To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991
© 1991 Kluwer Academic Publishers - Manufactured in The Netherlands

An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*

CRAIG CHAMBERS (*craig@self.stanford.edu*)
DAVID UNGAR† (*ungar@self.stanford.edu*)
ELGIN LEE‡ (*ehl@parcplace.com*)

Computer Systems Laboratory, Stanford University, Stanford, California 94305

Abstract. We have developed and implemented techniques that double the performance of dynamically-typed object-oriented languages. Our SELF implementation runs twice as fast as the fastest Smalltalk implementation, despite SELF's lack of classes and explicit variables.

To compensate for the absence of classes, our system uses implementation-level *maps* to transparently group objects cloned from the same prototype, providing data type information and eliminating the apparent space overhead for prototype-based systems. To compensate for dynamic typing, user-defined control structures, and the lack of explicit variables, our system dynamically compiles *multiple versions* of a source method, each *customized* according to its receiver's map. Within each version the type of the receiver is fixed, and thus the compiler can statically bind and *inline* all messages sent to **`self`**. *Message splitting* and *type prediction* extract and preserve even more static type information, allowing the compiler to inline many other messages. Inlining dramatically improves performance and eliminates the need to hard-wire low-level methods such as **`+`**, **`==`**, and **`ifTrue:`**.

Despite inlining and other optimizations, our system still supports interactive programming environments. The system traverses internal dependency lists to invalidate all compiled methods affected by a programming change. The debugger reconstructs inlined stack frames from compiler-generated debugging information, making inlining invisible to the SELF programmer.

---

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

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

‡Author's present address: ParcPlace Systems, 1550 Plymouth Street, Mountain View, CA 94043.

This paper was originally published in *OOPSLA '89 Conference Proceedings* (*SIGPLAN Notices*, 25, 10 (1989) 49-70).

1 Introduction

SELF [32] is a dynamically-typed object-oriented language inspired by the Smalltalk-80[1] language [12]. Like Smalltalk, SELF has no type declarations, allowing programmers to rapidly build and modify systems without interference from out-of-date type declarations. Also, SELF provides *blocks* (lexically-scoped function objects akin to closures [24, 25]) so that SELF programmers may define their own control structures; even the standard control structures for iteration and boolean selection are constructed out of blocks. However, unlike Smalltalk and most other object-oriented languages, SELF has no classes.[2] Instead it is based on the *prototype object model*, in which each object defines its own object-specific behavior, and inherits shared behavior from its parent objects. Also unlike Smalltalk, SELF accesses state solely by sending messages; there is no special 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.

In a straightforward implementation, SELF's prototype-based model would consume much more storage space than other dynamically-typed object-oriented programming languages, and its reliance on message passing to access state would exact an even higher penalty in execution time. We have developed and implemented techniques that eliminate the space and time costs of these features. In addition, we have implemented other optimizations that enable SELF to run twice as fast as the fastest Smalltalk system. These same techniques could improve implementations of class-based object-oriented languages such as Smalltalk, Flavors [20], CLOS [3], C++ [27], Trellis/Owl [23], and Eiffel [19].

This paper describes our implementation for SELF, which has been running for over a year. First we review SELF's object and execution model in section 2. Then we describe SELF's object storage system in section 3, introducing *maps* and *segregation* and presenting object formats. Section 4 explains our byte-coded representation for source code. Section 5 reviews the compiler techniques, originally published in [6]. Section 6 explains how these optimizations can coexist with an exploratory programming environment that supports incremental recompilation and source-level debugging. Section 7 compares the performance of SELF to the fastest available Smalltalk system and an optimizing C compiler. It also proposes a new performance metric, *MiMS*, for object-oriented language implementations. We conclude with a discussion of open issues and future work.

[1] 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.

[2] To illustrate how unusual this is, note that some well-respected authorities have gone so far as to require that "object-oriented" languages provide classes [32]. Other prototype models are discussed in [4, 15, 17, 26].

2 Overview of SELF

SELF was initially designed 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 authors at Stanford University.

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 asterisks to their names). 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).

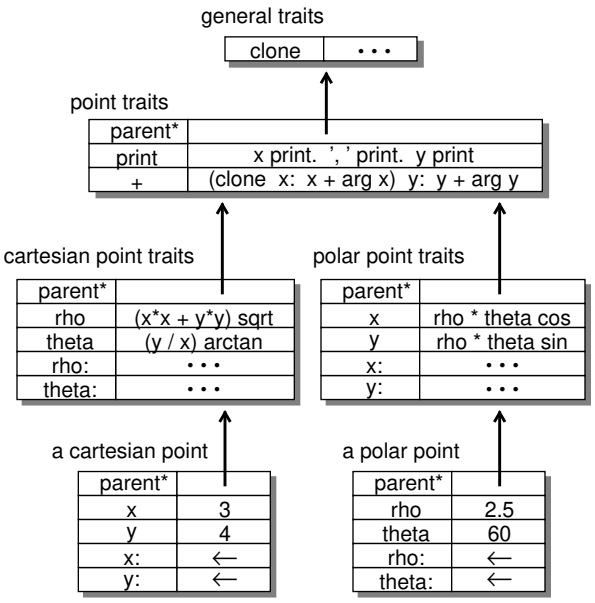
When a message is sent to an object (called the *receiver* of the message), the object is searched for a slot with the same name as the message. If a matching slot is not found, then the contents of the object's parent slots are searched recursively, using SELF's multiple inheritance rules to disambiguate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

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 method 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 its code. The **`self`** slot is a parent slot so that the cloned activation record inherits from the receiver of the message send.

For instance, in the point example shown on the next page, sending the **`x`** message to the cartesian 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, finding 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. The method would get cloned and executed, producing the floating point result **`1.25`**.

If the **`print`** message were sent to a point object, the **`print`** slot defined in the point traits object would be found. The method contained in the slot prints out the point object in cartesian coordinates. If the point were represented using cartesian coordinates, the **`x`** and **`y`** messages would access the corresponding data slots of the point object. But the **`print`** method works fine even for points represented using polar coordinates: the **`x`** and **`y`** messages would find the conversion methods defined in the polar point traits object to 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. 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 no



Six SELF objects. The bottom objects are two-dimensional point objects, the left one using cartesian coordinates and the right one using polar coordinates. The ← represents the assignment primitive operation, which is invoked to modify the contents of corresponding data slots. The cartesian point traits object is the immediate parent object shared by all cartesian point objects, and defines four methods for interpreting cartesian points in terms of polar coordinates; the polar point traits object does the same for polar point objects. The point traits 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, which defines even more general behavior, such as how to copy objects.

---

corresponding assignment slot is called a *constant* or *read-only slot*, since a running program cannot change its value. For example, most parent slots are constant slots. However, our object model allows a parent slot to be assignable just like any other slot, simply by defining its corresponding assignment slot. Such an assignable parent slot permits an object's inheritance to change on-the-fly, perhaps as a result of a change in the object's state. For example, a collection object may wish to provide different behavior depending on whether the collection is empty or not. This *dynamic inheritance* is one of SELF's linguistic innovations, and has proven 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 special 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.

Two other kinds of objects appear in SELF: object arrays and byte arrays. Arrays contain only a single parent slot pointing to the parent object for that kind of array, but contain a variable number of element objects. As their names suggest, object arrays contain elements that are arbitrary objects, while byte arrays contain only integer objects in the range 0 to 255, but in a more compact form. Primitive operations support fetching and storing elements of arrays as well as determining the size of an array and cloning a new array of a particular size.

The SELF language described here is both simple and powerful, but resists efficient implementation. SELF's prototype object model, in which each object can have unique format and behavior, poses serious challenges for the economical storage of objects. SELF's exclusion of type declarations and commitment to message passing for all computation—even for control structures and variable accesses—defeats existing compiler technology. The remainder of this paper describes our responses to these challenges.

3 The Object Storage System

The object storage system (also referred to as the *memory system*) must represent the objects of the SELF user's world, including references between objects. It creates new objects and reclaims the resources consumed by inaccessible objects. An ideal memory system would squeeze as many objects into as little memory as possible, for high performance at low cost. An earlier version of our SELF memory system was documented in [16].

Much of our memory system design exploits technology proven in existing high-performance Smalltalk systems. For minimal overhead in the common case, our SELF system represents object references using direct tagged pointers, rather than indirectly through an object table. Allocation and garbage collection in our SELF system uses Generation Scavenging with demographic feedback-mediated tenuring [29, 30], augmented with a traditional mark-and-sweep collector to reclaim tenured garbage. The following two subsections describe our new techniques for efficient object storage systems; the third subsection describes our object formats in detail.

