Object, Message, and Performance: How they coexist in SELF*

David Ungar	Sun Microsystems Laboratories, Inc., 2550 Garcia Ave., MTV 29-116, Mountain View, CA 94043	david.ungar@sun.com (415)336-2618
Randall B. Smith	Sun Microsystems Laboratories, Inc., 2550 Garcia Ave., MTV 29-116, Mountain View, CA 94043	randall.smith@sun.com (415)336-2620
Craig Chambers	Department of Computer Science and Engineering Sieg Hall, FR-35 University of Washington, Seattle, WA 98195	chambers@ cs.washington.edu (206)685-2094
Urs Hölzle	Computer Systems Laboratory Stanford University Stanford, CA 94305	urs@cs.stanford.edu (415)725-3701

Introduction

Applying object-oriented techniques to the art of computer programming confers many benefits, and like an older discipline, structured programming, is most effective when applied uniformly throughout a program. For example, the SELF programming language distills object-oriented computation down to a simple story based on copying prototypes to create objects, inheriting from objects to share their contents, and passing messages to invoke methods. SELF programs even send messages to alter the flow of control, access variables, and perform arithmetic. As a result, methods are oblivious to the representations of objects and are therefore easier to reuse.

For example, last year you might have written a routine to sort an array of numbers. It was a method defined for arrays that works by sending the **less-than** (<) message to the numbers. Today, you might need to sort an array of strings. In a pure object-oriented language, you could just call the same sort method. The old sort method would still work, because the objects in the array (now strings) respond to the < message. The code run for < is decided at runtime according to the type of the receiver, so the sort method works for any object that implements a < method. When the same code can be used for different types of objects, it is

* This work has been generously supported by a NSF Presidential Young Investigator Grant # CCR-8657631 and by Sun, IBM, the Powell Foundation, Apple, Cray, Tandem, NCR, TI, and DEC.

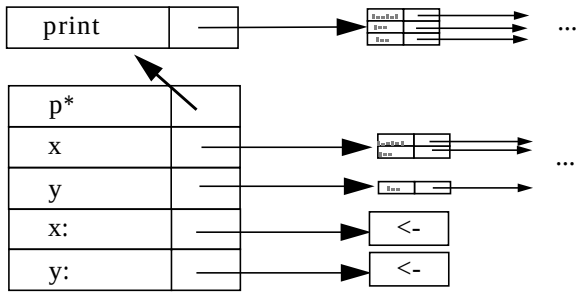
said to be *polymorphic*. With this kind of polymorphism, you do not have to explicitly parameterize the sort routine when you write it—it just works! This quality of unanticipated reusability may be one of the reasons programmers feel empowered by pure object-oriented languages.

However, unlike structured programming, pure object-oriented programming cannot be implemented efficiently with traditional compilation techniques because traditional optimizations rely on static declarations of representation types. In this paper, we will present the novel implementation techniques that recapture much of the efficiency that would seem to be lost in a pure object-oriented language. For many of the benchmarks we have measured, these techniques have provided a fivefold speedup, enabling SELF programs to come within a factor of two or three of optimized C.

Overview of SELF: A Simple, Pure, Object-Oriented Programming Language

SELF was initially designed at Xerox PARC by authors Ungar and Smith [4]. The designers employed a minimalist strategy, striving to distill an essence of object and message. The subsequent design evolution and implementation were undertaken by authors Chambers, Hölzle, and Ungar, and by Bay-Wei Chang at Stanford University. SELF today is a fairly large system, has hosted a few sizable projects, and has been tried at over 150 (mainly academically curious) sites.

The basic notion in SELF is an object consisting of slots. A slot has a name and a value. Slot names are always strings, but slot values can be any SELF object. Slots can be marked with an asterisk to show that they designate a *parent*. Here, an object represents a two-dimensional point with **x** and **y** slots, a parent slot called **p**, and two special assignment slots, **x:** and **y:,** that are used to assign to the **x** and **y** slots. The object's parent has a single slot called **print** (containing a method object).



When sending a message, if a no slot name matches within the receiving object, its parent's slots are searched, and then slots in the parent's parent, and so on. Thus our point object can respond to the messages **x**, **y**, **x:, y:,** and **p**, plus the message **print**, because it *inherits* the **print** slot from its parent. In SELF, any object can potentially be a parent for any number of children, or a child of any object. This gives the language an unusual kind of uniformity and flexibility.

In addition to slots, a SELF object can include code. Such objects are called *methods*; they correspond to subroutines in other languages. For example, the object in the **print** slot above includes code and thus serves as a method. When an object is found in a slot as a result of a message send it is *run*: an object without code runs by simply returning itself, but a method runs by invoking its code. Thus when the **print** message is sent to our point object, the code in the print slot's method will run immediately. The assignment slots, **x:** and **y:,** contain a special method (symbolized by the arrow), that takes an argument (in addition to the receiver) and stuffs it in either the **x** or **y** slot.

Messages can have extra arguments in addition to the receiver. For example, **3 + 4** sends the message **+** to the object **3** with **4** as argument. Because languages like Smalltalk or SELF do arithmetic by sending messages, programmers are free to add new numeric data types to the language, and the new types can inherit and reuse all the existing numeric code. Adding complex numbers or matrices to the system would be straightforward: after defining a **+** slot for matrices, the user could have the matrix freely inherit from some slot with code that sends **+**. Code that sends **+** would then work for matrices as well as for integers. However, the cost in performance will be severe if sending **+** to integers cannot be expressed efficiently as simple operations in the CPU (such as a single-cycle add instruction). Again, the extreme adherence to the object-oriented paradigm confers benefits, but raises performance issues.

In addition to messages with arithmetic symbols, messages with textual names can also have arguments. For example, the message **ifTrue: False:** takes two arguments, one for each word ending with a colon. For convenience, the arguments are sprinkled in amongst the pieces of a multiple-argument message name, and so we write **(a < b) ifTrue: case1 False: case2**. Here, the result of the **(a < b)** message send (in practice, either the object **true** or **false**) is sent the message **ifTrue: False:** with two arguments. The arguments of **ifTrue: False:** are typically "blocks" or "closures"—bundles of SELF expressions that can be evaluated by sending the **value** message*. The fact that flow-of-control statements such as **ifTrue:False:** are done with message sending has its benefits: for example, adding fuzzy logic to the system by redefining the **ifTrue:False:** method is relatively straightforward, as is adding any sort of user-defined control structure. However, this is yet another performance challenge: the compiler had better figure out how to replace the message sends with compare-and-branch machine instructions.

There is more to be said about the language SELF: new SELF objects are made simply by copying—there are no special class objects for instantiation. Our current implementation allows multiple inheritance, which requires a strategy for dealing with multiple parents. It has block closure objects, threads, and private slots

* We will use the terms "block" and "closure" interchangeably. The distinctions are subtle and not relevant for the purposes of this paper.

for encapsulation. SELF also has mirrors, for "structural reflection," so that, for example, a slot can be deleted, or set to contain a method. (A few somewhat obscure constructs, such as methods contained in the local slots of methods, are not yet supported by the implementation.)

