Smalltalk-80

Bits of History, Words of Advice

Glenn Krasner, Editor

Xerox Palo Alto Research Center

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California
London • Amsterdam • Don Mills, Ontario • Sydney

This book is in the
Addison-Wesley series in Computer Science
MICHAEL A. HARRISON
CONSULTING EDITOR

Cartoons drawn by Jean Depoian

Library of Congress Cataloging in Publication Data

Main entry under title:

Smalltalk-80 : bits of history, words of advice.

Bibliography: p.
Includes index.
1. Smalltalk-80 (Computer system) I. Krasner,
Glenn. II. Title: Smalltalk-eighty.
QA76.8.S635S58 1983 001.64'.25 83-5985
ISBN 0-201-11669-3

Reprinted with corrections, June 1984

ISBN 0-201-11669-3
CDEFGHIJ-AL-8987654

Design Decisions for Smalltalk-80 Implementors

Allen Wirfs-Brock
Tektronix, Inc.
Beaverton, Oregon

Abstract

The Smalltalk-80 virtual machine specification describes the required behavior of any Smalltalk-80 interpreter. The specification takes the form of a model implementation of a Smalltalk-80 interpreter. An implementor of a Smalltalk-80 interpreter is not required to exactly copy the data structures and algorithms of the model interpreter. The only requirement is that any Smalltalk-80 interpreter exhibit external behavior which is identical to that described by the formal specification. The implementor is free to make design tradeoffs that may increase the performance of the implementation while preserving the required external behavior. This paper identifies some of the design decisions which face a Smalltalk-80 implementor and discusses several design trade-offs.

Introduction

The Smalltalk-80 virtual machine specification as it appears in *Smalltalk-80: The Language and Its Implementation*[1] describes the required low level behavior of any Smalltalk-80 implementation. The

specification takes the form of a Smalltalk-80 "program" which exhibits this behavior. One approach to the implementation of a Smalltalk-80 interpreter is to literally translate this program into some appropriate implementation language. While this approach will result in an interpreter which exhibits the required behavior, the performance of the resulting interpreter may be unsatisfactory.

An alternate implementation approach is to construct an interpreter that uses algorithms and data structures which differ from those used in the formal specification. These would be chosen to optimize performance for the host implementation environment. Such an interpreter may achieve higher performance but requires greater implementation effort.

This paper presents an overview of the design decision space which confronts the implementors of Smalltalk-80 interpreters. Specifically, it examines *some* of the potential design trade-offs concerning the host hardware and implementation language, the interpreter data structures, the actual execution of Smalltalk-80 instructions, and the creation and destruction of objects. Even though the design issues are examined assuming an interpreter implementation utilizing a conventional computer or microprocessor as a host, many of the trade-offs should be applicable to a microcoded or hardware implementation.

The Formal Specification

The first part of the Smalltalk-80 virtual machine specification defines the virtual machine architecture. This includes the definition of the primitive data types, the instruction set, and the interface to the object memory manager. The second part describes the internal operation of the object memory manager. An implementation of the Smalltalk-80 virtual machine is commonly referred to as a Smalltalk-80 interpreter. The formal specification completely defines the required behavior of a Smalltalk-80 interpreter.

The formal specification takes the form of a collection of Smalltalk-80 methods which implement a Smalltalk-80 interpreter. It is, in effect, an implementation of a "model interpreter." Within this model the "registers" of the virtual machine are represented as Smalltalk-80 instance variables, the data structures are explicitly defined via constant field offsets and bit masks, and the required semantics of the interpreter are implicit in the behavior of the methods. The model bytecode interpreter implementation can be viewed as the definition of the correct behavior of a Smalltalk-80 implementation.

The specification does not place any particular requirements upon the internal implementation of the object memory manager. Of course, it assumes that any implementation will correctly preserve stored data and that this data will be available to the interpreter when requested. The memory manager implementation chapter may also be viewed as a model for how an object memory manager may be implemented.

An implementor of a Smalltalk-80 interpreter must design and construct an interpreter whose behavior conforms to that defined by the formal specification. One method of accomplishing this is to directly translate the Smalltalk-80 methods of the model implementation into an appropriate implementation language. One might even consider using a program to perform this translation. Figure 4.1 gives an example of a method from the formal specification and Figure 4.2 shows how it might be translated into Pascal.