3.1 Maps

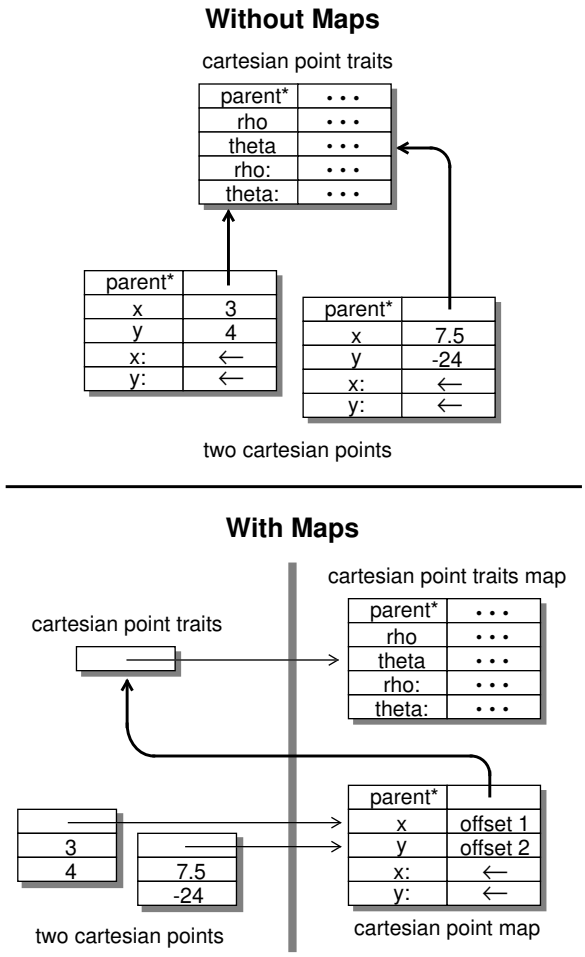
A naive implementation of SELF's prototype object model would waste space. If SELF were based on classes, the class objects would contain the format (names and locations of the instance variables), methods, and superclass information for all their instances; the instances would contain only the values of their instance variables and a pointer to the shared class object. Since SELF uses the prototype model, each object must define its own format, behavior, and inheritance, and presumably an implementation would have to represent both the class-like format, method, and inheritance information and the instance-like state information in *every* SELF object.

Luckily, we can regain the storage efficiency of classes even in SELF's prototype object model. Few SELF objects have totally unique format and behavior. Almost all objects are created by cloning some other object and then modifying the values of the assignable slots. Wholesale changes in the format or inheritance of an object, such as those induced by the programmer, can only be accomplished by invoking special primitives. We say that a prototype and the objects cloned from it, identical in every way except for the values of their assignable slots, form a *clone family.*

We have invented *maps* as an implementation technique to efficiently represent members of a clone family. In our SELF object storage system, objects are represented by the values of their assignable slots, if any, and a pointer to the object's map; the map is shared by all members of the same clone family. 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). If the object has code (i.e., is a method), the map stores a pointer to a SELF byte code object representing the source code of the method (byte code objects are described in section 4).

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, starting a new 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. In addition, the map of an object conveys its static properties to the SELF compiler. Section 5 explains how the compiler can exploit this information to optimize SELF code.



An example of the representations for two cartesian points and their parent. Without maps, each slot would require at least two words: one for its name and another for its contents. This means that each point would occupy at least 10 words. With maps, each point object only needs to store the contents of its assignable slots, plus one more word to point to the map. All constant slots and all format information are factored out into the map. Maps reduce the 10 words per point to 3 words. Since the cartesian point traits object has no assignable slots, all of its data are kept in its map.

3.2 Segregation

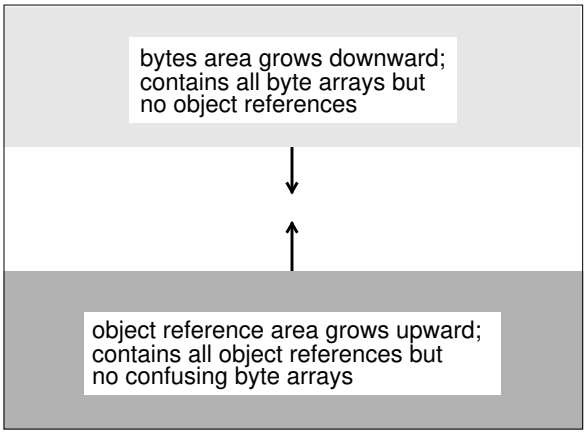
A common operation of the memory system is to scan all object references for those that meet some criterion:

- The scavenger scans all objects for references to objects in from-space.
- The reflective object modification and programming primitives have to redirect all references to an object if its size changes and it has to be moved.
- The browser may want to scan all objects for those that contain a reference to a particular object that interests the SELF user.

To support these and other functions, our SELF implementation has been designed for rapid scanning of object references.

Since the elements of byte arrays are represented using packed bytes rather than tagged words, byte array elements may masquerade as object references. Smalltalk systems typically handle this problem by scanning the heap object-by-object rather than word-by-word. For each object, the system checks to see whether the object contains object references or only bytes. Only if the object contains object references does the system scan the object for matching references, iterating up to the length of the object. Then the scanner proceeds to the next object. This procedure avoids the problems caused by scanning byte arrays, but slows down the scan with the overhead to parse object headers and compute object lengths.

In our SELF system, we avoid the problems associated with scanning byte arrays without degrading the object reference scanning speed by *segregating* the byte arrays from the other SELF objects. Each Generation Scavenging memory space is divided into two areas, one for byte arrays and one for objects with references. To scan all object references, only the object reference area of each space needs to be scanned. This optimization speeds scans in two ways: byte array objects are never scanned, and object headers are never parsed.



A SELF Memory Space

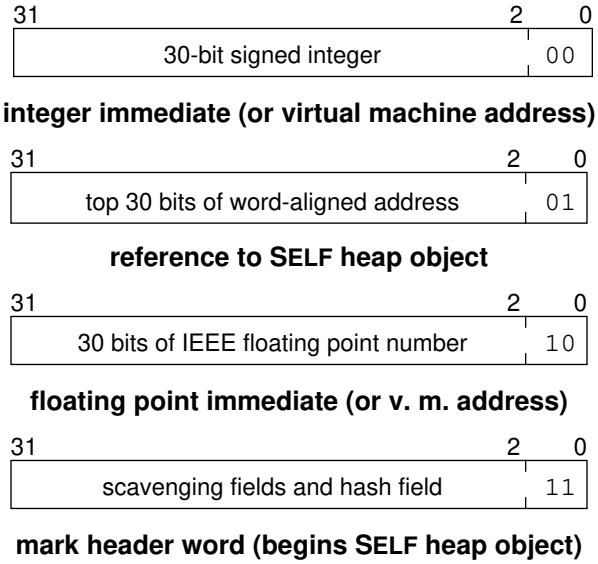
To avoid slowing the tight scanning loop with an explicit end-of-space check, the word after the end of the space is temporarily replaced with a *sentinel* reference that matches the scanning criterion. The scanner checks for the end of the space only on a matching reference, instead of on every word. Early measurements on 68020-based Sun-3/50's showed that our SELF system scanned memory at the rate of approximately 3 megabytes per second. Measurements of the fastest Smalltalk-80 implementation on the same machine indicated a scanning speed for non-segregated memory spaces of only 1.6 megabytes per second.

For some kinds of scans, such as finding all objects that refer to a particular object, the scanner needs to find the objects that *contain* a matching reference, rather than the reference itself. Our system can perform these types of searches nearly as fast as a normal scan. We use a special tag for the first header word of every object (called the *mark* word) to identify the beginning of the object. The scanner proceeds normally, searching for matching references. Once a reference is found, the object containing the reference can be found by simply scanning backwards to the object's mark word, and then converting the mark's address into an object reference.

3.3 Object Formats

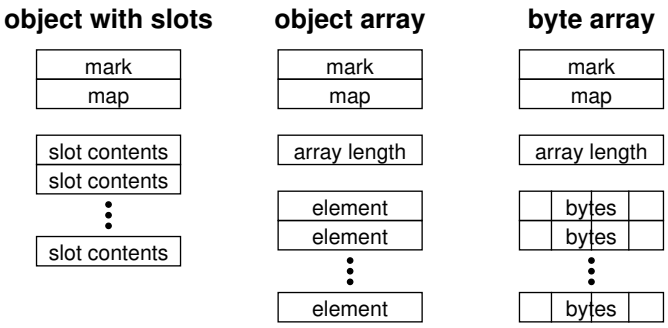
A SELF memory space is organized as a linear array of aligned 32-bit words. Each word contains a low-order 2-bit tag field, used to interpret the remaining 30 bits of information. A reference to an integer or floating point number encodes the number directly in the reference itself. Converting between a tagged integer immediate and its corresponding hardware representation requires only a shift instruction. Adding, subtracting, and comparing tagged integers require no conversion at all. References to other SELF objects and references to map objects embed the address of the object in the reference (remember that there is no object table). The remaining tag format is used to mark the first header word of each object, as required by the scanning scheme discussed in the previous subsection. Pointers to virtual machine functions and other objects not in the SELF heap are represented using raw machine addresses; since their addresses are at least 16-bit half-word aligned the scavenger will interpret them as immediates and won't try to relocate them.

Each object begins with two header words. The first word is the mark word, marking the beginning of the object. The mark contains several bitfields used by the scavenger and an immutable bitfield used by the SELF **`hash`** primitive. The second word is a tagged reference to the object's map. A SELF object with assignable slots contains additional words to represent their contents. An array object contains its length (tagged as a SELF integer to prevent interactions with scavenging and scanning) and its elements (either 32-bit tagged object references or 8-bit untagged bytes, padded out to the nearest 32-bit boundary).

