Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle
Craig Chambers
David Ungar[†]

Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{urs,craig,ungar}@self.stanford.edu

Abstract: *Polymorphic inline caches* (PICs) provide a new way to reduce the overhead of polymorphic message sends by extending inline caches to include more than one cached lookup result per call site. For a set of typical object-oriented SELF programs, PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when *recompiling* a method. An experimental version of such a system achieves a median speedup of 27% for our set of SELF programs, reducing the number of non-inlined message sends by a factor of two.

Implementations of dynamically-typed object-oriented languages have been limited by the paucity of type information available to the compiler. The abundance of the type information provided by PICs suggests a new compilation approach for these languages, *adaptive compilation*. Such compilers may succeed in generating very efficient code for the time-critical parts of a program without incurring distracting compilation pauses.

1. Introduction

Historically, dynamically-typed object-oriented languages have run much slower than statically-typed languages. This disparity in performance stemmed largely from the relatively slow speed and high frequency of message passing and from the lack of type information which could be used to reduce these costs. Recently, techniques such as type analysis, customization, and splitting have been shown to be very effective in reducing this disparity: for example, these techniques applied to the SELF language bring its performance to within a factor of two of optimized C for small C-like programs such as the Stanford integer benchmarks [CU90, CU91, Cha91]. However, larger, object-oriented SELF programs benefit less from these techniques.[‡] For example, the `Richards` operating system benchmark in SELF is four times slower than optimized C.

In addition, techniques like type analysis lengthen compile time. In an interactive environment based on dynamic compilation, compilations occur frequently, and a slow compiler may lead to distracting pauses. Thus, although techniques such as type analysis can improve code quality significantly, they may sometimes degrade overall system performance.

[†] Current address: David Ungar, Sun Labs, Sun Microsystems, Mail Stop MTV 10-21, 2500 Garcia St., Mountain View, CA 94043.

[‡] By "C-like" we mean programs that operate on relatively simple data structures like integers and arrays; but unlike in C, all primitive operations are safe, e.g. there are checks for out-of-bounds accesses and overflow. By "object-oriented" we mean programs which manipulate many user-defined data structures (types) and exploit polymorphism and dynamic typing.

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

We propose a new approach to the optimization of dynamically-typed object-oriented languages based on *polymorphic inline caches* (PICs). As an immediate benefit, PICs improve the efficiency of polymorphic message sends. More importantly, they collect type information which may be used by the compiler to produce more efficient code, especially for programs written in an object-oriented style where type analysis often fails to extract useful type information. In addition, the new wealth of type information enables design trade-offs which may lead to faster compilation.

The first part of the paper describes polymorphic inline caches, explains how they speed up polymorphic message sends, and evaluates their impact on the execution time of some medium-sized object-oriented programs. With PICs, the `Richards` benchmark in SELF runs 52% faster than without them.[†]

The second part explains how the type information accumulated in the PICs can be used to guide the compilation of programs and evaluates the impact of these techniques using an experimental version of the SELF compiler. For a set of typical object-oriented programs, the experimental system obtains a median speedup of 27% over the current SELF compiler and significantly reduces the number of non-inlined message sends.

The third part outlines a framework for efficient implementations of dynamically-typed object-oriented languages based on *adaptive compilation*. This framework offers a wide variety of trade-offs between compile time and execution efficiency and promises to produce systems which are simpler, more efficient, and less susceptible to performance variations than existing systems.

2. Background

To present PICs in context, we first review existing well-known techniques for improving the efficiency of dynamically-typed object-oriented languages. All of these techniques have been used by Smalltalk-80[‡] implementations.

2.1. Dynamic Compilation

Early implementations of Smalltalk interpreted the byte codes produced by the Smalltalk compiler [GR83]. The interpretation overhead was significant, so later implementations doubled performance by dynamically compiling and caching machine code [DS84]. This technique is known as *dynamic compilation* (called "dynamic translation" in [DS84]).

Translation to native code is the basis for efficient implementations; most of the techniques described here would not make sense in an interpreted system. We therefore assume for the rest of this paper that methods are always translated to machine code before they are executed, and that this translation can occur at any time, i.e. may be interleaved with normal program execution. This means that the entire source program must be accessible at all times so that any part of it can be compiled at any time.

2.2. Lookup Caches

Sending a dynamically-bound message takes longer than calling a statically-bound procedure because the program must find the correct target method according to the run-time type of the receiver and the inheritance rules of the language. Although early Smalltalk systems had simple inheritance rules and relatively slow interpreters, method lookup (also known as message lookup) was still responsible for a substantial portion of execution time.

Lookup caches reduce the overhead of dynamically-bound message passing. A lookup cache maps (receiver type, message name) pairs to methods and holds the most recently used lookup results. Message sends first consult the cache; if the cache probe fails, they call the normal (expensive) lookup routine and store the result in the cache, possibly replacing an older lookup result. Lookup caches are very effective in reducing the lookup overhead. Berkeley Smalltalk, for example, would have been 37% slower without a cache [UP83].

[†] In this paper, we will consistently use speedups when comparing performance; for instance, "X is 52% faster than Y" means that Y's execution time is 1.52 times X's execution time.

[‡] Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

2.3. Inline Caches

Even with a lookup cache, sending a message still takes considerably longer than calling a simple procedure because the cache must be probed for every message sent. However, sends can be sped up further by observing that the type of the receiver at a given call site rarely varies; if a message is sent to an object of type *X* at a particular call site, it is very likely that the next time the send is executed it will also have a receiver of type *X*.

This locality of type usage can be exploited by caching the looked-up method address at the call site, e.g. by overwriting the call instruction. Subsequent executions of the send code jump directly to the cached method, completely avoiding any lookup. Of course, the type of the receiver could have changed, and so the prologue of the called method must verify that the receiver's type is correct and call the lookup code if the type test fails. This form of caching is called *inline caching* since the target address is stored at the send point, i.e. in the caller's code [DS84].

