A Comparative Performance Evaluation of Write Barrier Implementations*

Antony L. Hosking J. Eliot B. Moss Darko Stefanović

Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

Generational garbage collectors are able to achieve very small pause times by concentrating on the youngest (most recently allocated) objects when collecting, since objects have been observed to die young in many systems. Generational collectors must keep track of all pointers from older to younger generations, by "monitoring" all stores into the heap. This *write barrier* has been implemented in a number of ways, varying essentially in the granularity of the information observed and stored. Here we examine a range of write barrier implementations and evaluate their relative performance within a generation scavenging garbage collector for Smalltalk.

1 Introduction

Generational collectors achieve short collection pause times partly because they separate heap-allocated objects into two or more generations and do not process all generations during each collection. Empirical studies have shown that in many programs most objects die young, so separating objects by age and focusing collection effort on the younger generations is a popular strategy. However, *any* collection scheme that processes only a portion of the heap must somehow know or discover all pointers outside the collected area that refer to objects within the collected area. Since the areas not collected are generally assumed to be large, most generational collectors employ some kind of pointer tracking scheme, to avoid scanning the uncollected areas. Again, empirical studies show that in many programs the older-to-younger pointers of interest to generational collection are rare, so avoiding scanning presumably improves performance.

To avoid scanning, the system must maintain some kind of table enabling the collector to find all the interesting pointers; we call this abstraction the *interesting pointers table* (IPT). Interesting pointers are created when a pointer (as opposed to non-pointer data) is stored in a heap object (as opposed to some other place) and the modified object resides in an older generation than the object that is the target of the pointer. Thus, certain of the program's stores must somehow create IPT entries. The action required has been called a *store check* or a *write barrier* by different authors. The general approach is to add an entry to the IPT whenever an interesting pointer is (or might be) created. The collector uses and rebuilds the IPT, discarding any entries that do not describe interesting pointers. Such entries can come about either because the system, as it runs, is imprecise about what is interesting, or because later changes overwrite interesting pointers with uninteresting data. Note that if the system is imprecise, it must err on the side of putting too many entries in the IPT rather than too few, since the IPT must allow the collector to find *all* interesting pointers.

In this paper we are concerned with direct comparison of various methods of implementing the write barrier. We will describe: our collector, the specific write barrier methods we compare, the benchmarks we used,

*This work is supported by National Science Foundation Grant CCR-8658074 and by Digital Equipment Corporation and Apple Computer. The authors can be reached electronically via Internet addresses {hosking,moss,stefanov}@cs.umass.edu.

OOPSLA'92, pp. 92-109

the experiment setup and methodology, and the results. We also discuss related work and present the conclusions we draw from the results. We offer two principal contributions here: the experimental results, which, like most benchmark-based studies, are not conclusive but nevertheless are interesting and useful; as well as the unique (to our knowledge) experimental setup that allows very direct and meaningful comparisons of the various schemes.

2 Overview of the garbage collector

We now describe the garbage collector used for the performance studies reported here. Its basis is the UMass Language-Independent Garbage Collector Toolkit, to which we add language specific code for our Smalltalk system. We first offer a condensed description of the toolkit and continue with appropriate details of the Smalltalk system. For a more detailed discussion of the toolkit see [4].

2.1 The toolkit concept

The toolkit divides the responsibility for and support of garbage collection into two parts: a language-independent part, supplied by the toolkit, and a language-specific part, nominally supplied by the language implementor. The language-independent part consists mostly of the data structures and code for managing multiple generations and the allocation of heap objects. The language implementor must supply the following capabilities: locating at scavenge time all *root pointers* (those pointers outside the scavenged generations that refer to objects in the scavenged generations), and locating all pointers within a heap object given a pointer to the start of the object. The toolkit includes a library of routines that an implementor can use to support the IPT; it remains the implementor's responsibility to locate roots lying in the stack(s), registers, and any other areas outside the heap.

2.2 The structure of the heap

The toolkit defines the structure of the heap and supplies the necessary allocation routines. The heap consists of a number of *generations*, ordered by age. We number them $0, 1, 2, \ldots$, in order of increasing age. In any given collection some generation and all younger generations will be scavenged. The number of generations may vary over time.

Each generation consists of a number of *steps*. Steps segregate objects by age within a generation, and during scavenging all surviving (reachable) objects in a given step are copied to some other step. This *promotion step* may belong to the same or a different generation. By adjusting the promotion steps before scavenging one can introduce new steps, combine existing steps, and so on, allowing the number of steps in a generation to vary over time. The primary function of steps is to eliminate the need for storing or maintaining any age information in individual objects. This reduces storage and time costs, but also gives the collector age information without imposing any requirements on object formats (which are entirely the responsibility of the language implementor). While the meaning of steps is somewhat arbitrary, we impose a convention that objects in the lower numbered steps are younger than those in the higher numbered steps, numbering the steps $0, 1, 2, \ldots$, such that every step in the system has a unique number.

For example, generation 0 might have steps 0 and 1, generation 1 might have steps 2 through 4, and so on. A simple promotion policy is to promote survivors of step k to step $k{+}1$. In that case, the number of steps in a generation determines the number of scavenges (of that generation) necessary to promote objects to the next generation.

Each step consists of a number of *blocks*. A block is 2^n bytes, aligned on a 2^n-byte boundary for some value of n chosen when the system is built. A typical block size might be 64K bytes. The number of blocks in a step may vary over time. While the blocks of a step are usually not contiguous, a *nursery* may be set up to consist of a number of contiguous blocks, so that one might more readily use a page trap to detect nursery overflow and trigger a scavenge. This avoids the need for an explicit limit check at every allocation.

Blocks have four primary advantages. First, they allow sizes of steps and generations to vary easily since the storage of a step need not be contiguous. Second, they allow speedy determination of the generation, step, and promotion step of an object: one merely shifts the address of the object right by n bits and indexes a block table containing the needed information. Third, blocks match naturally with page trapping or card marking schemes (to be discussed in detail below). Fourth, they reduce the storage needed under some circumstances, compared with copying collectors that use semi-spaces. If b bytes are present in a generation before a scavenge and the survivors consume a bytes, then a semi-space scheme uses $2b$ bytes whereas our scheme uses $b+a$ bytes (modulo rounding resulting from the block size). The degree of advantage depends on the survival rate a/b, but may be significant in some applications.

