High Performance Storage Reclamation in an Object-Based Memory System

Scott B. Baden

**Computer Science Division,
University of California,
Berkeley, California**

June 18, 1982

Abstract

Measurements of the Smalltalk Virtual Machine indicate that 20% - 30% of the time is spent reclaiming disused storage. Following the work of Deutsch, Bobrow, and Snyder [Deutsch and Bobrow 76] [Deutsch 82a] [Deutsch 82b] [Snyder 79] we introduce a strategy that reduces the overhead of storage reclamation by more than 80%. We discuss the design of hardware to support this strategy, and compare the hardware to a traditional software implementation. We conclude by suggesting directions for future research.

Table of Contents

List of Tables

List of Figures

1. Introduction

Smalltalk, an interactive system developed at Xerox PARC, is based on Alan Kay's Dynabook concept [Kay 69]. Dynabook is a *personal* information system comprising tools for communication and modeling of information [Goldberg 80]. Although Smalltalk possesses a powerful graphics interface, its most distinctive feature is its object-based representation of information. This abstraction mechanism is useful in the development of large software systems since it allows the user to interact with information while hiding the implementation details.

For many years Smalltalk was used solely in-house; recently, Xerox released the system. Berkeley is a member of the Smalltalk community, thanks to the efforts of both Xerox PARC and Hewlett Packard Laboratories. Last fall Hewlett-Packard provided a prototype version of the Smalltalk Virtual Machine (STVM); under the direction of Prof. David Patterson, students in the Computer Science department ported Smalltalk to a research VAX-11/780 and analyzed several aspects of STVM performance [Baden 81] [Cole 81] [Hagmann 81].

As a result of these studies we discovered that a large percentage of STVM execution time was spent reclaiming storage – 20% - 30%. In this report we show how to avoid storage reclamation overhead by changing the STVM and by providing some hardware assist to the host CPU.

First we introduce the Smalltalk system. This covers a discussion of common terminology, attractive characteristics, and performance bottlenecks. We focus our discussion on a few of these bottlenecks and present interesting results about them. We then propose enhancements and hypothesize their effects on performance. Finally, we verify our claims through experimentation and assess the benefits provided by the proposed strategy.

We will assume that the reader is familiar with the Smalltalk system; if not he should consult the special issue of *Byte* for a useful tutorial on Smalltalk [Byte 81].

2. Background

The execution environment of the Smalltalk system consists of a set of *objects* that communicate by sending *messages* to each other. The user in turn communicates with Smalltalk via a graphical display and a keyboard.

In Smalltalk the uniform representation of information is called an *object*. This is true for both user information (e.g. a phone number directory) and system information (e.g. process lists, windows, compiled code). Each object consists of a collection of *fields* and it has has an associated object-pointer, called an *oop*. The *object table* maps an object's oop onto its physical address. The system avoids the object table indirection for certain frequently accessed objects by cacheing their physical addresses in special registers.

Messages are polymorphic; that is, they can be understood by different kinds of objects. This means that an object's response to a message cannot be determined until runtime. Objects respond to messages in one of two ways: by making a call to a primitive STVM routine, or by executing a compiled *method*. A method is analogous to a procedure in traditional programming languages and primitive methods are analogous to single opcode instructions, e.g., arithmetic and special system instructions in a traditional instruction set architecture (The latter are included for completeness, such as I/O, or for efficiency, such as integer arithmetic).

Most of the Smalltalk system is written in Smalltalk itself; we refer to this portion of the system as the *Virtual Image*. The stack-based STVM interprets virtual instructions, or *bytecodes*, that the Smalltalk compiler generates. Bytecodes may be 1, 2, or three bytes long, although most bytecodes executed are 1 byte long (92% of the bytecodes executed by the STVM are 1 byte long). There are 5 groups of bytecodes:

(1) Stack/memory reference operations.

(2) Branches.
(3) Primitive Integer Arithmetic.
(4) Primitive methods.
(5) Sends and Returns.

Smalltalk reclaims storage automatically since the user cannot explicitly deallocate memory, as in UNIX [Joy and Babaoglu 79]. The STVM has been implemented on at least four computers, including the Alto [Thacker 79], the Dorado [Lampson 81], the Dolphin, and on the VAX-11/780 [DEC 81], under UNIX [Joy and Babaoglu 79] [HP 81] [Ungar and Patterson 82]. The fastest implementation runs 250 times faster than the slowest.

The Smalltalk system is attractive for a several reasons:

(1) *Ease of modification.* Since the virtual image is written in Smalltalk, the user may modify it to suit his or her own needs.

(2) *Windowing.* The windowing mechanisms provide a convenient "modeless" interface [Tesler 81]. For example, if the user is composing a letter within one window, then he could activate another window to check the progress of an executing program without first having to pass through several layers of software.

(3) *Polymorphic messages.* The use of polymorphic messages eases the task of program verification since every object can respond meaningfully to any message.

(4) *Ease of Debugging.* Smalltalk has a powerful window-based debugger. The debugger was easy to implement due to the uniformity of object representation.

(5) *Graphics.* Smalltalk provides full graphics capabilities on its bit-mapped display.

These advantages do not come without a cost; Table 1 shows a comparison of the "Towers of Hanoi" benchmark, executed with arguments (18,3,1,2), that was executed in:

(1) UNIX C on the VAX-11/780 and 11/750 [Ritchie 78].
(2) Berkeley Pascal on the VAX-11/780 and 11/750 (interpreted and compiled) [Joy 79].
(3) Berkeley Pascal on the 68000 (compiled).
(4) Franz Lisp on the VAX-11/780 and 11/750 (interpreted and compiled) [Foderaro 80].
(5) Xerox Smalltalk on the Dorado and the Dolphin.
(6) Berkeley Smalltalk on the VAX-11/780 [Ungar 82].[1]

Language	Machine	Running Time (sec)	Relative Time (Time/Dorado)
Smalltalk (Compiled)	Dorado	16	1.0
	Dolphin	205	13
	VAX-11/780[a]	294	19
C	VAX-11/780	14	0.9
	VAX-11/750	20	1.2
Pascal (Interpreted)	VAX-11/780	173	11
	VAX-11/750	338	21
Pascal (Compiled)	VAX-11/780	20	1.3
	VAX-11/750	30	1.9
	68000	20	1.1
FRANZ LISP (Interpreted)	VAX-11/780	505	32
	VAX-11/750	830	52
FRANZ LISP (Compiled)	VAX-11/780	52	3.2
	VAX-11/750	82	5.1

Table 1. Comparative Benchmark Times

[a] Berkeley Smalltalk [Ungar and Patterson 82].

[1] This system was written in C to run under UNIX. Both Xerox implementations are microcoded.

3. An Overview of Performance Limitations

In this section we outline two kinds of performance bottlenecks in the STVM: those owing to implementation specifics and those common to all existing implementations.

3.1. Implementation-Specific Performance Bottlenecks

There are two major bottlenecks:

(1) *Bytecode interpretation vs. compiliation.* Compiling Smalltalk directly into VAX native mode code would be more efficient than having the VAX interpret bytecodes.

(2) *Graphics.* We use an external display (Barco monitor and an AED-512 graphics interface). These are connected to the VAX via a 9600 baud serial line. The VAX does most of the graphics processing since the AED has primitive graphics processing capabilities. This is inefficient both from an I/O and from a computational standpoint (see [Cole 81]).