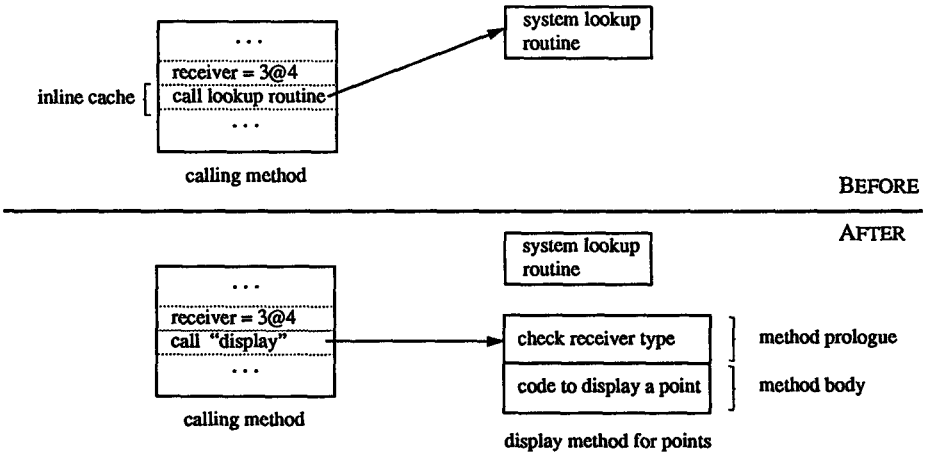

Figure 1. Inline Caching

Inline caching is surprisingly effective, with a hit ratio of 95% for Smalltalk code [DS84, Ung86, UP87]. SOAR (a Smalltalk implementation for a RISC processor) would be 33% slower without inline caching [Ung86]. All compiled implementations of Smalltalk that we know of incorporate inline caches, as does our SELF system [CUL89].

3. Handling Polymorphic Sends

Inline caches are effective only if the receiver type (and thus the call target) remains relatively constant at a call site. Although inline caching works very well for the majority of sends, it does not speed up a polymorphic call site with several equally likely receiver types because the call target switches back and forth between different methods.† Worse, inline caching may even slow down these sends because of the extra overhead associated with inline cache misses. The performance impact of inline cache misses becomes more severe in highly efficient systems, where it can no longer be ignored. For example, measurements for the SELF system show that the `Richards` benchmark spends about 25% of its time handling inline cache misses [CUL89].

An informal examination of polymorphic call sites in the SELF system showed that in most cases the degree of polymorphism is small, typically less than ten. The degree of polymorphism of sends seems to have a trimodal distribution: sends are either *monomorphic* (only one receiver type), *polymorphic* (a few

† We will use the term "polymorphic" for call sites where polymorphism is *actually* used. Consequently, we will use "monomorphic" for call sites which do not actually use polymorphism even though they might *potentially* be polymorphic.

receiver types), or *megamorphic* (very many receiver types). This observation suggests that the performance of polymorphic calls can be improved with a more flexible form of caching. This section describes a new technique to optimize polymorphic sends and presents performance measurements to estimate the benefits of this optimization.

3.1. Polymorphic Inline Caches

The *polymorphic inline cache* (PIC) extends inline caching to handle polymorphic call sites. Instead of merely caching the last lookup result, PICs cache *all* lookup results for a given polymorphic call site in a specially-generated stub routine. An example will illustrate this.

Suppose that a method is sending the `display` message to all elements in a list, and that so far, all list elements have been rectangles. (In other words, the `display` message has been sent monomorphically.) At this point, the situation is identical to normal inline caching:

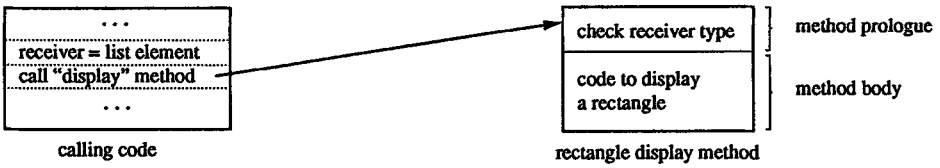

Figure 2. Inline cache after first send

Now suppose that the next list element is a circle. The inline cache calls the `display` method for rectangles which detects the cache miss and calls the lookup routine. With normal inline caching, this routine would rebind the call to the `display` method for circles. This rebinding would happen every time the receiver type changed.

With PICs, however, the miss handler constructs a short stub routine and rebinds the call to this stub routine. The stub checks if the receiver is either a rectangle or a circle and branches to the corresponding method. The stub can branch directly to the method's body (skipping the type test in the method prologue) because the receiver type has already been verified. Methods still need a type test in their prologue because they can also be called from monomorphic call sites which have a standard inline cache.

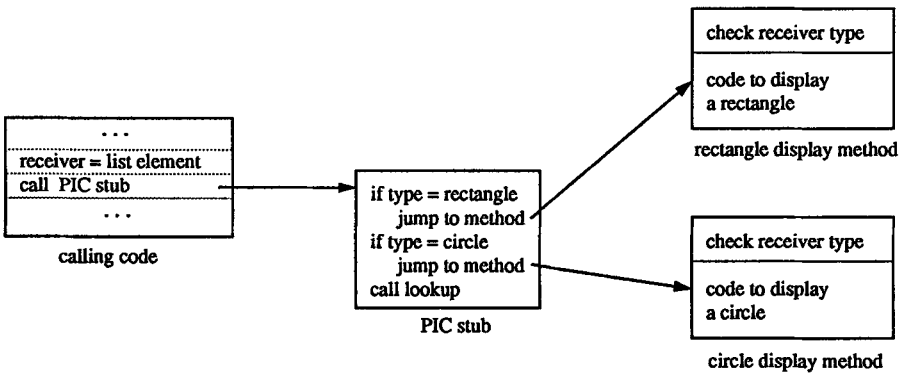

Figure 3. Polymorphic inline cache

If the cache misses again (i.e. the receiver is neither a rectangle nor a circle), the stub routine will simply be extended to handle the new case. Eventually, the stub will contain all cases seen in practice, and there will be no more cache misses or lookups. Thus, a PIC isn't a fixed-size cache similar to a hardware data cache; rather, it should be viewed as an extensible cache in which no cache item is ever displaced by another (newer) item.

3.2. Variations

The PIC scheme described above works well in most cases and reduces the cost of a polymorphic send to a few machine cycles. This section discusses some remaining problems and possible solutions.

Handling megamorphic sends. Some send sites may send a message to a very large number of types. For example, a method might send the `writeSnapshot` message to every object in the system. Building a large PIC for such a send wastes time and space. Therefore, the inline cache miss handler should not extend the PIC beyond a certain number of type cases; rather, it should mark the call site as being megamorphic and adopt a fall-back strategy, possibly just the traditional monomorphic inline cache mechanism.

Improving linear search. If the dynamic usage frequency of each type were available, PICs could be reordered periodically in order to move the most frequently occurring types to the beginning of the PIC, reducing the average number of type tests executed. If linear search is not efficient enough, more sophisticated algorithms like binary search or some form of hashing could be used for cases with many types. However, the number of types is likely to be small on average so this optimization may not be worth the effort: a PIC with linear search is probably faster than other methods for most situations which occur in practice.

