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

Implementing the Smalltalk-80 System: The Tektronix Experience

*Paul L. McCullough**
Tektronix, Inc.
Beaverton, Oregon

Introduction

The Tektronix Smalltalk-80 implementation went through a number of hardware and software phases. Our experience will probably prove to be similar to that of other research and prototype groups desiring to implement the Smalltalk-80 system. At best, we will point out some mistakes to avoid; at the very least we can provide an entertaining view of our successes and follies.

This paper gives an overview of our initial hardware and software environments and our initial implementation design. We then present a fairly detailed account of debugging our first system. Next, we describe the evolution of our hardware, software, and development tools. We conclude with some observations and conclusions about the Smalltalk-80 system and its implications for the future.

Readers should note that we were debugging both our implementation and the formal specification. Although we detected a number of errors in the formal specification, these errors have since been corrected and are discussed herein to provide historical perspective.

*Mr. McCullough is currently employed by Xerox Palo Alto Research Center, Palo Alto, California.

Initial Goals

Initially we had four goals for our Smalltalk-80 work:

- Learn about the Smalltalk-80 system, in particular the implementation of the virtual machine,
- Learn about programming in the Smalltalk-80 language,
- Report on errors in the book draft, and
- Implement the virtual machine, realizing that it would not be our final implementation.

Tektronix had no previous experience with object-oriented software, so we were very interested in having a system with which we could interactively program in the Smalltalk-80 language, and in studying the Smalltalk-80 virtual machine. As part of our agreement with Xerox, we were to use our implementation as a means to detect errors in the book draft and to identify ways in which the book might be made clearer. We realized that our initial implementation would suffer from performance problems, but felt that a timely implementation was more desirable than a high performance one.

Initial Hardware

Our initial hardware consisted of:

- Motorola 68000 processor (8 MHz)
- 4 MHz proprietary bus
- 768 Kbytes of RAM
- Tektronix 4025 terminal
- A microprocessor development system, used as a file server

The choice of hardware was based on the availability of a Tektronix designed 68000-based system, along with the need for a large, preferably linear, address space. We also wanted to use a processor amenable to the construction of personal workstations. The Tektronix 4025 terminal is a raster graphics terminal, primarily oriented toward drawing vectors. While our bitmapped display was being designed, the 4025 served as an interim display device. Because the initial Smalltalk-80 virtual image did not depend on the use of a file system, we only used the microprocessor development system as a file server to load and store virtual images.

Software Development Environment

Our virtual machine was developed in a cross-compilation environment using a DECSYSTEM-20. The bulk of the virtual machine was written in a dialect of the proposed ISO Standard Pascal. This particular dialect of Pascal supports the independent compilation of modules and produces assembly language files which are assembled and linked. The resulting executable file is downloaded to the 68000-based system over a 1200 baud serial line. Though the 1200 baud line was an obvious bottleneck, the Pascal software already existed on the DECSYSTEM-20 and we had no desire to port it.

Initial Software

According to Dan Ingalls: "an operating system is a collection of things that don't fit into a language. There shouldn't be one"[1]. Taking those words to heart, we chose to implement our virtual machine on a system that had no operating system. This choice meant that we could not rely on runtime services normally provided by an operating system and had to write those portions that we needed, such as routines to handle input/output to an RS-232 port and perform IEEE 32-bit floating point arithmetic.

Our software implementation team consisted of three software engineers. We chose to partition the programming task into three distinct parts:

- Object Memory Manager
- Interpreter and Primitives
- BitBlt

Object Memory Manager

The initial object memory manager was, for the most part, a strict translation from Smalltalk-80 to Pascal of the methods presented in the formal specification. During the translation phase, we noted four minor typographical errors in the draft of the book involving improper bit masks, or incorrect variable, constant, or method names. We chose to implement the reference-counting garbage collector. Later, because the image creates circular garbage, we added a simple, recursive, mark-sweep collector. The translation process took less than one week and resulted in a working memory manager that maintained a very clear boundary between itself and the rest of the virtual machine. As discussed below, this clear differentiation is both a blessing and a curse.