3.2. Fundamental Limitations

Three activities consume a large portion of execution time in several different implementations (see below):

(1) *Message sending.* Message sending overhead includes: run-time type checking, context changes and context memory management.

(2) *Object-table indirection* must be done in roughly 50% of object memory accesses.

(3) *Storage reclamation* consumes 20% - 30% of the execution time.

We consider improving only storage reclamation overhead as the other issues are beyond the scope of this report. In the next section we give a thorough discussion of storage reclamation in the STVM.

4. Storage Reclamation

Smalltalk uses the reference count technique to reclaim storage, augmented with occasional garbage collection. There are several descriptions of reference count algorithms in [Cohen 81] [Knuth 73] [Standish 80].

Smalltalk has three types of reference count operations, listed in order of increasing cost.

(1) Reference count requests that cannot be satisfied (the object cannot be reference counted).
(2) Reference count increment (*refI*).
(3) Reference count decrement (*refD*).

(1) is decided by a simple check of the object table. (2) or (3) occur depending on the outcome of the check in (1), and (3) is accompanied by a check for zero, since the object might be free.

Next we argue the case for and against the reference count technique.

4.1. Advantages of Using the Reference Count Strategy

(1) By reclaiming storage incrementally the system will never run out of free storage.

(2) The strategy avoids costly scans of main memory. Straightforward garbage collection algorithms scan main memory *at least* once during the marking phase and once during the gathering phase.

4.2. Disadvantages of the Reference Count Technique

(1) Overhead is associated with each pointer manipulation. On a storePointer operation the STVM decrements the reference count of the displaced pointer and increments the reference count of the replacing pointer. Other operations have extra overhead: pushes onto stack (which do a

storePointer), pops, and returns from messages.

(2) The reference counts are larger than mark bits and may overflow. If the reference count overflows, then the system must resort to an alternative reclamation scheme, such as garbage collection.

(3) The reference count technique cannot recover cyclic structures. Consider Figure 1. When →**A** is removed the reference counts of **A** and **B** are still non-zero, even though neither **A** nor **B** is accessible. The only solution is to garbage collect. In Smalltalk, the creation of cyclic structures happens often enough to require occasional garbage collection[2].

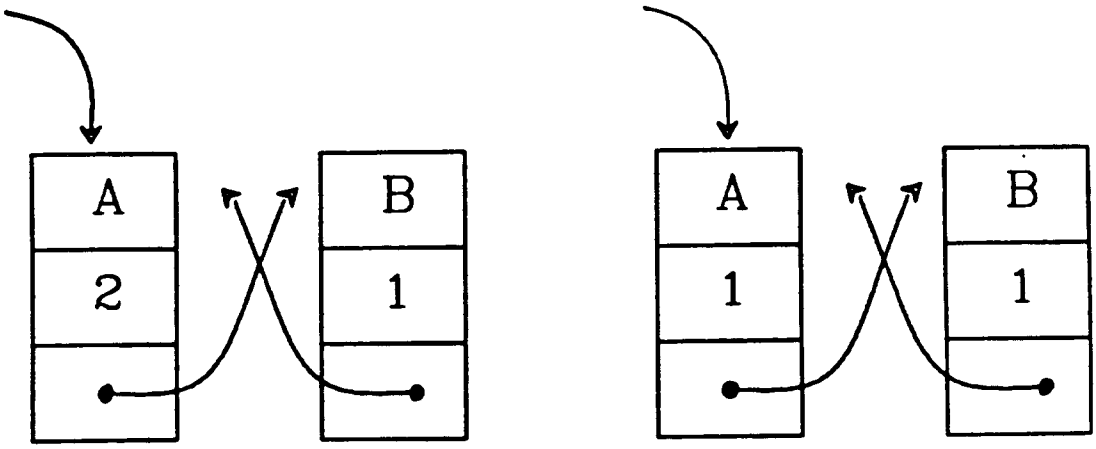

Figure 1. Attempting to free a cyclic structure.

(4) Recursive freeing gives rise to unpredictable work-loads. When dereferencing **A** (ref. Figure 2) we implicitly de-reference **B**, **C** and **D**. Whenever a

[2]From discussions with Peter Deutsch and Dan Halbert.

reference count reaches zero then the reference counts of all objects reachable from the object's fields (this is called "chasing the pointers") must be decremented recursively. Obviously it is not possible to predict *a priori* the amount of work required to do any single recursive freeing operation.

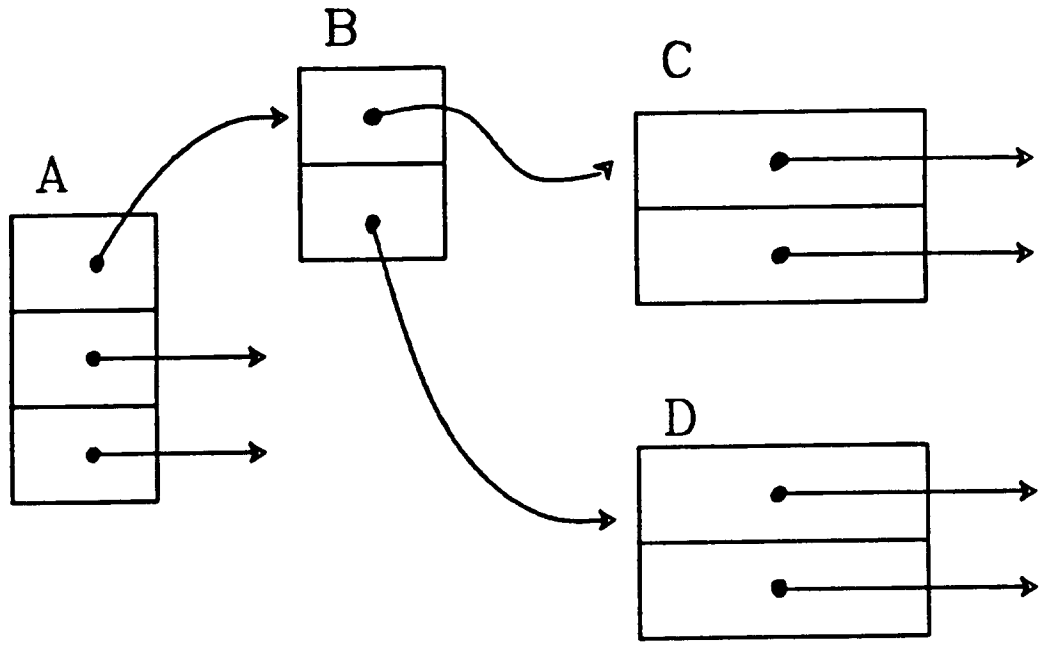

Figure 2. An example of recursive freeing.

5. Interesting Results

5.1. Where the STVM spends its time

Table 2 shows that Smalltalk spends most of its time dealing with message sends/returns and reclaiming storage.

Measure	%		
	HP VAX/UNIX	Dorado[a]	Dolphin[a]
Sends/Returns	27	38	34
Deallocation	24	20	4[b]
Allocation	5	6	15[c]
Other bytecodes	18	16	22
Primitives	14	12	16
Misc	12	5	11

Table 2. Where the STVM spends its time.