Inlining short methods. Many methods are short: for example, it is very common to have methods which just return one of the receiver's instance variables. In SELF, many of these sends are inlined away by the compiler, but non-inlined access methods still represent about 10%-20% of total runtime (30%-50% of all non-inlined sends) in typical programs. At polymorphic call sites, short methods could be integrated into the PIC instead of being called by it. For example, suppose the lookup routine finds a method that just loads the receiver's x field. Instead of calling this method from the stub, its code could be copied into the stub, eliminating the call / return overhead.[†]

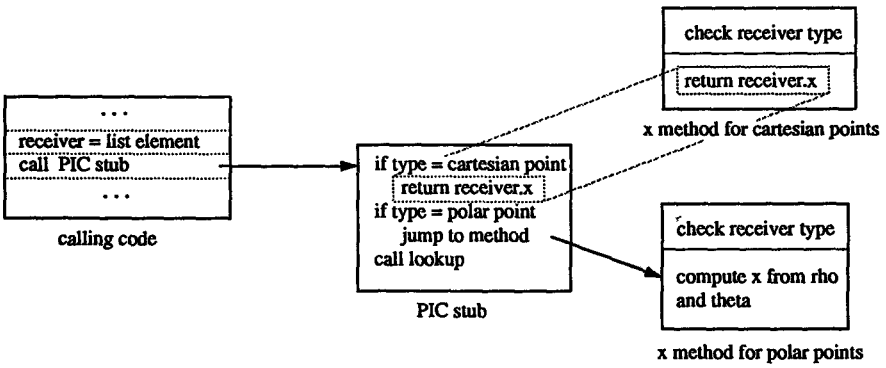

Figure 4. Inlining a small method into the PIC

Improving space efficiency. PICs are larger than normal inline caches because of the stub routine associated with every polymorphic call site. If space is tight, call sites with identical message names could share a common PIC to reduce the space overhead. In such a scenario, PICs would act as fast message-specific lookup caches. The average cost of a polymorphic send is likely to be higher than with call-site-specific PICs because the number of types per PIC will increase due to the loss of locality (a shared PIC will contain all receiver types for the particular message name, whereas a call-specific PIC only contains the types which actually occur at that call site). If the number of types is large, a shared PIC implemented with a hash table should be faster than the global lookup cache because the message name need not be verified and because the hit ratio will approach 100%.

[†] The CLOS implementation described in [KiRo89] uses a similar technique to speed up access methods (called "reader methods" in CLOS). The authors report that access methods represent 69% of the dynamically executed method calls for a set of large CLOS applications.

3.3. Implementation and Results

We implemented PICs for the SELF system, an efficient implementation of a dynamically-typed object-oriented language [CUL89, CU90, CU91]. All measurements were done on a lightly-loaded Sun-4/260 with 48 MB of memory. The base system used for comparison was the current SELF system as of September 1990. It uses inline caching; a send takes 8 instructions (9 cycles) until the method-specific code is reached (see Appendix B). An inline cache miss takes about 15 microseconds (250 cycles). This time could be reduced by some optimizations and by recoding critical parts in assembly. We estimate that such optimizations could reduce the miss overhead by about a factor of two. Thus, our measurements may overstate the direct performance advantage of PICs by about the same factor. On the other hand, measurements of the ParcPlace Smalltalk-80 system indicate that it also takes about 15 microseconds to handle a miss (see Appendix A), and thus our current implementation does not seem to be unreasonably slow.

Monomorphic sends in our experimental system use the same inline caching scheme as the base system. For polymorphic sends, a stub is constructed which tests the receiver type and branches to the corresponding method. The stub has a fixed overhead of 8 cycles (to load the receiver type and to jump to the target method), and every type test takes 4 cycles. The PICs are implemented as described in section 3.1. None of the optimizations mentioned in the previous section are implemented except that a call site is treated as megamorphic if it has more than ten receiver types (but such calls do not occur in our benchmarks). Appendix B contains an example of a PIC stub generated by our implementation.

In order to evaluate the effectiveness of polymorphic inline caches, we measured a suite of SELF programs. The programs (with the exception of `PolyTest`) can be considered fairly typical object-oriented programs and cover a variety of programming styles. More detailed data about the benchmarks is given in Appendix A.

`Parser`. A recursive-descent parser for an earlier version of the SELF syntax (550 lines).

`PrimitiveMaker`. A program generating C++ and SELF stub routines from a description of primitives (850 lines).

`UI`. The SELF user interface prototype (3000 lines) running a short interactive session. Since the Sun-4 used for our measurements has no special graphics hardware, runtime is dominated by graphics primitives (e.g. polygon filling and full-screen bitmap copies). For our tests, the three most expensive graphics primitives were turned into no-ops; the remaining primitives still account for about 30% of total execution time.

`PathCache`. A part of the SELF system which computes the names of all global objects and stores them in compressed form (150 lines). Most of the time is spent in a loop which iterates through a collection.

`Richards`. An operating system simulation benchmark (400 lines). The benchmark schedules the execution of four different kinds of tasks. It contains a frequently executed polymorphic send (the scheduler sends the `runTask` message to the next task).

`PolyTest`. An artificial benchmark (20 lines) designed to show the highest possible speedup with PICs. `PolyTest` consists of a loop containing a polymorphic send of degree 5; the send is executed a million times. Normal inline caches have a 100% miss rate in this benchmark (no two consecutive sends have the same receiver type). Since `PolyTest` is a short, artificial benchmark, we do not include it when computing averages for the entire set of benchmarks.

The benchmarks were run 10 times and the average CPU time was computed; this process was repeated 10 times, and the best average was chosen. A garbage collection was performed before every measurement in order to reduce inaccuracies. Figure 5 shows the benchmark results normalized to the base system's execution time (see Appendix A for raw execution times). For comparison, the execution times for ParcPlace Smalltalk-80 V2.4 are 262% for `Richards` and 93% for `PolyTest` (i.e. in Smalltalk, `Richards` runs slower and `PolyTest` slightly faster than the base SELF system).

