Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language*

Craig Chambers
David Ungar
Stanford University

Abstract

Dynamically-typed object-oriented languages please programmers, but their lack of static type information penalizes performance. Our new implementation techniques extract static type information from declaration-free programs. Our system compiles several copies of a given procedure, each *customized* for one receiver type, so that the type of the receiver is bound at compile time. The compiler *predicts* types that are statically unknown but likely, and inserts run-time type tests to verify its predictions. It *splits* calls, compiling a copy on each control path, optimized to the specific types on that path. Coupling these new techniques with compile-time message lookup, aggressive procedure inlining, and traditional optimizations has doubled the performance of dynamically-typed object-oriented languages.

1. Introduction

Object-oriented languages contain a number of features that make programs easier to write but slower to run. Chief among these is *message passing*, in which procedures (called *methods*) are invoked indirectly, based on the type (or *class*) of the first argument (the *receiver*);** message passing can be significantly more expensive to implement than normal procedure calls. To take full advantage of the extra level of indirection for procedure calls, object-oriented programs tend to be composed of many small procedures, increasing the relative overhead of procedure calls and further aggravating the implementation problem. In addition, *pure* object-oriented languages use message passing for all computation, avoiding built-in operators and control structures. The resulting call density of pure object-oriented programs is staggering, and brings naive implementations to their knees.

Some languages minimize message passing by including static procedure calls and built-in operators for non-object-oriented code. For example, C++ [Str86] includes C's repertoire of built-in operators and control structures, and some object-oriented Lisps [Moo86, Bob88] include normal Lisp functions, such as `car` and `cdr` which only work on `cons`-cells. While these impure object-oriented languages can avoid the cost of message passing with non-object-oriented constructs, the resulting programs are significantly restricted in flexibility and reusability.

To reduce the cost of message passing, some object-oriented languages, such as C++, Trellis/Owl [Sch86], and Eiffel [Mey86], include explicit type declarations. This allows the implementation to reduce the cost of a message send (or virtual function call) to no more than an indirect procedure call, but forces the programmer to enter and maintain the type declarations, and constrains him to write programs that will statically type-check in the language's type system.

Pure dynamically-typed object oriented languages, such as the Smalltalk-80 language*** [GR83], defer type-checking until run-time, freeing the programmer from the burden of explicit type declarations and allowing the programmer more flexibility in the kinds of programs that can be written easily. Unfortunately, the dearth of static information dramatically increases the cost of message passing. Dynamically-typed object-oriented systems historically have suffered from poor performance; even the fastest Smalltalk-80 implementation, chock full of clever compiler techniques and run-time support, is still more than 10 times slower than optimized C, according to our measurements of some small C benchmarks translated into Smalltalk.

We are working on SELF [US87], a pure dynamically-typed object-oriented language that contains even *less* static information than Smalltalk. Like Smalltalk, SELF eschews explicit type declarations and built-in control structures. However, unlike Smalltalk and most other object-oriented languages, SELF has no classes. Instead it is based on the *prototype object model*, in which each object defines its own object-specific

* This work has been generously supported by a National Science Foundation Presidential Young Investigator Grant # CCR-8657631, and by IBM, Texas Instruments, NCR, Tandem Computers, Apple Computer, and Sun Microsystems.

** Some object-oriented languages dispatch based on the types of several arguments.

*** Smalltalk-80 is a trademark of ParcPlace Systems, Inc. Hereafter when we write "Smalltalk" we will be referring to the Smalltalk-80 system or language.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-306-X/89/0006/0146 $1.50

behavior, and inherits shared behavior from its parent objects.* Also unlike Smalltalk, SELF accesses state solely through messages; SELF has no syntax for accessing a variable or changing its value. These two features, combined with SELF's multiple inheritance rules, help keep programs concise, malleable, and reusable. Unfortunately, these features make the implementation's job harder, even variables would have to be accessed using full message sends.

We have invented new implementation techniques that enable our initial SELF compiler to generate machine code that is twice as fast as the fastest Smalltalk-80 implementation, and only 4 to 5 times slower than optimized C. Our compiler creates and preserves compile-time type information wherever possible by performing *customized dynamic compilation*, *static type prediction*, and *message splitting*. This static type information makes it possible to look up methods at compile-time and to compile those methods in line. In addition, the SELF system supports garbage collection, incremental recompilation for short programming turnaround times, and complete source-level debugging; these features are beyond the scope of this paper.

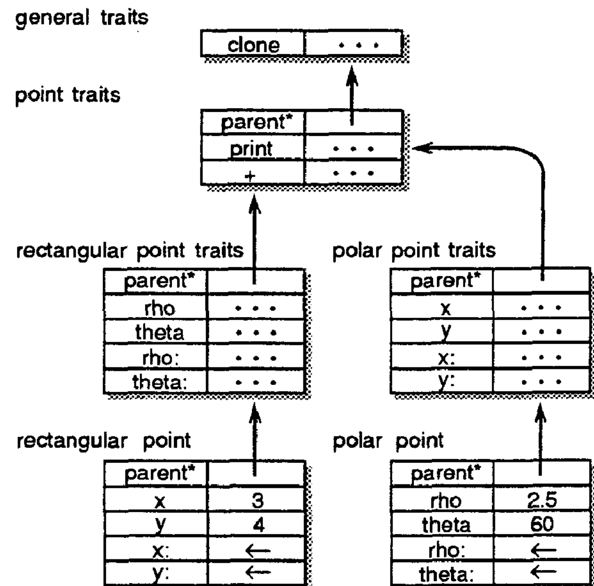
This paper describes our new techniques in detail. The next section presents a simple overview of the SELF object and execution model. In section 3 we describe relevant parts of our memory system; in section 4 we describe our compiler. Section 5 compares actual performance measurements for our initial SELF system to measurements of a fast Smalltalk-80 implementation and an optimizing C compiler. The last sections discuss current, future, and related work.

2. Overview of SELF

SELF objects consist of *named slots*, each of which contains a reference to some other object. Some slots may be designated as *parent* slots (by appending an asterisk to their name). Objects may also have SELF source code associated with them, in which case the object is called a *method* (similar to a procedure). To make a new object in SELF, an existing object (called the *prototype*) is simply *cloned* (shallow-copied).

When a message is sent to an object (called the *receiver* of the message), the object is searched for a slot with the same name as the message. If a matching slot is not found, then the contents of the object's parent slots are searched recursively, using SELF's multiple inheritance rules to disambiguate any duplicate matching slots. If a matching slot is found, its contents is *evaluated* and the result is returned as the result of the message send.

* To get an idea of how unusual this is, note that some respected authorities have gone so far as to define object-oriented languages as those with classes [Weg87]. Other prototype models are discussed in [Bor86, Lie86, LTP86, Ste87].



Five SELF objects. The bottom objects are two-dimensional point objects, the left one using rectangular coordinates and the right one using polar coordinates. The middle left object is the immediate parent object shared by all rectangular point objects, and defines four methods for interpreting rectangular points in terms of polar coordinates; the middle right object does the same for polar point objects. The second-to-top object is a shared ancestor of all point objects, and defines general methods for printing and adding points, regardless of coordinate system. This object inherits from the top object, defining even more general behavior, such as how to copy objects.