With minor changes due to different dialects of Pascal, we were able to run programs that tested the object memory manager on the DECSYSTEM-20 with its more sophisticated debugging and perform-

ance monitoring software. The test programs read the virtual image, then made calls to the various entry points in the memory manager. Then, with Pascal write statements and the debugger, we were able to examine the state of the object space and determine whether the memory manager was working correctly. These tests indicated several errors in the book's methods: for example, the method that determined where to find the last pointer of an object was incorrect for CompiledMethods, and the recursive freer needed an extra guard to prevent SmallIntegers from being passed off to the pointerBitOf: routine.

At this point, we were able to run test programs that created instances of classes, stored object pointers in other objects, destroyed such links and thus invoked the deallocation of objects, and performed compactions of the object space. Further testing demonstrated that the book's method for swapPointersOf:and: was also incorrect.

In order to speed up the performance of the deallocation portions of the memory manager, we modified the countDown: routine to call forAllObjectsAccessibleFrom:suchThatDo: only when the object's reference count was going to drop to zero, thus saving a procedure activation that was usually unnecessary.

A few other minor changes provided us with a memory manager that was tested on the DECSYSTEM-20. Thus, we had a great deal of assurance that the memory manager would perform correctly on the 68000-based system. Also, we felt that when problems were encountered in our implementation of the virtual machine, we could concentrate on looking for the problem in the bytecode interpreter or primitives, and could ignore the memory manager. In actual practice we made many, many runs of the virtual machine before any problems were found in the memory manager. We heartily recommend having a trustworthy memory manager.

Interpreter and Primitives

In parallel with the development of the object memory manager, we coded the bytecode interpreter and primitives. The interpreter and many of the primitives were written in Pascal. The arithmetic primitives were coded in assembly language in order to simplify the maintenance of the small integer tag bit.

The outer block of the interpreter consists of a call to an initialization routine and a loop containing a very large case statement that acts as the bytecode dispatcher. While the memory manager was a fairly literal translation of the book's methods, much greater care was exercised in the construction of the interpreter. Code that in the book was several message sends was often collapsed into a single Pascal statement. We included in our interpreter the capability of producing traces which duplicate those supplied with the virtual image by Xerox.

In order to give the reader a measure of the complexity of implementing an interpreter (in Pascal), we present the lengths (in

printer pages at 60 lines per page) of some of the major routines. These figures include the length of tracing code:

- Looking up a message, including the perform: primitive: two and one-half pages
- Sending a message (including cache lookup): one and one-half pages
- Executing the current method, including the primitives written in Pascal: twelve pages
- Returning a value from the active context: one and one-half pages
- The scan characters primitive (used for text composition): three and one-half pages
- Large integer primitives: four pages
- Process primitives: five pages

We strongly recommend that the first implementation of an interpreter be in a high-level language. By writing the virtual machine in a high-level language, implementors gain a more thorough understanding of the virtual machine as well as a much more quickly completed implementation.

BitBlt

The BitBlt primitive handles all graphics in the Smalltalk-80 system. Due to its importance, we decided to have one person concentrate on its implementation. The routines to implement BitBlt were written in assembly language and closely reflect the structure of the BitBlt methods in the book. To assist in the debugging of BitBlt, there are many conditionally assembled calls to the Pascal runtime print routines. The main BitBlt routine accepts one argument, the address of a Pascal record containing the various BitBlt parameters. When called, the routines perform the following actions:

- Clip the source parameters to the physical size of the source form
- Clip the clipping rectangle to the true size of the destination form
- Clip and adjust the source origin
- Compute the masks necessary for the logical operations
- Check for possible overlap of the source and destination forms
- Calculate the offsets for the starting and ending words
- Copy the bits as appropriate

Certain optimizations are performed for the special cases of clearing, setting, or complementing the destination forms. BitBlt is approximately 2 Kbytes of assembly code.