Blocks do introduce a problem: they cannot handle objects larger than the block size. To handle such objects we provide a *large object space* (LOS), as suggested in [14]. Indeed, it is probably a good idea to put in LOS any object that consumes a significant fraction of a block; we used the heuristic threshold of 1/8 of a block. Further, as also discussed in [14], any object that has few pointers in it and that exceeds some threshold in size should be stored in LOS to avoid the overhead of copying. Without going into all the details, LOS uses free list allocation based on splay trees [10, 11, 5] and once allocated an LOS object is never moved. However, LOS objects still belong to a step, which is indicated by threading the objects onto a doubly linked list rooted in the step data structure. When an LOS object is promoted, we simply unchain it from one list and chain it into another. When scavenging is complete, any LOS objects remaining on a scavenged step's LOS list are freed.

While the generation, step, and block of a non-LOS object can be discovered via the simple shift and index technique, LOS may mix objects from different steps and generations in the same block. Therefore, we store a back reference from each LOS object's header to its containing step, allowing relatively easy determination of the step given a pointer to the object's base. Determining the step given a pointer into the middle of the object requires locating the object header, which is supported but involves additional work.

2.3 Phases of a scavenge

A scavenge consists of two phases. First, the root set for the scavenge is determined based on the IPT scheme employed (as well as the stack and register decoding approach). All objects directly reachable from the roots are copied into new space, and the roots updated. In the second phase all objects reachable from the new space objects are copied over using a non-recursive Cheney scan [2].[1] As each object is copied, a forwarding pointer is left in the old copy, so that other references to the object can be updated as they are encountered. Since the toolkit makes no object format assumptions, the details of forwarding pointer format are up to the language implementor. The toolkit does support automatic determination of where to allocate the new copy of the object, given the object's size (which must be determined by language-specific code).

Before a scavenge begins, the toolkit, following a dynamically modifiable plan supplied by the language implementor, determines the generations to be scavenged and creates new steps according to the number desired for each scavenged generation. It also sets up all the promotion step references. After a scavenge, all the old steps of the scavenged generations are deleted and their blocks become available for allocation.

2.4 Smalltalk details

Our Smalltalk system consists of a virtual machine of our own design. It includes a bytecode interpreter for the instruction set defined in [3], and we run a Smalltalk image cloned (converted into our format) from an earlier release of Smalltalk-80.[2] We manage contexts (stack frames) as described in [7]. In particular, a number of frames are preallocated and assembled on a doubly linked list. Ordinary calls traverse the list in one direction and ordinary returns traverse it the other way, with cost similar to a stack. When a block context (similar to a closure) is created, or a frame otherwise becomes referenceable as an object, it is removed from the ordinary linked list so that it will not be reused until the collector can establish that it is no longer referenced. We store frames in step 0 and they are never promoted. This means that we need never perform store checks on stores into frames (they are in the youngest generation, so such a store can never create an interesting pointer).

Non-frame objects are created in the nursery in step 1. Generation 0 includes steps 0 and 1, so in principle we can use a slightly cheaper store check for initializing stores (which seem to be the most common stores in the system): ignore stores if the modified object is in

[1] The toolkit might be adapted to support mark-sweep or other approaches to collection, but currently it provides only copying collection. Also, it would not be hard to incorporate suggestions such as hierarchical clustering [16].

[2] Smalltalk-80 is a registered trademark of PARC Place Systems.

generation 0 (regardless of the generation of the target of the pointer).[3] There is a total of five generations, with one step in each of generations 1, 2, 3, and 4. Each step (except step 0, which never promotes, and step 5 which is the oldest step) promotes to the next step. Generation 0 is collected if we run out of frames or step 1 exceeds its allocation of one block. Similarly, generations 1, 2, and 3 are scavenged if they exceed their respective limits of 1, 1, and 10 blocks. Generation 4 is never collected. The block size is 64K bytes. All objects larger than 8K bytes are stored in LOS, as are all bytes objects of size at least 496 bytes. We do not claim that this arrangement is necessarily well-tuned, but we held it fixed across all benchmark runs so the comparisons remain direct. Note that the system can easily be configured to have a different heap arrangement.

3 Write barrier implementations

As previously sketched, the write barrier consists of actions performed in conjunction with a store that might create an interesting pointer. The purpose of the write barrier is to support efficient location of all root pointers in the heap (i.e., to avoid scanning the generations not being collected). We have implemented several versions of the three most common write barrier approaches. They vary mostly in the granularity of the information they record.

The first scheme associates a *remembered set* with each generation [13], recording the objects or locations in older generations that *may* contain pointers into that generation. Any pointer store that creates a reference from an older generation to a younger generation is recorded in the remembered set for the younger generation. At scavenge time the remembered sets for the generations being scavenged include the heap root set for the scavenge.

The other schemes divide the heap into logical regions of size 2^k bytes, aligned on a 2^k-byte boundary, for some fixed k. We call these regions *cards*, after [12, 17]. Each card has a corresponding entry in a card table indicating whether the card might contain pointers into younger generations. Mapping an address to an entry in this table is simple: one shifts the address right by k and uses the result as an index into the table.

[3] We detail later the exact store checks (if any) we used with each write barrier implementation.

Whenever a pointer is stored into an object, the corresponding card is *dirtied*. At scavenge time all dirty cards of generations *not* being scavenged include the heap root set for the scavenge.

One variant of this scheme uses the page protection mechanism of the operating system to detect stores into clean cards. A card in this scheme corresponds to a page of virtual memory. All clean pages are protected from writes. When a write occurs to a protected page, the trap handler dirties the corresponding entry in the card table and unprotects the page. Subsequent writes to the now dirty page incur no extra overhead. Note that *all* writes to a clean page cause a protection fault, not just those that store pointers. An operating system could more efficiently supply the information needed in the page protection scheme if it offered appropriate calls to manipulate the page dirty bits maintained by most memory management hardware [8].

With each of these schemes we are faced with the choice of remembering either the slot that is updated or the object containing that slot. For remembered sets, this is simply a matter of entering the object pointer or the slot address in the appropriate remembered set. For card marking, remembering the containing object means dirtying the card containing the header of the object. Remembering the slot means dirtying the actual card in which the slot lies, which may be different. Naturally, the page protection scheme is only able to dirty the page containing the slot, since that is the location updated.

We now give a detailed description of our implementation of these schemes.

3.1 Remembered sets

Our remembered sets are implemented as circular hash tables using linear hashing. A remembered set is allocated as an array of 2^i+k entries. To enter an item in the set, we hash the item to obtain i bits and index the table. If the indexed location is empty then the item is stored in that slot and we are done. If the location already contains the item then we are done also. Otherwise, the immediately succeeding k slots are examined to try to place the item (this is not done circularly; hence the 2^i+k rather than simply 2^i). If an empty location still cannot be found then a circular search of the table is made to find an empty slot. The hash tables are kept relatively sparse by growing a table whenever an item cannot be placed in its natural hash slot or the k following slots, and 60% or more of the table's slots are full. We fixed k=2 and the growth policy is to increment i (i.e., basically double the table size when a table is grown).