An object without code evaluates to itself (and so the slot holding it acts like a variable). An object with code (a method) is a prototype activation record. When evaluated, the object clones itself, fills in its **self** slot with the receiver of the message, fills in its argument slots (if any) with the arguments of the message, and executes the code associated with the object. The **self** slot is a parent slot so that the cloned activation record inherits from the receiver of the message send.

For example, in the point example pictured here, sending the x message to the rectangular point object finds the x slot immediately. The contents of the slot is the integer 3, which evaluates to itself (it has no associated code), producing 3 as the result of the x message. If x were sent to the polar point object, however, x wouldn't be found immediately. The object's parents would be searched, which would find the x slot defined in the polar point traits object. That x slot contains a method that computes the x coordinate from the rho and theta coordinates:

```
rho * theta cos.
```

The method gets cloned and executed, producing the floating point result 1.25.

If the print message is sent to a point object, the print slot defined in the point traits object is found. The method contained in the slot prints out the point object in rectangular coordinates:

```
x   print.
', ' print.
y   print.
```

If the point is represented using rectangular coordinates, the x and y messages will access the corresponding data slots of the point object. But this method works fine even for points represented using polar coordinates: the x and y messages will find the conversion methods defined in the middle right object, and compute the correct x and y values.

SELF supports assignments to data slots by associating an *assignment slot* with each assignable data slot. The assignment slot contains the *assignment primitive* object (represented in the picture using ←). When the assignment primitive is evaluated as the result of a message send, it stores its argument into the associated data slot. A data slot with an associated assignment slot is called an *assignable slot*; a slot with no corresponding assignment slot is called a *constant* or *read-only slot*. A parent slot may be either a constant slot, yielding conventional inheritance semantics, or it may be an assignable slot, permitting an object's inheritance to change on-the-fly. This *dynamic inheritance* is one of SELF's linguistic innovations, and may prove to be a useful addition to the set of object-oriented programming techniques.

SELF allows programmers to define their own control structures using blocks. A block contains a method in a slot named value; this method is "magical" in that when it is invoked (by sending value to the block), the method runs as a child of its lexically enclosing activation record (either a "normal" method activation or another block method activation). The **self** slot is not rebound when invoking a block method, but instead is inherited from the lexically enclosing method. Block methods may be terminated with a *non-local return* expression, which returns a value not to the caller of the block method, but to the caller of the lexically-enclosing non-block method, much like a return statement in C.

2.1. An Example

Let's look at a small piece of SELF code; we will come back later to this example to illustrate the compiler's optimizations and transformations. This example sums up the numbers from the receiver to some upper bound, and is defined in a parent object inherited by all numbers:*

```
sumTo: upperBound = (
    | sum <- 0 |
    to: upperBound Do: [
        | :index |
        sum: sum + index ].
    sum )
```

In ANSI C the example would be written as:

```
int sumTo(int self, int upperBound) {
    int sum = 0;
    int index;
    for ( index = self; index <= upperBound; index ++)
        sum = sum + index;
    return sum;
}
```

The SELF method begins by specifying the method name sumTo: followed by the name of the local argument slot upperBound. The body of the method first declares a local data slot named sum, initialized to 0, and implicitly its assignment slot, sum:, containing the assignment primitive. The method performs three message sends:

1. The upperBound message is sent to **self**, with the lookup beginning with the current activation record. This message will fetch the contents of the upperBound argument slot at run-time.
2. The to:Do: message is sent to **self** with the result of the upperBound message as the first argument and a block literal object (enclosed in brackets) as the second argument. to:Do: invokes a standard for-loop-style iteration method, that calls the block argument on each of the numbers between the receiver and the first argument. The result of the to:Do: message is ignored.
3. On the last line, the sum message is sent to **self**, accessing the contents of the method's sum slot. The value of this last expression is returned as the result of the sumTo: method.

The block literal object contains a slot named value: holding a lexically-scoped method. The block method declares an argument slot named index and sends four messages:

1. The sum message is sent to **self**. Since block methods are lexically scoped, the sum message must fetch the contents of the sumTo: method's sum local slot. This works because the cloned block activation record will inherit from the cloned method activation record.
2. The index message is sent to **self**. This fetches the contents of the block method's index argument slot.
3. The + message is sent to the result of the sum message with the result of the index message as its argument. The + message will invoke the appropriate addition operation defined for the object stored in the sum slot (such as the integer addition method or the floating point addition method).

---

* SELF syntax is similar to Smalltalk syntax, especially for message sends. The main differences are that messages sent to **self** do not explicitly include self in the source text, keyword messages use capitalized names for all but the first keyword part and associate right-to-left, and slot declarations may include explicit initializers.

4. Finally the sum: message is sent to **self** with the result of the + message as its argument. This invokes the assignment primitive stored in the enclosing method's sum: slot, which stores the new sum in the method's sum slot.

The SELF sumTo: method works for different numeric types, and handles overflow from integer representation to floating point representation, all by using the polymorphic + message and dynamic typing. The C version only works for integers, and produces erroneous results in the case of overflow.

2.2. Byte-Coded Internal Representation

To avoid reparsing textual representations of programs, SELF source code is represented internally using a SELF-level byte code object. The SELF parser is invoked once when the textual source code is initially entered into the system, producing SELF prototypical activation records, each composed of a method object holding the arguments and local slots and a byte code object representing the source code; the SELF compiler generates machine code directly from the prototype activation records. The byte code object contains an array of byte-sized opcodes representing the original source code's abstract syntax tree. The following are the significant byte codes defined by our SELF system, specified as if for execution by a simple stack-oriented interpreter:

SELF push **self** onto the execution stack

LITERAL <value>
push a literal value onto the execution stack

SEND <message name>
send a message, popping the receiver and arguments off the execution stack, and pushing the result

NON-LOCAL RETURN
execute a non-local return from the lexically-enclosing method activation

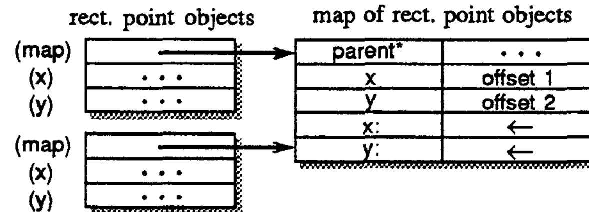
In SELF source code, primitive operations are invoked using normal message sending syntax, but with the message name beginning with an underbar ("_"). This allows the normal SEND byte codes to represent all primitive operations, and facilitates extensions to the set of available primitive operations.

3. Memory System Implementation

SELF's prototype model greatly impacts the design of the object storage system and, in turn, the compiler. Class-based languages can represent instances very concisely because each instance only needs space for its variables and class pointer; the class holds the *format* (names and locations of the instance variables), *behavior* (names and definitions of the class' methods), and *inheritance* (the class' superclass(es)) for all of its instances. On the other hand, a prototype-based language such as SELF has no classes; instead each object defines its own format, behavior, and inheritance. In principle each SELF object could be completely unique, requiring enough space to hold both the instance-like information and the class-like information.