SELF might be called an extremely object-oriented language, because what other languages do with special flow of control mechanisms, or special variable scoping and accessing mechanisms, or special primitive types and operations, is all done with objects and messages in SELF. This maniacal devotion to the object-message paradigm has forced us to develop the techniques described in the following sections.

Why Objects Hinder Performance: The Price of Passing Messages

We have shown how a pure object-oriented language like SELF replaces many conventional language idioms with message passing. But exactly why are messages slow? Message passing is a two-stage process:

- *Finding the method.* The system must search the message's receiver and its ancestors for a slot that matches the message name. Conceptually, this task involves a search of many objects, and a straightforward implementation would be slow. However, almost all systems optimize this lookup: for example, a hash table may be used to cache the results of prior lookups. Alternatively, message names can be encoded as offsets into function tables referenced from object headers. Or, the call instruction can be backpatched with the result of a prior lookup, as long as methods include a prologue to verify the caching. But even the fastest of these approaches (an indexed indirection through a function table) takes two loads. Typical implementations used for C++ require three loads and an addition, about eight cycles. (References 2, 3, and 5 discuss these techniques in more detail.)
- *Calling the method.* Once found, if a matching slot contains a data object, the slot contents must be loaded and returned. Or, if the matching slot contains a method object, the method must be invoked. This step resembles conventional procedure invocation and includes saving the PC and other registers, allocating a stack frame, passing any parameters, branching to the first instruction of the new method, executing the method, returning the result, restoring the PC and other registers, and popping the stack frame. Even on a CPU with a register-window architecture, a nontrivial procedure call typically requires four instructions.

By summing the cost of each step, we can tally the total cost of sending a message: more than ten cycles. When taken to its logical conclusion, object-oriented programming requires that a message be sent for each arithmetic operation, control operation, or even access operation—what should be simple one-cycle operations. When common single-cycle primitive operations are replaced by ten-cycle method invocations, programs run much slower: pure objects resist efficient implementation.

Consider a simple routine to compute the minimum of two quantities. Table 1 compares the object code for a procedural, monomorphic minimum routine with the object code for an object-oriented, polymorphic minimum routine. (It assumes throughout that the receiver, **self**, is **7**, and the argument, **x**, is **11**). On the left is the integer-specific C code. It executes three instructions and runs in 3 cycles. On the right is the SELF version. The example in the table will be detailed in coming sections. For now, note that the SELF source code works on any objects that understand <. As shown in the lower middle section, a traditional implementation would be forced to include three dynamic calls, and the result would be about a 40-cycle execution time. With the techniques described in this paper, the compiler can inline all the calls, reducing execution time to about 5 cycles. For object-oriented systems to be fast, they must *pretend* to send most messages instead of actually sending them.

BEGIN SIDE BAR

Traditional Ways To Achieve Performance in Object-Oriented Languages

Before examining the new techniques of customizing, inlining, and splitting, it will be helpful to consider more traditional strategies for optimizing object-oriented programs.

Special-Purpose Hardware

Hardware designers have tried to cater to object-oriented programming by adding architectural support for message passing. For example, the Smalltalk-on-a-RISC (SOAR) project designed a special-purpose microprocessor for Smalltalk, which ran quite efficiently for its time [2]. However, when each of its special features were evaluated, only register windows, byte manipulation, integer tagging, and one-cycle call instructions contributed more than 10% each to performance. Now, each of these effective features is available on various stock processors (e.g., all are found in the SPARC). Furthermore, more recent advances in compilation tend to diminish the importance of register windows and integer tagging.

Object-Oriented Extensions to Procedural Languages: Creating a Dilemma

Many pre-object-oriented languages, including C and Lisp, have received object-oriented extensions. The resulting hybrid object-oriented languages such as C++ and CLOS provide message passing but retain statically-bound procedure calls and primitive, non-object-oriented data types such as integers, arrays, and cons cells. These data types are accessed via built-in operators or procedure calls that are automatically inlined by the compiler to achieve good performance. Users of a hybrid language enjoy a choice of expressive

Table 1: Procedural and Object-Oriented Implementations of the "min" function

	Procedural			Object-Oriented				
Source Code	**self = 7, x = 11**			**methods involved (when self = 7, x = 11)**			**as inherited by**	

	int min(int self, int x) {
	if (self < x)
	return self;
	else
	return x;
	}
				min: x = (self < x ifTrue: self False: x)[a]			many objects	

				< x = (IntLTPrimitive: x IfFail: ...)[a]			integers	
				ifTrue: truePart False: falsePart = (truePart value)[a].			the **true** object	
Works on	**32-bit integers only**			**any object that inheriting a "<" method (integers, floats, complex numbers, strings, ...)**				
Object Code (for integers)	**Traditional Optimizing Compilation**			**Traditional Optimizing Compilation**			**Customizing, Inlining, & Splitting**	
	source	object code	cycles	source	object code	cycles		cycles
				<	send **<** message to **self**	10		
				IntLTPrim	check receiver's type	2		
					check x's type	2	check x's type	2
	<	cmp self, x	1		cmp self, x	1	cmp self, x	1
	if	branch LT	1		branch LT	1	branch LT	1
					return true	2		
				ifTrue:False:	send **ifTrue:False:** to **true**	10		
				value	send **value** to **self**	10		
		mov self, result	1		mov self, result	1	mov self, result	1
					return from value	2		
					return from ifTrue:False	2		
	rough total:		**3**	**rough total:**		**43**	**rough total:**	**5**

a. A word about syntax: the ". . . = (. . .)" form simply mean that the contents of the parentheses are the source code for the message in front of the equals sign.

medium, but the down side is that they must deal with a kind of schizophrenia, as two mindsets are needed to read and write code. Languages like C++ and partly object-oriented Lisps offer a choice between speed and generality: a routine can employ primitive types or procedural mechanisms and run fast (but have re-

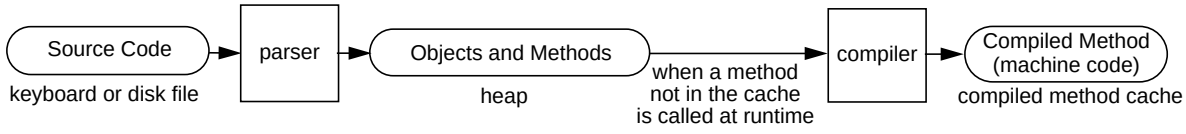
stricted reusability), or it can employ more versatile objects and run slower but be reusable. Traditional implementation techniques cannot provide both the speed of primitive operations and the reusability of message passing at the same time.