The principal advantage of the direct translation approach is that it is a simple method of obtaining a semantically correct interpreter. It also is a very good way for an implementor to learn how the interpreter works internally. The principal disadvantage associated with this approach is that the resulting interpreter may exhibit disappointing performance levels. The data structures and algorithms of the book's interpreter were selected to provide a clear definition of the required behavior; they will probably not be optimal for any particular host computer. The challenge for a Smalltalk-80 implementor is to design an interpreter which will yield acceptable performance within some particular host environment. At Tektronix, we utilized the direct translation approach (see Chapter 5) and were able to very quickly build a working (but slow) Smalltalk-80 implementation. Experience gained from this initial implementation enabled us to later design a significantly improved second generation interpreter.

```
initializeGuaranteedPointers
    "Undefined Object and Booleans"
    nilPointer ← 2.
    falsePointer ← 4.
    truePointer ← 6.
    "and so on ..."
pushConstantBytecode
    currentBytecode = 113 ifTrue: [↑self push: truePointer].
    currentBytecode = 114 ifTrue: [↑self push: falsePointer].
    currentBytecode = 115 ifTrue: [↑self push: nilPointer].
    currentBytecode = 116 ifTrue: [↑self push: minusOnePointer].
    currentBytecode = 117 ifTrue: [↑self push: zeroPointer].
    currentBytecode = 118 ifTrue: [↑self push: onePointer].
    currentBytecode = 119 ifTrue: [↑self push: twoPointer].
```

Figure 4.1

```
const
   {Undefined Object and Booleans}
   nilPointer = 2;
   falsePointer = 4;
   truePointer = 6;
   {and so on ...}
procedure pushConstantBytecode;
   begin
      case currentBytecode of
         113: push(truePointer);
         114: push(falsePointer);
         115: push(nilPointer);
         116: push(minusOnePointer);
         117: push(zeroPointer);
         118: push(onePointer);
         119: push(twoPointer);
      end {case}
   end {pushConstantBytecode};
```

Figure 4.2

The Host Processor

The first major design decision which will confront a Smalltalk-80 implementor will be the choice of the hardware which will host the implementation. In many situations the implementor will have little freedom in this area. Where the implementor has the freedom to select the host processor, there are a number of considerations which should enter into the decision process.

A processor which is to host a Smalltalk-80 interpreter should be fast. An interpreter which executes 10,000 bytecodes per second may be perceived by a Smalltalk-80 programmer to be quite slow. The original Tektronix implementation, which could execute 3500 bytecodes per second, was considered to be just barely usable. The Xerox Dolphin implementation executes 20,000 bytecodes per second and is considered to have "adequate" performance, while the Xerox Dorado at 400,000 bytecodes per second has excellent performance (see Chapter 9). At 10,000 bytecodes per second the interpreter will have, on the average, only 100 microseconds in which to fetch, decode, and execute each bytecode. At a more acceptable performance level of 100,000 bytecodes per second, the interpreter will have only 10 microseconds for each bytecode.

A Smalltalk-80 host processor architecture must support a large amount of main memory (either real or virtual). The standard Smalltalk-80 virtual image consists of approximately 500,000 bytes of

Smalltalk-80 objects. To this must be added the space for interpreter, the interpreter's data structures, the display bitmap, and additional space to contain objects created dynamically as the system runs. The total requirements of the system will easily approach one million bytes of memory with even a modest application. Although it may be possible to configure a virtual image with fewer features and more modest memory requirements, this can be most easily done utilizing an operational Smalltalk-80 system. For this reason, the implementor will need a development system with at least 1 megabyte of main memory.

By caching a number of variables which represent the execution state of a Smalltalk-80 method in internal registers, an implementation will probably get dramatically improved performance. A good host processor should have sufficient internal registers to allow these values to be cached in its registers. The exact number of registers needed to contain cached values will depend upon the specifics of the interpreter design. However, as a general rule, 8 is probably not enough while 32 is probably more than enough. For example, one of our implementations for the Motorola 68000 processor could have easily made use of several more than the 15 registers which were available.

Smalltalk-80 interpreters frequently look up values in tables and follow indirect references. For this reason it is desirable that the host processor provide good support for indexed addressing and indirection.

