Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback

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

David Ungar
Sun Microsystems Laboratories, Mountain View, CA
ungar@eng.sun.com

Abstract: Object-oriented programs are difficult to optimize because they execute many dynamically-dispatched calls. These calls cannot easily be eliminated because the compiler does not know which callee will be invoked at runtime. We have developed a simple technique that feeds back type information from the runtime system to the compiler. With this type feedback, the compiler can inline any dynamically-dispatched call. Our compiler drastically reduces the call frequency of a suite of large SELF applications (by a factor of 3.6) and improves performance by a factor of 1.7. We believe that type feedback could significantly reduce call frequencies and improve performance for most other object-oriented languages (statically-typed or not) as well as for languages with type-dependent operations such as generic arithmetic.

1. Introduction

Object-oriented programs are harder to optimize than programs written in languages like C or Fortran. There are two main reasons for this. 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 runtime 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 runtime 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 runtime, 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).

Pure object-oriented languages exacerbate this problem because *every* operation involves a dynamically-dispatched message send. For example, even very simple operations such as instance variable accesses, integer addition, and array accesses conceptually involve message sends in SELF [US87], the programming language used for this study. Consequently, a pure object-oriented language like SELF offers an ideal test case for optimization techniques tackling the problem of frequent dynamically-dispatched calls.

The rest of this paper describes our experience with a new optimization technique based on *type feedback*. With type feedback, our new compiler runs large SELF programs 1.7 times faster than without, and 1.5 times faster than the previous SELF compiler which uses extensive compile-time type analysis instead of type feedback. Although we have implemented type feedback only for the pure dynamically-typed object-oriented language SELF, the technique is language-independent and could be applied to statically-typed, non-pure languages as well.

2. Type Feedback

The key idea of type feedback is to extract type information from executing programs and feed it back to the compiler (Figure 1).

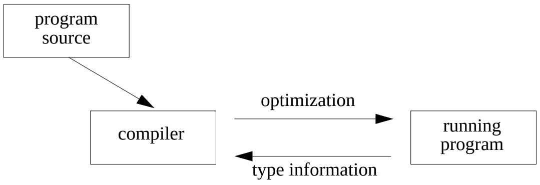

Figure 1. Overview of Type Feedback

In: *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

Specifically, we use an instrumented version of a program to record the program's *type profile*, i.e., a list of receiver types (and, optionally, their frequencies) for every single call site in the program. 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 we have observed in [HCU91], 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.

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* [CU90] 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 [Cha92] 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 [Wall91]. 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 Garrett et al. [G+94] that measured the stability of type profiles in SELF, C++, and Cecil programs confirms our experience.

3. Type feedback in the SELF system

This section describes the implementation of type feedback in SELF; although our implementation makes extensive use of possibilities opened by dynamic compilation, we wish to emphasize that dynamic compilation is not needed to implement type feedback. The reader who is not interested in the particular details of the SELF implementation may safely skip this section and continue with section 4. Section 5 discusses how type feedback could be implemented in a more conventional "batch-style" compilation environment.

Since SELF is dynamically-typed, it has no explicit notion of type. However, the implementation maintains internal type descriptors (called "maps") that describe the exact format of each object (i.e., its storage layout, inheritance structure, etc.). In the remainder of this paper, we will use "type" to refer to these internal implementation types. Translated into C++ parlance, "type" stands for "non-abstract (concrete) class."

3.1 Dynamic recompilation

The SELF-93 system uses dynamic recompilation not only to take advantage of type feedback but also to determine which parts of an application should be optimized at all. Figure 2 shows an overview of the compilation process of the system. When a source method is invoked for the first time, it is compiled quickly by a very simple, completely non-optimizing compiler. If the method is executed often, it is recompiled and optimized using type feedback. Sometimes, an optimized method is reoptimized to take advantage of additional type information or to adapt it to changes in the program's type profile. Combining the optimizing compiler with the fast non-optimizing compiler and dynamic recompilation

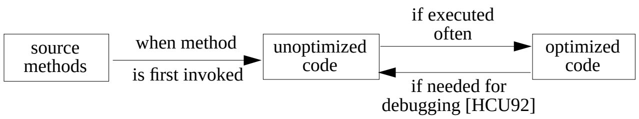

Figure 2. Compilation in the SELF-93 system

allows SELF-93 to achieve high performance while keeping compilation pauses in the sub-second range [Höl94].