[a] These numbers provided by Peter Deutsch at the Computer Science Systems Seminar given at Berkeley in Fall quarter, 1981.

[b] Only represents recursive freeing. Reference counting is distributed throughout the other figures, owing to quirks in the Dolphin μ-engine.

[c] In the Dorado this activity is accounted for in sends and returns. This figure drops to 8% if this is taken into account.

5.2. Bytecode Frequency Analysis

Table 3 presents an analysis of bytecode frequencies taken from a Smalltalk session of 559K executed bytecodes (the data appear in Appendix A). The data are condensed into groups as listed in the Smalltalk specification: pushes, sends, pops, returns, and branches [Goldberg 80]. The pushes and pops access local and receiver variables. Since some bytecodes claim membership in two groups (e.g. POP AND JUMP bytecodes are in both the pop and jump group), the total exceeds 100%. Smalltalk changes contexts at a rate of one in every five bytecodes executed. This figure could not be calculated from Table 3 since

some sends will execute primitively and some primitive sends will fail. We determined the context changing rate by direct measurement of context activations and deactivations. Table 4 lists the different causes of context changes in Smalltalk.

Bytecode Group	Relative Frequency
Push	43.9
Pops:	22.0
Return TOS	*6.8*
Pop and Store	*7.4*
Pop	*2.0*
Pop and jump	*5.8*
Sends:	30.2
Arith	*11.7*
Special	*7.5*
Others	*11.0*
Returns:	8.6
ret stk top	*6.7*
return	*1.9*
jumps	7.9
conditional	*5.8*
unconditional	*2.1*
Totals	112.6

Table 3. Bytecode Group Frequencies

	Activations	Returns	Totals
Method	46139	45379	91518
Block	3168	3166	6334
Totals	49307	48545	97852

Table 4. Context Changes

5.3. Memory Allocation

Table 5 shows that the STVM allocates small context objects more often than any other class of object.

Object Type	Tally	Freq %
Sm. Meth. Context	45734	82
Point	5266	10
Sm. Blk. Context	1185	2
Lg. Meth. Context	406	1
Others	3178	6

Table 5. Object Allocation Frequencies

Table 6 relates the frequency of various storage reclamation activities to the number of bytecodes executed. Figures in Roman and boldface Roman script represent frequencies in units of operations per bytecode while *italicized* figures designate less frequent activities in inverse units, e.g. bytecodes per operation. Since certain objects are not reference counted (either Nil, True, False, a small integer or a permanent object), some reference count operations will not occur. Measurements show this happens 80% of the time for reference count decrements (*refD's*) and 60% of the time for reference count increments (*refI's*). The reader should note that although refI's are *requested* less often then refD's, the two requests are *satisfied* at the same rate. This is due to inefficiencies in the reclamation of method contexts which we discuss in §7.4.

Measure	Tally	Freq ops/bytecode *bytecodes/op*
Bytecodes	558889	-
All refD's	1935592	3.5
Actually Done	**410331**	**0.7**
Recursive refD's	926016	1.6
Actually Done	**142263**	**0.2**
Requested refI	1004897	2.9
Actually Done	**413905**	**0.7**
Frees:	54430	*10*
All Rec. Frees	6936	*81*
Method Contexts	*44953*	*12*
Other Objects	*9477*	*59*
Allocations	55769	*10*

Table 6. Summary Table

5.4. Reference Count Memory

In a traditional implementation (e.g. using off-the-shelf equipment) reference count operations must be done serially, limiting the speed of object memory references. As discussed in [Baden 81] it would be possible to add a *Reference Count Memory* (RCM) to allow reference counting to proceed in parallel with bytecode execution. The RCM contains the reference count of every object plus a tag bit that identifies permanent, e.g. unfreeable, objects (As shown in Appendix A, Table 14, the reference count needs to be only 3 bits long). The RCM modifies its contents in response to external requests generated by the STVM. To initiate a request the STVM writes to a distinguished memory location while providing the oop and command code on the memory data-bus. The STVM need never wait on the RCM since the latter is guaranteed to satisfy requests more quickly than the STVM can generate them. Since the STVM does not have to wait for a reply, all reference count operations cost a single memory write operation.[3]

[3] In traditional software implementations a read-modify-write operation must be done. The refD also includes a check for zero. We assume that the modify or check step takes one memory access time, so the RCM speeds up reference counting by 70%.

5.5. Context Registers

Hagmann observed highly localized call nesting in Smalltalk [Hagmann 81], with a maximum nesting depth of 74. Contexts are stored in a stack of windows as in RISC-I [Patterson and Séquin 81], although window overlap is not included. Table 7 shows that for a stack of 8 windows, only 2% of all sends and returns result in a window underflow or overflow (on an underflow or overflow the cache misses since it must transfer a window between the cache and the memory). Hagmann also shows that the optimal number of windows to save or restore on an overflow or underflow is only one. Since about 40% of all object memory accesses are to contexts [Blau 82], use of a stack of context registers would significantly reduce main memory traffic.

Depth (Windows)	Miss Rate
1	100
2	46
3	22
4	14
5	8
6	5
7	4
8	2
9	2
10	2
11	1
12	1
13	1
14	1
15	1

Table 7. Context Cacheing

Deutsch, Bobrow, and Snyder mention that most reference count operations concern stack and local variables [Deutsch and Bobrow 76] [Deutsch 82a] [Snyder 80]. They advocate avoiding the reference counting of these fields alto-

gether. We introduce a variation of their strategy in the next section.

6. Our Approach

In our scheme the STVM buffers, in registers, the n most recent context activations. Cached references are not normally accounted for; that is, they are not reference counted (we will discuss exceptions to this rule later). The system stores the contexts in FIFO order from a fixed region of physical memory, and it does not assign oops to these contexts unless the region overflows (on an overflow it assigns oops from the bottom of the stack).

Two pointers, the *Top Window Pointer* and the *Bottom Window Pointer* mark the physical memory bounds of cached contexts. The system uses these pointers, as in in RISC-I, to resolve references to contexts (e.g., is the context in memory or in the registers?) [Patterson and Séquin 81]. Attached to the main memory address and data busses is the Reference Count Memory (RCM), discussed in § 5.4.

We hypothesize that our cacheing strategy reduces reference counting activity by 80%. This savings improves the STVM performance by 16%-19% (see Table 2). Owing to the speedup offered by the RCM our strategy speeds up the remaining reference count operations by 70% which improves the overall performance by an additional 2%-3%. The 80% reduction in object allocations and deallocations accounts for another 4%-5%. We postulate a general STVM performance improvement of 22%-27%. Figure 3 shows the system configuration (we describe the ZCT in § 6.1).