With PICs, the median speedup for the benchmarks (without `PolyTest`) is 11%. The speedup observed for the individual benchmarks corresponds to the time required to handle inline cache misses in the base system. For example, in the base system `PolyTest` spends more than 80% of its execution time in the

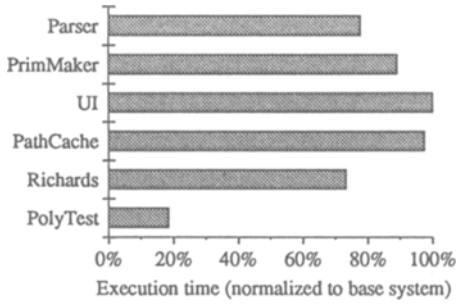

Figure 5. Impact of PICs on performance

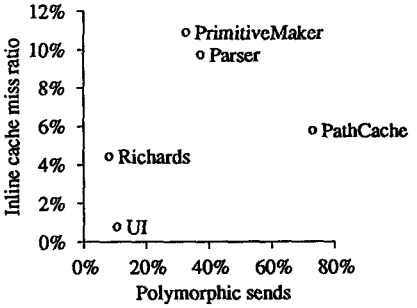

Figure 6. Inline cache miss ratios

miss handler, and thus it is more than five times faster with PICs. Overall, the performance impact of PICs is relatively small since the SELF compiler is able to inline many sends.

Interestingly, there is no direct correlation between cache misses and the number of polymorphic call sites (Figure 6). For example, more than 73% of the messages sent in `PathCache` are from polymorphic call sites, but the inline cache miss ratio is only 5.6%, much lower than `Parser`'s miss ratio despite the higher percentage of polymorphic sends. This suggests that one receiver type dominates at most call sites in `PathCache`, whereas the receiver type frequently changes in `Parser`'s inline caches. Thus, ordering a PIC's type tests by frequency of occurrence (as suggested in section 3.2) might be a win for programs like `PathCache`.

The space overhead of PICs is very low, typically less than 2% of the compiled code (see Appendix A). This low overhead is also observed in our daily use of the SELF system, where the space used by PICs usually does not exceed 50Kbytes in a system which contains about 2 Mbytes of compiled code.

4. Background on Inlining and Type Information

The techniques described so far strive to reduce the cost of sending a message to that of calling a procedure. But even if these techniques were completely successful, the extremely high call frequency would still impose a severe limit on the performance of dynamically-typed object-oriented programs: even the fastest procedure call is too slow. For example, the Sun-4/260 on which our measurements were made executes about 10 million native instructions per second. The optimal calling sequence consists of two instructions per call. This would seem to limit SELF programs to significantly less than 5 million message sends per second (MiMS; see [CUL89]) even if every send was implemented optimally. However, many programs execute at 5 MiMS in our current system, and some benchmarks exceed 20 MiMS. How is this possible?

The best way to significantly speed up a call is by not executing it at all, i.e. by *inlining* the called method into the caller, thus eliminating the calling overhead entirely. In addition, inlining introduces opportunities for other optimizations like constant folding, common subexpression elimination, and better global register allocation. The benefits obtained through these optimizations often overshadow the savings from just removing the call/return overhead and are essential in order to optimize user-defined control structures.

Therefore, the SELF compiler tries to inline as many message sends as possible. However, inlining requires that the type of the target of a message send be known at compile time so that its definition can be looked up and inlined. Hence, many optimization techniques have focused on ways to obtain and exploit type information [Joh87]. The remainder of this section describes existing techniques to extract, preserve, and exploit this precious type information.

4.1. Type Prediction

Certain messages are almost exclusively sent to particular receiver types. For such messages, the compiler can *predict* the type of the receiver based on the message name and insert a run-time type test before the

message send to test for the expected receiver type. Along the branch where the type test succeeds, the compiler has precise information about the type of the receiver and can statically bind and inline a copy of the message. For example, existing SELF and Smalltalk systems predict that '+' will be sent to an integer [UP82, GR83, DS84], since measurements indicate that this occurs 90% of the time [UP87]. Type prediction improves performance if the cost of the test is low and the likelihood of a successful outcome is high.

4.2. Customization

Customization is another technique for determining the types of many message receivers in a method [CUL89]. Customization extends dynamic compilation by exploiting the fact that many messages within a method are sent to `self`. The compiler creates a separate compiled version of a given source method for each receiver type. This duplication allows the compiler to *customize* each version to the specific receiver type. In particular, knowing the type of `self` at compile time allows all self sends to be inlined, without inserting type tests at every message send. Customization is especially important in SELF, since so many messages are sent to `self`, including instance variable accesses, global variable accesses, and many kinds of user-defined control structures.

4.3. Type Analysis and Splitting

Type analysis tries to get the most out of the available type information by propagating it through the control flow graph and by performing flow-sensitive analysis [CU90, CU91]. The compiler uses the type information obtained through the analysis to inline additional message sends and to reduce the cost of primitives (either by constant-folding the primitive or by avoiding run-time type checks of the primitive's arguments).

Often the compiler can infer only that the type of the receiver of a message is one of a small set of types (such as either an integer or a floating point number). This union type information does not enable the message to be inlined, since each possible receiver type could invoke a different method.

One approach to solving this problem is to insert type tests before the message send and create a separate branch for each of the possible types. This technique, *type casing*, is similar to type prediction and to the case analysis technique implemented as part of the Typed Smalltalk system [JGZ88].†

Splitting is another way to turn a polymorphic message into several separate monomorphic messages. It avoids type tests by copying parts of the control flow graph [CUL89, CU90, CU91]. For example, suppose that an object is known to be an integer in one branch of an `if` statement and a floating-point number in the other branch. If this object is the receiver of a message send following the `if` statement, the compiler can copy the send into the two branches. Since the exact receiver type is known in each branch, the compiler can then inline both copies of the send.

Type analysis and splitting provide a significant amount of additional type information that may be used to optimize object-oriented programs. These techniques work especially well for inferring the types of local variables and optimizing user-defined control structures. Nevertheless, there are classes of variables and expressions that type analysis cannot analyze well. One such class is the types of arguments to the method (our SELF system customizes on the type of the receiver, but not on the type of arguments). Another important class is the types of instance variables and array elements (actually, any assignable heap cell). These weaknesses of type analysis can be quite damaging to the overall performance of the system, especially for typical object-oriented programs.

5. PICs as Type Sources

PICs have a valuable property that can be used to inline many more message sends than with existing techniques. A PIC can be viewed as a *call-specific type database*: the PIC contains a list of all receiver types seen in practice at that call site. If the compiler can take advantage of this type information, it should be able to produce much more efficient code.

† The type information in Typed Smalltalk is provided by the programmer in the form of type declarations, while a dynamically-typed system would rely on type analysis to determine the set of possible receiver types.

Unfortunately, the information present in a method's PICs is not available when the method is first compiled, but only after it has been executing for a while. To take advantage of the information, the method must be *recompiled.* The rest of this section describes and evaluates the optimizations that may be performed when PIC-based type information from a previously compiled version of a method is available. Section 6 describes an extension to the recompilation scheme that leads to an adaptive system.