Summary of Runs

We maintained a fairly detailed log of our attempts to get the virtual machine up and running. The comments we made for each of these runs may be helpful to future implementors of the virtual machine. This summary should provide a sense of the types of errors and problems one should expect when implementing the virtual machine.

1. Reached the first send, then encountered an error in a debugging routine we had written.
2. Reached the first send again, encountered another error in a debugging routine.
3. Encountered a Pascal compiler bug.
4. Reached first send of the @ selector, and discovered that we had transcribed the constant for class SmallInteger incorrectly.
5. The method specified in the book for initializing the stack pointer of a new context was incorrect.
6. We forgot to initialize the sender field when creating a context.
7. In the book, the method returnValue:to: caused the reference count of the sender context to go to zero (thereby making the sender garbage) just before returning to that context. We had to explicitly increase the reference count of the sender context, perform the return, then explicitly decrement the reference count.
8. We had decided to implement the "common selector" bytecodes using full message lookup. Unfortunately, the method header for selector == in class Object did not specify the execution of a primitive. We patched the image to specify the correct primitive number.
9. The first conditional branch we encountered failed because we did not advance the instruction pointer past the second byte of the instruction.
10. We discovered that the source code for SmallInteger < did not specify a primitive, resulting in an infinite recursion. We patched the image again.
11. Discovered that other methods of class SmallInteger did not have primitives specified. We retrenched to executing the following selectors without lookup: class, ==, arithmetics, relationals.

12. Selector at: failed. Our fault, in the routine positive16BitValueOf: a " > " should have been a " < ".

13. Multiply primitive failed due to an assembly language coding error.

14. All relational primitives written in assembly language had an incorrect (and uninitialized) register specified.

15. Made it through the first trace. (Listings of four traces of the interpreter's internal operations were included with the first distribution of the virtual image. Subsequent distributions included three traces.)

16. The book's method for the primitive value: caused the stack to be off-by-one.

17. Once again, we found an error initializing the stack pointer of new contexts.

18. Again, the stack pointer is off. These three errors were caused by an incorrect constant in the book draft.

19. A message selector was not found. Another run is necessary to determine what happened.

20. At the beginning of execution for a block, the cached stack pointer is one too large. In the past, message sends and returns have worked because the routine that stored the stack pointer decremented it.

21. We had coded the at:put: primitive incorrectly: we forgot to have it return anything, hence the stack was off-by-one.

22. We incorrectly coded the at:put: primitive with an uninitialized variable.

23. The at:put: primitive had a > that should have been a >=.

24. The SmallInteger bitShift: primitive added in the SmallInteger bit, but should have Or'ed it in.

25. Interpreting lots of bytecodes, unfortunately not the correct ones. Apparently, we took a bad branch somewhere.

26. We found that the book's methods for the bytecode "push self" did not necessarily work for block contexts.

27. Almost through the fourth trace when the SmallInteger division primitive failed to clear the high-order half of a register. The error was detected by a Pascal runtime check.

28. Through the fourth trace when Sensor primMousePoint dies because of a clash between the interpreter and the Pascal runtimes.

29. We are well beyond the fourth trace when we discover that the method frame:window:para:style:printing: has a MethodHeader extension that specifies primitive number 0. We had assumed that an extension always specified a valid primitive number, but find that it may also be used to specify a method with more than four arguments.

30. We have changed all unimplemented primitives so that they fail, and now correctly handle primitive 0 in MethodHeader extensions. By now, we should have something up on the 4025 display, but do not. Investigating, we find that the book says that the bitmap for a Form is the first field, whereas the sources say it is the second field.

31. We are halftoning the display. We have to make a few adjustments to prevent overrunning the display. Halftoning will take a long time, approximately two hours. After a while, a runtime exception was raised by a Pascal support routine that contained a bug.

32. The "T" for the TopView window title tab is present on the display. Interpreter stopped after sending copyTo: to a SmallInteger.

33. We have disabled halftoning to the 4025, continuing with the study of the problem of sending copyTo: to a SmallInteger.