However, in practice most SELF objects will form groups "struck from the same mold," cloned from the same "prototypical" object. For example, the SELF user could create a new kind of object, say a point object, and add a reference to the new point object in a standard object dictionary inherited by all objects. Then whenever the user's program needs a new point object, the program simply clones the original prototype point and sets the copy's x and y coordinates. All point objects have the same format, behavior, and inheritance. We call the groups of objects cloned from an original member of the group a *clone family*; all members of the family are identical except possibly for the contents of their assignable data slots (corresponding to instance variables in Smalltalk).

In our SELF implementation, we take advantage of clone families by factoring out the common information into a shared immutable object called a *map*. The representation for a SELF object consists only of the contents of the object's assignable slots (the "instance variables") plus a pointer to the shared map object. For each slot in the object, the map contains the name of the slot, whether the slot is a parent slot, and either the offset within the object of the slot's contents (if it's an assignable slot) or the slot's contents itself (if it's a constant slot, such as a non-assignable parent slot or a method). If the object has code (i.e., is a method), the map also stores a pointer to the byte code object representing the source code of the method.



An example of the map for two rectangular point objects (polar point objects are in a different clone family and have a different map). Each point object holds only assignable slot contents; all constant slots and format information are factored out into the map.

Maps are immutable so that they may be freely shared by objects in the same clone family. However, when the user changes the format of an object or the value of one of an object's constant slots, the map no longer applies to the object. In this case, a new map is created for the changed object, and the object now starts its own clone family; the old map still applies to any other members of the original clone family.

From the implementation point of view, maps look much like classes, and achieve the same sorts of space savings for shared data. But maps are totally transparent at the SELF language level, simplifying the language and increasing expressive power by allowing

objects to change their formats at will. Maps also play a crucial role in the compiler. An object's map represents its *data type*, and type information about objects (i.e., to what clone families the object might belong) is used throughout the compiler to drive the optimizations. For example, in the absence of dynamic inheritance, all objects that are members of a clone family (i.e., objects with the same data type) will have identical inheritance hierarchies, and so a method that applies to one member of a clone family will apply equally to all members of the clone family.

Our object storage system is responsible for more traditional functions as well. We use Generation Scavenging [Ung86, Lee88] for automatic storage reclamation; this algorithm requires little run-time overhead (around 3% of the CPU time) and can have split-second pause times. Like most systems supporting garbage collection, our SELF implementation tags data words to distinguish among pointers, integers, and floating point numbers. Unlike many Smalltalk systems, we do not use an object table, since the extra level of indirection would degrade performance unnecessarily.

4. The SELF Compiler

4.1. Previous Compilation Techniques

Unlike traditional batch compilers, our compiler is invoked on a per-method basis when a method is first invoked; this technique of *dynamic translation* was pioneered by Deutsch and Schiffman in their high-performance Smalltalk system [DS84]. Our compiler generates machine code from the byte code object, and caches the compiled code for use the next time this method is invoked. If the compiled code cache fills up, some previously-compiled methods are flushed from the cache, and recompiled when next needed.

In order to reduce the cost of message sends without the benefit of static type information, our SELF system, like the Smalltalk system developed by Deutsch and Schiffman, uses an *inline caching* technique. To implement a message send, the compiler generates a call to a run-time message lookup routine. The first time the lookup routine is invoked from a particular call site, it *backpatches* the call instruction to invoke the target method instead of the general lookup routine. The next time the message is sent from that call site, the target method will be invoked directly instead of going through a long message lookup. Inline caching has the effect of replacing the costly message send with a simple procedure call.

However, the receiver of a message may be in a different clone family than it was the previous time, and therefore may use a different method to implement the message. To handle this case, the first few instructions of the compiled method check to make sure that the receiver has the correct map for that method. If the receiver's map is correct, the method executes normally. If the map is incorrect, the full message lookup routine is invoked to find and backpatch the right method. If the type of the receiver of a particular send doesn't change very often, then this optimization gains a lot of performance; inline caching is successful around 95% of the time [DS84, Ung86, Lee88].

To speed method call and return, our SELF system, like many other high-performance Smalltalk and Lisp systems [DS84, Kra86], allocates cloned activation records on a stack rather than in the heap; local slots in the activation record may be allocated to machine registers rather than in memory. To speed certain common primitive operations, such as integer arithmetic and array indexing, our SELF compiler inlines the definition of the invoked primitive, rather than generating an external procedure call. This technique is common in traditional optimizing compilers and even in some Smalltalk systems [DS84, Joh88].

Our system supports a number of compiler techniques typically omitted in dynamically-typed object-oriented language implementations. First, our compiler propagates type information within an expression. Obvious sources of type information are literals such as integers, strings, and blocks. Another source in our implementation is the result of a primitive operation. The compiler includes information about each primitive, including whether the primitive may fail and the type of the result if the primitive succeeds. Possible result types include: the same object as the receiver (for assignments, for example), the same type as the receiver (for the clone primitive, for example), a specific type (such as the integer type for the result of integer arithmetic primitives), or a boolean result, which is either the **true** object or the **false** object (the result of comparison primitives).

Second, our system performs *compile-time message lookup*. If the map of the receiver is known at compile-time, the compiler searches the map for a slot that matches the message name. If found, the compiler has successfully resolved the message send at compile-time, reducing an expensive message send to a statically-bound procedure call. If the compiler doesn't find a matching slot, it recursively searches the receiver's parents for a matching slot, as long as the parents are constant (not assignable) and thus stored in the receiver's map. A special case of compile-time lookup handles local variable accesses: message sends that search the local slots of executing activation records.

Third, our system includes a *general message send inliner*, similar to a conventional procedure inliner, which can replace message sends which have been successfully looked-up at compile-time with faster code. If the slot found by the lookup routine is a local assignable data slot, the compiler simply uses the register or stack location allocated to the slot. If the slot is an assignable data slot in an object in the heap, the compiler

generates code to load the contents of the slot from the appropriate offset in the object. If the slot is a constant data slot, the compiler replaces the message send with the constant value of the slot. If the slot is an assignment slot, the compiler generates code to do the store into the corresponding data slot, whether it's in a register, a stack location, or an object in the heap.

If the slot contains a method, the inliner can elect to compile the body of the method inline, if it isn't recursive or too long. Since object-oriented programs tend to be composed of many small procedures, method inlining is important to reduce the call density of the resulting compiled code. If a block method is inlined and the original block object is no longer needed as a value, the compiler eliminates the expensive block creation operation; this optimization is especially important, since all control structures involve block objects [Kra86].

Our system also performs *constant folding* (executing a side-effect-free primitive with constant arguments at compile-time) and some *dead code elimination*. Our second-generation SELF compiler, currently under construction, supports additional optimizations such as *type and data flow analysis*, *common subexpression elimination*, *global register allocation*, and *instruction scheduling*. However, our compiler abstains from any optimizations that would destroy the illusion of interpreting byte codes or that would prevent complete source-level debugging. For example, tail recursion elimination is disallowed since it would destroy stack frames needed at debug-time. Fortunately, these restrictions have not severely hampered our compiler's ability to generate efficient code.