5.1. PIC-Based Type Casing

Type casing may be extended naturally in a system that recompiles methods based on PIC information. When the compiler encounters a send which it did not inline in the previous compiled version of a method, it can consult the corresponding PIC to obtain a list of likely receiver types for this send. The compiler then knows that the receiver type is the union of the types in the PIC, plus an unlikely unknown type (since a new receiver type might, but probably won't, be encountered in the future).

The compiler can then take advantage of the new type information by inserting run-time type tests and inlining the cases. For example, sending the x message to a receiver that was either a cartesian point or a polar point in the previous version's PIC would be compiled into the following code:

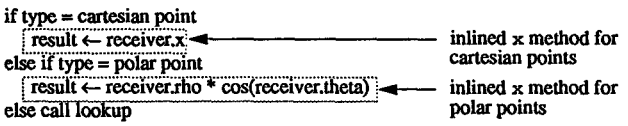

Figure 7. Inlining with type casing

5.2. Dynamic Type Prediction

PICs support better type prediction by replacing the *static* type prediction of existing systems with *dynamic* type prediction. Current systems hard-wire the set of predictions into the compiler and have no means to adapt if a particular application does not exhibit the predicted behavior. For example, if an application makes heavy use of floating point arithmetic, then predicting solely integer receivers for '+' messages penalizes the performance of floating point arithmetic.

Even worse, if an application makes heavy use of messages that are not recognized the compiler, these messages may run much slower than expected since the compiler is not using type prediction on them. For example, the initial SELF system defined a `predecessor` method for integers, and this message was type-predicted by the compiler. Later, programmers defined a `pred` method as a shorthand version of `predecessor`, but since this wasn't included in the compiler's static type prediction table, the performance of programs using `pred` was significantly worse than programs using `predecessor`.

These problems could be avoided by a system with PICs. The system would periodically examine all PICs, looking for messages with very skewed receiver type distributions. Those messages that clearly are only used with one or two receiver types should be type predicted. The compiler would augment or replace its built-in initial type prediction table with one derived from the actual usage patterns. As these usage patterns changed, type prediction would naturally adapt.

With PICs and recompilation, static type prediction theoretically could be eliminated, since the recompiled version will obtain the benefits of type prediction via type casing. However, since type prediction usually works very well for a few messages like '+' and `ifTrue:`, it is doubtful that the relatively minor simplification of the system is worth the expected loss of performance in the initial version of the compiled code.

5.3. PIC-Based Type Information and Type Analysis

The type information provided by PICs is neither strictly more nor strictly less precise than that computed using type analysis. It can be less precise because the analysis may be able to prove that an expression can have only a certain set of types at run-time. If this set is a singleton set, then the compiler can inline messages without needing a run-time type test to verify the type. On the other hand, type analysis may fail

to infer anything about an expression's type (e.g. the type of an instance variable); in this case, the information provided by PICs is more precise because it includes specific types in addition to the unknown type.

The presence of PIC-based type information fundamentally alters the nature of optimization of dynamically-typed object-oriented languages. In "traditional" systems such as the current SELF compiler, type information is scarce, and consequently the compiler is designed to make the best possible use of the type information. This effort is expensive both in terms of compile time and compiled code space, since the heuristics in the compiler are tuned to spend time and space if it helps extract or preserve type information. In contrast, a PIC-based recompiling system has a veritable wealth of type information: *every* message has a set of likely receiver types associated with it derived from the previously compiled version's PICs. The compiler's heuristics and perhaps even its fundamental design should be reconsidered once the information in PICs becomes available; section 6 outlines such a system architecture designed with PICs in mind.

5.4. PIC-Based Type Information and Type Declarations

The type information present in PICs could be used in a programming environment. For every method which exists in compiled form, the system knows all receiver types that have occurred in practice. Thus, a browser could answer questions like "what kinds of objects does this parameter typically denote?" or "what kinds of objects is this message sent to?"[†] Such querying facilities could help a programmer to better understand programs written by other people and to verify her assumptions about the types of a parameter or local variable. Similarly, the system's type information could be used as a starting point for type checking in a system where type declarations are optional. Once an untyped program becomes stable, the system could automatically type-annotate all variables, and could quickly reject type declarations made by the user if they exclude types known to be used in practice.

In this scenario, type information would flow from the system to the user, in contrast to other approaches where type information flows from the user to the compiler [Suz81, BI82, JGZ88]. In our system, the programmer benefits from type information even for programs which do not contain any type declarations, and the declarations are not needed to obtain good performance. In fact, it is likely that our system can generate *better* code than existing systems based on user-specified type declarations since PICs contain only those types that are used in practice, whereas static type declarations must include types that theoretically might occur but rarely do so in practice. Thus PICs include useful information about the relative likelihood of the possible receiver types that is not present in traditional type declarations.

5.5. Implementation

We have built an experimental version of a recompiling system on top of the current SELF compiler in order to prove the feasibility of PIC-based adaptive recompilation and to estimate the quality of the code which could be produced using the type information contained in the PICs. In our experimental system, the current SELF compiler was augmented by a "type oracle" which provides the PICs' type information to the compiler. The compiler itself was not changed fundamentally, and it does not use dynamic type prediction. Each benchmark was run once to produce the first version of compiled methods and to fill the inline caches. Then a system flag was set and the benchmark was run again, which caused all methods to be recompiled using the type information contained in the inline caches. The second version was then measured to determine the improvement over the base system.

We measured the same benchmarks that were used in section 3. Figure 8 shows the performance of the experimental system and the system with PICs (described in section 3.3), normalized to the base system.

[†] Most of the mechanisms needed to find the appropriate compiled method(s) from the source method are already present in order to invalidate compiled code when a source method is changed.

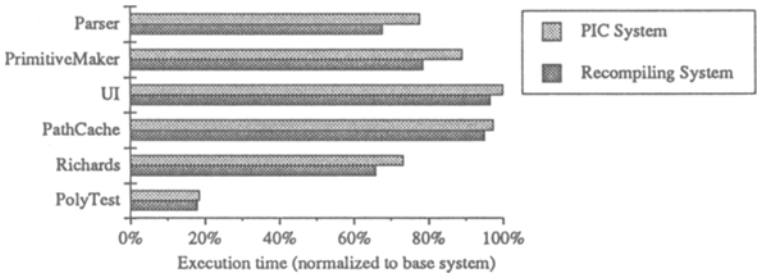

Figure 8. Performance of the Experimental System

With recompilation, the benchmarks show a median speedup of 27% over the base system and a median speedup of 11% over the system using PICs alone (excluding `PolyTest`). The experimental system is especially successful for `Richards` and `Parser` which are 52% and 48% faster than the base system, respectively. Because of several shortcomings of our experimental system, these numbers are conservative estimates of the performance achievable with our techniques. For example, the code for type cases generated by the experimental compiler is more than two times slower than an equivalent type case of a PIC because it reloads the receiver's type before every type test. The extra loads negate much of the savings achieved by inlining short methods; for example, `PolyTest` is only marginally faster even though recompilation has eliminated *all* message sends.

