELF-91	SELF-93	SELF-93 nofeedback
PrimMaker	3,934,308	819,277	76,273	602,217
Richards	6,962,721	839,478	151,819	888,817
Typeinf	2,363,131	288,982	101,858	293,815
UI1	1,727,021	256,573	213,145	288,176
UI3	1,274,863	274,262	101,884	301,344

Table A-3: Number of dynamically-dispatched calls

System	execution time (ms)	
	Richards	DeltaBlue
SELF-93	591	210
Smalltalk	2,580[a]	600[a]
C++ (all virtuals)	546	149
C++ (min. virtuals)	249	87
Lisp	2,010[a]	N/A

Table A-4: Performance of other systems

[a] elapsed time (see text)

System	code size (10^3 bytes)	
	Richards	DeltaBlue
SELF-93	11.3	39.9
Smalltalk	N/A	N/A
C++ (all virtuals)	7.6	13.5
C++ (min. virtuals)	7.1	9.3
Lisp	14.7	N/A

Table A-5: Size of compiled code