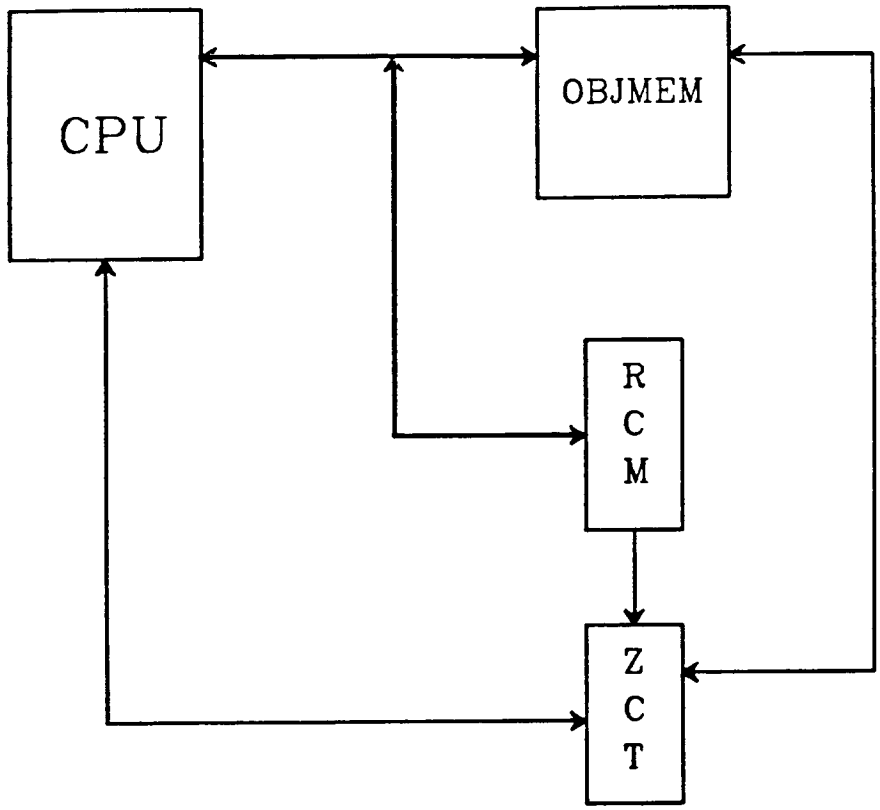

Figure 3. System Configuration

6.1. Difficulties

Although the savings look promising, we note that there are several complications associated with our strategy:

(1) Storage cannot be reclaimed during bytecode execution.

(2) Fields of overflowed and underflowed windows must be reference counted.

(3) Upward funargs, linearized message nesting, and sending messages to method contexts must be handled specially.

(4) The strategy places hardwired limits on message sending depth. This characteristic is inconsistent with the flexibility of the rest of the system.

To treat these problems we must change the strategy outlined earlier in this section.

6.2. Reclamation

When an object's reference count reaches zero it cannot be freed immediately since the reference count does not reflect cached references to it. Instead, the RCM marks a entry in a candidate "to be freed" table called the *Zero Count Table* (ZCT). The ZCT is a Content Associative Memory (CAM) that is 1 bit wide and 32K entries deep (one entry for each object).

The STVM reclaims storage periodically; it suspends normal execution, accounts for all of the cached references and frees any object marked in the ZCT (since the STVM cannot reclaim storage incrementally, we must ensure that it never runs out of storage between reclamation phases). To account for the references the STVM tells the RCM to refI its cached oops. We call this process *stabilization* [Deutsch 82a]. During stabilization the RCM may increment a zero reference count of an object – we call such a count a *spurious zero reference count*. The RCM clears the ZCT entry in response to a zero to one reference count transition, to prevent the STVM from freeing an object that had a false zero count. During the reclamation phase further storage may become free and the oops of these objects will be entered into the ZCT. When reclamation finishes, the system discounts all cacheable references and then it resumes suspended execution. To discount for the references the STVM tells the RCM to refD its cached oops. We call this process *volatilization* [Deutsch 82a].

Our implementation of the ZCT differs from the others. Since it uses a table instead of a queue [Deutsch and Bobrow 76] [Snyder 79] [Deutsch 82a] the table will not overflow. Another advantage lies in our treatment of the spurious zero reference counts. Owing to the overlap of the ZCT, RCM, and STVM, spurious zero counts do not slow down the system (spurious zero counts may be generated during execution as well as during the reclamation phase).

Since we implement the ZCT as a table we must speed up the time spent searching for a free object. This is why we use a CAM – the search time depends on the number of free objects, rather than on the number of possible objects. It is reasonable to build the ZCT as an associative memory since each entry is one bit wide.

6.3. Window Overflows and Underflows

Since the depth of the stack is limited, some sends will cause a window overflow and some returns will cause a window underflow. The system must stabilize the bottom window in the cache on an overflow and volatilize the top window in memory on an underflow. Hagmann's result tells us that only 2% of all message sends and returns cause the overflows and underflows [Hagmann 81] so we expect negligible overhead here.

Non-linearities in the context nesting sequence are caused by upward funargs (e.g. a message may return a block, the funarg, as a result), or by the debugger, which sends messages to contexts. Block activations[4] and messages to contexts[5] happen much less frequently than method activations and messages to other objects, so we assume that these activities are inexpensive (Deutsch has a set of schemes for linearizing the calling sequence and sending messages to contexts. He assures us that they are neither difficult nor expensive to implement [Deutsch 82b]).

[4] These account for only 6% of all activations, see Table 4.

[5] According to Peter Deutsch.

In fixing the region of memory devoted to contexts we do not place a hardwired limit on message nesting depth. Deutsch advocates migrating the least-recently used contexts into the object memory [Deutsch 82b]. We believe that this solution is reasonable; the system can store 1K 32-word contexts in only 32K words of memory. This number, in our experience [Hagmann 81], is much larger than most software will ever need. The system responds to extreme demands by a graceful degradation in performance. We assume this condition never arises.

We believe that the costs of our strategy will be unnoticeable, i.e., the benefits outweigh the costs. In the next section we provide experimental verification of our hypothesis.

7. Experiments and their interpretation

First we discuss our general methodology for gathering statistics, next, the actual experiments, and finally, our conclusions.

7.1. General Method

The monitored "session" involved the execution of system code and application code: browsing, compiling, and execution of simple messages (559K bytecodes were executed). As suggested, the initial portion of the trace (approximately 300K bytecodes) was discarded to avoid measuring transient behavior that occurs during system startup.[6]

The H-P code was written exactly as specified in the Smalltalk specification [Goldberg 80], hence it is highly modular and easy to change. We modified the VM code by inserting calls to special auditing routines that did not disturb the virtual machine. Appropriate oops, method headers, and so forth were written onto disk. Owing to its size (17 megabytes), the audit file was copied onto magnetic tape. A context cache simulator was written and ran directly from the

[6] Bob Ballance advises to gather statistics *after* the system can bring up a pop-up menu.

magnetic tape audit files. Complete documentation for the audit tape format appears in Appendix B.

Several activities were audited:

(1) Bytecode execution.
(2) Reference Counting.
(3) Deallocations and Allocations.
(4) Method Lookups.
(5) Context Activations and Returns.
(6) Primitive Successes Failures.

7.2. The Experiment

We measured the effects of volatilization in a non-volatilizing system. There are two experimental variables: stabilization period (in bytecodes) and context stack depth (in 32-word windows). We begin by introducing a set of criteria for assessing the validity of our approach. Next we discuss certain optimizations, not present in the H-P code, that reduce reference counting activity by 48% [Deutsch 82b] [Ungar 82]. Since the H-P code was written according to the specification in [Goldberg 80], we would not expect to find many of the optimizations in it. We mention the savings gained through these optimizations and compare the volatilizing system with the optimized system.

7.3. Figures of Merit

We evaluate our results by reporting the net savings in:

(1) Reference Counting,
(2) Allocation Activities, and
(3) Deallocation Activities.