Static Typing: Not Necessarily Sufficient for High Performance

Even statically-typed (but fairly pure) object-oriented languages like Trellis/Owl [6] and Eiffel must overcome the overhead of dynamically-dispatched message passing.* Static typing allows the compiler to check that an object will understand every message sent to it, and as a result the compiler can use a somewhat faster dispatching mechanism. However, because an instance of a subclass can always be substituted for an instance of a superclass, and because subclasses can provide alternate method implementations, static type-checking cannot in general determine if a single method will be invoked by a message at runtime, and thus it cannot optimize the call further. For example, if **min**'s arguments were declared to be a general type like **ComparableObjects**, then the compiler could guarantee that there would be *some* definition of < (less-than), but it could not statically determine *which* version of < would be needed. Therefore **min** would have to pay the price of a dynamic dispatch for <. In order for a statically-typed language to optimize **min**, its arguments will have to be statically typed as, say, 32-bit integers, but such specificity will prohibit **min**'s reuse with other types. Since conventional static typing alone does not enable static binding and inlining of messages, it cannot significantly reduce the overhead of message passing.

Dynamic Compilation: the Deutsch-Schiffman Technique

When the first Sun Workstation was announced, Peter Deutsch and Alan Schiffman set about the task of devising a high-performance Smalltalk-80 system for this machine [3]. As the price and physical size of main memory shrank, it became possible to include a simple compiler and a compiled code cache in main memory, along with the application. Their Smalltalk-80 system exploited a combination of *dynamic compilation*, compiled code caching, peephole optimization, hardwiring a few messages, stack-allocated activation records, and inline caching (backpatching call instructions) to obtain better runtime performance than an interpreter while reducing compile-time and code-space costs over a conventional static compiler. When a



programmer types in a method, a *parser* translates it into a simple byte-coded intermediate representation. A *compiler* compiles the method and performs peephole optimization. The space needed to store all of the

* And even these languages restrict common built-in types like integer and boolean to get better performance.

compiled code, and the time needed to recompile all of the code affected by a small change can be considerable. Consequently, the compilation step is *deferred until runtime,* when the method is actually invoked, and the resulting object code is cached for future use.

This system doubled performance over the fastest existing Smalltalk interpreter. It executed code approximately 24 times faster than an earlier and more straightforward interpreter whose implementors despaired of the practicality of pure object-oriented languages. But it was still an order of magnitude slower than optimized C.

END SIDE BAR

Customization, Inlining, and Splitting: Global Optimizations for Dynamically Compiling Methods

In the past few years, new techniques have been discovered for statically binding and inlining methods in order to realize high performance for dynamically-bound message sending. A common thread runs through all these techniques: creating multiple versions of machine code for a piece of source code, each version specialized for a particular situation. The generated code can thus be optimized for a specific context, even though the source code is completely general.

Customized Compilation

In Smalltalk, the **min:** method is defined in a general class (**Magnitude)**, and the Deutsch-Schiffman system compiles one machine code routine for it. Since many classes may inherit this method, the Smalltalk compiler cannot know the exact class of the receiver. Consequently, when **min** sends less-than, the target routine might be any less-than method in the system, and so a ten-cycle dynamically-dispatched call is needed to invoke a one-cycle compare instruction.

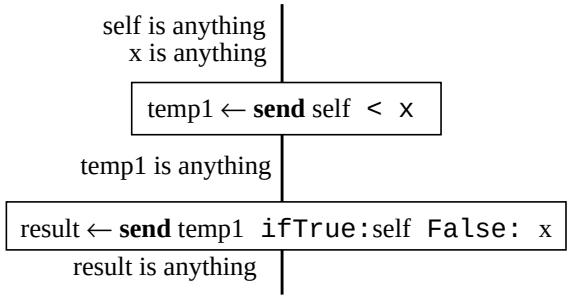
A customizing compiler, on the other hand, compiles a different machine code method *for each type of receiver* that runs a given source method. The advantage of this approach is that it gives the compiler precise information about the exact type of the receiver. As a result, the compiler can generate much better code for each specific version than it could for a single general-purpose compiled method.

Creating all possible versions after every source change would waste space and effort on many versions that will never run. Consequently, as with the Deutsch-Schiffman technique, compiled machine code is created on-demand, at run time. Dynamic compilation is an essential ingredient of the SELF implementation. The technique of dynamically compiling multiple specialized versions of a single source-code method is called *customized compilation*.

Consider the **min:** method defined in a slot inherited by many objects:

min: x = (self < x ifTrue: self False: x).

This notation means that there is a slot named "min:" and the code in the parentheses defines the method in that slot. Here is a general flow graph representation for this code (expensive operations are in bold face), including information about types (in this case for self, x, temp1, and result) at various points along the flow graph:*



This method could be invoked on integers, floating point numbers, strings, or any other objects that can be compared using `<`. Like other dynamic compilation systems, the SELF system waits until the **min:** method is first invoked before compiling any code for this method. Since the SELF compiler generates a separate version for each receiver type, it can customize the version to that specific receiver type, and use the new-found type information to optimize the < message by using the technique of *message inlining.*

Message Inlining

If the type of the receiver of a message is known at compile-time, the message lookup can be performed at compile-time rather than runtime. Since the compiler can then examine the slot found by the lookup, it can perform these critical optimizations:

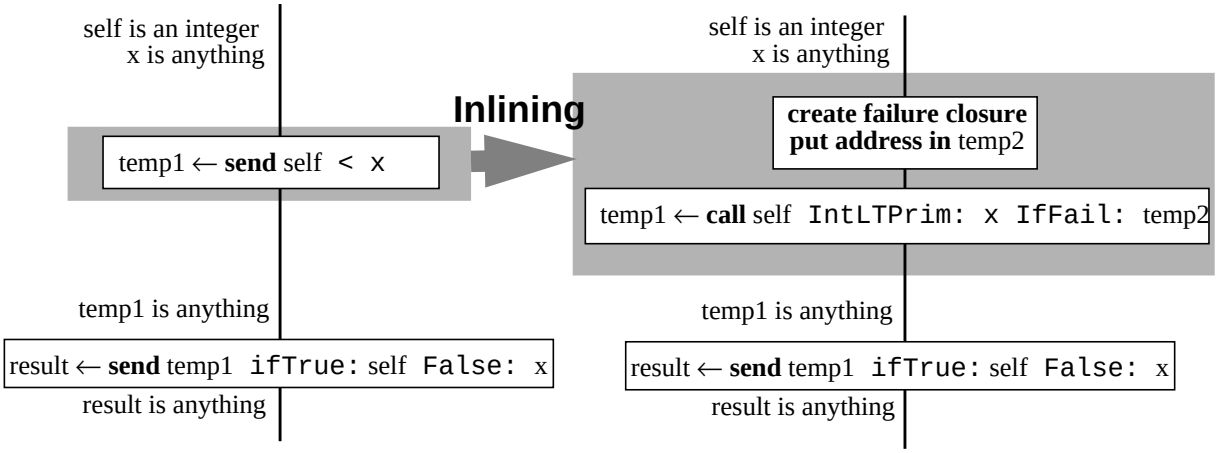
- If the slot contains a method, the compiler can *inline* the body of the method at the call site, if the method is short and nonrecursive.
- If the slot is a constant data slot, the compiler can replace the message send with the value of the slot (a constant known at compile-time).
- If the slot is an assignable data slot, the compiler can replace the message send with code to fetch the contents of the slot (e.g., a load instruction).
- If the slot is an assignment slot, the compiler can replace the message send with code to update the contents of the slot (e.g., a store instruction).