3.1.1 The write barrier

To avoid making the remembered sets too large we record only those stores that are interesting; we use the term *filtering* to indicate the process of determining whether an item is interesting. In Smalltalk we always do a pointer vs. non-pointer test on the item being stored. If the item is a pointer, this is followed by a generation test, which we perform by determining the generations of both the modified source object and the target object whose pointer is being stored, and comparing the two. Following Zorn [18], and based on our own run-time traces of the Smalltalk system which reveal that most stores occur to *initialize* newly allocated objects, we can frequently avoid the need to determine the generation of the target object by checking if the modified object is in generation 0. As mentioned earlier, determining the generation of an object involves shifting its pointer and indexing into the block table. Thus, our store filter involves a shift, index, and load to obtain the source object's generation, a conditional to filter initializing stores, followed by a shift, index, and load for the target object, and a comparison. If the store passes through this filter then it is interesting, so we invoke a subroutine to hash the modified object or slot into the appropriate remembered set. To avoid run-time code to determine precisely which remembered set to update, all interesting stores are actually hashed into a run-time *scratch* set.

On the MIPS R2000 initializing stores are filtered using 7 instructions. The remaining uninteresting stores are filtered using another 7 instructions. The entire inline sequence comes to a total of 17 instructions including the call to update the remembered set.

3.1.2 Scavenging

At scavenge time the remembered sets of the generations being scavenged plus the scratch set determine the heap root set. To eliminate duplicates in the root set we hash the remembered sets of the scavenged generations into the scratch set to form the union. Each entry in the scratch set is then processed to locate pointers into the scavenged generations: if we are remembering objects then the heap root set consists of all pointer locations in those objects; otherwise if slots are being remembered then they directly constitute the root set. As scratch set entries and promoted objects are processed, all interesting pointers that we encounter are recorded in their appropriate remembered set, in order to rebuild the remembered sets of the scavenged generations and to keep those of the older unscavenged generations up to date.

The apparent advantages of remembered sets are their conciseness and accuracy, achieved at the cost of filtering for interesting pointer stores before recording them in the appropriate remembered set, and of hashing to keep the sets small by eliminating duplicates. At scavenge time, unless there has been repeated mutation of an object or location, the remembered set is likely to be a very accurate characterization of the heap root set.

3.1.3 The sequential store buffer

For an interpreted language such as our Smalltalk system the space overhead of 17 instructions at every store site is not a problem, since stores occur at a relatively small number of fixed locations in the interpreter. However, for compiled languages this overhead will be incurred at every one of an arbitrary number of compiled store sites, which may be prohibitive. For this reason we have devised a scheme similar to that introduced by Appel [1], allowing batch filtering and recording of pointer stores, using a *sequential store buffer* (SSB) to buffer the necessary information. The SSB comprises some number of contiguous pages, bounded by a "guard" page that has been protected from writes. Recording a word of information in the SSB consists of storing to the next free location in the buffer and bumping the free pointer. If the free pointer is maintained in a register then this can be implemented on the MIPS R2000 using just two instructions: one to store the word and the other to increment the pointer.

At scavenge time the information recorded in the SSB is processed to update the scratch set, with filtering as described above. Overflow of the SSB at run time is trapped by the operating system when an attempt is made to store into the guard page. The trap handler processes the SSB and resets the free pointer to the beginning of the buffer.

We record two words of information in the SSB for each store to allow for efficient filtering of uninteresting pointers: when remembering slots we record the modified object as well as the updated slot;[4] when remembering objects we record both the modified source object and the target object to avoid scanning the entire modified object for interesting pointers when processing the SSB.

3.2 Card marking

Card marking requires that we allocate a contiguous card table containing an entry for every card in the heap. Our garbage collector allows the heap to grow as large as the operating system (and practical considerations) will allow, since blocks are incrementally added to the heap as they are needed. While we envision a scheme where the card table grows incrementally, in the benchmark runs we imposed an upper bound on heap growth and allocated a fixed-size card table during memory manager initialization.

3.2.1 The write barrier

One of the most attractive features of card marking is the simplicity of the write barrier. For this reason we have chosen to implement the card table as a byte array rather than a bit map.[5] By interpreting zero bytes as dirty entries and non-zero bytes as clean, a pointer store can be recorded using just a shift, index, and byte store of zero. On the MIPS R2000 this comes to just 4 instructions: a load to get the base of the card table, a shift to determine the index, an add to determine the byte entry's address, and a byte store of zero.

3.2.2 Scavenging

At scavenge time the dirty cards of the generations not being scavenged determine the root set. We must scan each card to find all references into the generations being scavenged. If we are remembering objects (i.e., if pointer stores dirty the containing object's card) then every pointer slot of every object whose header lies in a dirty card must be examined. If we are remembering slots (i.e., if stores dirty the updated slot's card) then the root set consists of all pointers that lie in dirty cards. Either way, locating pointers within cards is complicated by the mixing of bytes and pointers in Smalltalk objects, and the potential for objects to span multiple cards.

To find the pointers in a card we must be able to find the object headers in the card, which encode the formats of the objects allowing us to locate their pointers. To support locating object headers, we maintain a table of card offsets parallel to the dirty card table, indicating the location of the *last* (highest address) object header within each card. This requires every allocation of an object in any generation but the youngest to update the card offset table. These updates are unconditional, since we allocate from low to high addresses, so the most recent allocation in a card is always the offset of the last object in the card. Since new objects are always allocated in the youngest generation this allocation overhead is incurred only upon promotion of objects at scavenge time.[6] A negative offset entry indicates that the card contains no object header—the object header must be in some previous card. A positive offset indicates the *longword* of the card at which the last object's header begins. Using longword offsets allows us to keep the offset table entries to just one byte for cards of 512 bytes or less. For larger cards we use a two-byte entry.

Before scanning a dirty card for pointers, we first mark it clean. Then if we find any interesting pointer in the card (even if the generation of the target is not being scavenged), we dirty the card for future scavenges. Note that a dirty card becomes clean if the scan certifies that the card contains no interesting pointers. We reduce scanning overhead by scanning all contiguous dirty cards as a group, running from the first to the last. Promoted objects are always allocated in newly allocated blocks whose cards are assumed to be clean, so as promoted objects are scanned we also update their card entries.