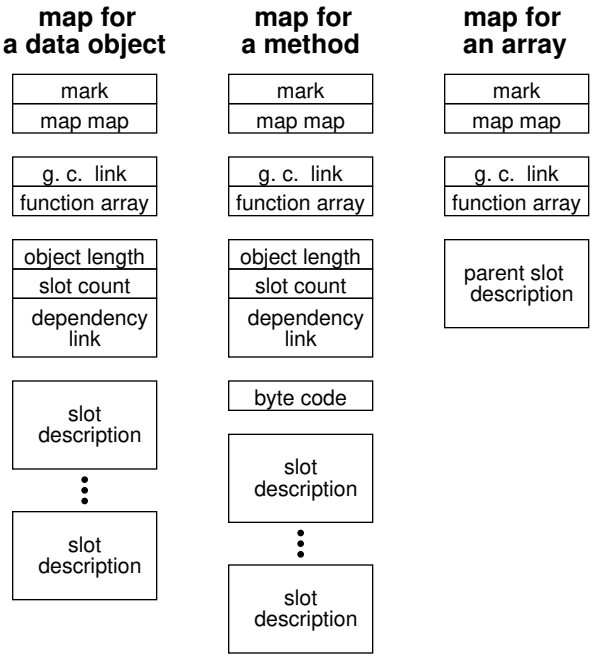


The representation of a map is similar. Map objects begin with mark and map words. All map objects share the same map, called the "map map." The map map is its own map. All maps in new-space are linked together by their third words; after a scavenge the system traverses this list to finalize inaccessible maps. The fourth word of a map contains the virtual machine address of an array of function pointers;[3] these functions perform format-dependent operations on objects or their maps.

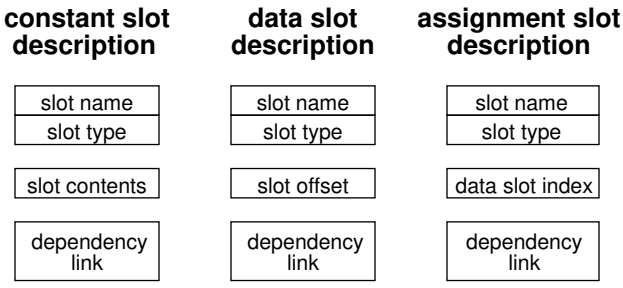
For maps of objects with slots, the fifth word specifies the size of the object in words. The sixth word indicates the number of slots in the object. The next two words contain a change dependency link for the map, described in section 6.1. These four words are tagged as integers. If the map is for a method, the ninth word references the byte code object representing the method's source code.



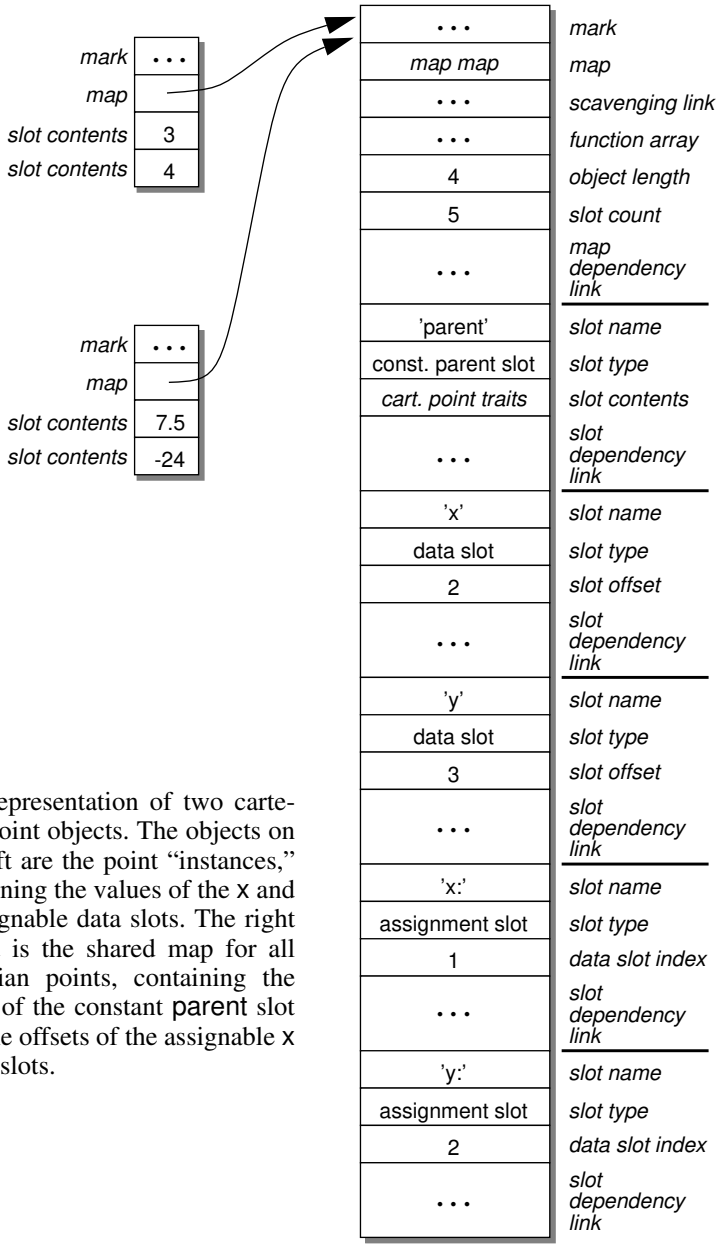
[3]This function pointer array is exactly the virtual function array generated by the C++ compiler.



Finally, the map includes a five-word description for each of the object's slots. The first word points to the SELF string object representing the name of the slot; the next word describes both the type of the slot (either constant data slot, assignable data slot, or assignment slot) and whether the slot is a parent slot.[4] The third word of a slot description contains either the contents of the slot (if it's a constant slot), the offset within the object of the contents of the slot (if it's an assignable data slot), or the index of the corresponding data slot (if it's an assignment slot). The last two words of each slot contain a change dependency link for that slot, described in section 6.1.



[4]In SELF parents are prioritized; the priority of a parent slot is stored in the second word of the slot description.



The representation of two cartesian point objects. The objects on the left are the point "instances," containing the values of the x and y assignable data slots. The right object is the shared map for all cartesian points, containing the value of the constant parent slot and the offsets of the assignable x and y slots.

From the above object formats, we can determine that the total space cost to represent a clone family of n objects (each with s slots, a of which are assignable) is $(2 + a)n + 5s + 8$ words. For the simple cartesian point example, s is 5 (**x**, **x:**, **y**, **y:**, and **parent**) and a is 2 (**x** and **y**), leading to a total space cost to represent all point objects of $4n + 33$ words. Published accounts of Smalltalk-80 systems [11, 29] indicate that these systems use at least two extra words per object: one for its class pointer and another for either its address or its hash code and flags. Therefore, maps allow objects in a prototype-based system like SELF to be represented just as space-efficiently as objects in a class-based system like Smalltalk.

4 The Parser

To minimize parsing overhead, textual SELF programs are parsed once when entered into the system, generating SELF-level *byte code* objects, much like Smalltalk **CompiledMethod** instances. Each method object represents its source code by storing a reference to the pre-parsed byte code object in the method's map; all cloned invocations of the method thus share the same byte code object. A byte code object contains a byte array holding the byte codes for the source, and an object array holding the message names and object literals used in the source. Each byte code in the byte array represents a single byte-sized virtual machine instruction, and is divided into two parts: a 3-bit opcode and a 5-bit object array index. The opcodes are specified as if for execution by a stack-oriented interpreter; in actuality, our SELF compiler dynamically translates byte code objects into native machine instructions just prior to execution. The only opcodes used to represent SELF programs are the following:

SELF
: push **self** onto the execution stack

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

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

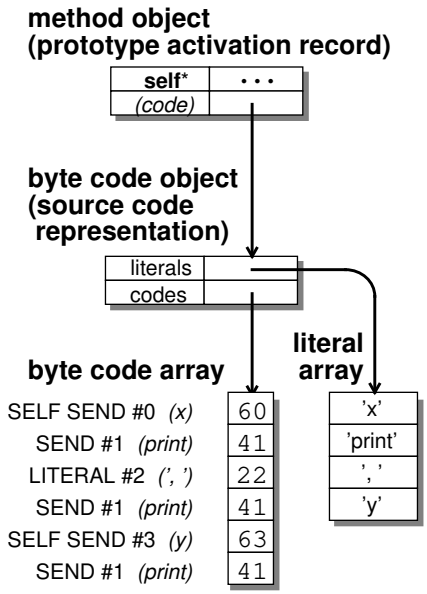
SELF SEND <message name index>
: send a message to **self**, popping the arguments off the execution stack and pushing the result

SUPER SEND <message name index>
: send a message to **self**, delegated to all parents, popping the arguments off the execution stack and pushing the result

DELEGATEE <parent name index>
: delegate the next message send to the named parent

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

INDEX-EXTENSION <index extension>
: extend the next index by prepending the index extension



The representation of the point **`print`** method. The top object is the prototype activation record, containing placeholders for the local slots of the method (in this case, just the **`self`** slot) plus a reference to the byte code object representing the source code (actually stored in the method's map). The byte code object contains a byte array for the byte codes themselves, and a separate object array for the constants and message names used in the source code.

The index for the opcodes is an index into the accompanying object array. The 5-bit offset allows the first 32 message names and literals to be referred to directly; indices larger than 32 are constructed using extra INDEX-EXTENSION instructions.