The savings in (2) and (3) equal the number of allocated and deallocated method contexts. We do not include block contexts owing to the difficulties with handling upward funargs (§ 6.1). This will not effect or results because only 7% of all contexts allocated were block contexts. The savings in (1) equals the

number of reference counts of cached method context fields (e.g. in active and deactivated contexts, also in initialized, but inactive, contexts) minus a small overhead.

7.4. The Optimizations

Table 8 shows the reduction in reference counting due to four optimizations:

(1) Moving oops without doing any reference counting.
(2) Nilling the TOS on a pop.
(3) Distinguished use of non-reference countable objects, such as small integers, nil, true, false.
(4) Special treatment of the context class-defining object.

These optimizations cut reference counting in half.

Object	refI's			refD's		
	Requested	Done	Not Done	Requested	Done	Not Done
All objects	1004897	413905	590992	1935592	410331	1525261
Vol. Contexts	793973	301384	492589	1723823	302476	1420650
Savings	301083	104569	196514	875644	104569	771075
Oth. Objects	210924	112521	98403	211966	108855	104611
Savings	145177	97852	47325	145177	97852	47325
Tot. Savings	446260	202421	243839	1020821	201421	818400
(%)	44	49	41	53	49	54

Table 8. Savings in Reference Counting Due to optimizations

7.4.1. Moving Oops

On a move there is no net reference count change, but the specification says to *refI* the new reference and *refD* the old one. This is unnecessary in a system with an atomic move instruction. The STVM moves oops in four situations:

(1) When updating the active context oop register on a context change.
(2) On a pop and store into a literal or instance variable.
(3) On returns of TOS.
(4) When transferring the receiver and arguments to a newly activated context.

7.4.2. Nilling the Top of Stack on a Pop

In an unoptimized system the STVM reclaims context objects by examining every field, even if it were never used. On a pop, an optimized system refD's the top of the stack and replaces the old value with *nil*. This action is called *nilling* the top of stack. On a return every field above the top of stack will be nil, and hence, not subject to recursive freeing. The average method never uses more than 4 out of 12 context fields, (we ignore large method contexts, since they are used less than 1% of the time) so the optimization reduces context field reference counting by 67%. This optimization accounts for 25% of the savings. In a traditional software implementation the improvement is less significant, since the saved operations do not include an expensive reference count modification.

7.4.3. Use of Distinguished Values

Since nil, true, false, and small integers are not subject to reference counting, we can optimize distinguished use of these values. The STVM uses distinguished values in four scenarios:

(1) The ip and sp fields of contexts are small integers.
(2) "Distinguished push" bytecodes push: nil, true, false, -1, 0, 1, 2.
(3) "Distinguished return" bytecodes return: nil, true, false.
(4) Successful arithmetic primitives manipulate only small integers.

When instantiating a new object the system increments the reference of that object's class-defining object. This is unnecessary for contexts; the class-

defining object must never disappear or the system will fail.

7.5. Experimental Results – The Effects of Volatilization

We simplify the analysis of reference count savings by assuming a minimum cache depth of two windows. This condition forces most context references to always be in the cache, since they refer to contexts which are adjacent to one of the following:

(1) The home context.
(2) The sender (the caller context for blocks) context.
(3) A newly created context.

Only two of these three contexts need be cached at one time since the STVM disposes of the sender context when activating a new one. Although a reference from a deeply nested block context to its home or sender may not refer to an adjacent context, we ignore this case since it happens infrequently.[7] Table 9 shows that our volatilization reduces reference counting by 91% over the system which has been optimized as discussed in § 7.4.

Object	refI's	refD's	Totals
Volatile Contexts	492890	894427	1387317
Other Objects	65747	66789	132536
Savings (%)	88	93	91

Table 9. Savings Owing to Volatilization

Two events reduce these savings:

(1) Window underflows and overflows.
(2) Periodic stabilization.

[7] The number of reference counts of references to a block's home and sender context were insignificant. See Appendix A, Tables 19 and 20.

To measure these costs we simulate a register cache. The simulator stacks the active contexts (in memory and in the registers) and marks the bounds of the cached contexts with two pointers, the *top of stack* (tos) and *bottom of stack* (bos) pointer. For each window the simulator maintains four values:

(1) Current value of the context's Top-of-stack pointer (TOS), including the arguments and locals.
(2) Maximum value of TOS during the context's lifetime.
(3) Sender context (the caller if this is a block).
(4) Context size (large or small) and type (block or method).

On an overflow in the actual system, the STVM writes out part of the window to memory: the stack, header, and temporaries (an average of 7 fields out of 16). The RCM reference counts the fields while they are sent to memory. Owing to overlap, the cost of an overflow equals the number of fields written to memory and not on the number of reference count operations – hence we include the cost of writing the two header fields (*instruction pointer*, ip, and *stack pointer*, sp) which are not reference counted. Owing to linear context nesting the STVM can infer the sender from the top of stack pointer; hence, there is no need for a sender field and the context header is shortened to four fields.

On an underflow, the STVM restores the top window in memory into the registers; this is accompanied by concurrent *refD*'s. The cost of an underflow is the same as an overflow. As shown in Table 10, for an 8 window cache, the cost of underflows and overflows offsets our gains by only 2%.

If the system does not have an RCM then it must unload and refl the registers serially. Since we assume that a refl costs three memory access times, we expect an additional x2.3 increase in overflow and underflow penalties (the total penalties are: 13.8%, 4.6%, and 2.3% for 4, 8, and 16 windows respectively). The penalty is x2.3 rather than x3 since 2 of the 8 context fields (ip and sp) are not subject to reference counting.

Activity	Cache Depth (In Windows)		
	4	8	16
% overflows	19	3.0	0.0
% underflows	20	3.0	0.0
% degradation	6	2.0	1.0

Table 10. Cost of Overflows and Underflows

During reclamation the STVM accounts for all *reference-countable* cached references so it does not send the ip and sp fields to the RCM. When done it revolatilizes the cache. This overhead degrades the savings by less than 1% (see Table 11), and, as we show later, it has a negligible effect on reclamation latency time.

Volatilization Period (bytecodes x 1000)	Cache Depth (Windows)					
	4		8		16	
	Penalty (Ops)	Penalty (%)	Penalty (Ops)	Penalty (%)	Penalty (Ops)	Penalty (%)
8	3354	0.2	6707	0.4	13414	0.8
16	1676	0.1	3354	0.2	6707	0.4
32	838	0.06	1676	0.1	3354	0.2
64	419	0.03	838	0.06	1676	0.1
128	210	0.01	419	0.03	838	0.06

Table 11. Losses owing to Periodic Stabilization and Volatilization

Table 12 gives the time (in μs) taken to account for references in the cache. We assume that the cache is full (e.g. all windows in use), and that it takes 125 ns to send a cached field to the RCM. From our observations we noted that the average method uses 3 stack and temporary fields; two others (the method and receiver) must also be reference counted during the reclamation phase.