---

[4]Recording the slot alone would be sufficient. However, we can take advantage of the fact that our Smalltalk implementation allocates all object headers in small object space. Large objects are represented by a header in small object space with a pointer to the body of the object in large object space. This makes determining the generation of a slot much simpler if we are given a pointer to its containing object's header rather than the address of the slot itself. By recording the modified object as well as the slot we avoid unnecessarily complicating SSB filtering.

[5]We first heard of this idea from Paul Wilson.

[6]There is one rare exception to this brought about by our implementation of the Smalltalk primitive method `become:`.

An unresolved question is just how large cards should be. There is an obvious tradeoff in that large cards mean fewer cards and smaller tables, but larger cards also imply a larger root set at scavenge time. There is also the question of filtering. As for remembered sets we filter non-pointer stores to avoid unnecessarily marking cards. However, there is the possibility that generation filtering might also improve the accuracy of the root set by reducing the number of marked cards to be scanned at scavenge time.

3.3 Page protection

The final scheme is a variant of card marking where the write barrier is implemented by using the paging hardware's capability to trap writes to protected pages. Rather than recording *every store* at run time, we trap only writes to clean pages. This means that there is no overhead for writing to *dirty* pages at run time, but stores to clean pages will incur the significant overhead of fielding a signal from the operating system, unprotecting the appropriate page, and resuming ($\sim 250\mu s$ round trip as measured in a tight loop under Ultrix 4.1 on the DECStation 3100).

At scavenge time we process dirty pages (of generations not being scavenged) essentially as for card marking, except that any dirty page certified as clean must be protected. We scan runs of contiguous dirty pages as a group. Similarly, to protect a run of contiguous ex-dirty pages we issue just one system call for the entire run, to minimize system call overhead.

Unlike card marking, where we allocate promoted objects in newly allocated blocks whose cards are assumed to be clean, the page protection scheme assumes that the pages of all newly allocated blocks are dirty. This means that there is no need to record interesting pointers as promoted objects are scanned. It also means that no page is ever protected in the youngest generation, where new objects are allocated, so allocating and storing into a new object never causes a trap.

4 Benchmarks

We chose a set of five Smalltalk programs to run as benchmarks under each of the write barrier implementations. The first two benchmarks are real applications, the second two are synthetic benchmarks designed to reveal the behavior of the garbage collector, and the last is intended to reveal the behavior of the garbage collector in an "interactive" session. We now describe each benchmark and characterize its behavior:

Richards: This is the Richards operating system simulation benchmark. It is a computation-intensive program, and preallocates most of its data. Most subsequent allocations consist of frames. We chose this benchmark to reveal the cost of garbage collection in a program that does little allocation and creates little garbage.

Lambda: This is a pure λ-calculus interpreter of our own devising. It represents λ-expressions as directed graphs, internally consisting of small fixed size Smalltalk objects. It models β- and η- reduction. Internally, it implements normal order reduction by copying the argument subexpression. This entails intensive allocation activity (for each occurrence of the bound variable, it allocates objects for the argument copy) and garbage generation (following the substitution, the original argument is garbage). In addition, variable bindings are handled internally using Smalltalk dictionaries, giving rise to a large number of `become:` operations to grow the dictionaries.

Swap—trees with mutation: This synthetic benchmark first builds a complete tree of branching factor 4 and height 6. Each node consists of an array of pointers to the node's children and a small data array. The total size of the tree is 600K bytes. Once the tree is built the program loops swapping random subtrees of height 3. This benchmark reveals the efficiency of the write barrier.

Destroy—trees with destructive updates: This synthetic benchmark builds a complete tree of branching factor 6 and height 5, similar to the tree of the Swap benchmark. The total size of the tree is 900K bytes. However, instead of swapping subtrees, Destroy replaces a subtree of height 3 (size about 25K bytes) with a newly allocated subtree of the same size. The total amount of data processed during a run is about 24 megabytes. This benchmark explores the cost of applications that generate garbage rapidly.

Interactive—the "macro" benchmarks: For this benchmark we iterate 10 times through the full set of "macro" benchmarks. These benchmarks are part of the standard suite of benchmarks [6] used to compare the relative performance of different Smalltalk implementations. They measure system support for the programming activities that constitute typical interaction with the Smalltalk system, such as keyboard activity, compilation of methods to bytecodes, and browsing.

5 Experiments

To ensure that each benchmark exhibited the same behavior from run to run we modified the Smalltalk interpreter to record and replay sessions. Thus, every run sees exactly the same Smalltalk events, such as allocation, system time, keyboard/mouse events, interrupts, etc. We note that the toolkit and write barrier software design is such that each scavenge is presented with exactly the same heap layout, collection of objects, blocks, etc., even to the point that the offset of the objects in blocks will be the same. Indeed, the memory contents can differ only in the sizes and locations of the write barrier data structures (card table, remembered sets) and the placement and order of the blocks (the presence of the write barrier structures may cause blocks to be allocated in different places under different schemes).

Naturally, there will still be some variation from run to run due to context switching by the operating system, but we minimized this by doing all timing tests in single user mode, disconnected from the network. We ran each benchmark several times under various implementations of the write barrier on a DECStation 3100 running Ultrix 4.1.[7] There was adequate real memory to prevent paging.

We measured elapsed time using a custom timer board with a resolution of 100 ns. Extracting the value of the timer involves reading 4 contiguous words from a memory location to which the timer device has been mapped, resulting in little timing overhead. The fine-grained accuracy of this timer allowed us to measure the elapsed time of each phase of execution separately: running time between scavenges, processing of the root set, scanning of promoted objects, and other overheads of garbage collection. To obtain dynamic counts of allocations, pointer stores, etc., we built an instrumented version of the interpreter and did a separate set of runs (i.e., the counter instrumented interpreter was not used for timing purposes).

Our experiments included runs for the two versions of the remembered set scheme (one remembering objects, the other slots), object and slot versions of the card scheme, with card sizes varying from 16 to 4096 bytes by powers of 2, and the page protection scheme (the page size is 4096 bytes). We also measured the SSB variant of the remembered set scheme for both objects and slots with a 10-page SSB, and a variant of the most promising card scheme using the same generation filter as for remembered sets to minimize the number of dirtied cards.

[7] The operating system had some official patches installed that fix bugs in the `mprotect` system call.