Hardware support for the Smalltalk-80 graphics model is another major consideration. Smalltalk-80 graphics is entirely based upon the manipulation of bitmaps. Although some implementations have simulated this model using other display technologies (for example, by using a vector oriented raster terminal), the results have been less than satisfactory (see Chapter 5). Acceptable results will only be achieved if an actual hardware bitmapped display is provided. A frequent concern of new implementors is the performance of BitBlt, the bitmap manipulation operation. One concern is whether specific hardware support will be required for this operation. Our experience with the 68000 was that adequate BitBlt performance was easy to achieve with straightforward coding, while adequate bytecode interpreter performance was very difficult to achieve. This leads us to believe that a host processor capable of achieving adequate performance when interpreting bytecodes will probably perform adequately when BitBlt-ing. In particular, the processor's ability to perform shifting and masking operations will affect the overall performance of BitBlt.

The Implementation Language

The choice of an implementation language for a Smalltalk-80 interpreter is typically a trade-off between the ease of implementation of the interpreter and the final performance of the system. Implementors should

consider using a high-level programming language as the first implementation tool. A high-level language based interpreter can be quickly implemented and should be relatively easy to debug. Unfortunately, the final performance of such implementations may be disappointing. This may be the case even if a very good optimizing compiler is used.

It is generally accepted that the code generated for a large program by an optimizing compiler will be "better" than that which a human assembly language programmer would write for the same problem. Conversely, for short code sequences, a human programmer can usually write better code than that generated by an optimizing compiler. Although a Smalltalk-80 interpreter may appear to be a complex piece of software, it is actually a relatively small program. For example, our assembly language implementation for the Motorola 68000 contains approximately 5000 instructions. Furthermore, a large portion of the execution time tends to be spent executing only a few dozen of the instructions. These instruction sequences are short enough that carefully written assembly code can achieve significantly better performance than optimized compiler generated code. Our 68000 bytecode dispatch routine consists of five instructions, while the bodies of many of the push and pop bytecodes consist of only one or two instructions.

A successful Smalltalk-80 interpreter design will consist of an efficient mapping of the virtual machine architecture onto the available resources of the host processor. Such a mapping will include the global allocation of processor resources (registers, preferred memory locations, instruction sequences, etc.) for specific purposes within the interpreter. An assembly language programmer will have complete freedom to make these allocations. Such freedom is typically unavailable to a high-level language programmer who must work within a general purpose resource allocation model chosen by the designers of the compiler.

Object Pointer Formats

The most common form of data manipulated by a Smalltalk-80 interpreter are Object Pointers (commonly referred to as Oops). An Oop represents either an atomic integer value in the range -16,384 to 16,383 or a reference to some particular Smalltalk-80 object. The formal specification uses a standard representation for Oops. This representation defines an Oop to be a 16-bit quantity. The least significant of the 16 bits is used as a tag which indicates how the rest of the bits are to be interpreted. If the tag bit is a 0 then the most significant 15 bits are interpreted as an object reference. If the tag bit is a 1 then the most significant 15 bits are interpreted as a 2's complement integer value.

Note that the size of an Oop determines both the total number of objects which may exist at any time (32,768) and the range of integer values upon which arithmetic is primitively performed.

Because Oops are used so frequently by the interpreter, their format can have a large impact upon the overall performance of the interpreter. The most common operations performed upon Oops by the interpreter are testing the tag bit, accessing the object referenced by an Oop, extracting the integer value from an Oop, and constructing an Oop from an integer.

Even though the standard Oop format pervades the formal specification, use of a different format will not violate the criteria for conformance to the specification. This is possible because the internal format of an Oop is invisible to the Smalltalk-80 programmer.

There are several possible alternative Oop formats which may offer varying performance advantages. One alternative is to change the position of the tag bit.

Placing the tag bit in the least significant bit position (the position in the standard Oop format) is most appropriate for a processor which reflects the value of this bit in its condition codes. This is the case for the Xerox processors[2] upon which the Smalltalk-80 system was originally developed, and for some common microprocessors. Using such a processor, the tag bit is automatically "tested" each time an Oop is accessed. A simple conditional branch instruction can then be used by the interpreter to choose between integer and object reference actions. Processors which lack this feature will require a more complex instruction sequence, shifting the Oop, a masking operation, and comparison to perform the same test.