34. The problem is that the BitBlt primitive, copyBits did not return anything, thus forcing the stack off by one. Similarly, beDisplay, and beCursor did not return anything. We have added more display memory to the 4025.

35. Hurray! "Top View" window title tab is on the screen. Pascal runtime checks detected an out-of-range scalar while setting up arguments for copyBits. We have always assumed that BitBlt arguments are unsigned, but that is not so. We were told that BitBlt should do source clipping, so we will add that to BitBlt.

36. The entire "Top View" window is on the display, then erased. We eventually crashed because we are out of object space, but unsure why.

37. We are out of object space because the book's methods for superclass send was incorrect: another level of indirection is necessary.

38. We now have "Top View" window, Browser, and Transcript window on the display. Interpreter stopped when the mouseButtons primitive failed.

39. We turned on halftoning to see what would happen. This was a mistake because windows are halftoned black and then white. We decided to reload and try again without halftoning.

40. We have reached the idle loop, checking if the mouse is in any window. We changed the position of the mouse (by altering two memory locations) and placed it within the browser. The browser awoke and refreshed four of its panes. The fifth pane (code pane) caused an interpreter crash with a Pascal out-of-range error due to a minor bug in the mod primitive.

41. Great excitement! We have refreshed the window containing a "Congratulations!!" message. Eventually we crashed because the Float < primitive fails. The system tried to put up a Notify window, but had difficulty because of other primitive failures. However, it was able to put up messages in the Transcript window. For a system that is not yet fully implemented, it is amazingly robust. We noticed that certain BitBlt operations seem to put up incorrect information, then erase it. For example, putting up the "Top View" title tab, the text reads "Top Vijkl" for a short time, and the incorrect part is then repainted. Investigation showed the method computeMasks to have a < selector that should have been a <=, an error carried over from the book.

42. Generally poking around with the system. We have found that we need floating point primitives in order for scroll bars to work, so we have implemented all but the fractionalPart primitive. Rather than develop an IEEE Floating Point package, we acquired one from another group at Tektronix. We have also speeded up BitBlt by using 4010-style graphics commands with the 4025.

43. We have implemented object memory statistics to report the number of fetchPointers, storePointers, etc. performed. We have also added a lookup cache for faster message send processing. A clerical error in the caching routines crashes the virtual machine.

44. An uninitialized variable causes the cache to misbehave.

45. The cache is functioning well. Our initial algorithm is to exclusive-or the Oops of the receiver's class and the method, then extract bits 3-7 and index a 256 element array of 8 byte entries. The interpreter definitely runs faster with the cache. The cache consists of the Oop of the selector, Oop of the receiver's class, Oop of the method, the most significant byte of the method header, and one byte indicating either the primitive index or 0.

46. Tried a new hash function, shifting two bits to the left before the exclusive-or because we observed that the Oops of different selec-

tors in the same class are very similar to one another. Some speedup was noted.

47. Another hash function, this time adding the Oops rather than exclusive-oring them. No noticeable change. We did move the mouse to the first pane of the Browser and crashed the system when the interpreter passed a SmallInteger to the memory manager.

48. Further examination of the previous problem shows that we did not cut the stack back far enough after a value: message. This bug was carried over into our code from the book, but only appears when sending value: within an iterative loop.

49. We have fixed value:, now we need to write the perform: primitive.

50. We have installed perform:, but get an infinite recursion because the floating point package is *not* IEEE format. We will write one in Pascal.

51. With the new floating point code, we can now cut text, pop up menus, and so on. This is great!

At this point, we added some simple performance monitoring code. We counted the number and type of object memory references, the number of bytecodes executed, and information concerning the performance of the lookup cache. For each bytecode executed, an average of just under 10 object memory references were made. The majority were calls to fetchPointer:, then storePointer:, fetchByte:, and fetchClass:. The various lookup cache algorithms were found to perform either fairly well (50 to 70% hit rate) or very poorly (20% or worse hit rate). Evidently, caching algorithms either perform quite well or miserably.