In the following sections, we will briefly discuss our implementation of dynamic recompilation; more details can be found in [Höl94].

3.2 When to recompile

Any dynamic recompilation system needs to decide when to recompile code. If the system recompiles too eagerly, compilation time is wasted; if it recompiles too lazily, performance will suffer. SELF-93 uses invocation counts to drive recompilation. Each unoptimized method has its own counter that is incremented in the method prologue. When the counter exceeds a certain limit, the recompiler is invoked to decide which method (if any) should be recompiled. If the method overflowing its counter isn't recompiled, its counter is reset to zero. Counter values decay exponentially with time (i.e., the system monitors invocation rates, not pure invocation counts).

Originally, we envisioned counters as a first step, to be used only until a better solution was found. However, in the course of our experiments we came to realize that the trigger mechanism ("when") is much less important for good recompilation results than the selection mechanism ("what").

3.3 What to recompile

To find a "good" candidate for recompilation, the recompiler walks up the call chain and inspects the callers of the method triggering the recompilation. A caller is recompiled if it performs many calls to unoptimized or small methods (the hope being that these calls will be eliminated), or if it creates closure objects. (SELF implements all control structures using message passing and closures; when control structures are inlined, the closures can typically be eliminated.) A simpler recompilation strategy would always recompile the method whose counter overflowed, since it obviously was invoked often. But suppose that the method just returns a constant. Optimizing this method would not gain much; rather, the method should be inlined into its caller, and thus it is necessary to inspect the callers before deciding what to recompile.

If a recompilee is found, it is (re)optimized, and the old version is discarded. During the compilation, the compiler marks the restart point (i.e., the point where execution will be resumed) and tries to compute the contents of all live registers at that point. If this is successful,[†] the reoptimized method replaces the corresponding unoptimized methods on the stack, possibly replacing several unoptimized activation records with a single optimized activation record. (This process is the reverse of dynamic deoptimization as described in [HCU92]; that paper also describes how the compiler represents the source-level state of optimized code.)

The system tries to optimize 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.) Thus, if the newly optimized method isn't at the top of the stack, recompilation continues with the method's callee. If the old method cannot be replaced on the stack, it is left to finish its current activation(s), but subsequent invocations will always use the new, optimized version.

Finally, the recompilation 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.

3.4 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. Essentially, 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.

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. 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.

3.5 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 [CUL89], and splitting [CU90]—and generates a graph of intermediate code nodes. The back end performs only very 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 runtime 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.

The main differences between SELF-93 and the SELF-91 compiler described by Chambers [Cha92] are that we have substituted type feedback for iterative type analysis, and that our back end is less ambitious. As a result, SELF-93 is considerably simpler (11,000 vs.

[†] 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 situations and signal them to the recompilation system.

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 [Cha92] 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 the next section is probably a conservative indication of what a fully optimizing SELF compiler with type feedback could achieve.

4. Results

To evaluate the performance of the SELF-93 compiler and the contribution of type feedback, we measured the runtime performance of several large SELF programs (see Table A-1 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 1 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 [Cha92] 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 [DS84])
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

4.1 Methodology

To accurately measure execution times, the programs were run under a SPARC simulator based on the Spa [Irl91] and Shade [CK93] tracing tools and the Dinero cache simulator [Hill87]. 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 [KLS92], [Rei93], [DTM94].

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.[†]

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 beyond the scope of this paper and will be the subject of a separate study.) SELF-91 and SELF-93-nofeedback do not use recompilation, so we used the second run for our measurements.

4.2 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 1. Figure 3 on the next page shows the results (Table A-2 in the appendix contains detailed data). 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 4.4 and 4.5 will analyze the reasons for the increased performance of SELF-93 in more detail.) SELF-93 also outperforms SELF-91 by a considerable 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.