Figure 9 shows the impact of recompilation on the number of message sends. For each benchmark, three configurations are shown. The first bar represents the number of messages sent by the benchmark when compiled with the base system. The middle bar represents the number of message sends when using the experimental system with one recompilation. The third bar represents the steady state achieved after several recompilations; it can be viewed as the best possible case for the current compiler, i.e. what the compiler would produce if it had complete type information and inlined every message it wanted. Thus, the third scenario shows the minimum number of message sends given the inlining strategies used by the compiler.† The bars are normalized relative to the base system.

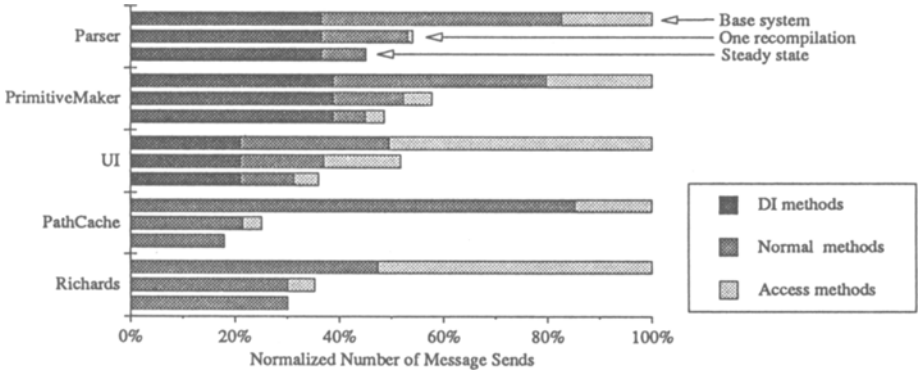

Figure 9. Impact of Recompilation on Number of Message Sends

† Some methods will not be inlined even when their receiver type is known, for example if they are too large. Because of limitations of the experimental system, the steady states for `PrimitiveMaker` and `UI` are not available; the data shown in the graph represent the best case which the experimental system currently can achieve.

The messages labelled "DI methods" invoke methods which use dynamic inheritance (DI), a feature of SELF which allows objects to change their parents on the fly. In the current SELF system, the use of DI prevents messages from being inlined even if their receiver type is known. Therefore, the experimental system cannot eliminate these sends. The messages labelled "normal methods" and "access methods" invoke ordinary methods and "wrapper" methods that only access an instance variable, respectively.

For our benchmarks, recompilation is extremely successful in reducing the number of dynamically executed message sends: recompiling once *halves* the total number of message sends. If DI methods are subtracted out, the median reduction of the number of message sends is a factor of 3.2. With several recompilations, a factor of 5.6 is achieved for `PathCache`. This surprising reduction provides strong evidence that a compiler using only static techniques such as type analysis, type prediction, and splitting cannot infer the receiver types of many message sends in object-oriented programs. PICs provide this missing information, and our experimental system is able to optimize away most eligible sends with just one recompilation.

6. Adaptive Compilation

Our experimental implementation demonstrates that code quality can be significantly improved by using the type information contained in PICs. However, it is built on top of a compiler which was designed with fundamentally different assumptions about what is important and cost-effective and therefore does not realize many of the potential benefits of a recompilation-based system. In this section we outline the benefits and problems of a new framework for efficient implementations of dynamically-typed object-oriented languages, based on the idea of incremental optimization of compiled code in an environment where type information is relatively abundant. In the tradition of Hansen [Han74] we will call this mode of compilation *adaptive compilation.*

One goal of such a system is to maximize overall system performance, i.e. to minimize the sum of compile time and execution time over the life of the system. Another somewhat conflicting goal is to minimize pauses caused by dynamic compilation so that applications always appear to the user to be making progress. These goals can be thought of as maximizing throughput (overall system performance) while minimizing latency (time to reach a certain point in an application), and as with other systems must be balanced against each other.

In our adaptive framework, methods are initially compiled with little or no optimization. As methods are used often, they are recompiled with more optimization. The recompilation can take advantage of the type information present in the previous version's PICs to generate reasonably good code without too much expensive analysis. Only those methods that are executed most frequently are eventually recompiled with full optimization, using relatively expensive techniques to generate the best possible code.

6.1. Faster Compilation

Adaptive compilation promises a number of important benefits. Perhaps the most important of these is that overall system performance should improve dramatically. We believe that a relatively simple and very fast compiler can generate good code using the type information provided by PICs because the greater amount of inlining should compensate for the optimization possibilities lost by not using expensive global techniques. The compiler could refrain from using expensive optimizations as long as possible and rely on the wealth of type information to generate good code quickly.

A fast compiler also means short compile pauses. Even for the critical methods which need to be optimized, compile pauses could be kept short by distributing the optimization effort over several recompilations or by compiling in the background. Also, recompilation could be performed in the background or during the user's "think pauses," similar to Wilson's opportunistic garbage collection [WM89].

Of course, recompilation has costs. Some time will be wasted by executing unoptimized code, and some time will be wasted because some work is repeated with every recompilation (e.g. code generation). However, we believe that the time saved by recompiling only the frequently-used methods will more than offset this additional overhead.

6.2. Trade-offs

To work well, an adaptive system needs to quickly adapt to the application to maximize overall performance. Specifically, the system must make three decisions:

- *What to recompile.* The system needs to identify the dominating parts of an application (the "hot spots") in order to optimize them. A simple approach is to count the number of times a method is executed. This should work well for relatively unoptimized programs which contain message sends in the body of every loop. The counters of the leaf methods called by the innermost loop will overflow first, and the system can then search up the call stack to find the loop. Once the methods become more optimized and contain inlined loops, standard profiling methods such as interrupt-driven PC-sampling can be used to find the methods responsible for significant amounts of execution time.
- *When to recompile.* When a method exceeds a certain threshold in the number of invocations, it will be recompiled. Thus the value of this threshold is one of the primary points of control of the adaptive system, and the system needs to do a good job in setting this value. The threshold could be determined empirically by estimating the cost and gains of recompilation. A more dynamic approach could estimate the recompilation time based on the size (and maybe previous compilation time) of a method, and adjust the recompilation threshold accordingly.
- *How much to optimize.* Spending more effort during recompilation can result in bigger savings in execution time and reduce the need for or number of future recompilations, but it will also lead to longer compile pauses. In some situations (e.g. when starting up a new application), latency is more important than absolute speed, and too much optimization would impair the responsiveness of the system.