In SELF source code, primitive operations are invoked with the same syntax used to send a message, except that the message name begins with an underscore ("_"). Every call of a primitive operation may optionally pass in a block to be invoked if the primitive fails by appending **`IfFail:`** to the message name. If invoked, the block is passed an error code identifying the nature of the failure (e.g. overflow, divide by zero, or incorrect argument type). The normal SEND byte codes are used to represent all primitive operation invocations, simplifying the byte codes and facilitating extensions to the set of available primitive operations. By contrast, Smalltalk-80 primitives are invoked by number rather than name, and may only be called at the beginning of a method. The rest of the method is executed if the primitive fails, without any indication of why the primitive failed.

The byte codes needed to express SELF programs fall into only three classes: base values (LITERAL and SELF), message sends, and non-local return. This small number results from both the simplicity and elegance of the SELF language and the lack of elaborate space-saving encodings. Smalltalk-80 defines a much larger set of byte codes [12], tuned to minimize space and maximize interpretation speed, and includes byte codes to fetch and store local, instance, class, pool, and global variables and shortcut byte codes for common-case operations such as loading constants like **`nil`**, **`true`**, and **`0`**.

Smalltalk-80 systems also use special control flow byte codes to implement common boolean messages like **`ifTrue:ifFalse:`** and **`whileTrue:`**; the Smalltalk parser translates the message sends into conditional and unconditional branch byte codes, open-coding the argument blocks. Similarly, the **`==`** message is automatically translated into the identity comparison primitive operation byte code. A similar optimization is included for messages like **`+`** and **`<`**, which the parser translates into special byte codes. When executed, these byte codes either directly invoke the corresponding integer primitive operation (if the receiver is an integer), or perform the message send (if the receiver isn't an integer).

Although this special processing for common messages may significantly improve the performance of existing Smalltalk systems, especially interpreted ones, they violate the extensible and flexible spirit of Smalltalk:

- The source code for the hard-wired methods is relegated to documentation, and all changes to the hard-wired source code are ignored by the system. Any definitions of **`ifTrue:ifFalse:`**, **`whileTrue:`**, and **`==`** for other types of objects are ignored.
- The receiver of an **`ifTrue:ifFalse:`** message must evaluate to either the **`true`** or the **`false`** object at run-time and the arguments must be block literals at parse-time; the receiver and argument to **`whileTrue:`** must be block literals at parse-time, and the receiver block must evaluate to either the **`true`** or the **`false`** object at run-time.
- Perhaps the worst aspect of these parser "optimizations" is that they tempt programmers to select inappropriate control structures like **`whileTrue:`** instead of **`to:do:`** to obtain the performance of the hard-wired message.

In effect, these hard-wired messages have become the non-object-oriented built-in operators of Smalltalk. Our SELF system incorporates none of these tricks. Instead our compilation techniques achieve better performance without compromising the language's conceptual simplicity and elegance, preserving the message passing model for *all* messages.

5 The Compiler

The SELF compiler is a significant part of our efficient implementation [6]. It is similar to the Deutsch-Schiffman translator described in [11] (and implemented in the ParcPlace Smalltalk-80 system) in that it supports *dynamic translation* of byte-coded methods into machine code transparently on demand at run-time, and it uses an *inline caching* technique to reduce the cost of non-polymorphic message sends. However, although the Deutsch-Schiffman system is the fastest Smalltalk system (as of July 1989), it still runs about 10 times slower than optimized C. By combining traditional optimizing compiler technology, techniques from high-performance Smalltalk systems, and some critical new techniques we developed, our SELF compiler has already achieved a level of performance more than twice as fast as the Deutsch-Schiffman system, and only 4 to 5 times slower than optimized C. We hope that our second-generation system under construction (and described in section 7) will achieve even better levels of performance.

The main obstacle to generating efficient code from Smalltalk programs, as many people have noted before [1, 2, 14], is that very little static type information is available in the Smalltalk source. Only literal constants have a known class at compile-time; without detailed analysis, no other types are known. Type inferencing is difficult for Smalltalk programs, especially when the compiler is using the inferred types to improve performance [5, 8, 28]. Even if the Smalltalk programmer were willing to sacrifice many of the benefits of his exploratory programming environment and annotate his programs with static type declarations, designing an adequate type system for Smalltalk would be hard [1, 14]; the more flexible the type system, the smaller the performance improvement possible and the smaller the reward for including type declarations in the first place.

SELF programs are even harder to compile efficiently than Smalltalk programs. All the problems of missing static type information that Smalltalk compilers face are also faced by our SELF compiler. In addition, all variables in SELF are accessed by sending messages, rather than being explicitly identified as variables in the source code and byte codes. And since there are no classes in SELF, some of the class-based techniques used to optimize Smalltalk programs, such as inline caching, type inferencing, and static type checking, cannot be directly used in our SELF system.

Rather than compromising the flexibility of SELF programs with a static type system, or compromising the execution speed of programs by interpreting dynamic type information, we have developed compilation techniques that automatically derive much of the type information statically specified in other type systems. By combining this extra information with a few general-purpose techniques from optimizing compilers for traditional languages like Fortran and C, our compiler achieves good performance without sacrificing any of the comforts of an interactive, exploratory programming environment: fast turnaround for programming

changes, complete source-level debugging, and a simple, elegant programming language unfettered by static type declarations. The next few subsections summarize our new compilation techniques; a more detailed discussion may be found in [6, 7].

5.1 Customized Compilation

The Deutsch-Schiffman Smalltalk-80 system compiles a single machine code method for a given source code method. Since many classes may inherit the same method the Smalltalk-80 compiler cannot know the exact class of the receiver. Our SELF 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 our SELF compiler can know the type of the receiver of the message at compile-time, and can generate much better code for each of the specific versions of a method than it could for a single general-purpose compiled method. We call this technique of dynamic translation of multiple specially-compiled methods for a single source-code method *customized compilation*.

Consider the **`min:`** method defined for all objects:

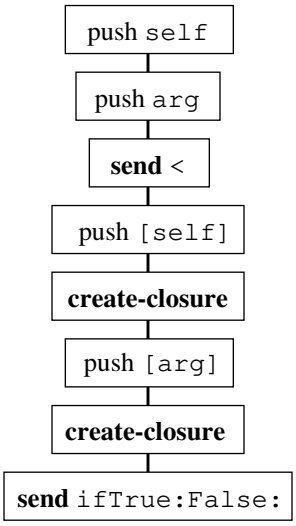
```
min: arg = ( < arg ifTrue: [self] False: [arg] ).
```

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, our SELF system waits until the **`min:`** method is first invoked before compiling any code for this method. Other systems would compile this method once for all receiver and argument types, which would require generating the code for a full message dispatch to select the right **`<`** comparison routine. Since our SELF compiler generates a separate compiled 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.

Let's trace the operations of our SELF compiler to evaluate the expression **`i min: j`**, where **`i`** contains an integer at run-time. Assuming this is the first time **`min:`** has been sent to an integer, our compiler will generate code for a version of **`min:`** that is customized for integer receivers. The compiler first builds the internal flow graph pictured at the top of the next page (expensive operations are in bold face).[5]

Many of the expensive operations can be eliminated by *inlining* messages sent to receivers of known type, as described next.

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



5.2 Message Inlining

Our compiler uses sources of type information, such as the types of source-code literals and the type of **self** gleaned from customized compilation, to perform *compile-time message lookup* and *message inlining*. If the type of the receiver of a message is known at compile-time, the compiler can perform the message lookup at compile-time rather than wait until run-time. If this lookup is successful, which it will be in the absence of dynamic inheritance and programming errors, our compiler will do one of the following:

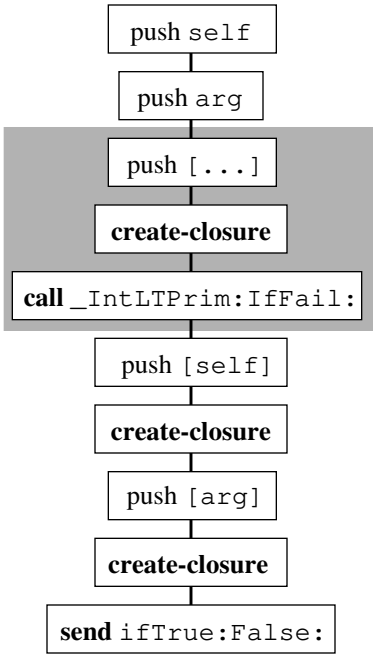
- If the slot contains a method, the compiler will *inline* the body of the method at the call site, if the method is short enough and nonrecursive.
- If the slot contains a block **value** method, the compiler will inline the body of the block **value** method at the call site, if it is short enough. If after inlining there are no remaining uses of the block object, the compiler will eliminate the code to create the block at run-time.
- If the slot is a constant data slot, the compiler will 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 will 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 will replace the message send with code to update the contents of the slot (e.g. a store instruction).

After inlining all messages sent to receivers of known type, the compiler will have inlined all messages that in an equivalent Smalltalk program would have been variable references or assignments, thus eliminating the overhead in SELF of using message passing to access variables. In addition, many more messages have been inlined that in a Smalltalk system would have remained full message sends.