4.3 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

[†] 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.

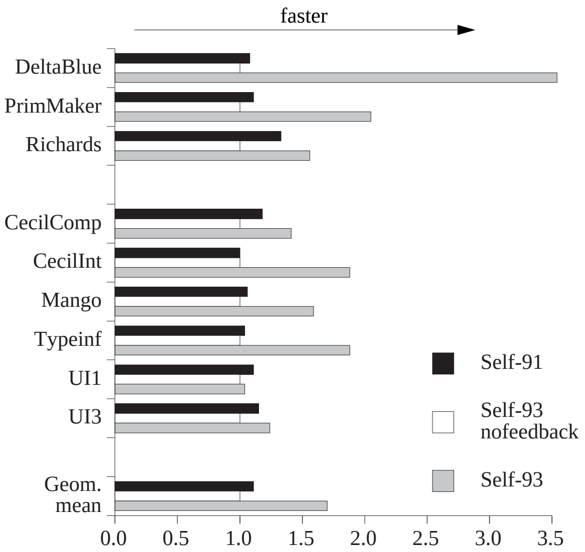

Figure 3. Performance impact of type feedback (all speeds relative to Self-93-nofeedback)

accesses to instance variables in the receiver). Both Self-91 and Self-93 run many times faster than unoptimized programs.

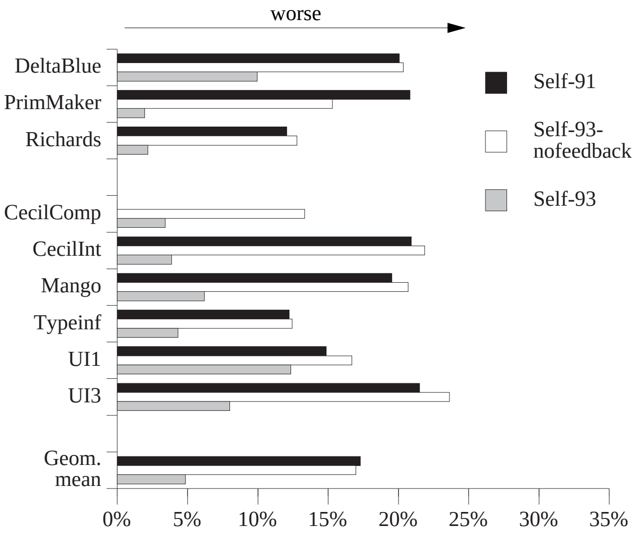

Figure 4. Impact of type feedback on number of calls (all numbers are relative to unoptimized Self)

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 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.

4.4 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*). The latter are used because in dynamically-typed languages it is harder (but not impossible [Dri93]) to use indirect function calls for dynamically-dispatched calls. Instead, Self uses *Polymorphic Inline Caches* [HCU91] which implement a dynamically-dispatched call as a typecase statement (to determine the receiver type) followed by a direct call. This implementation of dynamic dispatch can compete well with the indirect-call implementation typically used by C++ systems: on the SPARCstation-2, Self-93 uses an average of 12 cycles per dispatched call (including cache effects) for the programs measured, whereas a C++ virtual call uses 10 cycles (excluding cache effects).

The average number of type tests executed per send (i.e., the number of branches in the `if` statement 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.

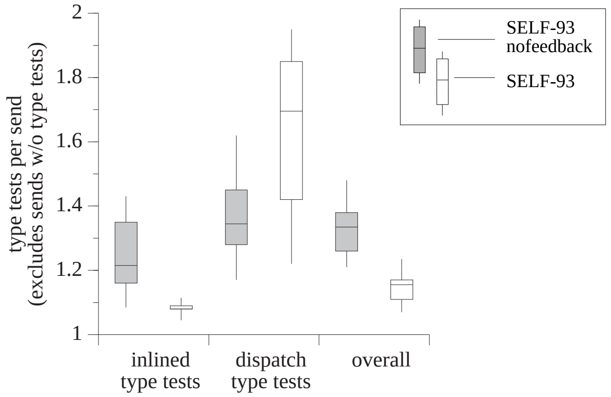

Figure 5. Number of type tests per dispatched send

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). Where the median and mean differ significantly, we indicate the mean with a dot (•).