Cache Depth (Windows)	4	8	16
Accounting Time (μs)	4.2	8.4	17

Table 12. Cost of Periodic Volatilization and Stabilization

Besides register examination, reclamation includes pointer chasing (number of recursive *refD*'s done), exclusive of those done to method contexts, plus object deallocations. Since this overhead is deferred (i.e. it is done incrementally in a non-volatilizing system) it does not affect reference counting activity but increases reclamation latency time. To calculate reclamation latency we assume that the STVM executes 128K bytecodes/second and that it takes 400 ns to read out and nill a memory location. In addition we assume that it takes one 400 ns cycle to both locate a free object in the ZCT and to read and modify the object table entry corresponding to that object. This is reasonable if we pipeline ZCT scanning and OT access. Table 13 lists the latency period for different combinations of the experimental variables. In all cases the latency time is less than 0.1% of the execution time between reclamation phases (the time spent stabilizing and volatilizing the cache is insignificant compared to the time spent reclaiming), so it does not slow down the system appreciably. A system implementor will adjust the reclamation period to suit any response time constraints, e.g., the 6.9 ms latency due to a 128K bytecode reclamation period may be too long.

Reclamation Period (K BC)	Execution Time (ms)	Frees	Fields Chased	Reclamation Time (ms)	Overhead (%)
8	63	136	811	0.4	0.6
16	125	271	1622	0.7	0.6
32	250	543	3245	1.5	0.6
64	500	1085	6489	3.0	0.6
128	1000	2170	12978	6.1	0.6

Table 13. Reclamation Latency

8. Evaluation

Volatilization reduces storage reclamation time by 80% - 90% (82% of the object allocations, 90% of the reference counts). The strategy incurs an insignificant overhead and does not have an appreciable affect on response time. Unusable storage accumulates at a very low rate, so the system will not run out of storage.

Our conclusions were based on a 400 ns processor cycle time. If a faster or slower one is available then an implementor need only adjust the reclamation period – he does not have to change the cache depth. For interactive use, we recommend 128K bytecode periods. At this rate the accumulation of unusable storage will still be reasonable – only 5K words (2 word points account for most of the deallocated objects), and the latency time is short (6.1 ms). We recommend an eight window cache. Eight Windows are far superior to four, but we appear to reach a diminishing rate of return at 8; 16 windows do not improve performance significantly.

9. Conclusions

By treating context objects specially it is possible to realize a tremendous savings in dynamic storage reclamation time. The hardware is inexpensive and a 20% overall improvement in performance seems likely. The savings could be as

high as 30%, depending on the implementation.

Although the strategy looks attractive we must caution the reader that we have not dealt with two significant issues: (1) How to reclaim cyclic garbage and (2) how to handle the special cases (e.g. mentioned in § 6.1, items (3) and (4)). In the first case we must resort to garbage collection or develop a scheme to keep track of cycles [Deutsch 82b]. In the second case the system implementor must weigh the technique's benefits against its complexity. The special cases are not straightforward and their complexity may make our scheme appear less attractive, i.e., we may want to replace the reference count technique by garbage collection. No clear-cut answer has been found. The low-level details of implementing a volatilizing Smalltalk system are under development at Xerox.

10. Future Directions

There are a number of issues which would improve the reliability of our numbers:

(1) We must justify our implementation of the Zero Count Table by measuring spurious zero reference counts.

(2) We must deal with process switches properly. At present we use only one stack and do not consider the cost of flushing the registers on a context swap. Owing to our results regarding periodic stablization these times should not be significant. The omission of this detail probably has an adverse effect on the maximum context nesting depth – we measured a maximum depth of 794 contexts. This is much higher than previous observations, and runs contrary to our intuition. Since we were very careful in metering the code and in writing the simulator, we suspect that process switching caused the problem.

(3) We may find that alternative reclamation strategies, such as garbage collection, may be easier to implement and provide similar gains.

(4) We have ignored most of the bottlenecks mentioned in § 3.1 and 3.2, since they do not affect our results. We would expect that changes in either the compiler [Hagmann 81] or the bytecode architecture [Deutsch 82b] would affect our results dramatically.

11. Postscript and Acknowledgements

This project has given me the opportunity to study a non-traditional computing environment, both its pitfalls and its strengths. Smalltalk employs powerful abstraction mechanisms, and as we have seen, these incur a significant performance penalty. Throughout the last six months I have studied several tradeoffs and have considered multi-disciplinary aspects of system design: hardware, architecture, operating systems, graphics, tools, performance analysis, and even a little psychology about interactive computing.

I'd like to thank my colleagues here at Berkeley: Ricki Blau, Clem Cole, John Foderaro, Robert Hagmann, Peter Kessler, Ed Pelegri, Richard Probst, Russell Wayman, and especially David Ungar, with whom I spent many enjoyable evenings discussing Smalltalk. At Xerox: Adele Goldberg, Dan Ingalls, Ted Kaehler, Glen Krasner; also Peter Deutsch, who has shared with me and my colleagues a good deal of his insight into the Smalltalk system. At Hewlett-Packard, I'd like to thank Bob Ballance, Ted Laliotis, and Jim Stinger. Without their help this work would never have been possible.

Two faculty members have been instrumental in the execution of this project: Yale Patt and David Patterson. Yale kindly offered his time as second reader. Dave made Smalltalk a reality at Berkeley. As my advisor he succeeded in his mission: his constant coaching sharpened my resolve to complete the work and he provided a good deal of moral support all the way through to the end. I am very grateful for his time and for his consideration throughout the project.

Bibliography

[Baden 81]
Scott B. Baden, "Architectural Enhancements for an Object-Based Memory System", EECS 292R Class project, Fall 1981, Univ. of California, Berkeley, California.

[Blau 82]
Ricki Blau, "The Virtual Memory Performance of Smalltalk under VM/UNIX", EECS 258 Class project, Winter 1982, Univ. of California, Berkeley, California.

[Byte 81]
Special Smalltalk Issue, *Byte*, **6**(8), August 1981.

[Cohen 81]
Jacques Cohen, "Garbage collection of Linked Data Structures", *ACM Computing Surveys*, **13**(3):341-367, Sept. 1981.

[Cole 81]
Clement T. Cole, Eduardo Pelegri, David M. Ungar, Russell J. Wayman, "Limits to Speed: A Case Study of a Smalltalk Implementation Under VM/UNIX", EECS 292R Class project, Fall 1981, Univ. of California, Berkeley, California.

[DEC 81]
"Vax Architecture Handbook", Digital Equipment Corporation, 1981.

[Deutsch and Bobrow 76]
L.P. Deutsch, and D.G. Bobrow, "An Efficient Incremental Automatic Garbage Collector", *CACM*, **19**(9):522-526, September 1976.

[Deutsch 82a]
L. Peter Deutsch, Lecture given to the Berkeley Smalltalk Seminar, Feb. 5, 1982.

[Deutsch 82b]
L. Peter Deutsch, *Private communications*.

[Foderaro 80]
John K. Foderaro, "The FRANZ LISP Manual", University of California, Berkeley, California, 1980.

[Goldberg 80]
Adele Goldberg, D. Robson, D. H. H. Ingalls, "Smalltalk-80: The Language and its Implementation", Draft Edition, Xerox Corp., 1980.

[Hagmann 81]
Robert Hagmann, "Some Smalltalk Performance Measurements", EECS 292R Class project, Fall 1981, Univ. of California, Berkeley, California.

[HP 81]
Private communications with the Hewlett-Packard Smalltalk Group.

[Joy 79]
William N. Joy, Susan L. Graham, Charles B. Haley, "Berkeley Pascal User's Manual: Version 1.1 – April, 1979", Computer Science Division, University of California, Berkeley, California.