For example, in the version of **`min:`** customized for integers, the compiler can statically look up the definition of **<** defined for integers:

```
< arg = ( _IntLTPrim: arg IfFail: [...] ).
```

This method simply calls the integer less-than primitive with a failure block (omitted here for brevity). The compiler inlines this **<** method to get to the flow graph below:



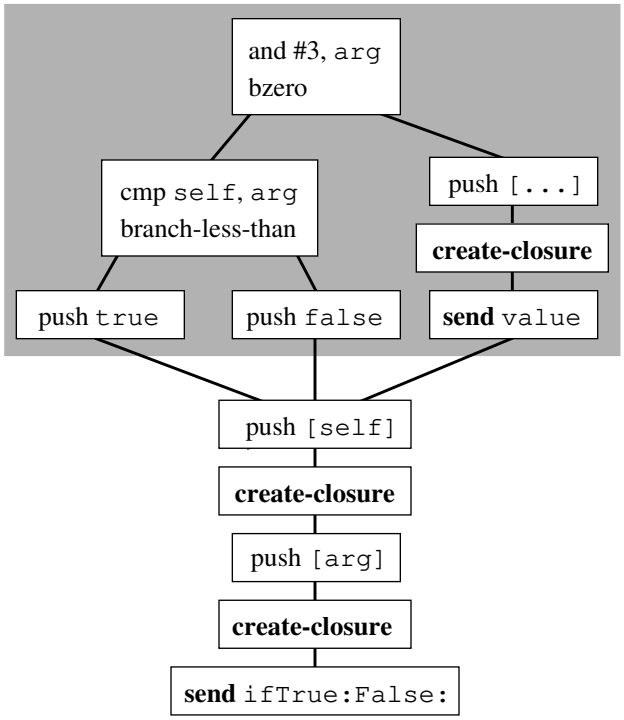
The overhead for sending the **<** message has been eliminated, but calling a procedure to compare integers is still expensive. The next section explains how our compiler open-codes common primitive built-in operations to further increase performance.

5.3 Primitive Inlining

Primitive inlining can be viewed as a simpler form of message inlining. Calls to primitive operations are normally implemented using a simple procedure call to an external function in the virtual machine. However, like most other high-performance systems, including some Smalltalk systems [11, 12], our SELF compiler replaces calls of certain common primitives, such as integer arithmetic, comparisons, and array accesses, with their hard-wired definitions. This significantly improves performance since some of these primitives can be implemented in 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 actually calls the primitive at compile-time, replacing the call to the primitive with the result of the primitive; this is SELF's form of *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:



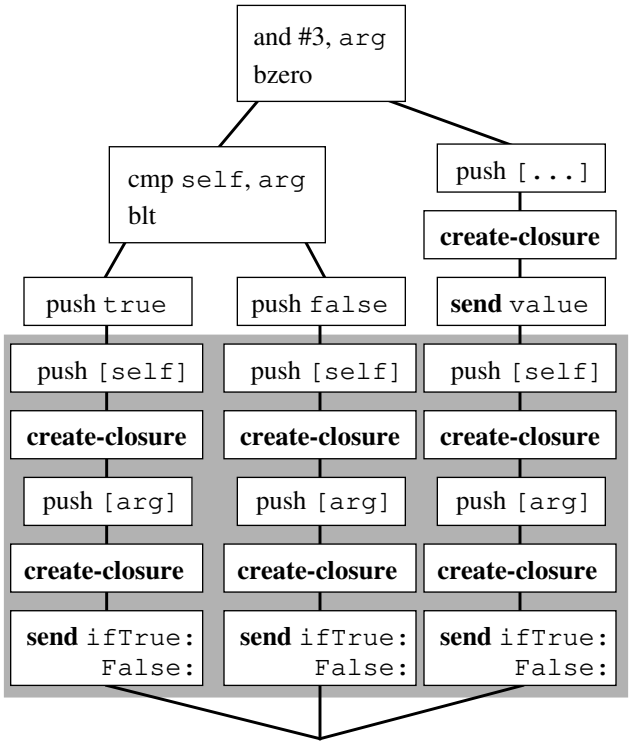
The first compare-and-branch sequence verifies that the argument to the **`_IntLTPrim:IfFail:`** call is also an integer (the receiver is already known to be an integer courtesy of customization); if not, the failure block is created and invoked. If the argument is an integer, then the two integers are compared, and either the **`true`** object or the **`false`** object is returned as the result of the **`<`** message.

The next message considered by our compiler is the **`ifTrue:False:`** message. If **`arg`** is an integer—the common case—the receiver of **`ifTrue:False:`** will be either **`true`** or **`false`**; otherwise it will be the result of the **`value`** message (unknown at compile-time). Normally, this would prevent inlining of the **`ifTrue:False:`** message, since the type of its receiver cannot be uniquely determined. However, by compiling multiple versions of the **`ifTrue:False:`** message, one version for each statically-known receiver type, our SELF compiler can handle and optimize each case separately. This technique is explained next.

5.4 Message Splitting

When type information is lost because the flow of control merges (such as happens just prior to the **`ifTrue:False:`** message in the **`min:`** example), our SELF compiler may elect to *split* the message following the merge into separate messages at the end of each of the preceding branches; the merge is postponed until after the split message. The compiler knows the type of the receiver for some of the copies of the message, and can perform compile-time message lookup and message inlining to radically improve performance for these versions. The proper semantics of the original unsplit message is preserved by compiling a real message send along those branches with unknown receiver types. 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.

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



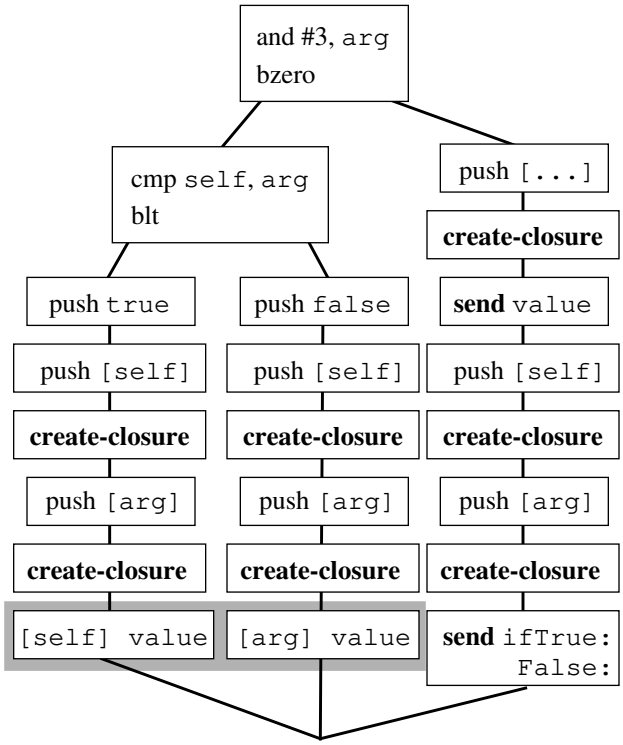
Now the compiler can inline the definition of **`ifTrue:False:`** for the **`true`** object:

```
ifTrue: trueBlk False: falseBlk = ( trueBlk value ).
```

and for the **`false`** object:

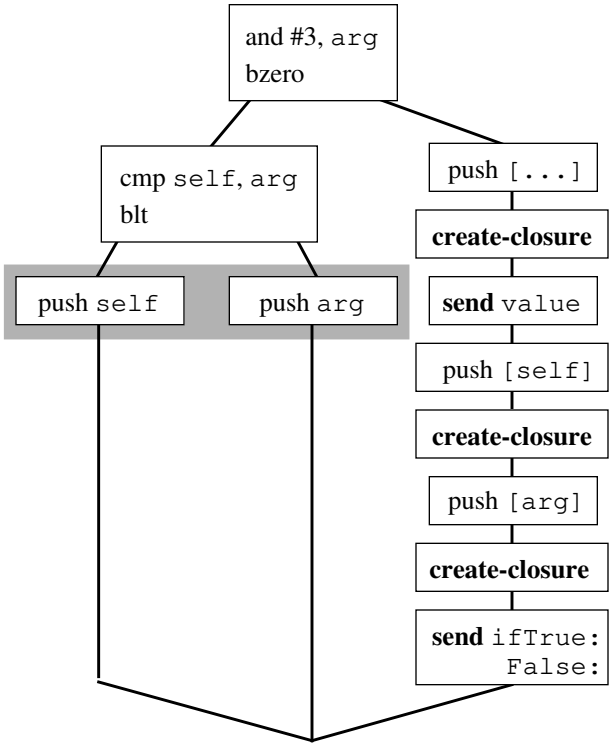
```
ifTrue: trueBlk False: falseBlk = ( falseBlk value ).
```

to get to the following flow graph:



The two **`value`** messages can be inlined, replaced by the bodies of the blocks. Since none of the receiver and arguments of the inlined **`ifTrue:False:`** messages need to be created at run-time any more, the compiler eliminates them from the control flow graph, producing the flow graph at the top of the next page.

Let's assume that the failure block for integer comparisons is too complex to inline away. The compiler won't inline the **`value`** message, and so the **`value`** message's result type is unknown at compile-time. Thus the receiver type of the **`ifTrue:False:`** message is unknown, and a simple SELF compiler wouldn't be able to inline this message away either. However, the next subsection describes how our compiler uses known patterns of usage to predict that the receiver of the **`ifTrue:False:`** message will be a boolean and optimizes the message accordingly.