Self-93-nofeedback executes some inlined type tests because it uses static type prediction [DS84] 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 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 its target. 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-

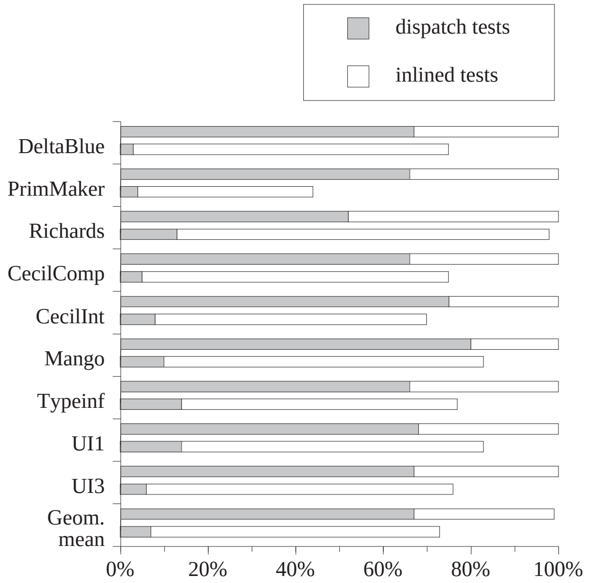

Figure 6. Number of type tests relative to SELF-93-nofeedback (upper bars: SELF-93-nofeedback, lower bars: SELF-93)

93 programs execute 27% fewer type tests. At first sight, such a reduction seems impossible: since 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, 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.)

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 very rudimentary dataflow analysis. With a more sophisticated analysis, this "bonus" might be even higher.

4.5 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 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 (because of increased register pressure), or higher instruction cache misses. (All of the above measurements include cache effects.)

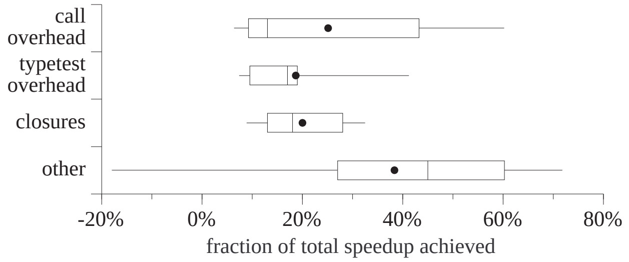

Figure 7. Reasons for SELF-93's improved performance

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.

4.6 Code growth

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 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%.

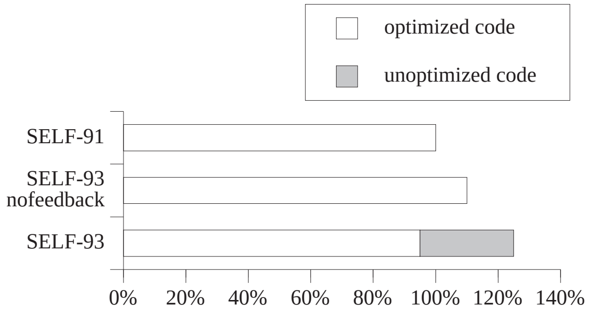

Figure 8. Size of compiled code relative to SELF-91

4.7 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 1 for details about the C++ and Smalltalk systems, and Table A-5 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 10 shows the results.