Placing the tag in the most significant bit position causes the tag to occupy the sign-bit position for 16-bit 2's complement processors. For a processor that has condition codes which reflect the value of the sign bit, a test of the tag becomes a simple branch on positive or negative value.

Other factors which will affect the tag bit position might include the relative performance cost of setting the least significant bit as opposed to the most significant bit (is adding or logical or-ing a 1 less expensive than the same operation involving 32,768) for converting an integer into an Oop, and the relative cost of shifts as opposed to adds for converting Oops into table indices.

The standard format uses a tag bit value of 1 to identify an integer value and a tag bit value of 0 to identify an object identifier. Inverting this interpretation has potentially useful properties, some of which are also dependent upon the choice of tag bit position. For example, if a tag value 0 is used to indicate an integer valued Oop and the tag occupies the least significant bit position, then SmallInteger values are, in effect, 2's complement values which have been scaled by a factor of 2. Such

values can be added and subtracted (the most common arithmetic operations) without requiring a conversion from the Oop format and the result will also be a valid SmallInteger Oop. Only one of the operands of a multiplication operation will need to be converted from the Oop format for the operation to yield a valid SmallInteger Oop.

If a tag value of 0 is used to indicate object identifier Oops and the tag occupies the most significant bit position, then object identifier Oops can serve as direct indices into a table of 8-bit values on byte addressable processors. This would allow reference counting to be implemented using an independent table of 8-bit reference-count values which is directly indexed using Oops. For a word addressed processor, the standard format allows Oops to be used to directly index a 2 word per entry object table.

The Object Memory

The object memory implementation described in the formal specification views the object memory as being physically divided into 16 physical segments, each containing 64K 16-bit words. Individual objects occupy space within a single segment. Object reference Oops are translated into memory addresses using a data structure known as the Object Table. The object table contains one 32-bit entry for each of the 32K possible object referencing Oops. Each object table entry has the following format:

Bits 0-15 (lsb):	The word offset of the object within its segment
Bits 16-19:	The number of the segment which contains the object
Bit 20:	Reserved
Bit 21:	Set if the Oop associated with this entry is unused
Bit 22:	Set if the fields of this object contain Oops
Bit 23:	Set if object contains an odd number of 8 bit fields
Bits 24-31 (msb):	This object's reference count

For each segment there is a set of linked lists which locate all free space within the segment. In addition there is a single list which links all unassigned Oops and object table entries. Objects are linked using Oop references.

The above design includes several implicit assumptions about the memory organization of the host processor. It assumes that the unit of memory addressability is a 16-bit word. It assumes that the processor uses a segmented address space and that each segment contains 64K

words. Finally, it assumes that at most 1024K words (16 segments) are addressable. This organization may be considerably different from that of an actual host processor. Many processors support a large, byte addressable, linear address space. Although the formal specification's design can be mapped onto such a memory organization, such a mapping will result in reduced interpreter performance if it is carried out dynamically.

An object memory design will consist of two inter-related elements, the organization of the actual object space and the format of the object table. The goal of the design will usually be to minimize the time required to access the fields of an object when given an Oop. However, if main memory is limited, the goal of the design may be to limit the size of the object table. A performance oriented object table will usually be represented as an array which is directly indexed by Oops (or a simple function on Oops). A hash table might be used for a space efficient object table representation[3].

The most important component of an object table entry is the field which contains the actual address of the associated object within the object space. Ideally this field should contain the physical memory address of the object represented so that it may be used without any masking or shifting operations. Such a format will permit the contents to be used to directly address the associated object, either by loading the field into a processor base register or by some type of indirect addressing mechanism. In this case, the size of the address field will be the size of a physical processor address.

If the host processor's physical address is larger than the 20-bits used in the formal specification, the size of an object table entry will have to be increased beyond 32-bits or the size of the reference count and flag bits will have to be decreased. Since Oops are typically used as scaled indexes into the object table, it is desirable that the size of an object table entry be a power-of-two multiple of the processor's addressable word size so that object table offsets may be computed by shifting instead of multiplication. For most conventional processors, 64-bits (8 bytes, four 16-bit words, two 32-bit words) would be the next available size. However, a 64-bit object table entry will require 256K bytes and will probably contain many unused bits. An alternate approach is to use separate parallel arrays to hold the address fields and the reference count/flag fields of each entry. This results in an effective entry size which is greater than 32-bits without requiring a full 64-bit entry. Decreasing the size of the reference-count field is another valid alternative. Since most reference count values are either very small (8 or less) or have reached the overflow value where they stick[4], a reference-count field size of 3 or 4 bits should be adequate. The main consideration will be whether the host processor can efficiently access such a field.