* To simplify the discussion, message sends that access local slots within the executing activation record (e.g., arguments) are assumed to be replaced with register accesses immediately.

For example, let's trace the operations of the SELF compiler to evaluate the expression **i min: j** for the case in which **i** is an integer. Assuming this is the first time **min:** has been sent to an integer, the compiler will generate code for a version of **min:** that is customized for integer receivers. In this case, the SELF compiler can statically look up the definition of **<** defined for integers:

< x = (IntLTPrim: x IfFail: [... ***code for handling the failure*****...] .)**

This method simply calls the integer less-than primitive and supplies a failure closure (whose code is omitted here for brevity). The compiler inlines the **<** method to transform the flow graph:

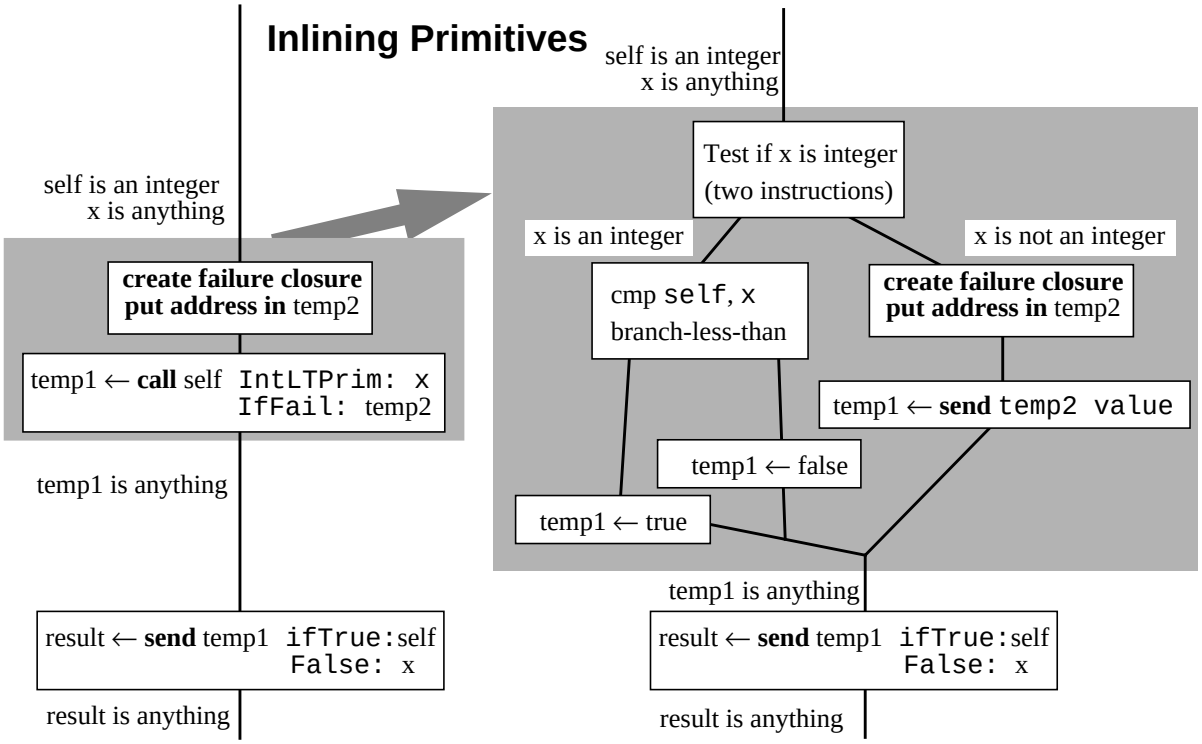


The overhead for sending the **<** message has been eliminated, but calling a procedure to compare two integers would still be expensive. The next section explains how a compiler can open-code primitive built-in operations to eliminate even this call overhead.

Primitive Inlining

Primitive inlining can be viewed as a simpler form of message inlining. Calls to primitive operations are normally implemented by calling an external function in the virtual machine. However, like most other high-performance systems, the SELF compiler replaces calls of certain common primitives, such as integer arithmetic, comparisons, and array accesses, with equivalent in-line instruction sequences. This substitution significantly improves performance since some of these primitives can be implemented in only two or three machine instructions if the overhead of the procedure call is removed. If the arguments to a side-effect-free primitive, such as an arithmetic or comparison primitive, are known at compile-time, the compiler can even call the primitive at compile-time, replacing the entire computation with the result of the primitive; this is how an inlining compiler can do *constant folding*.

In our ongoing **min:** example, the compiler inlines the **IntLTPrim:IfFail:** call (the definition of the integer less-than *primitive*, but not the integer less-than *method*, is hard-wired into the compiler) to get the following flow graph:

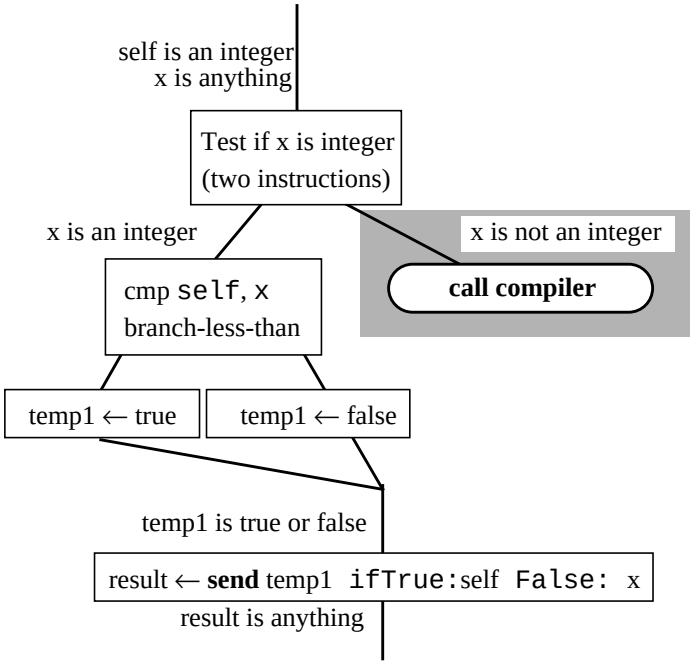


The first test is a compare-and-branch sequence which verifies that the argument to the **IntLTPrim:IfFail:** call is also an integer (the receiver is already known to be an integer because the method is being customized for integer self); if not, the failure closure is created and invoked. Thus this optimization not only eliminated the call overhead but also saved considerable time by not creating the failure closure unless really needed. On the most common branch (the argument is an integer), the two integers get compared, and either the **true** object or the **false** object is returned as the result of the < message.