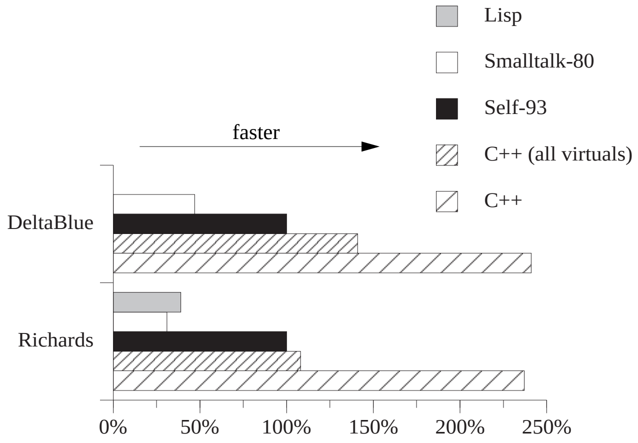

Figure 9. Execution speed (Self-93 = 100%)

For DeltaBlue and Richards, Self-93 runs 2.2 and 3.3 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 [Cha92]. For Richards, Self-93 runs 2.6 times faster than an equivalent CommonLisp program compiled with maximum optimization and minimum safety (i.e., the Lisp code would not detect some runtime 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, the C++ programmer has a choice of programming style: either she uses virtual functions liberally to get more flexibility, reusability, and maintainability, or she minimizes 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 clearly 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 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).

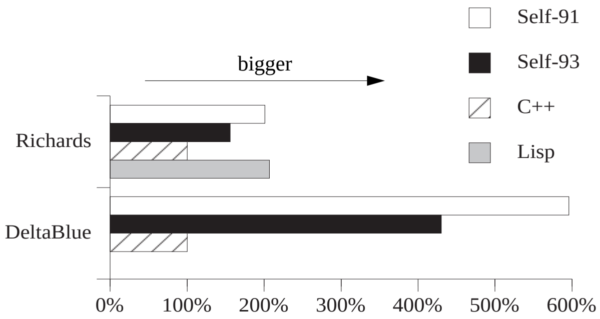

Figure 10. Code size relative to GNU C++

Figure 9 shows that for Richards and DeltaBlue, the additional inlining performed by Self-93 actually decreases code size relative to Self-91 (see Table A-6 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).† Fortunately, our experience with larger applications suggests that DeltaBlue is a pathological case rather than the norm.

5. Applicability to other systems

As demonstrated by the above measurements, 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?

5.1 Type feedback and static compilation

Type feedback is in no way 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.

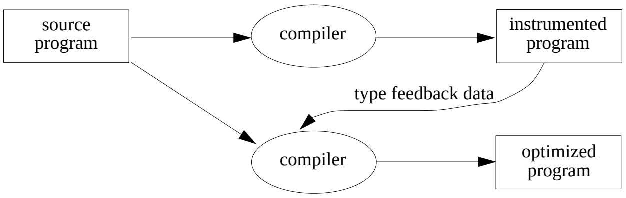

Figure 11. Type feedback in a statically compiled system

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 [GKM83]. (The standard `gprof`

† 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).

profiler already collects almost all information needed by type feedback, except that its data is 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.

As mentioned above, 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 representative. On the other hand, a dynamic recompilation system has a significant advantage because it can dynamically adapt to changes in the program's behavior.

5.2 Applicability to other languages

Obviously, type feedback could be used for 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? We cannot give a definitive answer since would require measurements of actual implementations, which are not available. Instead, we discuss the applicability of type feedback using Smalltalk and C++ as examples.

Type feedback is directly applicable to Smalltalk, and we expect the resulting speedups to be similar to those achieved for SELF. Despite some language differences (e.g. prototype- vs. class-based inheritance), the two languages have very similar execution characteristics (e.g., a high frequency of message sends, intensive heap allocation, use of closures to implement user-defined control structures, etc.) and thus very similar sources of inefficiency.

C++'s execution behavior (and language philosophy) is much further away from SELF, but we believe it will nevertheless benefit from type feedback. First, measurements of large C++ programs [CGZ94] have shown that calls are almost five times more frequent in C++ programs than in C programs, and that the average size of a C++ virtual function is only 30 instructions, six times smaller than the average C function. Second, the two C++ programs we measured in section 4.7 slowed down by factors of 1.7 and 2.2 when using virtual functions everywhere, demonstrating that current C++ compilers do not optimize such calls well. Third, we expect that C++ programmers will make even more use of virtual functions in the future as they become more familiar with object-oriented programming styles; for example, recent versions of the Interviews framework [LVC89] use virtual functions more frequently than previous versions.