The Bytecode Interpreter

The bytecode interpreter performs the task of fetching and executing individual Smalltalk-80 bytecodes (virtual machine instructions). Before examining the actual functioning of the bytecode interpreter, we will consider the general question of time/space trade-offs within Smalltalk-80 implementations. A complete, operational Smalltalk-80 system requires approximately one million bytes of storage to operate. The actual interpreter will occupy only a small fraction of this. (Our first implementation, which was very bulky, required approximately 128K bytes for the interpreter. A later assembly language implementation for the same host needed less than 25K bytes.) Since Smalltalk-80 interpreters seem to strain the computational resources of conventional processors, most interpreter designs will tend towards reducing execution time at the expense of increasing the total size of the implementation.

The model implementation in the formal specification takes an algorithmic approach to interpretation. The interpreter fetches a bytecode, shifts and masks it to extract the operation code and parameter fields, and uses conditional statements to select the particular operation to be performed. While this approach is quite effective for illustrating the encoding of the bytecodes it is often not suitable for a production interpreter because of the computation required to decode each bytecode. A more efficient implementation technique for the bytecode dispatch operation may be to use the bytecode as an index into a 256-way dispatch table which contains the addresses of the individual routines for each bytecode. For example, rather than using one routine, as in the example in Fig. 3.1, there could be seven individual routines, each one optimized for pushing a particular constant value.

The model implementation exhibits a high degree of modularity. This is particularly true in the area of the interface between the bytecode interpreter and the object memory manager. The bytecode interpreter makes explicit calls to object memory routines for each memory access. The performance of a production implementation can, however, be improved by incorporating intimate knowledge of the object memory implementation into the bytecode interpreter. Many object memory accesses may be performed directly by the interpreter without actually invoking separate routines within the object memory manager.

As mentioned earlier, the selection of which interpreter state values to cache is a critical design decision for the bytecode interpreter. The designer must evaluate the cost of maintaining the cached values (loading the values when a context is activated and storing some of the values back into the context when it is deactivated) relative to the actual performance gains from using the cached values. The evaluation should consider the average duration of an activation. Our observations indicate that most activations span a small number of bytecodes (less than

10). Caching too much of the active context can thus lead to situations where considerable execution time is spent caching values that are not used over the span of the activation.

The model implementation caches the actual Oop values of several context fields. This implies that these values must be decoded into real memory addresses (performing an object table lookup or conversion from SmallInteger format) each time they are used. An alternative is to decode these values when they are fetched from the active context and to cache the addresses. This means that the cached program counter would be the actual address of the next bytecode and that the cached stack pointer would be the actual address of the top element of the active context's stack. If this technique is used, care must be taken that the cached values are correctly updated, e.g., when the memory manager changes the physical location of objects (performs a segment compression). It is also essential that the values of the stack pointer and program counter field get updated when the active context changes.

The Smalltalk-80 system's required support for multiple processes, when implemented in an obvious manner, can impose an overhead upon each bytecode. The formal specification requires that a process switch may occur before each bytecode is fetched. An obvious way to implement this requirement is to have a global boolean flag which indicates that a process switch is pending, and to test this flag before fetching each bytecode. This technique has the disadvantage that the overhead of testing this flag occurs for each bytecode executed even though actual process switches are infrequent. Since the number of instructions required to implement most bytecodes is relatively small, this test can be a significant overhead. Alternative implementations techniques can avoid this overhead. For example, the address of the bytecode dispatcher might be stored in a processor register. Routines which implement bytecodes would then terminate by branching to the address contained in the registers. A pending process switch could then be signaled by changing the address in the register to the address of the routine which performs process switches. When the current bytecode finishes, control would therefore be transferred to the process switcher.

Memory Management

The routines of the formal specification's object memory manager may be grouped into two categories. The first category consists of those routines which support accesses to objects. The second category consists of those routines which support the allocation and deallocation of objects.