Summary of Initial Software

We feel that we were able to implement a relatively complex piece of software in less than six weeks (that is, from nothing to a working system) in less than 60 runs for several reasons:

- We were fortunate to have very good software engineers.
- We had a well-defined task.
- Because it took so long to load the virtual image (about 10 minutes) from the file server and so long (again, 10 minutes) to download our virtual machine from the host, we were very careful in coding and in analyzing crashes. We were also sharing the hardware with another group, so we made good use of our time on the machine.
- The specification, though not without error, was well written.

The Second Virtual Image

About this time, we received the second virtual image from Xerox Palo Alto Research Center (PARC). With this image, the handling of primitive methods was much cleaner, access to BitBlt was improved, the kernel classes were rewritten, and a source code management system was added. Several significant changes to the virtual machine specification were made, with the intention that these would be the final modifications. The second image also made use of the process primitives, while the first image did not.

Because a general cleanup of our interpreter seemed a good idea, and because a fair amount of the interpreter needed to be changed to support processes and new primitive numbers, we rewrote much of it. A history of our runs for the second virtual image follows:

1. We got our "Initializing . . ." message, and the system crashed because we were trying to initialize the cursor frame buffer. Since our bitmap display was not yet available, the presence of this code was premature.
2. We are through one-third of the first trace, but a conditional branch bytecode branched the wrong way.
3. Several problems noted:
   - Metaclass names no longer print properly on our traces.
   - We encountered off-by-one errors in stack operations while handling bytecode 187 because we forgot to adjust the stack index.
   - We encountered off-by-one errors in stack operation for SmallInteger //.
   - Our trace does not print operands for SmallInteger * properly.
   - We need to carefully check the code for all stack operations.
4. M68000 stack overflow causes parity errors.
5. We are through trace 1, and three-quarters through trace 2 when Pascal detects an out-of-range scalar because the routine returnValue:to: returned to a deallocated block context. We had failed to increase a reference count.
6. We are almost halfway through trace 3 when we hit an unimplemented process primitive. We also noticed the primitive return of an instance variable did not pop the receiver, thus causing the stack to be off-by-one.

7. We are about 60% through trace 3 when we try to add nil to an instance of class Rectangle. Caused by our coding error: when a direct execution send fails, we fail to tidy up the stack pointer.
8. We find that we need to implement the process primitives.
9. BitBlt fails to clear the high-order bits of a register causing a crash on the 21380th message sent.
10. Sending the selector + to an Array fails. Stack is off-by-one because the copyBits primitive failed to return self.
11. We find that the resume: primitive does not work due to an uninitialized variable.
12. More problems with resume:, it fails to set a boolean.
13. More problems with the resume: primitive: the process to be resumed has nil as its instruction pointer because the initial instruction pointer is not set in primitiveBlockCopy.
14. The resume: primitive works finally! Unfortunately, the wait primitive does not because of an incorrectly coded branch.
15. The wait primitive works, and we are through the third trace correctly. We forgot to code the setting of the success boolean for primitive become:, so a notify window is created.
16. Fired up the system. We have executed more than 15,000,000 bytecodes and it is still alive!

In order to improve performance, we made many changes to the interpreter and the memory manager. Changes to the interpreter included the caching of absolute addresses in the interpreter, thus employing considerably fewer calls to the memory manager. For example, to extract the fields of a source form, rather than a fetchPointer call to the memory manager for every field, the interpreter merely cached an absolute address and stepped through a range of offsets. Within the memory manager, many procedure calls were replaced with macro calls that were expanded by a macro preprocessor. Not only did this save the overhead of procedure calls, but quite often allowed common subexpression elimination to occur, thus actually decreasing the amount of compiler-generated code.

We also sped up certain parts of the interpreter based on where we believed the interpreter was spending its time. With these optimizations, performance is approximately 470 bytecodes a second.