Let's consider applying these optimizations to the simple sumTo: example, to investigate the state-of-the-art in implementing dynamically-typed object-oriented languages. Recall that the example is:

```
sumTo: upperBound = (
    | sum <- 0 |
    to: upperBound Do: [ | :index | sum: sum + index ].
    sum )
```

As described so far, the SELF compiler would generate code for the sumTo: source method from the pre-parsed byte code object when this method is first called. It would allocate the receiver, the argument upperBound, and the local slot sum to registers; the argument index of the block would be allocated to a register in the block method's stack frame. The sends of upperBound and sum could be successfully looked-up at compile-time (since they access local slots) and replaced by simple register accesses. However, given only the techniques discussed above, the to:Do: message cannot be successfully looked-up at compile-time, since the compiler doesn't know the specific type of **self** (it could be any object that happens to inherit the sumTo: method being compiled). Instead, the to:Do: message must be implemented as a full message send, and the block literal object must be cloned at run-time and initialized with the cloned method activation record as its lexical scope.

To do better, the compiler needs more type information. In general, since the call density of methods compiled using only the above techniques would be so high, other optimizations like common subexpression elimination and global register allocation wouldn't have much impact; they typically don't work well across procedure calls, and any performance improvement they make would be swamped by the cost of the procedure calls.

The next three subsections describe the new techniques we use in our SELF compiler to create more static type information for the compiler. The extra type information allows the compiler to statically bind and inline many message sends, greatly reducing the call density of the resulting compiled methods. Not only does this directly improve performance, it opens the door for more traditional optimizations like common subexpression elimination and global register allocation, allowing compiled SELF programs to close in on the performance of traditional compiled languages.

4.2. Customized Compilation

An important contribution of our work is *customized compilation*. We use this term to mean that the dynamic compilation of a given source method is customized by important characteristics of the calling site; thereafter the customized compiled method is only used for calls with the same characteristics as the original calling site. Since not all call sites have the same characteristics, multiple compiled versions of the same source method may coexist in the compiled code cache. Although customized code occupies more space and takes longer to compile, it has the potential to run much faster.

In our current implementation, we customize the compiled method by the map of the message's receiver; only members of the receiver's clone family may use the customized method.* The compiler knows the type of **self**, and, in the absence of dynamic inheritance, can successfully look-up *all* sends to **self** at compile-time. This in turn leads to many more message sends being candidates for inlining.

Using customized compilation in the sumTo: example leads to some remarkable results. The compiler generates a compiled version of sumTo: customized for the type of the receiver, for example an integer (different versions of the sumTo: method would be compiled and customized for floating point numbers and arbitrary-precision "bignums"). Since the compiler knows the map of the receiver, it can find and inline the to:Do: method which is defined in the parent object for integers:

```
to: end Do: block = ( to: end By: 1 Do: block )
```

* Special precautions have to be taken to handle dynamic inheritance properly; these are briefly described in section 6.

This method merely supplies a default increment of 1 and calls to:By:Do:, a slightly more general iteration method. After inlining this method, the original sumTo: becomes (the changed text appears in boldface):

```
sumTo: upperBound = (
    | sum <- 0 |
    to: upperBound By: 1 Do: [ |:index| sum: sum + index].
    sum )
```

Again, because the code is customized according to the receiver, the compiler can find the appropriate implementation of to:By:Do::

```
to: end By: step Do: block = (
    step = 0
        ifTrue: [ error: 'step is zero' ]
        False: [
            step < 0
                ifTrue: [ "stepping down" ... ]
                False: [ "stepping up"
                    | i |
                    i: self.
                    [ i <= end ] whileTrue: [
                        block value: i.
                        i: i + step. ] ] ] )
```

This general iteration method tests the step value to select the code for either counting up or counting down. We show only the relevant portion that counts up. This portion defines an assignable slot i which is used to hold the index, initializes it to self, then invokes the whileTrue: method. Its receiver is the block [i <= end] (the looping condition), and its argument is the block [block value: i. i: i + step.] (the body of the loop). This latter block first evaluates the argument block (which when called by sumTo: will be [| :index | sum: sum + index]), passing in the iteraton counter, i, as the argument to the block, and then increments the counter by the value of step (in this example the constant 1).

The compiler inlines this to:By:Do: method into sumTo:. After replacing formals with actuals, the compiler is faced with this intermediate form:

```
sumTo: upperBound = (
    | sum <- 0 |
    1 = 0
        ifTrue: [ error: 'step is zero' ]
        False: [
            1 < 0
                ifTrue: [ "stepping down" ... ]
                False: [ "stepping up"
                    | i |
                    i: self.
                    [ i <= upperBound ] whileTrue: [
                        [ | :index | sum: sum + index ]
                            value: i.
                        i: i + 1. ] ] ].
    sum )
```

The next step in customization is to resolve the = in 1 = 0. Since 1 is a constant, the compiler can look up = in 1 to find the following method in the parent object for integers:

```
= x = ( _Integer_EQ_Primitive: x IfFail: [ "fail block" . . . ] )
```

This method simply calls the integer equality primitive (known to the compiler) and executes the fail block if the primitive fails (e.g., if x is not an integer).

So as it inlines this method the compiler derives

```
1 _Integer_EQ_Primitive: 0 IfFail: [ . . . ]
```

Since all the primitive's arguments are constants, and the primitive has no side-effects, the compiler evaluates the primitive at compile time, replacing the call with the **false** object:

```
sumTo: upperBound = (
    | sum <- 0 |
    false
        ifTrue: [ error: 'step is zero' ]
        False: [
            1 < 0
                ifTrue: [ "stepping down" ... ]
                False: [ "stepping up"
                    | i |
                    i: self.
                    [ i <= upperBound ] whileTrue: [
                        [ | :index | sum: sum + index ]
                            value: i.
                        i: i + 1. ] ] ].
    sum )
```

The next message that the compiler attacks is the first occurrence of ifTrue:False:. Since the **false** object is a constant object computed at compile-time, the compiler can look up ifTrue:False: in **false** to find:

```
ifTrue: trueBlock False: falseBlock = ( falseBlock value )
```

After inlining this occurrence of ifTrue:False: the code becomes:

```
sumTo: upperBound = (
    | sum <- 0 |
    [ 1 < 0
        ifTrue: [ "stepping down" ... ]
        False: [ "stepping up"
            | i |
            i: self.
            [ i <= upperBound ] whileTrue: [
                [ | :index | sum: sum + index ] value: i.
                i: i + 1. ] ] ] value.
    sum )
```

At this point the next message to be compiled is the outermost send of value, which is sent to a block literal. Once more the receiver is known at compile-time, so the compiler can find the value method (just the body of the block literal) and inline it:

```
sumTo: upperBound = (
    | sum <- 0 |
    1 < 0
        ifTrue: [ "stepping down" ... ]
        False: [ "stepping up"
            | i |
            i: self.
            [ i <= upperBound ] whileTrue: [
                [ | :index | sum: sum + index ] value: i.
                i: i + 1. ] ].
    sum )
```

The compiler proceeds by inlining 1 < 0, ifTrue:False:, and value (within the ifTrue:False: method) to reduce the code to:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
    [ i <= upperBound ] whileTrue: [
        [ | :index | sum: sum + index ] value: i.
        i: i + 1. ].
    sum )
```

The compiler allocates the i slot to a register, and generates the code to store the receiver (**self**) into that register. The compiler looks up whileTrue: relative to the block literal [i <= upperBound] and finds it in the parent object of all blocks:

```
whileTrue: block = (
    [
        value ifTrue: block False: [ ^ nil ]
    ] loop )
```

This method sends the loop message to the outer block, which evaluates the block over and over again. Each time the block is invoked, it first evaluates the loop condition (by sending value to **self**) and then either evaluates the loop body (the argument block) or exits out of the whileTrue: method (the circumflex (^) is the SELF non-local return operator).

Inlining whileTrue: produces (using goto statements and labels, in italics, to represent low-level control flow):

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
    [
        [ i <= upperBound ] value
            ifTrue: [
                [ | :index | sum: sum + index ] value: i.
                i: i + 1. ]
            False: [
                goto exit ].
    ] loop.
exit:
    sum )
```

Next, the compiler looks up loop, a simple method that evaluates its receiver and then invokes the special looping primitive to transfer control to the beginning of the method:

```
loop = ( value. _Restart. ).
```

After inlining this method, the value message, and the _Restart primitive, sumTo: becomes:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    [ i <= upperBound ] value
        ifTrue: [
            [ | :index | sum: sum + index ] value: i.
            i: i + 1. ]
        False: [
            goto exit ].
    goto loop
exit:
    sum )
```

The compiler inlines the value and value: messages sent to block literals to get to this simplified form:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    i <= upperBound ifTrue: [
        sum: sum + i.
        i: i + 1.
    ] False: [
        goto exit
    ].
    goto loop
exit:
    sum )
```

Next the compiler tries to inline the i <= upperBound expression. Since the initial SELF compiler doesn't know the types of the contents of any assignable slots, it doesn't know the type of the i slot and so cannot inline the <= message. Similarly, the compiler doesn't know the type of the result of the unknown <= method, so it can't inline the following ifTrue:False: message. The compiler (if using only customized compilation) is stuck here, and is forced to generate code to send the <= message followed by the ifTrue:False: message with two cloned block objects as arguments.

The compiler has used type information about the receiver and constants to inline to:Do:, to:By:Do:, whileTrue:, ifTrue:False:, loop, value, value:, =, and <, as well as all local slot accesses. All but two of the block literals in the methods have been inlined and so aren't cloned at run-time. Customized compilation, when coupled with compile-time lookup and general message inlining, can yield a significant gain in run-time performance. But even more can be done to inline messages in the common cases, such as the <= and ifTrue:False: messages above. The next two sections describe two general techniques, *message splitting* and *type prediction*, that together allow the compiler to eliminate these remaining message sends in the common case.

4.3. Message Splitting

As with most languages, a SELF program's flow of control may pass along one of a number of paths. For example, primitive operations may either succeed or fail at run time. After such a primitive, the control flow graph splits into two branches, one for the success case and one for the failure case. Additionally, comparison primitives (such as integer equality) have two success branches: one for success resulting in the **true** object, and one for success resulting in the **false** object.

Normally, the flow of control rejoins after the result of the primitive operation is computed, in anticipation of a message being sent to the result of the primitive. However, the type of the result of the primitive may be

different along the different branches, and this precious information will be lost if control rejoins immediately. For example, in our SELF system the type of the result of a primitive is known at compile-time in the success case(s), but is typically unknown in the failure case; after rejoining the flow of control, the least specific type must be used, which is typically the unknown type.

In situations where multiple control branches merge with different result types along the different branches, our compiler can *split* the following message, compiling a copy of the message send for each branch; control flow then merges after the split message. This operation can be visualized as *pushing* the message send through the merge point, producing a copy of the message on each incoming branch. The advantage of doing this message splitting is that now the compiler can take advantage of the more specific type information along each branch, customizing the message send for each specific case.

For example, consider the fragment

```
0 = i
    ifTrue: [ "true block" . . . ]
```

After inlining the = message down to the integer equality primitive operation, the compiler produces (using italicized Algol-like statements and operators to represent low-level operations):

```
if hasIntegerTag(i) then
    if 0 == i then
        temp := true
    else
        temp := false
else
    temp := . . . "failure block; unknown result type"
temp ifTrue: [ "true block" . . . ]
```

By splitting the ifTrue: message for both the **true**, the **false**, and the unknown cases, the compiler can produce:

```
if hasIntegerTag(i) then
    if 0 == i then
        true ifTrue: [ "true block" . . . ]
    else
        false ifTrue: [ "true block" . . . ]
else
    temp := . . . "failure block; unknown result type"
    temp ifTrue: [ "true block" . . . ]
```

By inlining the two sends of ifTrue: to compile-time constants, and then the resulting value messages, the compiler reaches the simple, low-level code:

```
if hasIntegerTag(i) then
    if 0 == i then
        "true block" . . .
    else
        "nothing"
else
    temp := . . . "failure block; unknown result type"
    temp ifTrue: [ "true block" . . . ]
```

In the final code, three separate copies of the ifTrue: message get compiled: one for the **true** case, one for the **false** case, and one for the failure case. In the two success cases, the ifTrue: message gets inlined, since the type of the receiver is known at compile-time; the failure case typically remains a full message send, since the receiver type is usually unknown along the failure branch. The compiler optimized out the generation of an explicit **true** or **false** object (since the ifTrue: methods didn't require the value of their receivers), instead using the flow of control to represent the boolean result value, just like a good C compiler would. Message splitting can be thought of as an extension to customized compilation, by customizing individual messages along particular control flow paths, with similar improvements in run-time performance.

4.4. Type Prediction

Customized compilation provides type information for any sends to **self**. Message splitting provides type information for the successful results of primitive operations. Unfortunately, the compiler still doesn't know the types of the receivers of many other messages. To compile some of these messages more efficiently, our compiler uses a *static type prediction* scheme, reminiscent of static branch prediction schemes, to generate extra compile-time type information.

In normal SELF environments, certain messages are more likely to be sent to some types of receivers than others. Measurements of Smalltalk-80 benchmarks show that operators like +, -, and < had integer arguments 90% of the time; messages like ifTrue: had boolean receivers 100% of the time [Un86]. Our compiler takes advantage of these usage patterns (which can be thought of as static profile data) by inserting fast run-time tests for the expected values or types of the receiver, and conditional branches to split the flow of control based on the result of the test; along the "success" branch the value or type of the receiver is known.

The compiler can split the original arithmetic or boolean message, compiling a copy on each branch. The message will get inlined along the success branch, but will remain a full message send along the failure branch. If the run-time type test is fast enough, and the test successful often enough, the optimization will pay big dividends in performance.