The access routines (such as fetchPointer:ofObject: and storeByte:ofObject:withValue:) are used by the bytecode interpreter to store and retrieve the information contained in the fields of objects. In many implementations of the bytecode interpreter, these functions will not be performed by independent routines, but will be implicitly performed by inline code sequences within the routines of the interpreter. The object allocation and deallocation routines form the bulk of the memory manager.

Collectively, the memory management routines will probably comprise the most complex part of a Smalltalk-80 interpreter implementation. In addition, unless great care is taken in their design, the percentage of execution time spent in these routines can easily dominate the time spent in all other parts of the interpreter. Our initial implementation was found to be spending 70% of its time within memory management routines (see Chapter 5).

Object Allocation

The bytecode interpreter normally requests the allocation of an object in two circumstances. The first circumstance is the execution of a primitive method (most commonly the primitive new or new:) which explicitly calls for the creation of a new object. The second circumstance is the activation of a new method. This implicitly requires the creation of a context object to represent the state of the activation. The formal specification provides a single generalized set of routines which handle both types of allocation requests. These routines perform the following actions. First they must assign an Oop which will be used to refer to the new object. Second they must find an area of free storage within the object memory, large enough to contain the requested object. Next they must initialize any internal data structures (for example an object table entry or object length field) used to represent the object. Finally, they must initialize the fields of the object with a null value.

Observation of actual Smalltalk-80 implementations indicates that the vast majority of allocation requests are for the creation of context objects (see Chapter 11). In addition, most of these requests are for the smaller of the two possible context sizes. A memory manager design which optimizes the creation of a small context object should thus yield better performance.

There are a number of possible approaches to achieving such an optimization. A memory manager might have a dedicated list of available contexts. These available contexts might be preinitialized and have preassigned Oops associated with them. If the memory manager attempts to ensure that this list will not be empty (perhaps by using a background process to maintain the list), then a context could usually be allocated by simply removing the first element from the list.

A memory manager might choose to dedicate a memory segment to the allocation of contexts. Since such a segment would only contain objects of a single size, the actual allocation and deallocation process should be simplified.

Any scheme to optimize context allocation must, of course, conform to the formal specification's requirement that a context behaves as a normal Smalltalk-80 object. The representation of activation records (contexts) as objects contributes much to the power of Smalltalk-80 (it allows programs such as the Smalltalk-80 debugger to be implemented) but requires a large amount of system overhead to support. A major challenge to Smalltalk-80 implementors is to develop techniques to reduce this overhead while preserving the inherent power of context objects.

Storage Reclamation

Storage reclamation is the second major function of the Smalltalk-80 memory manager. While the Smalltalk-80 storage model allows a program to explicitly request the creation of an object, it does not require a program to explicitly request that an object be deallocated. Once an object has been allocated it must remain in existence as long as it is accessible from any other object. An object may only be deallocated if no references to it exist. It is the memory manager's responsibility to automatically deallocate all inaccessible objects. This process is commonly referred to as garbage collection[5]. The classical method (called mark/sweep) of performing garbage collection is to periodically halt processing, identify all inaccessible objects, and then deallocate them. This is commonly done as a two-phase process. First all accessible objects are marked. This requires starting at some root object and traversing all accessible pointers in the system. Second, all unmarked objects are deallocated. With a large object memory, such a process may consume a considerable period of time (ranging from several seconds to several minutes). Because of the interactive nature of the Smalltalk-80 system, such delays are unacceptable. Instead, a garbage collection technique which distributes the storage reclamation overhead over the entire computation is required. The most commonly known technique for achieving this is reference counting. This is the technique used by the formal specification's model implementation.

Reference counting requires that each object have associated with it a count of the number of pointers to it which exist in the system. Each time an Oop is stored into a field the reference count of the object associated with the Oop is incremented. Since storing an Oop into a field must overwrite the previous contents of the field, the reference count associated with the old value is decremented. When the reference count of an object reaches zero, the object is deallocated. The deallocation of

an object invalidates any object references contained in it and hence will decrement their reference counts. This may recursively cause other objects to be deallocated.

Although reference counting eliminates the long delays characteristic of mark/sweep collection, it introduces considerable overhead into the normal operations of the system. We have found that for our host processor (a Motorola 68000), the code sequences that implement simple bytecodes such as the push and pop operations using reference counting are several times longer than the equivalent routines without reference counting. A Smalltalk-80 interpreter design that can decrease this overhead should have greatly improved performance.