An observation: Utilizing a raw computer (that is, one without an underlying operating system) to implement a Smalltalk-80 system is a double-edged sword: on the one hand, you can place data structures and

code anywhere in the system, and you have complete control of the hardware. On the other hand, the lack of performance monitoring tools and underlying file systems can be a problem because it takes time to implement them, rather than just interfacing to them.

Second Version of the Hardware

At about this time, we added floppy disks to the system, as well as a utility program that could save and restore arbitrary memory locations on the disks, thus freeing us from the microprocessor development system file server. The 10 minute delay for the loading of a virtual image was reduced to about 45 seconds. A more dramatic change to the hardware was the addition of our bitmap display. No longer would we have to translate bitmap operations to vector drawing commands on the 4025, nor wait for a window to be halftoned. We also added a standard Tektronix keyboard and a mouse. In order for the mouse and keyboard (as well as portions of the Smalltalk-80 software) to work, we also added a one millisecond timer interrupt.

As part of another project, a new M68000 processor board was made available to us. Recall that the bus that we were using ran at 4 MHz, which introduced wait states into the M68000. The new processor board used a one longword data cache and a one longword instruction cache to reduce bus requests. This resulted in a 70% speedup in system performance, to approximately 800 bytecodes per second.

The Third Virtual Image

At this point, our goal became to build a virtual machine that was clearly faster (approximately 4000 bytecodes per second), but to do it quickly and at relatively low expense. The method we chose was to develop a performance analysis tool and, using the results of the measurements, to rewrite time consuming portions of the virtual machine in assembly language. The following sections summarize our findings and our techniques for speeding up the virtual machine.

Performance Modeling Tool

To monitor the execution of the virtual machine, we developed a simple analysis tool that was called by the one millisecond timer interrupt routine. Each time it was called, it stored the value of the interrupted

M68000 program counter. By changing a memory location, a routine could be activated to print a histogram showing ranges of program addresses, the number of times the program counter was found to be within the range, and the percentage of time spent within the range. The size of the address range for each line of the histogram was selectable by the user. We mapped routine addresses to these ranges so that the histogram showed time spent in each routine. This tool proved to be invaluable in speeding up the virtual machine.

Prior to utilizing this tool, we decided to measure how much time was spent in the interrupt service routine. The Smalltalk-80 virtual machine expects a timer interrupt every millisecond and the routine checks the mouse and keyboard motion registers. If a change has occurred, the routine makes note of the change so that the bytecode dispatch loop can create a Smalltalk-80 event. Like much of our virtual machine, our timer interrupt routine was initially written in Pascal. Because the interrupt routine has many basic blocks, and the optimizer of the Pascal compiler operates only upon one basic block at a time, the interrupt service routine spent a great deal of time reloading registers with previously loaded values. We discovered that an amazing 30% of the M68000 cycles were going to the interrupt service routine! One of the first optimizations that we performed was to take the Pascal compiler-generated code and to perform flow analysis on it. The new interrupt service routine consumed 9% of the M68000 cycles. Future plans call for hardware to track mouse and keyboard events, and for timers to interrupt the M68000 only when necessary (for example, when an instance of class Delay has finished its wait period).

The Results of Performance Monitoring

The performance monitoring tool showed us some statistics that were surprising to us (the percentage figures presented below do not include time spent in the interrupt service routine nor the performance monitoring tool). Approximately 70% of the M68000 cycles were being spent in the memory manager, 20% in the interpreter and primitives, and 10% in BitBlt. The bulk of the time in the memory manager was spent in only a few routines: fetchPointer:ofObject:, storePointer:ofObject:-withValue:, fetchClassOf:, countUp:, countDown:, and two sets of routines generally referred to as the recursive freer and the niller. Previous statistics we gathered had indicated that fetchPointer:ofObject: and store-Pointer:ofObject:withValue: were popular routines, but they were relatively short and (so it seemed) should consume relatively little processor time.