Let's return to the sumTo: example, and apply type prediction and message splitting. Using just customized compilation, the compiler had inlined the sumTo: invocation down to the following:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    i <= upperBound ifTrue: [
        sum: sum + i.
        i: i + 1.
    ] False: [
        goto exit
    ].
    goto loop
exit:
    sum )
```

Using type prediction the compiler guesses that the type of the receiver of the <= message (the local slot i) is likely to be an integer. The compiler inserts a run-time integer tag check of the value of i, and *splits* the <= message, with one copy for the integer case and one copy for the "otherwise" case (if i is a floating point number, for instance):

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    if hasIntegerTag(i) then
        "i is known to be an integer"
        temp := i <= upperBound
    else
        "i remains of unknown type"
        temp := i <= upperBound
    temp ifTrue: [
        sum: sum + i.
        i: i + 1.
    ] False: [
        goto exit
    ].
    goto loop
exit:
    sum )
```

Along the integer case, the compiler inlines the <= message for integers, which calls the integer less-than-or-equal-to primitive operation. This operation is, in turn, inlined. The primitive first checks that upperBound is also an integer; if this check fails, the failure branch is taken. Otherwise, the primitive does a simple compare and branch to either the **true** success branch or the **false** success branch. At this point, sumTo: has been compiled to:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    if hasIntegerTag(i) then
        if hasIntegerTag(upperBound) then
            if i <= upperBound then
                temp := true
            else
                temp := false
        else
            temp := . . . "failure block"
    else
        "i remains of unknown type"
        temp := i <= upperBound
    temp ifTrue: [
        sum: sum + i.
        i: i + 1.
    ] False: [
        goto exit
    ].
    goto loop
exit:
    sum )
```

The compiler now splits the ifTrue:False: message along the four previous branches, and inlines ifTrue:False: and then value for the **true** and **false** branches (omitting details in the two unknown cases):

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    if hasIntegerTag(i) then
        if hasIntegerTag(upperBound) then
            if i <= upperBound then
                sum: sum + i.
                i: i + 1.
            else
                goto exit
        else
            "failure block" ifTrue: [ ... ] False: [ ... ]
    else
        "i remains of unknown type"
        i <= upperBound ifTrue: [ ... ] False: [ ... ]
    goto loop
exit:
    sum )
```

The only expressions left to compile now are sum + i and i + 1. Again using type prediction, the compiler guesses that sum and i are integers, and conditionally compiles the + messages for the integer and non-integer branches, inlining the + message to the integer add primitive along the integer branches. These transformations lead the compiler to the following final form:

```
sumTo: upperBound = (
    | sum <- 0. i. |
    i: self.
loop:
    if hasIntegerTag(i) then
        if hasIntegerTag(upperBound) then
            if i <= upperBound then
                if hasIntegerTag(sum) then
                    if hasIntegerTag(i) then
                        sum := sum + i.
                        if overflow then
                            i := . . . "failure block"
                    else
                        i := . . . "failure block"
                else
                    "sum of unknown type"
                    sum: sum + i.
                if hasIntegerTag(i) then
                    i := i + 1.
                    if overflow then
                        i := . . . "failure block"
                else
                    "i of unknown type"
                    i: i + 1.
            else
                goto exit
        else
            "failure block" ifTrue: [ ... ] False: [ ... ]
    else
        "i of unknown type"
        i <= upperBound ifTrue: [ ... ] False: [ ... ]
    goto loop
exit:
    sum )
```

This is how our first-generation SELF compiler generates code for this simple example. It eliminates *every* message send inside the sumTo: method for the common case that our type prediction guesses are correct and no overflow occurs; these conditions are the *only* conditions under which the C version works. Our compiler avoids generating explicit **true** or **false** objects as the

results of primitive comparison operations, instead encoding the result in the subsequent flow of control. The inner loop still contains five run-time type tests (three as part of type prediction, two as part of the inlined primitive operations); a more sophisticated compiler could fold these tests into the two overflow tests on an architecture with hardware tag checking (such as the SPARC [Gar88]), completely eliminating the type-checking overhead in the common case. Our second-generation compiler, described in section 6, will eliminate the run-time type tests entirely.

Smalltalk-80 implementations use a technique similar to type prediction [GR83]. However, they *hard-wire* the definitions of certain common Smalltalk methods into the parser and compiler. The source code for the hard-wired methods is relegated to documentation, and any changes to that source code are ignored. In addition, for boolean messages like ifTrue:ifFalse:, the receiver must be **true** or **false** at run-time and the arguments must be block literals at parse-time; looping messages like whileTrue: have similar restrictions.

By building on the compile-time message lookup system and general message inliner, our SELF implementation achieves the same level of performance improvement without sacrificing any source-level compatibility; the SELF programmer may change the definition of the ifTrue: method or the integer + method at any time, and the system's behavior will immediately be updated. The type prediction optimization simply lets the compiler produce more efficient code for the common cases of certain messages, completely transparently to the SELF user.

5. Performance Comparison

SELF is implemented in 33,000 lines of C++ code and 1,000 lines of assembler, and runs on both the Sun-3 (a 68020-based machine) and the Sun-4 (a SPARC-based machine); we are working on an Apple Macintosh-II version. We have written almost 9,000 lines of SELF code, including a hierarchy of collection objects, a recursive descent parser, and the beginnings of a graphical user interface.

We compare the performance of our first-generation SELF implementation with a fast Smalltalk implementation and the standard Sun optimizing C compiler on a Sun-4/260 workstation. The fastest Smalltalk system currently available (excluding graphics performance) is the ParcPlace V2.4 β2 Smalltalk-80 virtual machine, rated at about 4 Dorados* [PP88]; this system includes the Deutsch-Schiffman techniques described earlier. We compare transliterations of the Stanford integer benchmarks [Hen88] from C into Smalltalk and SELF, the 1 sumTo: 10000 example, the Smalltalk-80 testActivationReturn micro-benchmark [McC83] (which tests the speed of procedure call and return and integer comparison and subtraction), and the Richards operating system simulation benchmark [Deu88]. We also rewrote some of the Stanford integer benchmarks in a more SELFish programming style; measurements for the rewritten benchmarks are presented in columns labeled SELF′, with the times in parentheses for the benchmarks that were not rewritten. The most representative rows of the table are probably the row for the median of the Stanford integer benchmarks and the row for the larger Richards benchmark.

The entries in the following table are the ratios of the running times of the benchmarks for the given pair of systems; we include the original raw running times in Appendix A. From our point of view, bigger numbers are better in the first two columns, while smaller numbers are better in the last two columns.

	Smalltalk/ SELF	Smalltalk/ SELF′	Smalltalk/ C	SELF/ C	SELF′/ C
perm	2.4	3.7	13.0	5.5	3.5
towers	2.4	3.8	11.2	4.7	2.9
queens	1.7	1.8	8.6	5.2	4.7
intmm	1.5	(1.5)	9.3	6.1	(6.1)
puzzle	3.1	(3.1)	21.4	6.9	(6.9)
quick	1.8	2.0	11.3	6.3	5.5
bubble	1.8	2.4	17.5	9.8	7.2
tree	1.0	1.2	2.1	2.1	1.8
min	**1.0**	**1.2**	**2.1**	**2.1**	**1.8**
median	**1.8**	**2.2**	**11.2**	**5.8**	**5.1**
max	**3.1**	**3.7**	**21.4**	**9.8**	**7.2**
sumTo:	1.4	(1.4)	6.2	4.5	(4.5)
activationRtrn	2.7	3.2	5.3	1.9	1.6
richards	2.8	(2.8)	10.6	3.8	(3.8)

Our SELF implementation outperforms the Smalltalk implementation on every benchmark; in many cases SELF runs more than twice as fast as Smalltalk. Not surprisingly, an optimizing C compiler does better than the SELF compiler. Some of the difference in performance may be attributed to the robust semantics of primitive operations in SELF: arithmetic operations always check for overflow, array accesses always check for indices out of bounds, method calls always check for stack overflow. Some of the difference is due to significantly poorer implementation in the SELF compiler of standard compiler techniques such as register allocation and peephole optimization. The rest of the difference is probably due to the lack of type information, especially for arguments, local slots, and the receiver's local slots. We are remedying these deficiencies to a large extent in the second-generation SELF system currently being implemented.