There are several possible approaches to achieving this improved performance. One technique which reduces the actual counting overhead is called deferred reference counting[6]. It is based upon the observations that the most frequent and most dynamic object references occur from context objects and that many of these references are quite transitory. For example, assigning an object to a variable causes the object's reference count to be first increased by one as it is pushed onto the context's stack, then decreased by one as it is popped from the stack, and finally increased by one as it is stored into the variable. Our measurements show that "store instance variable" bytecodes (the most common means of creating an object reference from a non-context object) account for less than 4% percent of the dynamically executed bytecodes. If the need to perform reference counting for references contained within contexts is eliminated, then almost all of the reference counting overhead will have been eliminated.

A Second Generation Design

The first Tektronix Smalltalk-80 interpreter was implemented in Pascal on a Motorola 68000 (see Chapter 5). Even though the performance of this implementation was so poor that it was only marginally useful, the experience gained from this effort enabled us to design a new interpreter which exhibits much better performance. In developing this second generation interpreter we encountered many of the design tradeoffs mentioned in the previous sections of this paper. The new interpreter was designed and implemented by the author over a period of approximately nine months.

We choose to continue using a 68000 as the host for the new interpreter but component advances enabled us to use a 10 Mhz processor with one memory wait state instead of an 8 Mhz processor with two wait states. We choose to implement the interpreter in assembly lan-

guage. In addition, great care was taken in choosing the code sequences for all of the frequently executed portions of the interpreter. The common byte codes are all open coded with separate routines for each possible instruction parameter.

The active context's stack pointer, instruction pointer, and home context pointer are cached in 68000 base registers as 68000 addresses. The stack pointer representation was chosen such that 68000 stack-oriented addressing modes could be used to access the active context stack. Other registers are dedicated to global resources such as addressing the object table and accessing free context objects.

The Oop format chosen requires only a simple add instruction to convert an Oop into an object table index. Object table entries can be directly loaded into base registers for accessing objects. A separate reference-count table is used. Deferred reference counting is used to limit the reference-counting overhead and to streamline the code sequences for push/pop bytecodes. Complete context objects are not created for leaves of the message send tree. Context objects are only created if a method references the active context or causes another new context to be activated.

The initial (before any tuning and without some optional primitives) performance benchmarks of our second generation interpreter (see Chapter 9) show that it is between five and eight times faster than our original implementation. We feel that these results demonstrate that it is feasible to build usable microprocessor based Smalltalk-80 implementations.

Summary and Conclusions

For any given host processor, its performance as a Smalltalk-80 host can potentially vary widely depending upon how the Smalltalk-80 interpreter is implemented. The goal of a Smalltalk-80 implementor should be to achieve the best possible mapping of the Smalltalk-80 virtual machine specification onto the chosen host computer. To accomplish this, the implementor will need to intimately understand both the internal dynamic behavior of the Smalltalk-80 virtual machine and the idiosyncrasies of the host processor. We would recommend that an implementor gain an understanding of the behavior of the virtual machine by first using a high-level language to implement the interpreter as described by the formal specification. This implementation can then be used to study the actual behavior of the Smalltalk-80 system and explore design alternatives. Finally, a new implementation should be designed which takes maximum advantage of the characteristics of the

host processor. We have presented a few of the design alternatives which should be considered by Smalltalk-80 implementors as they develop their interpreters.

References

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

2. Lampson, Butler W., "The Dorado: A High Performance Personal Computer," Xerox PARC Technical Report CSL-81-1, Jan. 1981.

3. Kaehler, Ted, "Virtual Memory for an Object-Oriented Language," *Byte* vol. 6, no. 8, pp. 378–387, Aug. 1981.

4. Baden, Scott, "Architectural Enhancements for an Object-Based Memory System," CS-292R Class Report, Computer Science Div., Dept. of E.E.C.S., University of California, Berkeley, CA, Fall 1981.

5. Cohen, Jacques; "Garbage Collection of Linked Data Structures", *ACM Computing Surveys* vol. 13, no. 3, pp. 341–367, Sept. 1981.

6. Deutsch, L. Peter, and Bobrow Daniel G., "An Efficient Incremental Automatic Garbage Collector," *Communications of the ACM* vol. 19, no. 9, pp. 522–526, Sept. 1976.