[Joy and Babaoglu 79]
W. N. Joy, O. Babaoglu, "UNIX Programmer's Manual", November 7, 1979", Computer Science Division, University of California, Berkeley, California.

[Kaehler 81]
Ted Kaehler, "Virtual Memory for an Object-Oriented Language", *Byte*, **6**(8):378-387, August 1981.

[Kay 69]
Alan Kay, "The Reactive Engine", Doctoral Dissertation, University of Utah, September 1969.

[Knuth 73]
Donald Knuth, "The Art of Computer Programming", vol. 1, Addison-Wesley, Reading, Mass., 1973.

[Krasner 81]
G. Krasner, "The Smalltalk-80 Virtual Machine", *Byte*, **6**(8):300-320, August 1981.

[Lampson 81]
Butler P. Lampson, Kenneth A. Pier, Gene A. McDaniel, Severo M. Ornstein, Douglas W. Clark, "The Dorado: A High Performance Personal Computer", CSL-81-1, Xerox PARC, Palo Alto, California, January 1981.

[Patterson and Séquin 81]
David A. Patterson, Carlo H. Séquin, "RISC I: A Restricted Instruction Set VLSI Computer", Eighth Symposium on Computer Architecture, Minneapolis, Minnesota, May 1981.

[Ritchie 78]
Dennis M. Ritchie, "The C Programming Language – Reference Manual", Bell Laboratories, Murray Hill, New Jersey, 1978.

[Snyder 79]
Alan Snyder, "A Machine Architecture to Support and Object-Oriented Language", Ph.D. Dissertation, MIT Laboratory for Computer Science, MIT/LCS/TR-209, March 1979.

[Standish 80]
Thomas A. Standish, "Data Structure Techniques", Addison-Wesley, Reading, Mass., 1980.

[Tesler 81]
Larry Tesler, "The Smalltalk Environment", *Byte*, 6(8):90-147, August 1981.

[Thacker 79]
C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs, "Alto: A Personal Computer", CSL-79-11, Xerox PARC, Palo Alto, California, 1979.

[Ungar 82]
David M. Ungar, *private communications*.

[Ungar and Patterson 82]
David Ungar and David Patterson, "Berkeley Smalltalk: Who Knows Where the Time Goes?, Or, Optimization Considered Essential", Dept. of Computer Science, University of California, Berkeley, 1982, *in preparation*.

Appendix A: Miscellaneous Data

From the following table we see that a 3 bit reference count is very reasonable since only 1% of all allocated storage will become non-reference countable owing to overflow.

Max Count	Tally	Cum. %	Storage	Cum. %
1	9851	39	177318	45
2	11237	84	189805	94
3	1070	89	9442	96
4	347	90	6186	98
5	522	92	3789	99
6	922	96	1835	99
7	254	97	592	99
8	78	97	211	99
9	302	99	1209	100
>9	361	100	809	100

Table 14. Cost of Reference Count Overflows

We measure the cost of an overflowed count in terms of the amount of additional storage which is no longer reclaimable (e.g. the size of the object). To gather these data, some auditing code was added to the *refI* routine.

Sums were maintained for each reference count from one to thirty-one (column four); when the reference count for an Oop was bumped to x then the sum for row x was increased by the size of the object (in sixteen bit words). The sum of all the rows from x-1 through 31 equals the amount of storage lost if the reference count is limited to x-1. The figures we are most interested in are the cumulative percentages (column five)-- what percentage of the storage referenced will remain reclaimable given a limitation x on the reference count (x+1 is the overflow value). For instance, if the maximum reference count were one, then only 4% of the storage could be reclaimed.

There are some additional figures also shown in the table: column two tells us how many times a particular reference count was reached, column three shows a cumulative percentage of each tally (what percentage of refI's involved an increment to a value of x or less). These figures are less useful than the weighted tallies mentioned in the previous paragraph.

Column five tells us that a reference count of six will be sufficient to reclaim 99% of object memory that is used (since the reference count won't overflow), assuming no cyclic structures. Two other values, zero and seven, must be reserved for zero and overflow conditions. Thus we need only three bits of reference count!

BC Groups	Tally	Freq	Cum %
pshTmpV	108354	19.4	19.4
push	66228	11.8	31.2
sndArith	65290	11.7	42.9
pshRcvV	46648	8.4	51.3
sndSpec	41677	7.5	58.7
retst	37731	6.8	65.5
send0	34926	6.2	71.7
PopJF	22273	4.0	75.7
pstTmpV	19375	3.5	79.2
send1	15385	2.8	81.9
pop	11141	2.0	83.9
return	10814	1.9	85.9
popJLF	10361	1.8	87.7
pstRcvV	8545	1.5	89.2
pstExt	8445	1.5	90.8
pshLitV	8440	1.5	92.3
jmpLong	8366	1.5	93.9
send2	7545	1.4	95.1
pushExt	7172	1.3	96.4
pshLitC	6891	1.2	97.6
storExt	4813	0.9	98.5
jump	3542	0.6	99.1
sendExt	1843	0.3	99.4
pushAct	1496	0.3	99.7
sndSExt	1395	0.2	100
send2Ext	180	0.0	100
dupe	13	0.0	100
popJLT	0	0.0	100
sndS2Ext	0	0.0	100
Total/avg	558889	100	100

Table 15. Bytecode Group Tallies Sorted by Frequency

Table 16 lists the access frequencies of variables by variable type – the majority of accesses are to temporaries and most of the rest are to the receiver fields.

Variable Access	% of all bytecodes	% of Var. Accesses
Temp	22.9	60
Receiver	9.9	26
Literals	2.7	7
Variable	*1.5*	*4*
Constant	*1.2*	*3*
Extended	*2.7*	*7*
Totals	*40*	*100*

Table 16. Variable Access Frequencies

Table 17 gives a breakdown of message argument counts. Most sends have few arguments, the average is 0.7 per send. These data are derived from the method header of the executed method and *not* from bytecode statistics, since some send bytecodes invoke a primitive method (48% according to our figures). 'Rest' designates extended send bytecodes whose argument counts could not be determined from the trace.

Arg. Count.	% of all sends
0	46
1	37
2	11
3	2
4	0
rest	2
Total	98

Table 17. Message Argument Count Frequencies

Size	Small Contexts			Large Contexts		
	Tally	Cum. %	Savings	Tally	Cum. %	Savings
1	2985	6	35820	0	0	0
2	5703	19	62733	0	0	0
3	11979	45	119790	0	0	0
4	11889	71	107001	0	0	0
5	6028	84	48224	0	0	0
6	3661	92	25627	0	0	0
7	1122	95	6732	0	0	0
8	560	96	2800	0	0	0
9	510	97	2040	0	0	0
10	510	98	1530	0	0	0
11	107	98	214	5	1	110
12	356	99	356	14	5	294
13	332	100	-	1	5	20
14	-	-	-	174	48	3306
15	-	-	-	87	69	1566
16	-	-	-	42	80	714
17	-	-	-	1	80	16
18	-	-	-	6	81	90
19	-	-	-	0	81	0
20	-	-	-	69	98	897
21	-	-	-	0	98	0
22	-	-	-	0	98	0
23	-	-	-	0	98	0
24	-	-	-	0	98	0
25	-	-	-	0	98	0
26	-	-	-	1	99	7
27	-	-	-	0	99	0
28	-	-	-	5	100	25
29	-	-	-	0	100	0
30	-	-	-	0	100	0
31	-	-	-	0	100	0
32	-	-	-	0	100	0
Totals	45742	100	412867	405	100	7045