* A "Dorado" is a measure of the performance of Smalltalk implementations. One Dorado is defined as the performance of an early Smalltalk implementation in microcode on the 70ns Xerox Dorado [Deu83]; until recently it was the fastest available Smalltalk implementation.

6. Future Work

6.1. The Second-Generation SELF System

We are in the process of reimplementing our entire SELF system to clean up our code, simplify our design, and include better algorithms for data flow analysis, common subexpression elimination, global register allocation, and instruction scheduling. As of this writing (March 1989), we have completely rewritten the object storage system and the run-time/compile-time message lookup system. We have implemented the core of the second-generation compiler, and it is now working for very simple examples.

The new compiler performs type flow analysis to determine the types of many local slots at compile-time. It also includes a significantly more powerful message splitting system. The initial message splitter described in this paper only splits a message based on the type of the result of the previous message; the second-generation message splitting system can use any of the type information constructed during type flow analysis, especially the types of local slots. The type splitter may elect to split messages even when the message is not immediately after a merge point, splitting all messages that intervene between the merge that lost the type information and the message that wants to take advantage of the type information.

Our goal for the combined type analyzer and extended message splitter is to allow the compiler to split off entire sections of the control flow graph that correspond to the most common data types. Along these common-case sections, the types of all used variables will be known at compile-time, leading to maximally-inlined code with no run-time type checks; in the other sections, less type information is available to the compiler, and more full message sends will be generated. For most executions everything will run fast, possibly just as fast as for a C program; however, in exceptional cases, for example when an overflow actually occurs, the flow of control will transfer to a more general section.

This splitting design preserves the meaning of the program and the robust semantics of the primitives without penalizing the common cases. For example, in the sumTo: example, the type of the argument upperBound will be tested once at the head of the loop, and from then on, barring overflows, the types of sum, i, and upperBound will be known to be integers without run-time type checks. If an overflow occurs, control will transfer out of the highly optimized inner loop into a more general version that sends messages to perform its work.

Our second-generation compiler also performs data flow analysis, common subexpression elimination, code motion, global register allocation, and instruction scheduling. We hope that the addition of these optimizations will allow our new SELF compiler to compete with high-quality production optimizing compilers.

6.2. Open Issues

Method arguments are one of the largest sources of "unknown" type information in the current compiler. We want to extend our second-generation system to customize methods by the types of their arguments in addition to the receiver type. This extension provides the compiler with static type information about arguments, leading to better generated code. These benefits have to be balanced against the cost of verifying the types of arguments in the prologue of the method at run-time.

The compile-time lookup strategy works nicely as long as all the parents that get searched are constant parents; if any are assignable, then the compile-time lookup fails, and the message isn't inlined. Our second generation system provides limited support for dynamically-inherited methods by adding the types of any assignable parents traversed in the run-time lookup to the customization information about the method; the method prologue tests the values of the assignable parents in addition to the type of the receiver. We are investigating techniques that might lead us to an even faster implementation of dynamically-inherited methods.

The message inliner needs to make better decisions about when to inline a method, and when not to. The inliner should use information about the call site, such as whether it's in a loop or in a failure block, to help decide whether to inline the send, without wasting too much extra compile time and compiled-code space. It should also do a better job of deciding if a method is short enough to inline reasonably; counting the byte codes with a fixed cut-off value as it does now is not a very good algorithm. Also, our implementation of type prediction hard-wires both the message names and the predicted type; a more dynamic implementation that used dynamic profile information or analysis of the SELF inheritance hierarchy might produce better, more adapting results.

The current implementation of the compiler, though speedy by traditional batch optimizing compiler standards, is not yet fast enough for our interactive programming environment. The compiler takes over seven seconds to compile and optimize the Stanford integer benchmarks (almost 900 lines of SELF code), and almost three seconds to compile and optimize the Richards benchmark (over 400 lines of SELF code). We plan to experiment with strategies in which the compiler executes quickly with little optimization whenever the user is waiting for the compiler, queuing up background jobs to recompile unoptimized methods with full optimization later.

Work remains in making sure that our techniques are practical for larger systems than we have tested. To fully understand the contributions of our work, we need to analyze the relative performance gains and the associated space and time costs of our techniques. This analysis will be performed as part of the first author's forthcoming dissertation.

7. Related Work

Many techniques exist for efficiently compiling statically-typed procedure-oriented programming languages. These techniques include type flow analysis (both for type-checking purposes and to resolve overloaded built-in operators), data flow analysis, interprocedural analyses, procedure call inlining, common subexpression elimination, code motion, induction variable elimination, strength reduction, constant folding, dead code elimination, register allocation, peephole optimization, and instruction scheduling [ASU86].

Message passing can be implemented in statically-typed object-oriented languages by embedding a pointer to an array of function pointers in each object. Whenever a message is sent that requires a run-time dispatch (known as a *virtual function call* in C++), the compiler generates code to load the address of the function to call from the function pointer array associated with the receiver object; the static class hierarchy is used to assign a particular array index to each message name. This technique entails an overhead of 2 indirections over a direct procedure call; faster message dispatch code can be generated in some circumstances [Ros88]. However, this technique incurs a large hidden cost because none of these virtual function calls may be inlined; our techniques would help here to statically bind many virtual function calls at compile-time, speeding their calling sequences and making them candidates for inlining.

Some work has been done on compiling efficient code for dynamically-typed object-oriented languages, culminating with the Deutsch-Schiffman Smalltalk system [DS84]. As described before, their system pioneered *dynamic translation** and *inline caching*, as well as stack allocation of activation records and inlining calls to primitive operations.

Some researchers have required the Smalltalk programmer to add explicit type declarations to his programs to achieve better performance. Atkinson's Hurricane compiler [Atk86] used this explicit type information to statically bind and inline message sends and block evaluations. Running a small benchmark on a 68020-based Sun-3, Hurricane achieved a factor of two speedup over the Deutsch-Schiffman system. Unfortunately, as far as we know, Hurricane was never finished. Type declarations could not be statically verified, forcing the compiler to add run-time type tests to check the declarations. Incorrect type declarations could lead to erroneous results for non-idempotent methods. Our SELF compiler suffers from none of these problems and achieves the same speedup.

Johnson's TS Typed Smalltalk system also requires the addition of type declarations [Joh88]. Unlike Hurricane, TS type declarations are statically checkable, but all Smalltalk source code used by a program must be annotated with type declarations before the compiler will accept it; this has prevented Johnson from testing more than trivial examples. However, Johnson did add type declarations to the `sumTo:` benchmark and compile it in his system; on a roughly 2-MIP 68020-based machine, the example ran in 62ms [Joh88]. On a 7- to 8-MIP SPARC-based machine, our first-generation SELF system runs the same example *without* type declarations in only 18ms. Our SELF system appears to achieve performance comparable to these systems without requiring the programmer to add explicit type declarations to his source code.