One approach to managing the optimization strategy would be to have the system monitor the ratio of compile time to program execution time. When this ratio is high (when compilation time dominates execution time), the compiler would optimize less aggressively and recompile less frequently. Thus, the compiler would compile a new application's working set as quickly as possible. Once execution time returned to the forefront, the compiler could adopt a more aggressive recompilation stance to increase the performance of the hot spots of the application.

6,3. Preserving Information Across Code Cache Flushes

Recompilation may interact poorly with dynamic compilation and caching. If the cache for compiled code is too small to hold all of the system's code (especially with a system whose programming environment shares the address space with user applications), an optimized method might be flushed from the cache because it had not been used recently. However, the fact that the method was optimized indicates that when it is used again, it is likely to be used intensively. If the compiled code (and all the information it contains) is simply discarded when being flushed from the cache, the system will forget that the method is a "hot spot." When the code is needed again, time will be wasted by first compiling an unoptimized version, only to discover later that the method needs optimization.

To solve this problem, the system could keep some information about optimized methods even after they have been flushed from the code cache. For example, the system could keep a compressed form of the method's type information so that an efficient compiled version could be regenerated immediately without going through several recompilations. If there is not enough space to keep such compressed methods, the preserved information could be the mere fact that the method is important (time-critical); when the method has to be regenerated the system could recompile it more aggressively than it would do normally and thus could produce optimized code more quickly.

7. Related Work

Statically-typed languages can handle polymorphic sends in constant time by indexing into a type-specific function table, thus reducing the lookup to an indirect procedure call. In C++, for example, a dynamically-bound call takes between 5 and 9 cycles on a SPARC [Ros88, DMSV89, ES90, PW90]. This is possible because static type checking can guarantee the success of the lookup, i.e. the result of the table lookup need not be verified. Inline caching techniques are less attractive in this context because a direct call plus a

type test take about the same time (6 cycles) as a full "lookup". However, statically-typed object-oriented languages could benefit from customization, type casing, and inlining [Lea90].

Kiczales and Rodriguez [KiRo89] describe a mechanism similar to PICs for a CLOS implementation. Their implementation of message dispatch does not use inline caching per se but it does use special dispatch handlers for some cases, e.g. when a call site uses only one or two distinct classes. In the general case, the lookup uses specific hash tables specific to the message names.

The concept of adaptive systems is not new. For example, Hansen describes an adaptive compiler in [Han74]. His compiler optimized the inner loops of Fortran programs at run-time. The main goal of his work was to minimize the total cost of running a program which presumably was executed only once. All optimizations could be applied statically, but Hansen's system tried to allocate compile time wisely in order to minimize total execution time, i.e. the sum of compile and run-time.

Some modern compilers for conventional languages use profiling information to perform branch scheduling and to reduce cache conflicts [MIPS86]. The optimizations enabled by this form of feedback are typically very low-level and machine-dependent.

Mitchell [Mit70] converted parts of dynamically-typed interpreted programs into compiled form, assuming that the types of variables remained constant. Whenever the type of a variable changed, all compiled code which depended on its type was discarded. Since the language did not support polymorphism and was not object-oriented, the main motivation for this scheme was to reduce interpretation overhead and to replace generic built-in operators by simpler, specialized code sequences (e.g. to replace generic addition by integer addition).

Suzuki [Suz81] reports that a type accumulation phase for Smalltalk-80 was suggested to him by Alan Perlis as an alternative to type analysis. In this approach, a program would be run in interpreted form against some examples and then compiled into more efficient code. However, the information would only be used to avoid lookups by inserting a type test for a likely receiver type and branching to the corresponding method (inline caching was not yet known at that time). As far as we know, Suzuki never implemented such a system. Furthermore, he maintained that the information obtained by a training run could never give useful information on polymorphic types, which is contradicted by our results.

8. Conclusion

Polymorphic inline caches (PICs) significantly speed up polymorphic sends: some programs making frequent use of polymorphism run up to 37% faster with PICs. More importantly, polymorphic inline caches are an important source of type information since they record the set of likely receiver types for every send; such type information is essential for optimizing compilers to generate efficient code. By taking advantage of this information, our experimental implementation of the SELF language executes some fairly typical object-oriented programs up to 52% faster than the base system and reduces the number of dynamically executed message sends by a factor of two to four.

The presence of PIC-based type information may fundamentally alter the problem of optimizing dynamically-typed object-oriented languages. In current systems, type information is *scarce*, and consequently the compiler needs to work hard in order to preserve and exploit the scarce information it has. Unfortunately, such techniques are expensive both in terms of compile time and compiled code space. In contrast, type information is *abundant* in a system using adaptive compilation. Such a system may reduce compilation times significantly by eliminating most of the computationally expensive type analysis phase and by only optimizing the most-used parts of programs. In addition, adaptive compilation could overcome some drawbacks of existing optimization techniques such as static type prediction. We are currently implementing such a system for the SELF language in order to validate our ideas.

Acknowledgments. We would like to thank Bay-Wei Chang and Ole Agesen for their helpful comments on earlier drafts of this paper.

9. References

[BI82] A. H. Borning and D. H. H. Ingalls, "A Type Declaration and Inference System for Smalltalk." In *Conference Record of the Ninth Annual Symposium on Foundations of Computer Science*, pp. 133-139, 1982.

[Cha91] Craig Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. Thesis, Stanford University. In preparation.

[CPL83] Thomas J. Conroy and Eduardo Pelegri-Llopart, "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations." In [Kra83].

[CU89] Craig Chambers and David Ungar, "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language." In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee, "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes." In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, 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*, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[CU91] Craig Chambers and David Ungar, "Making Pure Object-Oriented Languages Practical." To be presented at OOPSLA '91, Phoenix, AZ, October, 1991.

[Deu83] L. Peter Deutsch, "The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture." In [Kra83].

[DMSV89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan, "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance." In *OOPSLA '89 Conference Proceedings*, pp. 211-214, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[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.

[ES90] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[GJ90] Justin Graver and Ralph Johnson, "A Type System for Smalltalk." In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January, 1990.

[GR83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

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

[Hei90] Richard L. Heintz, Jr., *Low Level Optimizations for an Object-Oriented Programming Language*. Master's Thesis, University of Illinois at Urbana-Champaign, 1990.