Looking at the Pascal-generated code, we felt that we could do far better with assembly language, and we recoded all memory manager routines that the interpreter and primitives could call directly. Recoding fetchPointer:ofObject: resulted in a 4.5% speedup. Next, we recoded storePointer;ofObject:withValue: and achieved an additional 13% speedup. The major difference between these two routines is in reference counting: when storing pointers, reference counts must be updated; when fetching pointers they do not. Although we had previously concluded that reference counting was an expensive operation, we now had measurements of just how expensive. After recoding in assembly language all the routines callable by the interpreter and primitives, the system was an aggregate 19% faster.

Next, we considered routines that were private to the memory management module. From the histograms, it was obvious that we spent a great deal of time initializing just-instantiated objects to nil pointers (or zeroes for non-pointer objects). This inefficiency again arose from the strict basic block analysis of the Pascal compiler. For the price of a procedure call to an assembly language routine, we were rewarded with a speedup of nearly 10%.

Another major change to the memory manager came in the area of the so-called recursive freer. When an object's reference count drops to zero, this set of routines is activated to decrement the reference counts of the object's referents and, should their counts drop to zero, recursively free them. The first attempt at speeding up this process was done in Pascal and resulted in nearly a 10% speedup. Later on, we rewrote the recursive freer again in assembly language achieving an additional speedup.

The instantiation of objects was also expensive because several procedure calls were made. We rewrote this code (still in Pascal), collapsing several procedures into one. Later, the instantiation routines were rewritten in assembly language.

Changes to the interpreter and primitives were done in an interesting manner. Recall that we had a functioning, albeit slow, interpreter. With the belief that it is far better to change one thing at a time, rather than everything at once, we modified a small portion of the interpreter and tested the change. Once the change was shown to be satisfactory, we changed another part of the interpreter.

Initially, we rewrote the bytecode dispatch routine, but, in keeping with our philosophy of small changes, none of the bytecode interpretation routines. Thus, the assembly language bytecode dispatch routine set a boolean indicating that the assembly language dispatch had failed and that the Pascal routine would have to take over. Then we added bytecode interpretation routines, more or less one at a time. Eventually, we were able to discard the Pascal dispatch loop and bytecode interpreters completely.

Once all the bytecode interpretation routines were completed, we turned our attention to the primitive routines. These changes were accomplished in a similar manner: initially, all assembly language primitives failed, forcing the Pascal-coded primitives to run. We would then select a primitive, code it in assembly language, and test it. Once it was found to be acceptable, we selected another primitive to re-code. Finally, the Pascal primitives were discarded. Rather than call high-frequency primitive routines, we included many of them in-line.

In order to save some procedure calls to the memory manager when instantiating objects, the interpreter first tries to directly acquire the new object off the free lists. If the attempt fails, the interpreter calls the memory manager. Such "fuzzing" of the line between the pieces of the virtual machine seem necessary to achieve acceptable performance on current microprocessors. This demonstrates how a clear boundary between the memory manager and the rest of the virtual machine is both a blessing and a curse.

The changes to the memory manager and interpreter eventually resulted in a 3500 bytecode per second system.

The Third and Fourth Images

Our technique of making incremental changes to the virtual machine enabled us to use a working system and to bring up new virtual images as they were received from Xerox. A log of the attempts to run the third image follows:

1. At Xerox, the display bitmap is simply an object in the object space. In our implementation, the display bitmap lives at a specific address, and we encountered a problem because this image sends the become: primitive to the current display object. We modified our code in the become: routine.
2. We encountered a Pascal subscript-out-of-range error. The routine that returns instance variables was coded incorrectly, due to an error in the book's specification.
3. There are some new primitives related to the Xerox implementation in the image. We modified our interpreter to understand them.
4. A bit of Smalltalk folklore: "If 3 + 4 works, everything works." We typed 3 + 4 into a window and executed it. It did not work because the SmallInteger size message returned the wrong result.

5. Executing "Circle exampleOne" causes infinite recursion because the graphics classes were coded incorrectly by Xerox. They had not noticed this problem because the Xerox implementation of primitive new: did not comply with the formal specification, allowing their code to execute.
6. The system is up and working.

The fourth image was brought up on the first attempt.