6 Results

We now report the elapsed time performance of each benchmark in turn. To best eliminate any uncontrolled interference from the operating system, we take the *minimum* elapsed time for *each phase* (separately) over twenty runs. The phases include:

- *running*, the time spent in the interpreter as opposed to the collector (note that running includes the cost of store checks and/or page traps);
- *root* processing, the time spent scanning through remembered sets or card/page tables and copying the immediate survivors;[8]
- *promotion*, the time spent copying the remaining survivors; and
- *other*, time spent in any remaining activities, such as setting up internal tables, etc.

In addition, for the SSB variant of the remembered set scheme we measured the time spent processing the SSB prior to each scavenge. Note that any SSB processing required to handle SSB overflow is charged to the *running* phase. We exclude all image loading and initialization time (i.e., all actions prior to entering the main interpreter loop). We present results for the slot-based approaches first, and discuss the object remembering schemes later (results for the object-based schemes appear at the end of the paper).

[8] In Smalltalk the stack is stored as heap objects so there is no separate stack processing. In fact, all the process stacks are copied during each scavenge. Also, Smalltalk has only a few global variables, in the interpreter.

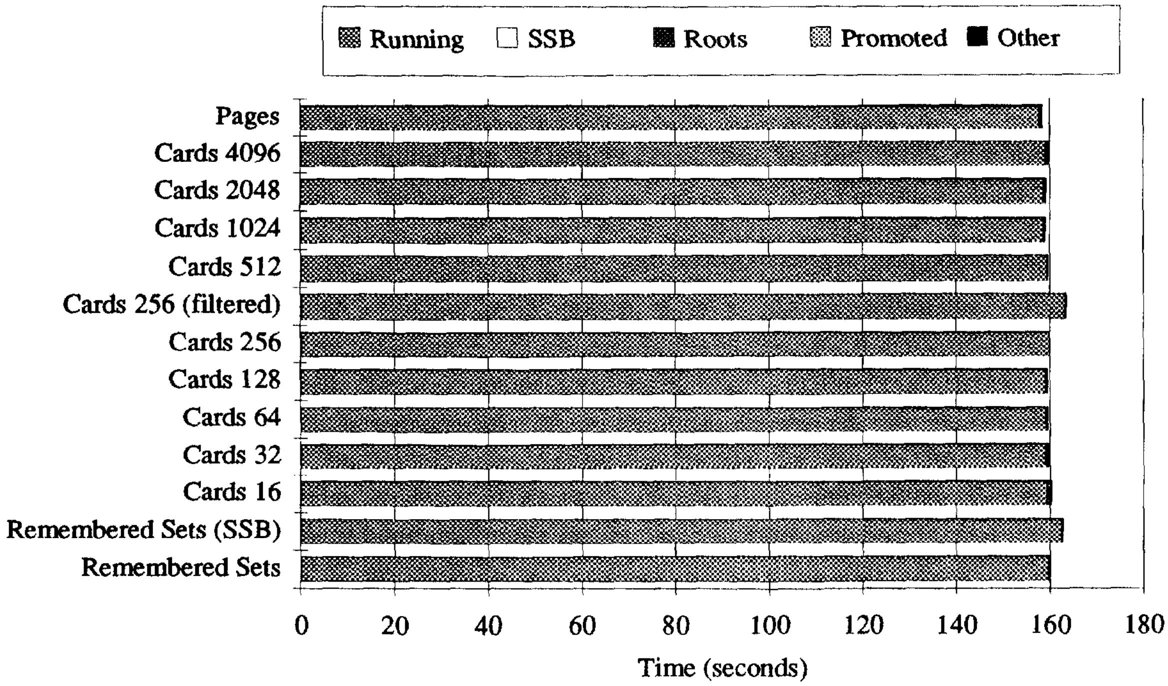

Figure 1: Elapsed time for Richards

6.1 Richards

The computation-intensive nature of the Richards benchmark is revealed in Figure 1. We see small gc overhead, indicating little need for scavenging apart from the recovery of block contexts (frames). Even so, expanding the scavenge part of the graph to examine gc overheads, we see the tradeoff in the card scheme between the size of the cards and the number of cards needing to be scavenged (Figure 2). For this benchmark the SSB is substantially more expensive, due in most part to the very high store rate and the low scavenge rate, so that the SSB overflows approximately 30 times between successive scavenges.

The high cost of the filtered card scheme is curious, considering that the same filter applied for remembered sets shows little extra overhead. Further, this overhead appears only in the results for the slot-based scheme. The corresponding object-based scheme is comparable with the other card schemes. Comparison of the compiled store check code reveals that the code generated by the compiler for the slot-based scheme is less efficient than that for the object-based scheme, perhaps because the store check code needs three quantities around (the object address, the slot address, and the new contents). Since the compiler can probably be convinced to generate more efficient object code through rearrangement of the source code, we anticipate eliminating this overhead, making the filtered card scheme more competitive. Moreover, compiled languages can ensure that the best code is generated for the store checks, since it has complete control over code generation.

A direct comparison of root processing times is given in Figure 3, showing that 256-byte cards appear to be optimal, and that at least for cards, remembering objects requires consistently less root processing. Moreover, filtering seems to be futile for the card schemes since it has little impact on the root processing time, and offers little (if any) improvement in running time. These root processing results hold across all the benchmarks, so we refrain from presenting separate root processing graphs for the remaining benchmarks.

Overall, the page trapping scheme is marginally better than the other schemes because most stores are to pages that are already dirty; remembered sets and cards are competitive. Filtering is of little use for cards, since it unnecessarily complicates the store check with little (if any) improvement in total time. The SSB is penalized by the high store rate and low scavenge rate of this benchmark, incurring substantial overhead to service SSB overflow traps.