[Ing86] Daniel H. Ingalls, "A Simple Technique for Handling Multiple Polymorphism." In *OOPSLA '86 Conference Proceedings*, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

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

[Joh87] Ralph Johnson, ed., "Workshop on Compiling and Optimizing Object-Oriented Programming Languages." In *Addendum to the OOPSLA '87 Conference Proceedings*, pp. 59-65, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 23(5)*, May, 1988.

[KiRo89] Gregor Kiczales and Luis Rodriguez, "Efficient Method Dispatch in PCL." Technical Report SSL-89-95, Xerox PARC, 1989.

[Kra83] Glenn Krasner, ed., *Smalltalk-80: Bits of History and Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Lea90] Douglas Lea, "Customization in C++." In *Proceedings of the 1990 Usenix C++ Conference*, pp. 301-314, San Francisco, CA, April, 1990.

[MIPS86] MIPS Computer Systems, *MIPS Language Programmer's Guide.* MIPS Computer Systems, Sunnyvale, CA, 1986.

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

[PW90] William Pugh and Grant Weddell, "Two-Directional Record Layout for Multiple Inheritance." In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation,* pp. 85-91, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6),* June, 1990.

[Ros88] John R. Rose, "Fast Dispatch Mechanisms for Stock Hardware." In *OOPSLA '88 Conference Proceedings,* pp. 27-35, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11),* November, 1988.

[ST84] Norihisa Suzuki and Minoru Terada, "Creating Efficient Systems for Object-Oriented Languages." In *Proceedings of the 11th Symposium on the Principles of Programming Languages,* Salt Lake City, January, 1984.

[Suz81] Norihisa Suzuki, "Inferring Types in Smalltalk." In *Proceedings of the 8th Symposium on the Principles of Programming Languages,* 1981.

[UBF+84] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC." In *Eleventh Annual International Symposium on Computer Architecture,* Ann Arbor, MI, June, 1984.

[Ung86] David Ungar, *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, Cambridge, MA, 1986.

[UP83] David Ungar and David Patterson, "Berkeley Smalltalk: Who Knows Where the Time Goes?" In [Kra83].

[UP87] David Ungar and David Patterson, "What Price Smalltalk?" In *IEEE Computer 20(1),* January, 1987.

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

Appendix A. Raw Benchmark Data

The following table gives the execution times (in seconds) of all benchmarks. We estimate that inaccuracies due to hardware caching and context switching effects are below 5%.

	base	PIC	recompiled	Smalltalk-80
Richards	2.95	2.16	1.94	7.74
Parser	3.32	2.58	2.25	
PrimMaker	3.13	2.79	2.46	
UI	5.97	5.97	5.77	
PathCache	1.62	1.58	1.54	
PolyTest	20.48	3.76	3.68	19.14†

A variation of the `PolyTest` benchmark which performs 1,000,000 monomorphic sends instead of the polymorphic sends runs in 4.2 seconds in ParcPlace Smalltalk-80. Thus, we estimate that a cache miss takes about 15 microseconds in this system, or about the same time as a SELF inline cache miss.

The space overhead of PICs is given in the next table. The first column lists the size in bytes of the compiled code (without PICs) for each benchmark; this includes the code for all parts of the system which are used by the benchmark (e.g. strings, collections, etc.). The code sizes given below include method headers and relocation tables but not debugging information. The second column contains the size in bytes of the PICs (including headers and relocation tables), and the third column shows the space overhead relative to the base version.

	code size	PIC size	overhead
Richards	30,000	240	0.8%
Parser	269,000	4,000	1.5%
PrimMaker	973,000	16,500	1.7%
PathCache	64,000	4,400	6.9%
UI		not available	

The next table describes the number of sends that the benchmarks execute when compiled with the base system (see Figure 9). The column labelled "polymorphic" lists the number of messages sent from polymorphic call sites; "misses" is the number of inline cache misses in the base system.

	normal	access	DI	total	polymorphic	misses
Richards	380,600	421,600	0	802,200	65,800	35,500
Parser	176,200	66,300	139,600	382,100	90,900	23,600
PrimMaker	129,500	64,400	123,000	316,900	63,400	21,100
UI	119,900	212,300	88,600	420,800	35,500	2,600
PathCache	60,700	10,500	0	71,200	52,000	4,100
PolyTest	1,000,000	0	0	1,000,000	1,000,000	1,000,000

For the experimental system, the number of sends are as follows (the DI numbers remain unchanged):

	after one recompilation		steady state	
	normal	access	normal	access
Richards	241,400	41,800	241,400	0
Parser	63,500	3,300	32,800	0
PrimMaker	42,700	17,500	19,600	11,600‡
UI	67,400	61,800	43,100	20,000‡
PathCache	15,300	2,600	12,800	0
PolyTest	0	0	0	0

† The main loop of the benchmark was hand-inlined so that the Smalltalk compiler could produce better code; otherwise, the time would be 36.7 seconds.

‡ Our experimental system cannot recompile this benchmark often enough to reach the steady state.

Appendix B. Example Code

The following code was produced by our system for a PIC containing two different receiver types; the code is given in SPARC assembler syntax. Branches have one delay slot on the SPARC; instructions in annulled delay slots (indicated by `,a`) are only executed if the branch is taken, except for branch always where the instruction in the delay slot is never executed. `sethi`/`add` combinations are needed to load 32-bit constants.

```
    andcc %i0, 1                    tag test for immediate (integer, float)
    bnz,a _mapTest                  branch to _mapTest if not an immediate
    load  [%i0+7], %g5              load receiver map (annulled delay slot)
    bra,a _miss                     immediate: branch to _miss
_mapTest:
    sethi %hi(type1), %g4           load first part of type 1 (32-bit literal)
    add   %g4, %lo(type1), %g4      load second part of type 1
    cmp   %g4, %g5                  compare with receiver type
    bne,a _next                     try next case if not equal
    sethi %hi(type2), %g4           load first part of type 2 (annulled delay slot)
    sethi %hi(method1), %g4         success: load first part of first method's address
    jmp   %g4 + %lo(method1)        and branch to the method
_next:
    add   %g4, %lo(type2), %g4      complete the load of type 2
    cmp   %g4, %g5                  compare with receiver type
    bne,a _miss                     no more types to check; goto _miss if not equal
    nop                             empty delay slot (could be eliminated)
    sethi %hi(method2), %g4         success: load first part of second method's address
    jmp   %g4 + %lo(method2)        and branch to the method
_miss:
    sethi %hi(Lookup), %g4          miss: load first part of lookup routine
    jmp   %g4 + %lo(Lookup)         and branch to the lookup routine
    nop                             empty delay slot
```