Some Observations

If we analyze the coding errors that we encountered in our various implementations, we find that most fall into the following categories:

- Off-by-one errors
- Failing to return the correct object, or failing to return any object (leading to off-by-one errors)
- Conditional branch reversals
- Errors in the specification

Perhaps the most painful part of debugging a virtual machine is finding the off-by-one errors. These errors typically arise in primitive handling and in the stack activation records. Certain primitives may fail, and Smalltalk-80 methods are expected to take over. During the development of the virtual machine, it is quite common to damage the object references on the stack or to misadjust the stack pointer resulting in off-by-one errors. When returning from a procedure call in many stack machines (the M68000 is an example), if the processor's stack has an extra argument or does not have a return value, the correct return address will not be found, and the processor will return to an erroneous location. The typical result is a system crash. In the Smalltalk-80 virtual machine, the return address (actually the sender field) of the activation record (an instance of either class MethodContext or class BlockContext) is always in a known place, and a correct return can always be made and the machine will definitely not crash. Nonetheless, the interpreter (or primitives) may have pushed an incorrect result value or left garbage on the stack. Only later will this type of error manifest itself. These errors can be time-consuming and relatively difficult to find.

Errors resulting from conditional branch reversals are common, and are not further discussed here.

We certainly found our share of errors in the specification of the Smalltalk-80 virtual machine. This statement should not be taken as an affront to the Software Concepts Group at Xerox PARC. They were both developing and documenting two complex software products (the Smalltalk-80 system itself and the underlying virtual machine), and it was our job to point out discrepancies. Indeed, they produced an amazingly well constructed software system, and future implementors should have fewer problems with their own implementations.

We have programmed very few application programs in the Smalltalk-80 language. However, we do have one very definite data point in this area. Our file system (see Chapter 16) was totally developed in the Smalltalk-80 system and in a relatively short time period. All debugging was done using the Smalltalk-80 system: we never used the Pascal or assembly language debugging tools.

A final observation: the routines collectively known as primitives are about one-third to one-half of the implementation effort. Bear this in mind when scheduling an implementation.

Conclusions

Our work with the Smalltalk-80 system has shown it to be a robust, well-engineered piece of software. The initial, albeit incomplete, virtual machine required six weeks of effort by three software engineers, primarily using a high-level language. This resulted in a slow but useable system. By monitoring where the virtual machine spent its time, we were able to construct a system with adequate performance. For first-time implementors, we heartily recommend a similar approach.

Without question, the Smalltalk-80 system will have a strong impact on many areas of computer science, including language design, system architecture, and user interfaces. Perhaps most importantly, the system and language cause the user to think about problems in new ways.

Acknowledgments

Many people contributed to our Smalltalk-80 effort. Allen Wirfs-Brock designed and implemented the Pascal-based interpreters and primitives and the initial assembly language enhancements. Jason Penney designed and implemented BitBlt, the floating point package, the floppy disk driver, and the assembly-enhanced interpreters. Joe Eckardt designed our excellent bitmap display and has made interesting modifications to the Smalltalk-80 code. Tom Kloos and John Theus designed

and maintained our M68000 system, as well as the interface to the mouse, keyboard, and floppy disks. Allen Otis graciously shared his hardware with us in the early days of the project and made some of the first measurements of the virtual machine. Larry Katz made many suggestions for the improvement of the book and served as our unofficial kibitzer during the implementation and provided much food for thought. We would like to acknowledge the various managers (Jack Grimes, Don Williams, Dave Heinen, George Rhine, and Sue Grady) who had the foresight and wisdom to allow us to work on the project. Glenn Krasner, of Xerox PARC, provided answers to our questions and provided us with ideas for speeding up our implementation. And, we would like to thank Adele Goldberg and the Software Concepts Group of Xerox PARC for including us in the book review and implementation process. Without them, we would have naught.

References

1. Ingalls, Daniel H. H., "Design Principles Behind Smalltalk", *Byte* vol. 6, no. 8, pp. 286–298, Aug. 1981.