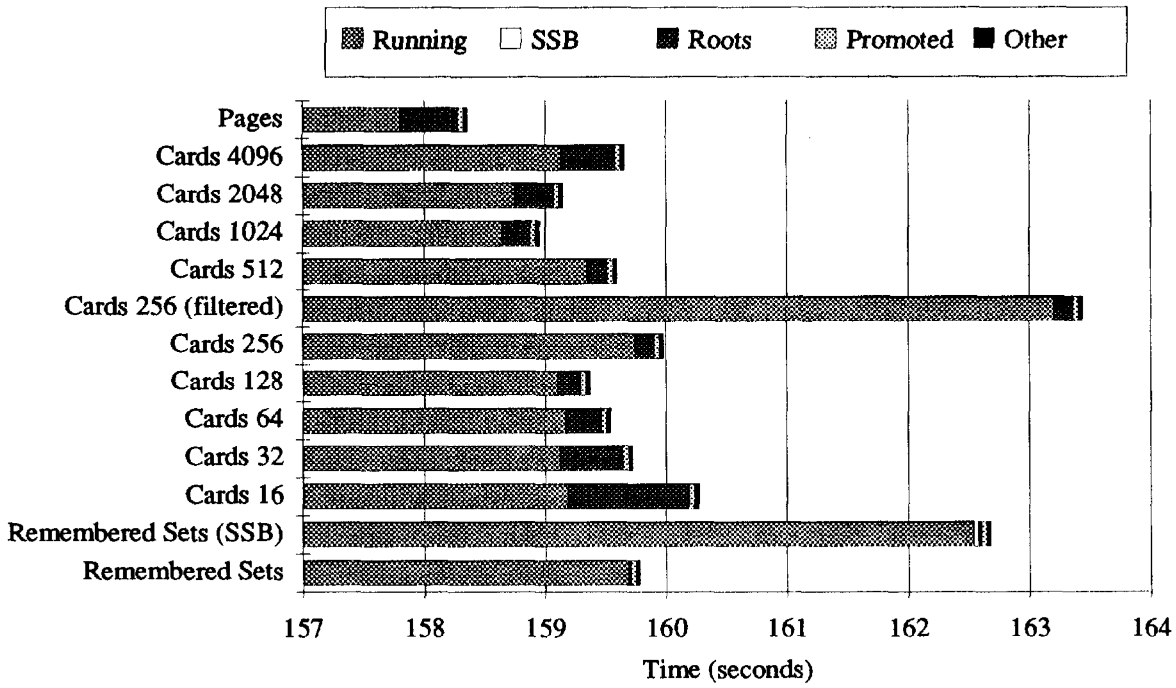

Figure 2: Elapsed time for Richards (expanded scale)