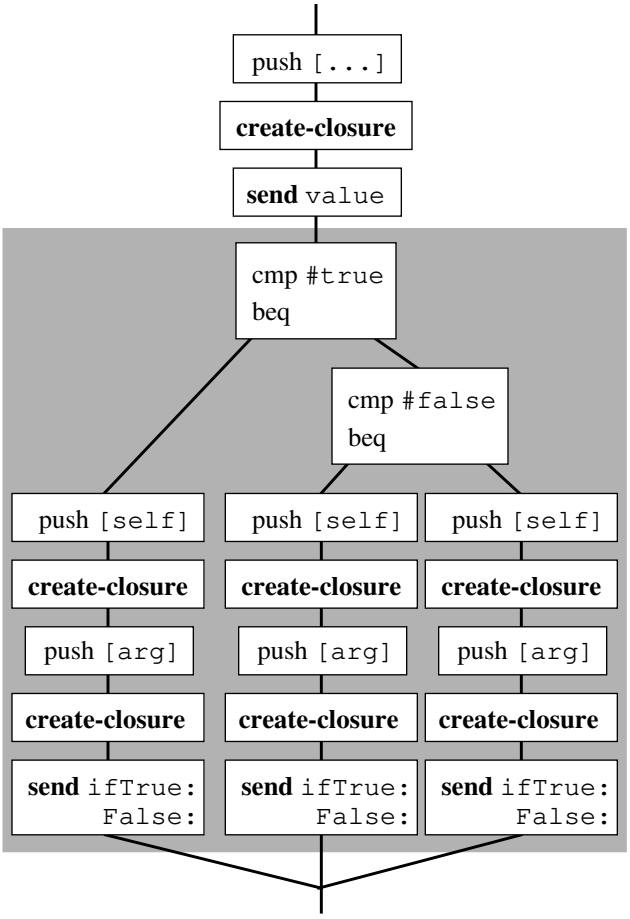


5.5 Type Prediction

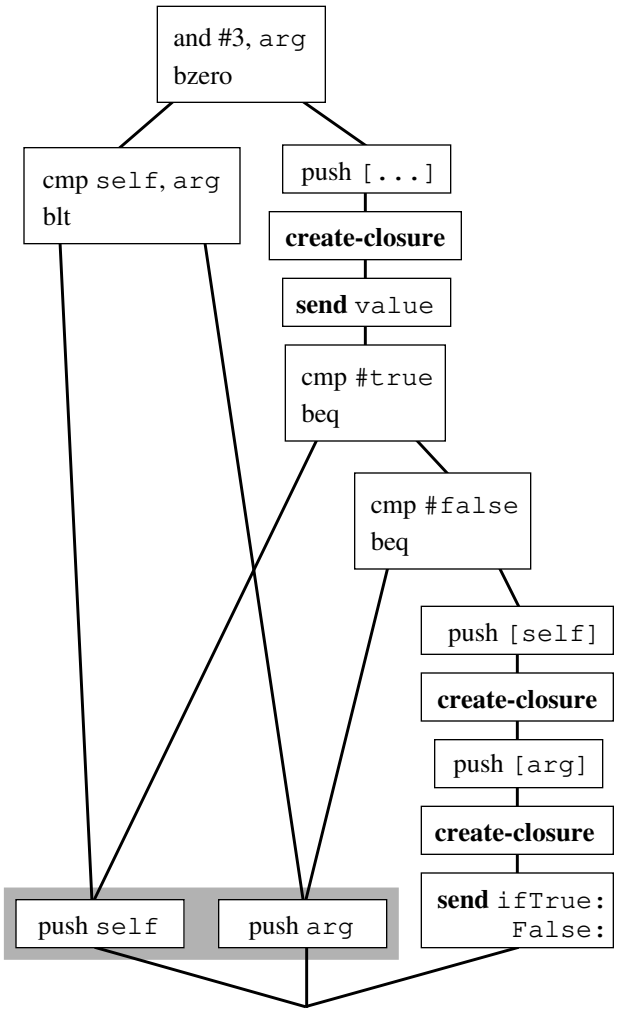
When the type of the receiver of a message is unknown at compile-time, the SELF compiler uses *static type prediction* to generate better code for some common situations. Certain messages are known to the compiler to be likely to be sent to receivers of certain types: **+** and **<** are likely to be sent to integers, and **ifTrue:False:** is likely to be sent to either **true** or **false**. The compiler generates a run-time test based on the expected type or value of the receiver, followed by a conditional branch to one of two sections of code; along the "success" branch, the type (or value) of the receiver is known (at compile-time), along the "failure" branch, the type is unknown. The compiler then uses the message splitting techniques to split the predicted message, compiling a copy of the message along each branch. Because the compiler now knows the type of the receiver of the split message along the "success" branch, it can inline that version of the message away, significantly improving performance for common operations like integer arithmetic and boolean testing. A real message send is executed in the case that the prediction fails, preserving the original message's semantics for all possible receivers.

This type prediction scheme requires little additional implementation work, since message splitting and inlining is already implemented. It is also much better than hard-wiring the **`ifTrue:ifFalse:`**, **`whileTrue:`**, **`==`**, **`+`**, and **`<`** messages into the parser and compiler as Smalltalk-80 systems do, since it achieves the same sorts of performance improvements but preserves the message passing semantics of the language and allows the programmer to modify the definitions of *all* SELF methods, including those that are optimized through type prediction.

Let's apply type prediction to the remaining **`ifTrue:False:`** message in the **`min:`** example. The compiler first inserts run-time tests for the **`true`** object and the **`false`** object, followed by several copies of the **`ifTrue:False:`** message (we'll just look at the remaining unoptimized branch):



In the left branch, the receiver of **`ifTrue:False:`** is known to be the value **`true`**; for the middle branch, the receiver is known to be the value **`false`**. As before, the compiler inlines these two **`ifTrue:False:`** messages, plus the corresponding **`value`** messages, and eliminates the closure creations to get to the final flow graph for the entire method:



In the common case of taking the minimum of two integers, our compiler executes only two simple compare-and-branch sequences, for fast execution. A similar savings will be seen if the user calls `min:` on two floating point numbers or two strings, since our compiler customizes and optimizes special versions for each of these receiver types. But even in the case of taking the minimum of two values of different types, such as an integer and a floating point number, our compilation techniques preserve the message passing semantics of the original source code, and execute the source code faithfully.

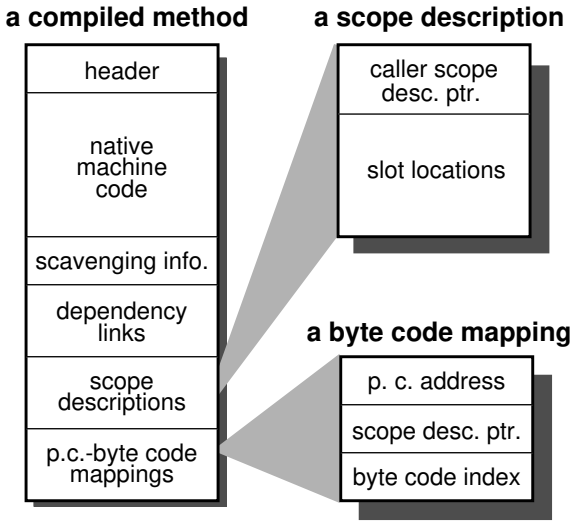
6 Supporting the Programming Environment

Our SELF system supports a high-productivity programming environment. This environment requires both rapid turn-around time for programming changes and complete source-level debugging at the byte code level. These features must coexist with our optimizing compiler techniques, including message inlining. The next two subsections describe the compiler-maintained change dependency links that support incremental recompilation of compiled code affected by programming changes, and the compiler-generated debugging information that allows the debugger to reconstruct inlined stack frames at debug-time. This information is appended to each compiled method object in the compiled code cache.

6.1 Support for Incremental Recompilation

A high-productivity programming environment requires that programming changes take effect within a fraction of a second. This is accomplished in our SELF system by *selectively invalidating* only those compiled methods that are affected by the programming change, recompiling them from new definitions when next needed. The compiler maintains two-way *change dependency links* between each cached compiled method and the slots that the compiled method depends on. The information used to compile code—object formats and the contents of non-assignable slots—is precisely the information stored in maps. Therefore we can confine our dependency links to maps. These links are formed in four ways:

- When a method is being compiled, the system creates a dependency link between the map slot description containing the method and the compiled code in case the definition of the method changes or its slot is removed.
- When the compiler inlines a message, the system creates a dependency link between the matching slot description (either a method slot, a data slot, or an assignment slot) and the compiled code in case the definition of the inlined method changes or its slot is removed.