Lazy Compilation of Uncommon Cases

At this point, a dynamic compiler can bring to bear an unorthodox optimization first suggested to us by John Maloney. If a certain primitive rarely fails, the compiler can save some time and improve its chances of producing better code by delaying the compilation of the failure branch. Instead of actually generating code, the compiler merely leaves a stub which will call the compiler to generate the failure code on-demand

if it is ever needed. The failure code will be an entirely different chunk of instructions and will not merge

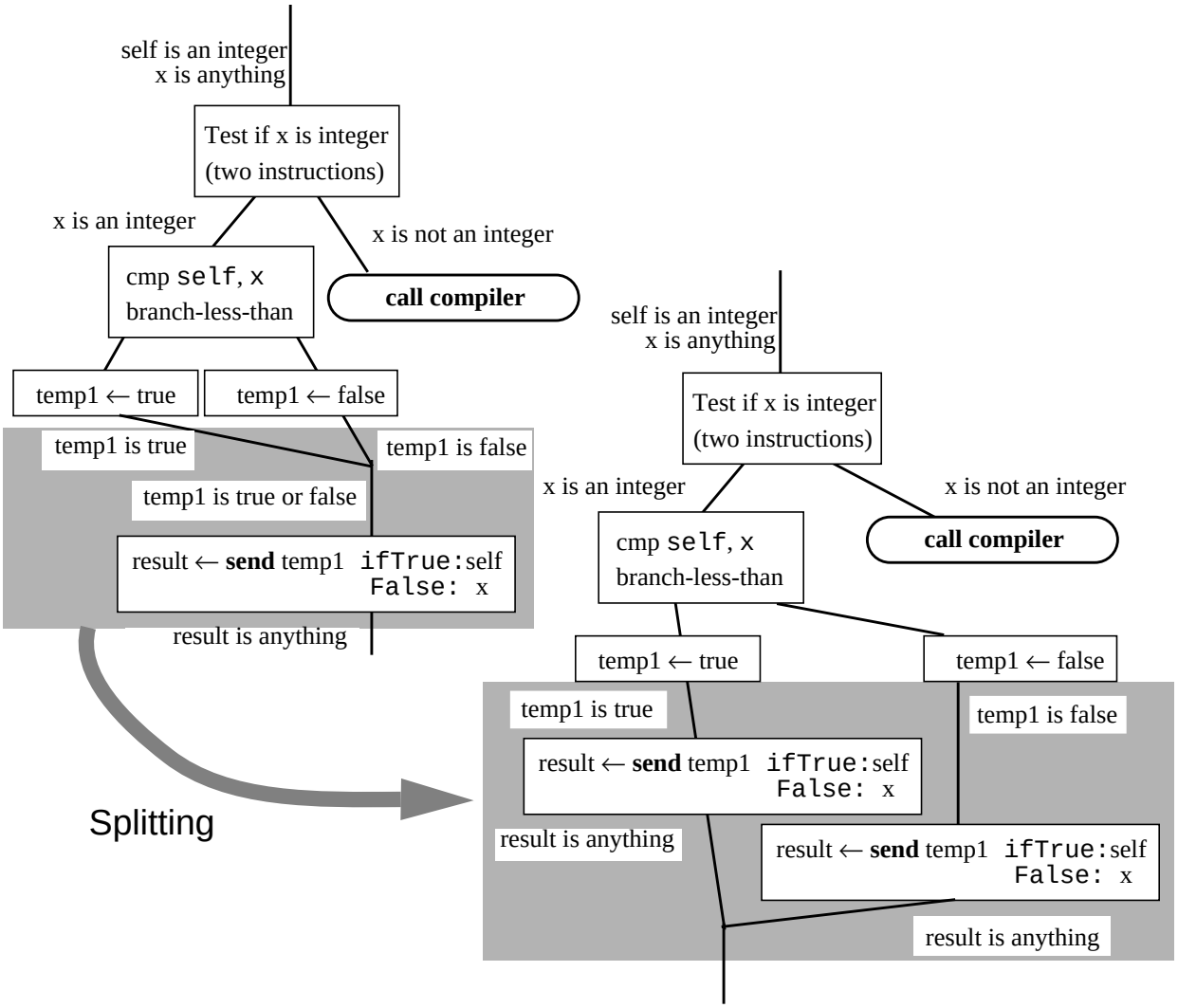


back into these instructions. Consequently, the compiler need not determine the effect of the failure code on the types of variables in this method, and the resulting code can therefore run faster. So at this point, the receiver of **ifTrue:False:** must be one of two objects: the true object or the false object. Normally, this would prevent inlining of the **ifTrue:False:** message, since the type of its receiver cannot be uniquely determined. However, by splitting the single call of **ifTrue:False:** into two calls, one for each statically-known receiver type, a compiler can handle and optimize each case separately. This technique is called *message splitting*.

Message Splitting

When several branches of the control flow graph merge, type information may be lost. In the **min:** example, for instance, there is a merge just prior to the **ifTrue:False:** message which loses type information about temp1. A message splitting compiler can postpone the merge until after the **ifTrue:False:** message send by duplicating this message send on both paths. Along each path, the compiler knows more type information, and can perform compile-time message lookup and message inlining to improve performance. In general, there may be a path for which the receiver type remains unknown, but the compiler can preserve the semantics of the original unsplit message by generating a real message send on this path. Message splitting can be thought of as an extension to customized compilation, because the compiler customizes individual message sends along particular control flow paths, with similar improvements in runtime performance.

For the **min:** example, the SELF compiler will split the **ifTrue:False:** message into two separate versions:



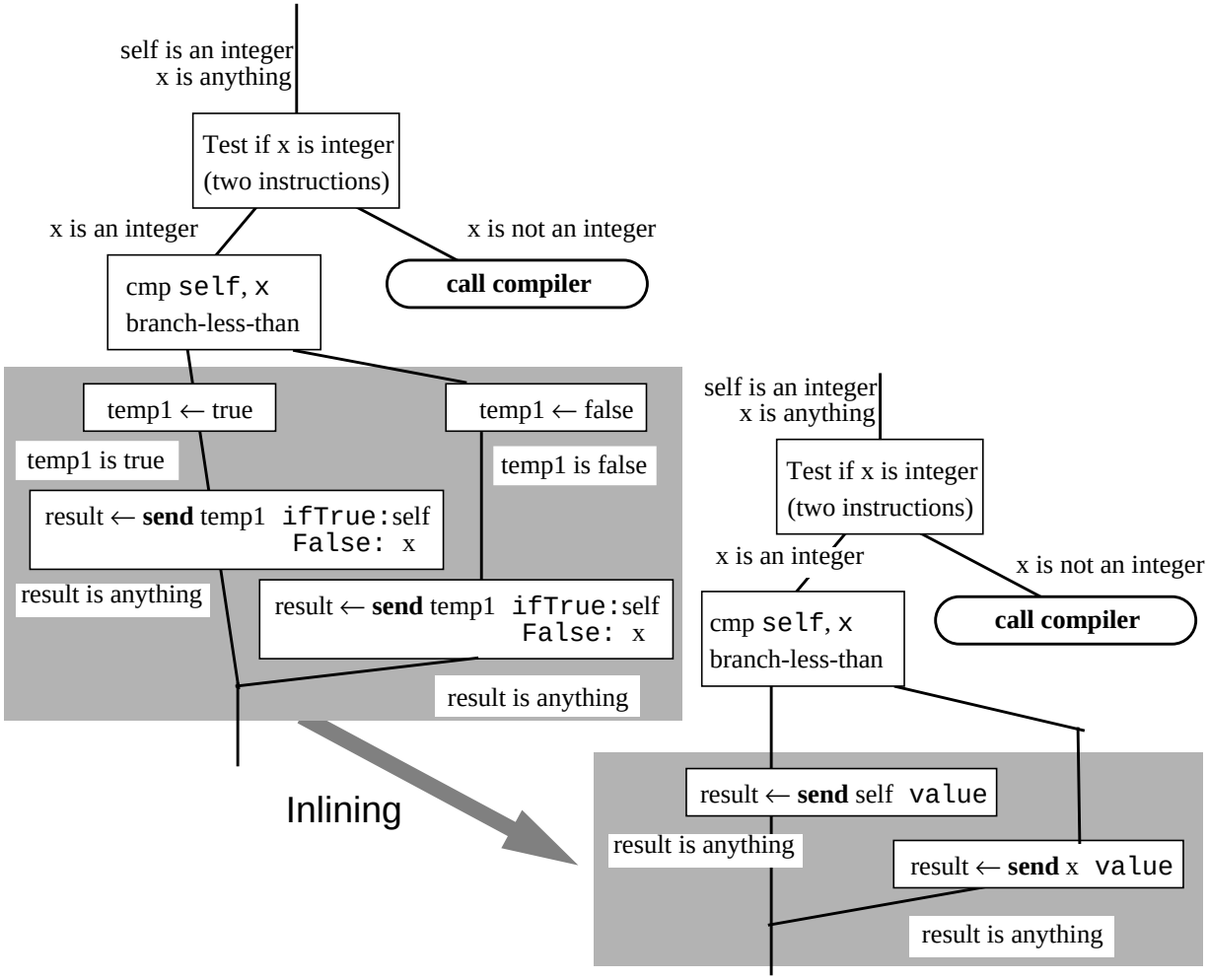
Now the compiler can inline the definition of **ifTrue:False:** for the **true** object:

ifTrue: truePart False: falsePart = (truePart value).

and for the **false** object:

ifTrue: truePart False: falsePart = (falsePart value).

to get to the following flow graph. Since the compiler knows that, at this point, both **self** and **x** must be inte-



gers, it can look up the **value** method. Integers inherit a general **value** method that just returns the receiver: `value = (self)`. This method is so short that the compiler can inline it and replace it with a register move instruction. Thus, in the common case of taking the minimum of two integers, the customized version of **min:** executes only two simple compare-and-branch sequences and a (register) move instruction. Similar savings will also be seen if the user calls **`min:`** on two floating point numbers or two strings, since customization, inlining, and splitting can also optimize special versions for each of these receiver types. Even though these compilation techniques can dramatically improve performance, they still preserve the message passing semantics of the original source code.

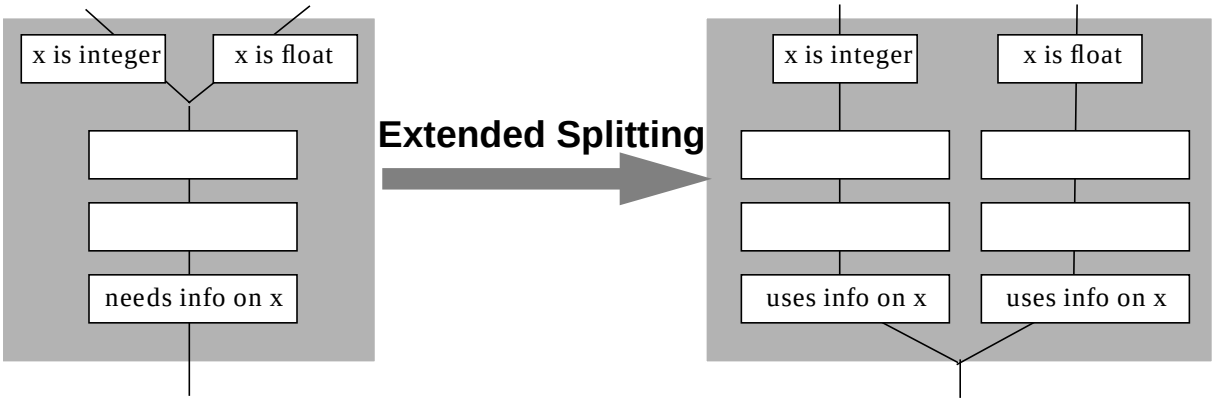
BEGIN SIDE BAR

Other Optimization Techniques

Besides dynamic customization and inlining, many more optimization techniques are possible. All of the following have been adopted in the current SELF system:

- **Type Prediction.** Sometimes, the compiler does not know the type of a message's receiver, but can make a good guess from the name of the message. For example, studies of Smalltalk code have shown that when the message is named **+**, **-**, **<**, etc. the receiver is an integer 90% of the time. Accordingly, a compiler can predict the types of receivers for these messages by inserting a type test into the control flow graph and assigning a high probability to the integer branch. Then splitting and inlining can take over to produce very fast code for the common case, and lazy compilation can save the compiler the trouble of compiling the non-integer case. Additionally, if the same objects are sent more messages later in the method, the compiler need not insert more type tests, since the objects are known to be integers along the common-case branch. Type prediction can, without adding much complexity, optimize the sort of integer-intensive code that many would expect to be fast. It was first used by Deutsch and Schiffman in their Smalltalk system [3]
- **Block (Closure) Elimination:** We have simplified the discussion of these techniques by ignoring issues raised by blocks. However, in a language like SELF that implements all control structures with them, block creation and invocation must be optimized away in order to achieve reasonable performance. This optimization is more difficult in SELF than, say, Scheme for two reasons: First, Scheme does not use blocks for if-s, but instead has a built-in construct, **cond**. Second, Scheme functions are easier to inline because function invocation does not perform dynamic-dispatching on user-defined types. Despite the difficulty of block elimination in a pure object-oriented language, the techniques outlined in this paper are equal to the task; they can keep track of which variables are bound to which blocks, and the same inlining techniques can inline-expand the code in the block.
- **Extended Splitting:** In the **min** example, the compiler split the **ifTrue:False:** message because the preceding node was a merge that lost type information. In other examples, there might have been other

messages between the merge and the would-be consumer of the lost information. Extended splitting allows the compiler to split whole paths instead of single messages.