To give a concrete example, the DOC document editor measured in [CGZ94] performs a virtual call every 75 instructions; given that a C++ virtual call uses about 5 instructions and usually incurs two load stalls and a stall for the indirect function call, we estimate that this program spends roughly 10% of its time dispatching virtual functions. If type feedback could eliminate a large fraction of these calls, and if the indirect benefits of inlining in C++ are similar to those measured for SELF (i.e., total savings are 4-6 times higher than the call overhead alone, see Figure 7), substantial speedups appear possible.

For type feedback to work well, the dynamic number of receiver types per call site should be close to one, i.e., one or two receiver types should dominate. A large fraction of call sites in C++ have this property [CG94][G+94], and it also holds in other object-oriented programming languages (e.g., Smalltalk, SELF, Sather, and Eiffel); this is the reason that *inline caching* [DS84], [HCU91] works well in these languages as an implementation of dynamic dispatch. Therefore, we expect type feedback to work well for these languages; the higher the frequency of dynamically-dispatched calls, the more beneficial type feedback could be.

6. Related work

Previous systems have used static type prediction to inline operations that depend on the runtime 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 runtime system. The Deutsch-Schiffman Smalltalk compiler was the first object-oriented system to predict integer receivers for common message names such as "+" [DS84]. However, none of these systems predicted types adaptively as does SELF-93.

Other systems have used some form of runtime type information for optimization, although not to the same extent as SELF-93 and not in combination with recompilation. For example, Mitchell's system [Mit70] specialized arithmetic operations to the runtime types of the operands (similar to SELF-89's customization [CUL89]). Similarly, several APL compilers created specialized code for certain expressions (e.g. [Joh79], [Dyk77], [GW78]). Of these systems, the HP APL compiler [Dyk77] 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 wildly 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.

Customization can be viewed as a restricted version of type feedback that attempts to minimize type tests by placing the receiver type test at the beginning of the method. Unlike type feedback, customization benefits only a restricted set of sends (namely those involving `self`). As implemented in SELF, customization is also more eager (i.e., all methods are always customized right away) and more static (all programs are treated the same way). In contrast, type feedback in SELF-93 is more lazy and adaptive.

The system described in this paper was inspired by the experimental proof-of-concept system described in [HCU91]. That system was the first one 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 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%). Based on measurements of C++ programs, Calder and Grunwald [CG94] argue that type feedback would be beneficial for C++; their proposed "if conversion" appears to be identical to inline caching [DS84] and PIC-based inlining [HCU91], except that it is performed statically.

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). The disadvantage of such a system is that it still leaves the procedure call overhead even for very simple callees, does not optimize polymorphic calls, and precludes extensibility through dynamic linking. (Srivastava and Wall [SW92] perform more extensive link-time optimization but do not optimize calls.)

Some type inference systems (e.g., [APS93], [PR94]) 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, the main problem with type inference is that it requires knowledge of the entire program, thus precluding dynamic linking.

Studies of inlining for more conventional languages like C or Fortran have found that it often does not significantly increase execution speed but tends to significantly increase code size (e.g., [DH88], [HwC89], [CHT91], [CM+92], [Hall91]). In contrast, inlining in SELF results in both significant speedups and only very 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 we have observed in section 5.2.

7. Conclusions

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). The process of collecting type information and the inlining transformations based on that information are both straightforward and do not pose significant implementation difficulties. 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).

We have implemented a compilation system for SELF that dynamically recompiles often-used code and uses type feedback to generate better code. The system uses simple heuristics to decide which methods to recompile, how much to rely on type feedback, and how much to optimize. The resulting implementation is stable enough to be used by other researchers as part of their daily work.

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 2.2 and 3.3, 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.

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. Many thanks also to all the people who have commented on earlier versions of this paper: Lars Bak, Roger Hayes, Peter Kessler, Brian Lewis, John Maloney, and Mario Wolczko, and the anonymous referees of PLDI ‘94 who provided valuable suggestions for improvements.