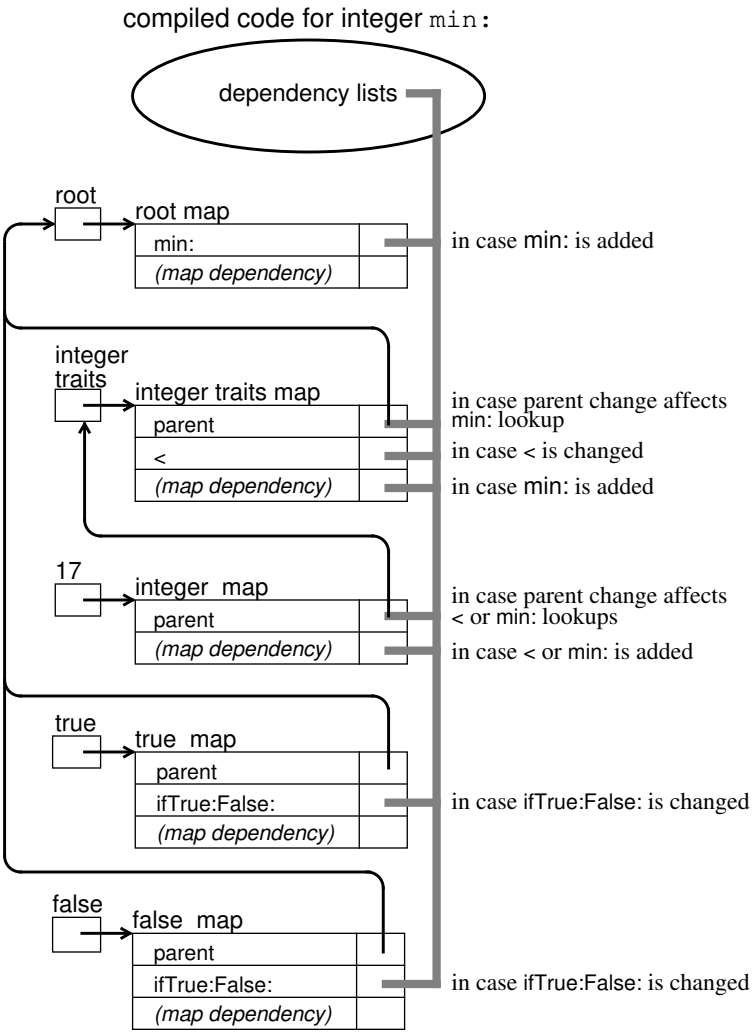


A compiled method contains more than just instructions. It includes a list of the offsets within the instructions of embedded object references, used by the scavenger to modify the compiled code if a referenced object is moved. The compiled method includes dependency links to support selective invalidation. It also includes descriptions of the inlined method scopes, which are used to find the values of local slots of the method and to display source-level call stacks, and a bidirectional mapping between source-level byte codes and actual program counter values.

- When the compiler searches a parent object during the course of a compile-time lookup, the system creates a dependency link between the slot description containing the parent and the compiled code in case the parent pointer changes and alters the result of the lookup.
- When the compiler searches an object unsuccessfully for a matching slot during compile-time lookup, the system creates a dependency link between the map of the object searched and the compiled code in case a matching slot is added to the object later.

These rules ensure that no out-of-date compiled methods survive programming changes, while limiting invalidations to those methods actually affected by a change.

A dependency link is represented by a circular list that connects a slot description or map to all dependent compiled methods. When the system changes the contents of a constant slot or removes a slot, it traverses the corresponding dependency list



The dependency lists for the compiled **`min:`** method customized for integers. The gray line represents eight separate circularly-linked dependency lists. Each list connects a slot description to its dependent compiled code objects. If any of the map information linked to the compiled code changes, the compiled code for **`min:`** (and for any other compiled methods that depend on the same changed information) will be thrown away and recompiled when next needed.

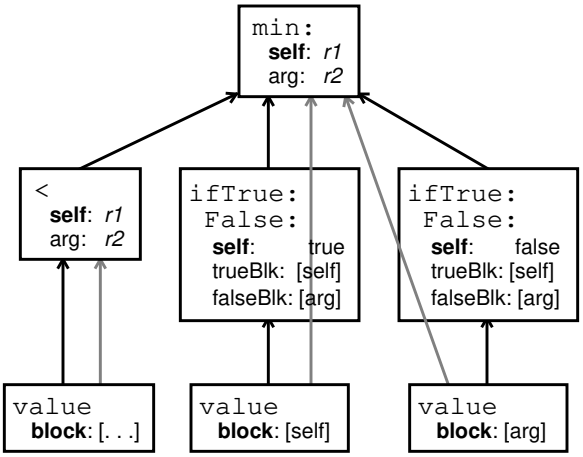
and invalidates all compiled code objects on the list. When the system adds a slot, it similarly traverses the map's dependency list and invalidates linked compiled code objects. Links must be removed from their lists when a method is invalidated or a map is garbage-collected; lists are doubly-linked to speed these removals.

Selective invalidation is complicated by methods that are executing when a programming change requires that they be invalidated. The methods cannot really be flushed, because they are still executing, and some code must exist. But neither can they remain untouched, since they have been optimized based on information that is no longer correct. One solution, which has not been implemented yet, would be to recompile executing methods immediately and to rebuild the execution stack for the new compiled methods. We do not know yet if this procedure would be fast enough to keep programming turn-around time short.

6.2 Support for Source-Level Debugging

A good programming environment must include a source-level debugger. The SELF debugger presents the program execution state in terms of the programmer's execution model: the state of the byte code interpreter, with *no* optimizations. This requires that the debugger be able to examine the state of the compiled, optimized SELF program, and construct a view of that state (the *virtual* state) in terms of the byte-coded execution model. Examining the execution state is complicated by having methods in the virtual call stack actually be inlined within other methods in the compiled method call stack, and by allocating the slots of virtual methods to registers and/or stack locations in the compiled methods. To allow the debugger to reconstruct the virtual call stack from the physical optimized call stack, the SELF compiler appends debugging information to each compiled method. For each scope compiled (the initial method, and any methods or block methods inlined within it), the compiler outputs information describing that scope's place in the virtual call chain within the compiled method's physical stack frame. For each argument and local slot in the scope, the compiler outputs either the value of the slot (if it's a constant known at compile-time, as many slots are) or the register or stack location allocated to hold the value of the slot at run-time.

Our SELF compiler also outputs debugging information to support computing and setting breakpoints. This information takes the form of a bidirectional mapping between program counter addresses and byte code instructions within a particular scope. One complexity with this mapping is that it is not one-to-one: several byte codes may map to the same program counter address (as messages get inlined and optimized away), and several program counter addresses may map to the same byte code (as messages get split and compiled in more than one place). To determine the current state of the program in byte code terms at any program counter address, the debugger first finds the *latest* program counter address in the mapping that is *less than or equal to* the current program counter, and then selects the *latest* byte code mapped to that address; this algorithm returns the last byte code that has been



The debugging information for the min: method. Each scope description points to its calling scope description (black arrows); a block scope also points to its lexically-enclosing scope description (gray arrows). For each slot within a scope, the debugging information identifies either the slot's compile-time value or its run-time location. For the min: example, only the initial arguments have run-time locations (registers r1 and r2 in this case); all other slot contents are known statically at compile-time.

---

started but not completed for any program counter address. The execution stack displayer uses this mapping information to find the bottommost virtual stack frame for each physical stack frame to display the call stack whenever the program is halted.

We have not implemented the breakpointing facilities in our debugger yet; the current "debugger" displays the virtual execution stack and immediately continues execution whenever the **`_DumpSelfStack`** primitive is called. However, our mapping system is designed to support computing and setting breakpoints in anticipation of breakpointing and process control primitives. To set a breakpoint at a particular source-level byte code, the debugger would find all those program counter addresses associated with the byte code and set breakpoints there. In cases where several byte codes map to the same program counter address, single stepping from one byte code to the next wouldn't actually cause any instructions to be executed; the debugger would pretend to execute instructions to preserve the illusion of byte-coded execution.

7 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 have written almost 9,000 lines of SELF code, including a hierarchy of collection objects, a recursive descent parser for SELF, and a prototype 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.[6] The fastest Smalltalk system currently available (excluding graphics performance) is the ParcPlace V2.4 β2 Smalltalk-80 virtual machine, rated at about 4 Dorados[7] [22]; this system includes the Deutsch-Schiffman techniques described earlier. We compare transliterations from C into Smalltalk and SELF of the Stanford integer benchmarks [13] and the Richards operating system simulation benchmark [10], as well as the following small benchmarks, adapted from Smalltalk-80 systems [18]:

```
sumToTest = ( 1 sumTo: 10000 ).
sumTo: arg = (
  | total <- 0 |
  to: arg Do: [ | :index |
    total: total + index.
  ].
  total ).
recurseTest = ( 14 recurse ).
recurse = (
  = 0 ifFalse: [
    (- 1) recurse. (- 1) recurse.
  ] ).
```

We also rewrote most of the Stanford integer benchmarks in a more SELFish programming style, using the first argument of a C function as the receiver of the corresponding SELF method. Measurements for the rewritten benchmarks are presented in columns labeled SELF-OO; times in parentheses mark those benchmarks that were not rewritten.

The following table presents the actual running times of the benchmarks on the specified platform. 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.

[6] Since this paper was originally published, these performance numbers have improved significantly, by a factor of two or three. See [7].

[7] 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 [9]; until recently it was the fastest available Smalltalk implementation.

Raw Running Times

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

The entries in the following table are the ratios of the running times of the benchmarks for the given pair of systems. From our point of view, bigger numbers are better in the first two columns, while smaller numbers are better in the last two columns. The most meaningful rows of the table are probably the rows for the median of the Stanford integer benchmarks and the row for the Richards benchmark.

Relative Performance of SELF

	Smalltalk/ SELF	Smalltalk/ SELF-OO	Smalltalk/ C	SELF/ C	SELF-OO/ 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**
richards	2.8	(2.8)	10.6	3.8	(3.8)
sumToTest	1.4	(1.4)	6.2	4.5	(4.5)
recurseTest	3.2	(3.2)	5.3	1.6	(1.6)

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 results from significantly poorer implementation in the SELF compiler of standard compiler techniques such as register allocation and peephole optimization. Some of the difference 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. The rest of the difference is probably caused by the lack of type information, especially for arguments and assignable data slots. We are remedying these deficiencies to a large extent in the second-generation SELF system described in the next section.