- **Loops:** Once it can split whole paths, a compiler need take only a short step to split off whole loops. This optimization can make a dramatic difference by hoisting type tests and message sends out of tight loops. For example, in a loop that increments an integer index, the compiler can generate a separate loop without any type tests by splitting on the type of the index.
- **Adaptive Recompilation:** When performed at runtime, the advanced optimizations, as well as the more conventional optimizations such as register allocation and code scheduling, can result in compile pauses that distract and disturb users. This problem can be alleviated by including a fast and simple compiler in the system [5]. This simple compiler generates unoptimized code when a method is first invoked. Because the simple compiler spends little time on optimizations, runtime compilation pauses are minimized.* The simple compiler also inserts code to increment a counter at each execution of the method. If any of the unoptimized code proves to be a bottleneck (as ascertained by checking the counters) the optimizing compiler is called in to recompile and optimize the hot spot. Many SELF users are now willing to use the optimizing compiler, thanks to the advent of adaptive recompilation.
- **Supporting the Programming Environment:** Although customization, inlining, and the other techniques move the object code farther from the source code, it is still possible to hide their existence in order to provide source-level debugging and allow the programmer to change programs while they are running. For example, the SELF compiler outputs debugging information about the layout of stack frames, maintains dependency lists, and deoptimizes existing code and stack frames when needed.

* For interactive systems, we consider pauses of more than a tenth of a second to be distracting. Currently, our optimizing compiler frequently exceeds this limit, especially when it has to compile several new methods in short succession. However, the simple compiler stays well below that limit, with compile times averaging less than a millisecond per method on a SPARCstation 1.

Consequently, to the user, the system behaves exactly like an extremely fast interpreter: in terms of debugging, all optimizations are invisible to the user.

END SIDE BAR

BEGIN SIDE BAR

Benchmarking Methodology

The standard Sun C compiler with optimization enabled (using the `-O2` option) established a goal for run-time performance. Compilation time for optimized C includes the time to read and write files but not the time to link the resulting `.o` files together.

In order to compare our approach to implementing a pure object-oriented language with competing approaches, we measured the ParcPlace Smalltalk-80 system (version 4) which incorporates the Deutsch-Schiffman techniques [3]. As far as we know, this is the fastest implementation of any other system in which nearly every operation is performed by sending a dynamically-dispatched message. (In Smalltalk-80, unlike SELF, variable accesses and some common control structures do not use messages.)

To compare our techniques against other (non-object oriented) systems supporting generic arithmetic, we also measured the ORBIT compiler (version 3.1) [7] for T, a dialect of Scheme. ORBIT is well-respected as a good optimizing compiler for a Scheme-like language. In addition to the normal T program, we also measured a hand-optimized version of the benchmarks that uses unsafe integer-specific arithmetic (e.g. `fx+` and `fx<`) instead of the general + function and contains explicit directives to the compiler to inline certain functions. These data are labeled T/ORBIT (integer only). Compilation time includes the time to read and write files but not the time to load the generated file into the running T system.

In some ways, comparing these systems is like comparing apples to oranges. The SELF system, unlike the C or T compilers, includes support for message passing at the most basic levels (including user-defined control structures), generic arithmetic, robust error-checking primitives (e.g. test for overflow in integer addition), and full source-level debugging of the optimized code. We have directed much of our effort toward developing optimization techniques that coexist with the advantages of the SELF language and environment so that programmers need not choose between clean semantics and performance.

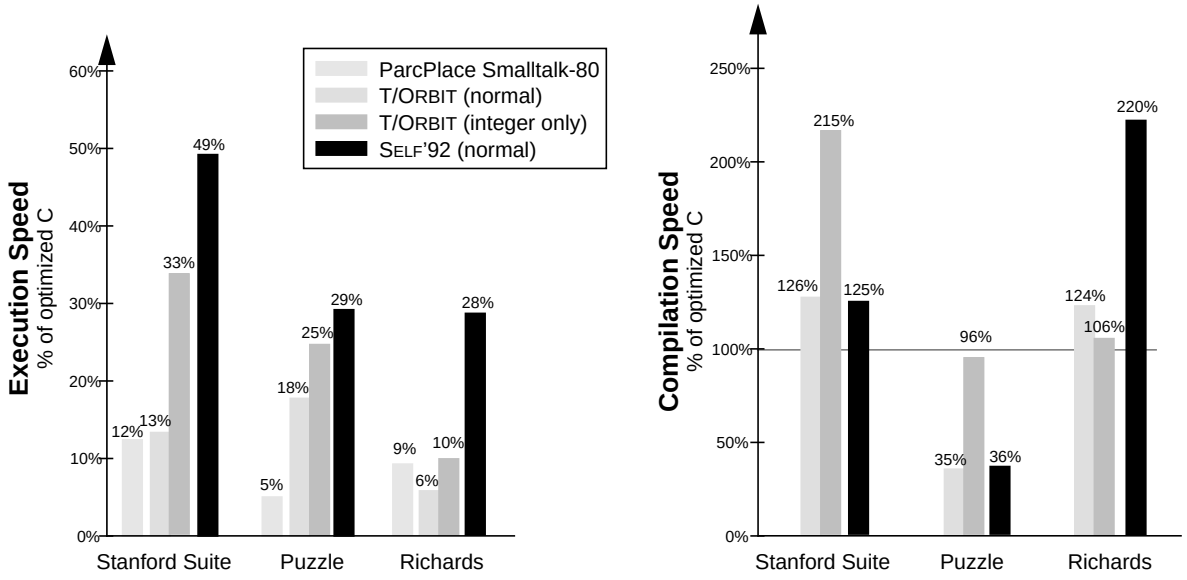
We measured the eight Stanford integer benchmarks and the Richards operating system simulation benchmark. The C version of the `richards` benchmark is actually written in C++ version 1.2 (using inline func-

tions whenever reasonable), translated into C using the standard `cfront` filter, and then optimized using the Sun C compiler. In the charts below, we report the geometric mean for the seven small Stanford integer benchmarks and `puzzle` and `richards` separately; this separates the benchmarks into rough "equivalence classes" based on benchmark size. All times were measured on a lightly-loaded SPARCstation-2. Raw data for each benchmark may be found in Appendix A.

END SIDE BAR

Performance results

The graphs below show the execution speed and compilation speed for our benchmarks, normalized to optimized C (taller bars are better).