The drawbacks of explicit type declarations could be avoided by inferring them from the program's source-code. Those who have attempted to perform type inferencing on Smalltalk programs [Suz81, BI82, Cur89] have had only limited success. Language features like user-defined control structures, initializing variables with **nil**, and special primitive operations (such as `perform:` and `become:`) interfere with inferring types. Some are confident that a new language could be designed from scratch to support effective type inferencing [Cur89]; however, the main benefit would be improved compile-time error detection since messages would remain dynamically bound. Type inferencing holds little promise for improving the performance of object-oriented dynamically-typed languages.

Other researchers have built systems for interpreted implementations that allow Smalltalk programmers to replace commonly-used Smalltalk methods with primitive methods written in a lower-level, non-object-oriented language [BMW86]. While speeding up the rewritten methods significantly over the original interpreted Smalltalk methods, this approach has the obvious disadvantage that the programmer has to explicitly rewrite those methods that need to execute faster. In addition, the low-level language is often overly restricted, preventing some methods from being rewritten at all.

* Deutsch and Schiffman use a different terminology than we do. What we call the parser they call the compiler (in the Smalltalk tradition), and what we call the compiler they call the translator.

8. Conclusions

We have doubled the performance of dynamically-typed object-oriented languages. Our system trades space for time by dynamically compiling and optimizing multiple copies of a single SELF method. Within each copy, customization, type prediction, and message splitting create and preserve static type information to drive compile-time lookup and aggressive message inlining, without sacrificing source-level compatibility. Our current system runs four to five times slower than optimized C. With better traditional optimization technology, global type analysis, and extensions to message splitting, our second-generation compiler may approach the performance of traditional optimizing compilers.

Once an implementation reaches the level of sophistication needed to achieve good performance, the information provided by variables and classes becomes redundant and unnecessary. Maps enable prototype-based languages to be as space-efficient as class-based ones, and provide a useful granularity of type information for the compiler. Customized method compilation combined with the compile-time lookup routine and the inlining facility serve to replace most message sends that access data slots with simple variable fetches and stores. The only novel feature of SELF that we have not yet implemented efficiently is dynamic inheritance, but class-based languages do not provide this capability at all.

The techniques presented in this paper directly apply to other dynamically-typed object-oriented languages, such as Smalltalk-80. These ideas could be applied to even statically-typed object-oriented languages such as C++ and Trellis/Owl as an extension of "copy-down"-style compilers. Whether an object-oriented language has type declarations or not, its programs suffer from many dispatched calls; our techniques provide opportunities to compile them out.

Since the first interpreter, programmers have been forced to choose between languages that cut programming time and languages that cut execution time. The benefits of an optimizing compiler have been denied those who opted for ease of programming. Now, the advent of larger main memories has opened up many new possibilities. This work shows that the right combination of compilation techniques can turn that extra memory into valuable type information, and so narrow the gap between productivity and performance.

Acknowledgments

SELF was initially developed by the second author and Randall B. Smith at Xerox PARC. The subsequent design evolution and implementation were undertaken beginning in mid-1987 by the SELF research group at Stanford University, composed of the authors and Elgin Lee; Elgin contributed significantly to the initial design and implementation of the memory system for SELF [Lee88]. We thank Bay-Wei Chang for his detailed comments that improved the presentation of this paper. We also thank Peter Deutsch for many instructive discussions and seminal ideas for the design and implementation of SELF.

Appendix A: Raw Running Times

The following table presents the actual running times of the benchmarks measured in the Performance section. All times are in milliseconds of CPU time, except for the Smalltalk times, which are in milliseconds of real time; the real time measurements for the SELF system and the compiled C program are practically identical to the CPU time numbers, so comparisons in measured performance between the ParcPlace Smalltalk system and the other two systems are valid.

	Smalltalk (real ms)	SELF (cpu ms)	SELF′ (cpu ms)	C (cpu ms)
perm	1559	660	420	120
towers	2130	900	560	190
queens	859	520	470	100
intmm	1490	970	---	160
puzzle	16510	5290	---	770
quick	1239	690	610	110
bubble	2970	1660	1230	170
tree	1760	1750	1480	820
sumTo:	25	18	---	4
activationReturn	169	62	52	32
richards	7740	2760	---	730

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Atk86] Robert G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 151-158, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[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.

[BMW86] Mark B. Ballard, David Maier, and Allen Wirfs-Brock. QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods. In *OOPSLA '86 Conference Proceedings*, pp. 140-150, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Bob88] D. G. Bobrow *et al.* Common Lisp Object System Specification X3J13 Document 88-002R. In *SIGPLAN Notices 23(9)*, September, 1988.

[Bor86] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pp. 36-40, Dallas, TX, 1986.

[Cur89] Pavel Curtis. Type inferencing in Smalltalk. Personal communication, March, 1989.

[Deu83] L. Peter Deutsch. The Dorado Smalltalk-80 Implementation. Hardware Architecture's Impact on Software Architecture. In [Kra83], pp. 113-126.

[Deu88] L. Peter Deutsch. Richards benchmark. Personal communication, October, 1988.

[DS84] 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.

[Gar88] Robert B. Garner *et al.* The Scalable Processor Architecture (SPARC). In *Proceedings of the Thirty-Second IEEE Computer Society International Conference (Spring CompCon)*, pp. 278-283, San Francisco, CA, 1988.

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

[Hen88] John Hennessy. Stanford integer benchmarks. Personal communication, June, 1988.

[Joh88] 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, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

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

[Kra86] David Kranz *et al.* ORBIT: An Optimizing Compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, pp. 219-233, Palo Alto, CA, 1986. Published as *SIGPLAN Notices 21(7)*, July, 1986.

[Lee88] Elgin Lee. *Object Storage and Inheritance for* SELF, *a Prototype-Based Object-Oriented Programming Language.* Engineer's thesis, Stanford University, 1988.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*, pp. 214-223, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[LTP86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 322-330, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[McC83] Kim McCall. The Smalltalk-80 Benchmarks. In [Kra83], pp. 153-174.

[Mey86] Bertrand Meyer. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*, pp. 391-405, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Moo86] David A. Moon. Object-Oriented Programming with *Flavors*. In *OOPSLA '86 Conference Proceedings*, pp. 391-405, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[PP88] ParcPlace Newsletter, Winter 1988, Vol. 1, No. 2. ParcPlace Systems, Palo Alto, CA, 1988.

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

[Sch86] Craig Schaffert *et al.* An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Ste76] Guy Lewis Steele Jr. LAMBDA: The Ultimate Declarative. AI Memo 379, MIT Artificial Intelligence Laboratory, November, 1976.

[SS76] Guy Lewis Steele Jr. and Gerald Jay Sussman. LAMBDA: The Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory, March, 1976.

[Ste87] Lynn Andrea Stein. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*, pp. 138-146, Orlando, FL, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[Suz81] N. Suzuki. Inferring Types in Smalltalk. In *8th Annual ACM Symposium on Principles of Programming Languages*, pp. 187-199, 1981.

[Ung86] David Michael Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* Ph.D. dissertation, the University of California at Berkeley, February, 1986. Published by the MIT Press, Cambridge, MA, 1987.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87 Conference Proceedings*, pp. 168-182, Orlando, FL, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987.