The previous tables show that the performance of object-oriented systems is improving dramatically. As a new metric for comparing the performance of these systems, we propose the *millions of messages per second (MiMS)* measure, analogous to the millions of instructions per second (MIPS) measure for processors. This number measures the performance of an object-oriented system in executing messages. To compute the MiMS rating of a system for a specific benchmark on a particular hardware platform, divide the number of messages the benchmark sends by its total running time. We define message sends as those invocations whose semantics include a dispatch; for SELF, this includes references to slots in the receiver ("instance variable" accesses), since the same reference could invoke a method, but excludes references to slots local to a method invocation ("local variable" accesses), since these could never do anything other than access data. We computed the MiMS rating of our first-generation SELF system for the Richards benchmark on the SPARC-based Sun-4/260 to be 3.3 MiMS, or a message executed every 300ns [16].

The *efficiency* of an object-oriented system is inversely proportional to the number of instructions executed per message sent. The cycle time on the Sun-4/260 is 60ns [21], giving our SELF system a cost per message of about 5 cycles. Since the SPARC has been clocked at 1.6 cycles per instruction [21] (accounting for cache misses and multicycle instructions), this would give our SELF system an efficiency rating of around 3 instructions per message sent. We are not aware of any other implementations of dynamically-typed object-oriented languages that approach this level of efficiency.

Other researchers have attempted to speed Smalltalk systems by adding type declarations to Smalltalk programs. Atkinson's Hurricane compiler compiles a subset of Smalltalk annotated with type declarations [1]. He reports a performance improvement of a factor of 2 for his Hurricane compiler over the Deutsch-Schiffman system on a 68020-based Sun-3; our initial SELF system already achieves the same performance improvement over the Deutsch-Schiffman system without type declarations. Johnson's TS Typed Smalltalk system type-checks and compiles Smalltalk-80 programs fully annotated with type declarations [14]. He reports a performance improvement of a factor of between 5 and 10 over the Tektronix Smalltalk-80 interpreter on a 68020-based Tektronix 4405. For a bench-

mark almost identical to our **`sumToTest`** benchmark, he reports an execution time of 62ms, which we executed in 18ms on a machine 3 to 4 times faster than his machine. This makes his system's performance roughly comparable to our system's performance, even though his system relies on type declarations while ours does not. These results suggest that our compilation techniques do a good job of extracting as much type information as is available to these other systems through programmer-supplied type declarations.

8 Future Work

SELF has not reached its final state. Although we have established the feasibility and rewards of the implementation techniques described in this paper, much work remains.

8.1 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 compilation algorithms. As of this writing (July 1989), we have completely rewritten the object storage system and unified the run-time/compile-time message lookup system. We have implemented the core of the second-generation compiler, and it now compiles and executes about half of our SELF code.

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 type information constructed during type flow analysis, especially the types of local slots. The message 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 needs 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, especially loop bodies, that manipulate the most common data types. Along these common-case sections, the types of most variables will be known at compile-time, leading to maximally-inlined code with few run-time type checks; in the other sections, less type information is available to the compiler, and more full message sends are generated. Under normal conditions the optimized code will be executed, and the method will run fast, possibly just as fast as for a C program. However, in exceptional situations, such as when an overflow actually occurs, the flow of control will transfer to a less optimized section of the method that preserves the message passing semantics.

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.

8.2 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 would provide the compiler with static type information about arguments so it could generate faster code. These benefits have to be balanced against the costs 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 cannot be 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 plan to investigate techniques to optimize 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. Finally, 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.

9 Conclusions

Many researchers have attempted to boost the performance of dynamically-typed object-oriented languages. The designers of Smalltalk-80 hard-wired the definitions of user-level arithmetic and control methods into the compiler, preventing the users from changing or overriding them. Other researchers added type declarations to Smalltalk, thereby hindering reuse and modification of code. We devised *dynamic customized compilation*, *static type prediction*, *type flow analysis*, *message splitting*, and *message inlining* to automatically extract and preserve static type information. Our measurements suggest that our system runs just as fast as Smalltalk systems with type declarations and at least twice as fast as those with hard-wired methods. Researchers seeking to improve performance should improve their compilers instead of compromising their languages.

SELF's novel features do not cost the user either execution time or storage space. Our virtual machine supports the prototype object model just as space- and time-efficiently as similar class-based systems; maps act as implementation-level classes and thus reclaim the efficiency of classes for the implementation without inflicting class-based semantics on the SELF user. SELF's use of messages to access variables has absolutely no effect on the final performance of SELF programs, since these message sends are the first to get inlined away. Once an implementation reaches this level of sophistication and performance, the information provided by classes and explicit variables becomes redundant and unnecessary. Prototype-based languages can run just as fast as class-based languages.

Our implementation introduces new techniques to support the programming environment. The segregation of object references from byte arrays speeds scavenging and scanning operations. Dependency lists reduce the response time for programming changes. Detailed debugging information maps the execution state into the user's source-level execution model, transparently "undoing" the effects of method inlining and other optimizations.

Our techniques are not restricted to SELF; they apply to other dynamically-typed object-oriented languages like Smalltalk, Flavors, and CLOS. Many of our techniques could even be applied to statically-typed object-oriented languages like C++ and Trellis/Owl. For example, customization and automatic inlining could be used to eliminate many C++ virtual function calls, encouraging broader use of object-oriented features and programming styles by reducing their cost. Compiler-generated debugging information could be used by the C++ debugger to hide the inlining from the user, just as our compiler generates debugging information to reconstruct the SELF virtual call stack.

SELF is practical: our implementation of SELF is twice as fast as any other dynamically-typed purely object-oriented language documented in the literature. The SELF compiler achieves this level of efficiency by combining traditional optimizing compiler technology like procedure inlining and global register allocation, specialized techniques developed for high-speed Smalltalk systems like dynamic translation and inline caching, and new techniques like customization, message splitting, and type prediction to bridge the gap between them. The synergy of old and new results in good performance.

10 Acknowledgments

We owe much to Randy Smith, one of the original designers of SELF. We also would like to thank Peter Deutsch for many instructive discussions and seminal ideas for the design and implementation of SELF. Bay-Wei Chang implemented our graphical SELF object browser and contributed to discussions on the future of the SELF language and implementation.

References

1. Atkinson, R. G. Hurricane: An Optimizing Compiler for Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 151-158.
2. Ballard, M. B., Maier, D., and Wirfs-Brock, A. QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 140-150.
3. Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification. Published as *SIGPLAN Notices*, 23, 9 (1988).
4. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.
5. Borning, A. H., and Ingalls, D. H. H. A type declaration and inference system for Smalltalk. In *Conference Record of the Ninth Annual Symposium on Foundations of Computer Science* (1982) 133-139.
6. Chambers, C., and Ungar, D. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.

7. Chambers, C., and Ungar, D. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. Published as *SIGPLAN Notices,* 25, 6 (1990) 150-162. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

8. Curtis, P. Type inferencing in Smalltalk. Personal communication (1989).

9. Deutsch, L. P. The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture. In Krasner, G., editor, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA (1983) 113-126.

10. Deutsch, L. P. Richards benchmark. Personal communication (1988).

11. Deutsch, L. P., and Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages* (1984) 297-302.

12. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).

13. Hennessy, J. Stanford integer benchmarks. Personal communication (1988).

14. Johnson, R. E., Graver, J. O., and Zurawski, L. W. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*. Published as *SIGPLAN Notices*, 23, 11 (1988) 18-26.

15. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.

16. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

17. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.

18. McCall, K. The Smalltalk-80 Benchmarks. In Krasner, G., editor, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, MA (1983) 153-174.

19. Meyer, B. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 391-405.

20. Moon, D. A. Object-Oriented Programming with Flavors. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 1-16.

21. Namjoo, M., Agrawal, A., Jackson, D. C., and Quach, L. CMOS Gate Array Implementation of the SPARC Architecture. In *COMPCON '88 Conference Proceedings* (1988) 10-13.

22. ParcPlace Systems. *ParcPlace Newsletter (Winter 1988)*, 1, 2 (1988).

23. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.

24. Steele, G. L., Jr. LAMBDA: The Ultimate Declarative. AI Memo 379, MIT Artificial Intelligence Laboratory (1976).

25. Steele, G. L., Jr., and Sussman, G. J. LAMBDA: The Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory (1976).

26. Stein, L. A. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 138-146.

27. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA (1986).

28. Suzuki, N. Inferring Types in Smalltalk. In *8th Annual ACM Symposium on Principles of Programming Languages* (1981) 187-199.

29. Ungar, D. M. *The Design and Evaluation of a High-Performance Smalltalk System*. Ph.D. thesis, the University of California at Berkeley (1986). Published by the MIT Press, Cambridge, MA (1987).

30. Ungar, D., and Jackson, F. Tenuring Policies for Generation-Based Storage Reclamation. In *OOPSLA '88 Conference Proceedings*. Published as *SIGPLAN Notices*, 23, 11 (1988) 1-17.

31. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

32. Wegner, P. Dimensions of Object-Based Language Design. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 168-182.