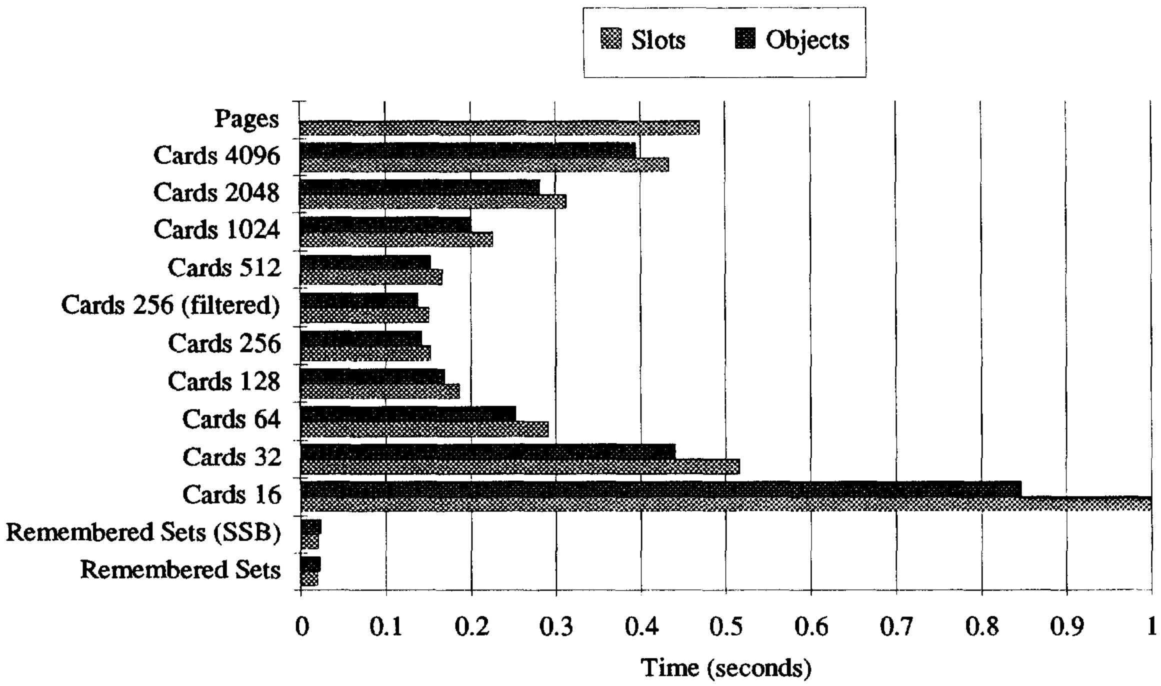

Figure 3: Root processing time for Richards

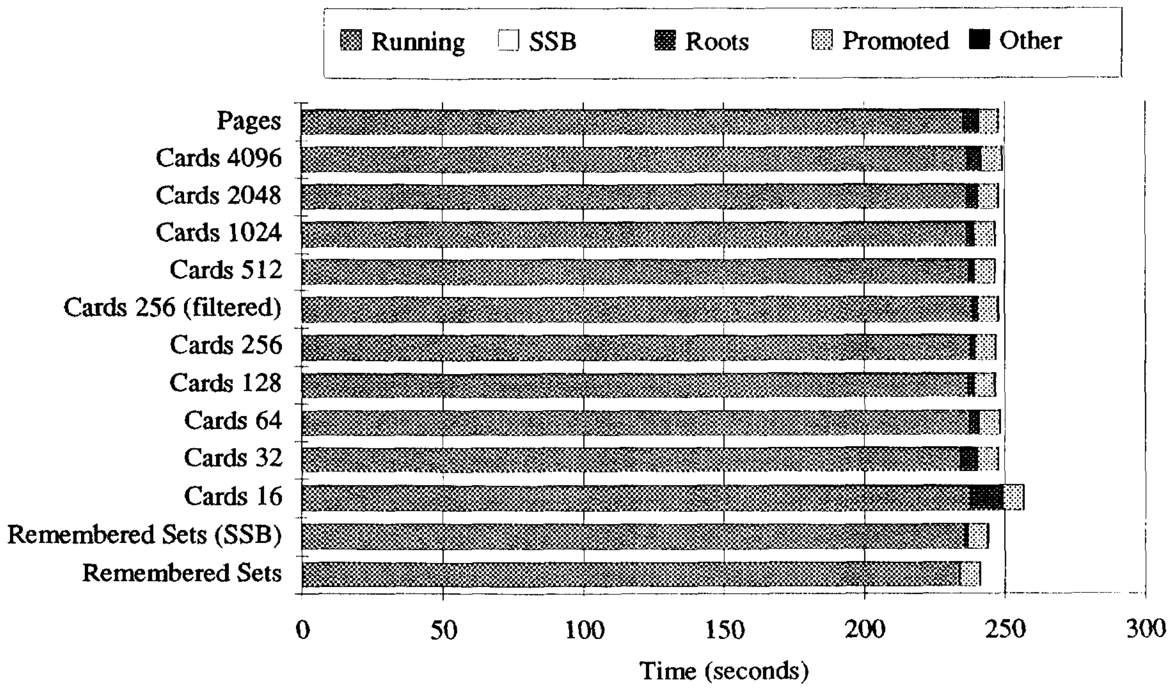

Figure 4: Elapsed time for Lambda

6.2 Lambda

Naturally since Lambda does much more allocation and creates much more garbage than Richards, we see that it exhibits higher collection overhead, ranging from 3.1% to 6.9% of the total time (Figure 4). This demonstrates how well even a minimally tuned generational collector can perform in a high garbage context. We still observe the characteristic tradeoff as card size varies, although the most effective scheme for this benchmark is remembered sets, both with and without the SSB. Tuned cards are competitive, and once again filtering offers little (if any) improvement.

6.3 Swap

Figure 5 shows the times for the Swap benchmark. Its behavior is generally similar to that of Lambda, except that the root processing time is proportionately much higher, as we would expect from the continual mutations, which force the collector to re-examine the objects. Once again, remembered sets come out best, with the SSB providing marginally better performance. The ratio of garbage collection time to total time ranges from 2.7% for remembered sets, through 5.4% for the best card scheme to 20% for 16-byte cards.

6.4 Destroy

In Figure 6 we see that Destroy incurs much higher promotion costs than Swap, which is to be expected. It also has larger root processing costs (as a fraction of run time), because it does more allocation and creates large amounts of garbage. Nevertheless, it produces the same relative standing of the schemes. The collection overhead is high: from 16% to 37% of total time. Note that this benchmark is probably not very characteristic of real programs—while some programs do mutate a long-lived heap, they generally do not do so rapidly, nor do they create garbage from their old objects in a rapid continual fashion.

The noticeable variation in running time amongst the card schemes is puzzling. Since the code for all the card schemes is exactly the same, barring the shift values and the card offset table entries, the variation can only be explained by cache and TLB effects. The DecStation 3100 has direct-mapped *physically-addressed* caches, so a bad assignment of virtual pages to physical pages may make certain lines of the cache more volatile than others, degrading performance if important data or instructions end up in those cache lines. Note also how filtering, which disturbs the store check code, and reduces the number of stores that must be recorded, im-