Table 18. Savings due to nilling the Top of stack on a Pop

Object	refI's			refD		
	Requested	Done	Not Done	Requested	Done	Not Done
MC Home	492443	145017	347426	492443	153950	338493
BC Home	5022	329	4693	5022	245	4777
New MC	264396	147678	116718	264396	0	264396
Dct MC	0	0	0	877110	135176	741934
Dct BC	0	0	0	45342	1937	43405
Act Blk	32112	8360	23752	32112	9809	22303
Caller	0	0	0	7201	1859	5342
Unknown	153819	99810	54009	98149	98050	99
Oth Con	22170	5890	16280	50238	1223	49015
Oth Obj	3612	1698	1914	16546	6933	9613
Rcvr	21083	5123	15960	21127	1149	19978
Point	10240	0	10240	25906	0	25906
totals	1004897	413905	590992	1935592	410331	1525261

Table 19. Reference Counting: refI's vs refD's

Object Type	Top-Level			Recursive		
	Requested	Done	Not Done	Requested	Done	Not Done
MC Home	492443	153950	338493	0	0	0
BC Home	5022	245	4777	0	0	0
New MC	264396	0	264396	0	0	0
Dct MC	14963	1605	13358	862147	133571	728576
Dct BC	45342	1937	43405	0	0	0
Act Blk	32112	9809	22303	0	0	0
Caller	0	0	0	7201	1859	5342
Unknown	98149	98050	99	0	0	0
Oth Con	22170	0	22170	28068	1223	26845
Oth Obj	3612	1323	2289	12934	5610	7324
Rcvr	21127	1149	19978	0	0	0
Point	10240	0	10240	15666	0	15666
totals	1009576	268068	741508	926016	142263	783753

Table 20. Refd's: Recursive vs Top Level

Activity	tally	refl Savings	refD Savings	Total
Context Changes	97852	1	1	195704
pop and Store	36365	1	1	72730
(ref. countable)	10156	1	1	20312
(not ref countable)	26209	1	1	52418
Return Stk Top	37721	1	1	75442
Arg + Rcvr Xfer On a Send	85294	1	1	170588
(ref. countable)	56892	1	1	113384
(not ref countable)	28602	1	1	57204
Nilling the TOS on a Pop	412867	0	1	412867
distinguished pushes	33124	1	0	33124
distinguished returns	3667	1	1	7334
Successful Arith Prims	59959	1	2	179877
Context Header Examination	44953	0	4	179812
Context Allocations	47325	1	1	94650
Total				1660608

Table 21. Savings Owing to the Optimizations

Activity	Stack Depth (In Windows)		
	4	8	16
Overflows (% of Sends)	8928 (19)	1448 (3.0)	157 (0.0)
Overflowed Fields (Avg/ovfl)	15641 (1.8)	1759 (1.2)	182 (1.2)
Overflowed Header fields	35712	5792	604
Underflows (% of Returns)	8921 (20)	1442 (3.0)	153 (0.0)
Underflowed Fields (Avg/unfl)	15636 (1.8)	1756 (1.2)	180 (1.2)
Underflowed Header fields	35684	5768	592
Total Fields	87037	13319	1378

Table 22. The Cost of Underflows and Overflows

Appendix B: Audit Tape Format

The audit tape consists of a stream of variable-length records. Each record begins with a 1 byte alphabetic activity code. To simplify the description we group the activities into six classes:

(1) Bytecodes.
(2) Reference Counting.
(3) Deallocations.
(4) Allocations.
(5) Message Sending.
(6) Miscellaneous.

1. Bytecodes

case 'A': 1 byte bytecode follows.

case 'B': 2 byte bytecode follows.

case 'C': 3 byte bytecode follows.

2. Reference Counting

case 'U': /* refI */

container-code || reference-code || reference-count

case 'u': /* unsatisfied refI request */

container-code || whynot

case 'D': /* Top-level refD */

container-code || reference-code || whynot || reference-count

if reference-count == 0 include a deallocation record.

```
case 'd': /* Recursive refD */
        container-code || reference-code || whynot || reference-count
        if reference-count == 0 include a deallocation record.

case 'W': /* Top-level unsatisfied refD request */
        container-code || whynot

case 'w': /* Recursive unsatisfied refD request */
        container-code || whynot
```

Container-code and reference code designate the class, and optionally, the oop of, respectively, the containing and referenced object. The codes are:

cases 'H/h': Method/Block context home.

case 'B': Active block context.

case 'S': Method context sender.

case 's': Block context sender.

case 'C': Block context caller.

case 'R': Receiver.

case 'M': Method.

case 'X': Other context (may be a newly created, but inactive context), oop (2 bytes).

case 'O': Other object (may be a newly created, but inactive context), oop (2 bytes).

case 'P': Point.

case 'Z': Unknown (some oops are contained by special registers,
e.g. activeContextOop register, rather than by objects).

"Whynot" is a 1 byte code designating the cause for not reference counting:

case 'N': Nil.

case 'T': True.

case 'F': False.

case 'I': Sm. Int.

case 'O': Miscellaneous.

The reference count value is 1 byte long; two values are reserved:

case 254: overflow.

case 255: negative reference count.

3. Deallocations

These records always follow a refD operation, so no activity code is needed. A deactivation record includes the pointers/no pointers designation:

case 'N': does not have pointers.

case 'P': has pointers.

Also included: a two byte size value, and the objects' class designation.
The scheme for encoding the class designation is the same as for

encoding the reference or container.

4. Allocations

case 'x': allocation with pointer.

case 'y': allocation with words.

case 'z': allocation with bytes.

Each of these codes is followed by its size (2 bytes),
a class-code, and, in some cases, an oop:

cases 'M'/'m': large/small method context.

cases 'B'/'b': large/small block context.

case 'P': Point.

case 'D': Other object, followed by its oop (2 bytes).

5. Message Sending

case 'b'/'m': Blk/Method Context Activation

Followed by the oop of the new context (2 bytes), the maximum
value of the old SP (1 byte), and the current value of the old SP.

case 'c'/'M': Blk/Method Return
Followed by the oops of the target context (e.g. where to return to),
the current context (2 bytes), and the maximum and final
values of the SP.

case 'N': Method lookup

Followed by the method's oop, the receiver oop, the method header, and the header extension if necessary (all 2 bytes long).

case 'E': Error-- selector not found.

cases 'n'/'s': Generation of a new method/block context.

Followed by the oop (2 bytes) and the size (1 byte).

Size codes are:

case 18: Large.

case 38: Small.

deafult: error.

cases 'Q'/'R': Initial failure of an arithmetic/common primitive.

cases 'S'/'F': Primitive success/failure.

case 'X': Fields transfer of arguments and receiver.

Followed by size of transfer and number of fields that could not be reference counted (both 2 bytes).

case 'Y': Fields xfer of arguments from an array.

6. Miscellaneous

case 'i':

Synchronization marker.

cases 'J'/'K': Reference countable/non-reference countable oop transfer on a pop and store bytecode.