References

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

[App88] Apple Computer, Inc. *Object Pascal User’s Manual*. Cupertino, 1988.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

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

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

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA ‘89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN ‘90 Conference on Programming Language Design and Implementation*, p. 150-164, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990.

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

[CM+92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience* 22 (5): 349-369, May 1992.

[CK93] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling.* Technical Report SMLI TR-93-12, Sun Microsystems Laboratories, 1993. Also published as Technical Report CSE-TR-93-06-06, University of Washington, 1993.

[CHT91] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience* 21 (6): 581-601, June 1991.

[DH88] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience* 18(8): 775-90, August 1988.

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

[DTM94] Amer Diwan, David Tarditi, and Eliot Moss. Memory Subsystem Performance of Programs with Intensive Heap Allocation. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 1-14, January 1994.

[Dri93] Karel Driesen. Selector Table Indexing and Sparse Arrays. *OOPSLA '93 Conference Proceedings*, p. 259-270, Washington, D.C., 1993. Published as *SIGPLAN Notices 28(10),* September 1993.

[Dyk77] Eric J. Van Dyke. A dynamic incremental compiler for an interpretative language. *HP Journal*, p. 17-24, July 1977.

[G+94] Charles D. Garrett, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report CSE-TR-94-03-05, University of Washington, February 1994.

[GKM83] S. L. Graham, P. B. Kessler, and M. K. McKusick. An Execution Profiler for Modular Programs. *Software—Practice and Experience* 13:671-685, 1983.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, p. 1-8, 1978.

[Hall91] Mary Wolcott Hall. *Managing Interprocedural Optimization.* Technical Report COMP TR91-157 (Ph.D. Thesis), Computer Science Department, Rice University, April 1991.

[Hill87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* Technical Report UCB/CSD 87/381, Computer Science Division, University of California, Berkeley, November 1987.

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

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code With Dynamic Deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, p. 21-38, San Francisco, 1992. Published as *SIGPLAN Notices 27(6),* June 1992.

[Höl94] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Ph.D. Thesis, Stanford University, Computer Science Department, 1994. (In preparation.)

[HwC89] W. W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, p. 246-57, Portland, OR, June 1989. Published as *SIGPLAN Notices 24(7)*, July 1989.

[Irl91] Gordon Irlam. *SPA—SPARC analyzer tool set.* Available via ftp from cs.adelaide.edu.au, 1991.

[Joh79] Ronald L. Johnston. The Dynamic Incremental Compiler of APL\3000. In *Proceedings of the APL '79 Conference*. Published as *APL Quote Quad 9(4)*, p. 82-87, 1979.

[KLS92] Philip Koopman, Peter Lee, and Daniel Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems 14 (2)*:265-297, April 1992.

[LVC89] Mark Linton, John Vlissides, and Paul Calder. Composing User Interfaces with Interviews. *IEEE Computer 22(2)*:8-22, February 1989.

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

[PR94] Hemant D. Pande and Barbara G. Ryder. *Static Type Determination for C++*. Technical Report LCSR-TR-197a, Rutgers University, 1994.

[Rei93] Mark Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. Technical Report MIT/LCS/TR-581 (Ph.D. Thesis), Massachusetts Institute of Technology, September 1993.

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

[SW92] Amitabh Srivastava and David Wall. *A Practical System for Intermodule Code Optimization at Link- Time.* DEC WRL Research Report 92/6, December 1992.

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

[Wall91] David Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, p. 59-70, Toronto, Canada, June 1991. Published as *SIGPLAN Notices 26(6)*, June 1991.

Appendix: Detailed Data

	Benchmark	Size[a]	Description
small benchmarks	DeltaBlue	500	two-way constraint solver [SM+93] 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 [Cha93] 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 [APS93]
	UI1	15,200	prototype user interface using animation techniques [CU93][b]
	UI3	4,000	experimental 3D user interface[b]

Table A-1: 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

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-2: 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-3 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-3: Performance of long-running benchmarks

[a] computed from the data in Table A-2

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
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-4: 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-5: 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-6: Size of compiled code