The execution speeds we report do not include the cost of compiling, thus they reflect performance after the methods have been optimized. How effective are the new compilation techniques? Comparing Smalltalk to SELF suggests that they provide a four-to-six fold performance improvement. They even do a better job of eliminating the overhead of generic primitives than do unsafe type declarations in the T systems. There still remains a two- to threefold gap between SELF and C. However, the SELF code is doing more useful work, checking arithmetic overflow and array bounds. Of course, the pure object-oriented source code lends itself to reuse, whereas the C source employs nonreusable specific types and functions. Because SELF's optimizations occur at runtime, compilation speed is quite important. Fortunately, even though it performs extensive optimizations, SELF's compile times are in the same ball park as optimized C's.

Related Work

Others have tackled the challenge of inferring type information for programs without explicit type declarations. While these approaches can help the programmer write programs, they usually compute high-level abstract types that describe an object's interface, not its representation. Therefore, they cannot supply the low-level representation information the compiler needs to generate good code. A recently proposed algorithm [8] may partly solve this problem, computing representation-level type information for programs written in a Smalltalk-like language. However, it is too early to judge the practicality of this approach for large real-world programs.

A different approach to achieving object-oriented performance come out of the work by the Typed Smalltalk (TS) project [10, 11]. In TS, programmers can annotate programs with type declarations. Such annotations can serve to make the code more readable, can lead to fewer runtime errors, and can help the TS optimizing compiler to produce better code by performing runtime type casing and message inlining. Unfortunately, the type annotations needed by the optimizing compiler use representation-level types and thus turn reusable methods into more restricted versions that cannot be safely reused with other types. Although performance date for the TS system is very scarce, the few published numbers appear to indicate that TS reaches only about half the speed of the current SELF system.

An early APL compiler [9] used techniques similar to customization to generate more efficient code. If a certain expression was first executed with two-dimensional integer arrays, the compiler would generate "hard" code specialized for this case. If the type of the variables changed in later executions, the compiler would produce less specialized "soft" fall-back code. Unlike the SELF compiler, the APL system did not generate multiple specialized copies of code in order to save compile time and code space. Some compilers designed to optimize scientific FORTRAN programs can also duplicate procedures in order to improve the effect of certain optimizations [12]. Similarly, the partial evaluation technique called polyvariant specialization involves replicating nodes based on unique values or types of variables.

Conclusions

Object-oriented programming promises to make it easier to write programs. Dynamically-bound messages allow methods to transcend the representation of the objects they manipulate, and inheritance allows different objects to share the same methods without copying them. SELF offers a particularly simple and pure version of object-oriented programming in which every action is performed by sending a message. SELF's utter uniformity allows code to be reused and refactored even more easily than in other object-oriented languages.

However, dynamically-bound messages cannot be optimized by conventional compilers. For example, a conventional implementation of SELF would probably run at least forty times slower than optimized C. Fortunately, this performance barrier can be overcome with new compilation techniques. *Customization* creates multiple copies of the same source method, each specialized for a particular receiver type. In every copy, many messages can subsequently be *inlined*, often completely eliminating the overhead of message passing. *Splitting* carries this idea one step further by generating specialized versions for parts of methods. Combined with other techniques such as type prediction, these new compilation methods can dramatically reduce the high frequency of dynamically-dispatched message sends. Subsequent conventional optimizations such as global register allocation and code scheduling can be used to speed up the code even further.

The new compilation techniques help make pure object-oriented languages practical. The measured benchmarks run four to six times faster in SELF than the next fastest pure object-oriented language implementation. In fact, the compilation techniques developed for SELF have narrowed the performance gap between SELF and optimized C to a factor of two for the Stanford integer benchmarks, despite the vastly different execution models of these two languages. Today, even a radically pure language such as SELF can be implemented efficiently, combining the benefits of ubiquitous objects with good performance.

Acknowledgments

Many people have done much to make this work possible. The other members of the SELF group, Bay-Wei Chang, Ole Agesen, and John Maloney have put their hearts and souls into other aspects of this project. Elgin Lee was part of the group for a time in the beginning and helped bring the first implementation of SELF to life. Peter Deutsch has been a constant source of interesting ideas and inspiration. Since 1991, the SELF project has found a congenial home in Sun Microsystems Laboratories, thanks to Wayne Rosing, Bill Joy, and Jim Mitchell. The first author would also like to thank David Patterson for teaching him how to think about optimizing and analyzing the performance of computer systems. Finally, we would like to express our gratitude for our families' constant support.

References

1 Joseph Falcone, James Stinger. The Smalltalk-80 Implementation at Hewlett-Packard, in *Smalltalk-80: bits of history, words of advice*. Addison-Wesley, Reading, MA, 1983.

2 David Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.

3 L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297-302, Salt Lake City, UT, 1984.

4 *Lisp and Symbolic Computation: An International Journal*, 4, 3 (1991), Kluwer Academic Publishers, The Netherlands, 1991. Special issue on SELF including revisions of SELF papers previously published in OOPSLA '87, OOPSLA '89, and PLDI '90.

5 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991.

6 Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA'86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.
7 David Andrew Kranz. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. thesis, Yale University, 1988.
8 Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *ECOOP'92 Conference Proceedings*, Utrecht, The Netherlands, June, 1992.
9 Ronald E. Johnston. The Dynamic Incremental Compiler of APL\3000. *ACM APL Quote Quad 9(4)*, June 1979, pp. 82-87.
10 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.
11 Justin O. Graver and Ralph E. Johnson. A Type System for Smalltalk. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pp. 136-150, San Francisco, CA, January, 1990.
12 Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, pp. 96-105, Oakland, CA, April 1992.

Appendix A Per-Benchmark Raw Data

Compile times are not available for Smalltalk-80. All times were measured on a lightly-loaded SPARCstation-2.

Run Times (milliseconds)	Smalltalk	T (normal)	T (int only)	SELF'92	C (opt)
bubble	1020	440	140	95	67
matrix multiply	640	1230	500	300	110
perm	540	460	110	96	44
queens	330	250	95	55	35
quicksort	470	700	340	105	41
towers	380	300	130	120	63
treesort	750	670	460	500	220
puzzle	6790	1830	1320	1170	335
richards	3250	4520	2860	1040	290

Compile Times (sec)	SELF'92	T (normal)	T (int only)	C (opt)
bubble	1.4	1.4	0.7	1.4
matrix multiply	1.5	1.5	0.7	2.3
perm	1.6	1.0	0.5	2.4
queens	2.8	1.7	0.8	1.4
quicksort	2.0	1.6	0.9	1.8
towers	0.8	1.7	1.8	2.6
treesort	1.1	1.7	1.2	1.6
puzzle	12.6	12.7	4.7	4.5
richards	3.1	5.5	6.4	6.8

BEGIN SIDE BAR

To probe further

More detailed papers on the SELF language and the compilation techniques mentioned here (including references 4 and 5) are available electronically in PostScript form. To obtain a copy, ftp to self.stanford.edu and read the README file in /pub/papers. The current SELF implementation is also available without charge from the same host; see the file /pub/README for details. Finally, there is a mailing list for people interested in SELF; send mail to self-request@self.stanford.edu to be added to it.

END SIDE BAR