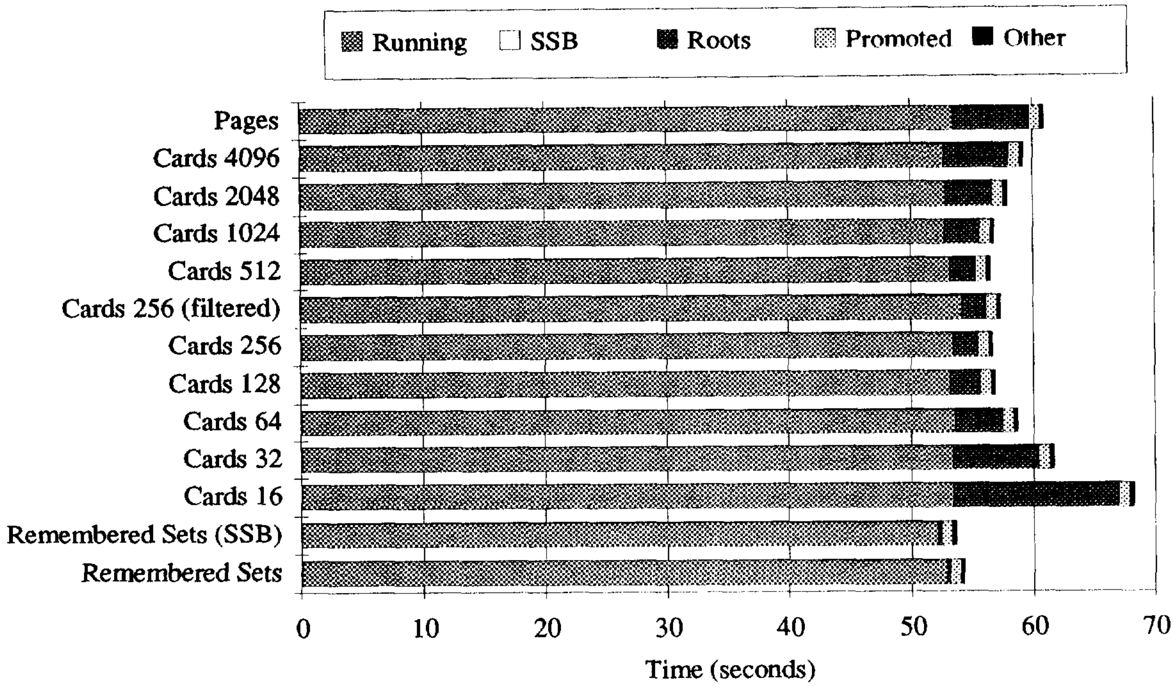

Figure 5: Elapsed time for Swap

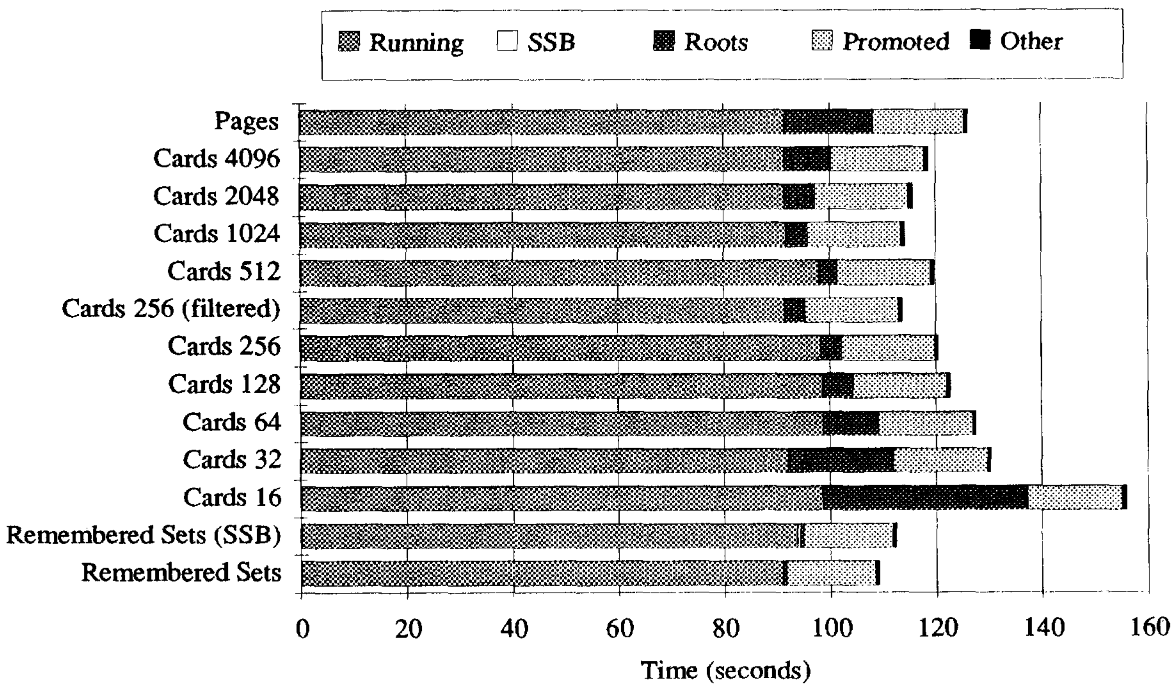

Figure 6: Elapsed time for Destroy

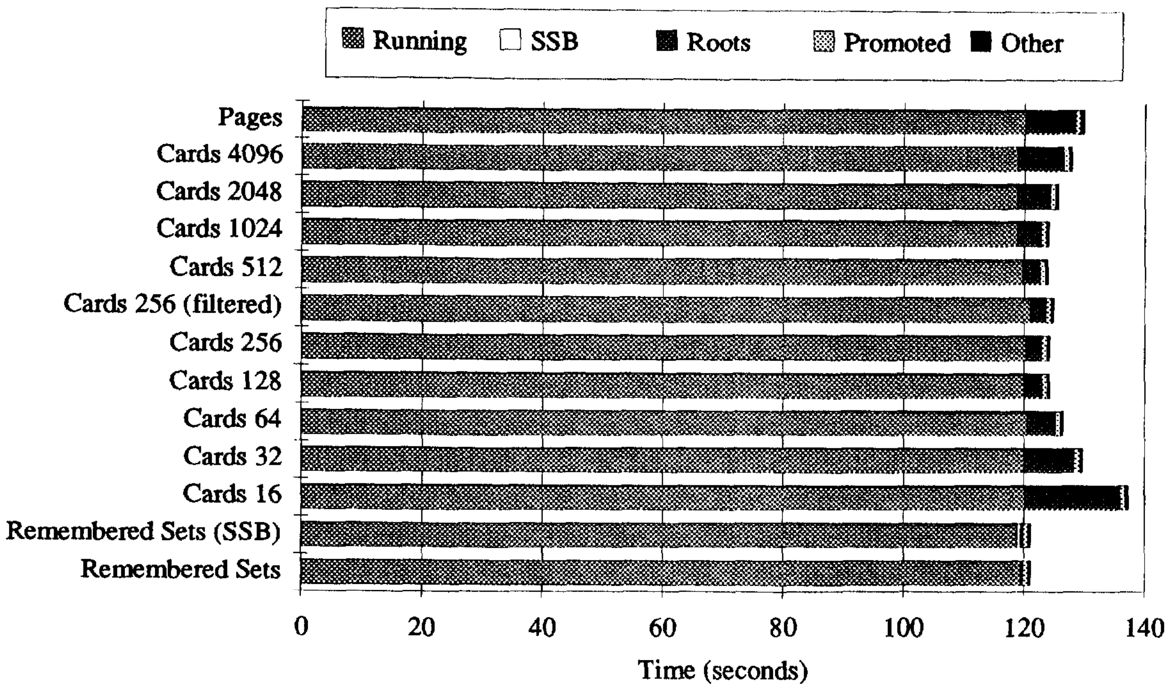

Figure 7: Elapsed time for Interactive

proves the running time for the 256-byte card scheme. Moreover, the object-based schemes, which again have slightly different store check code and dirty different entries in the card table, exhibit no such variation in running times.

6.5 Interactive

The Interactive benchmark yields similar results to the other benchmarks (Figure 7). Remembered sets are best, with gc overhead as good as 1.5% of total time, and cards ranging from 2.9% to 10% of total time.

6.6 Objects versus slots

There seems little to distinguish the approaches that remember objects from those that remember slots. Root processing for the slot-based card schemes costs a little more than for the object-based card schemes, with the effect more pronounced for small card sizes. We suggest that remembering objects is cheaper because when remembering slots, we must process any object of the previous card that continues into a dirty card, but when remembering objects, we can skip over such objects. This extra cost is essentially per-card (since the average object size remains fixed as we vary the card size within any given benchmark), so as the card size increases and the number of cards decreases, the extra cost fades away. For remembered sets, remembering slots is marginally cheaper than remembering objects, since the slots encode the interesting pointer information more exactly, whereas the remembered objects must be scanned to find their interesting pointers.

7 Related Work

Ungar introduced generation scavenging [13], building on earlier work on generational collection. Further details as to the cost of store checks appear in [15], with the conclusion that special hardware in the SOAR chip might offer a time performance improvement of 3% over a tightly coded inline check. While our checks are not as tightly coded, the apparent time penalty is still small, although it may be because our interpreter is relatively slow compared with the SOAR design.

Shaw considered the relationship of collection to virtual memory for LISP programs [9]. In particular, Shaw examined various write barrier methods, including hypothetical user access to page dirty bits maintained by the operating system [8].

The most directly related work of which we are aware is [18]. There, Zorn studies not only the write barrier, but also the read barrier, which is used in incremental collection. Our results agree with Zorn on the cost of the write barrier: even when implemented in software, its cost appears to be modest. However, Zorn focused primarily on the cost of the write barrier alone, rather than the total cost, and our results show that granularity is sometimes significant in the total cost. We somewhat disagree with Zorn on the cost of page traps for the write barrier, provided the operating system cost of delivering a trap to a user handler is reasonably low. Still, we agree in the conclusion that since software schemes offer generally better performance, port more easily, and do not rely on good operating system performance, software approaches might be more desirable.

8 Conclusions

There are several conclusions we draw from the benchmark results. First, the card marking scheme exhibited quite clearly the expected tradeoff with respect to card size. In this environment (hardware/memory architecture) a card size of 256 or 512 bytes gives the best performance of the card schemes. Note that because the card offset table entries are in terms of longwords, these sizes allow the offsets to fit in one byte, so the total overhead is two bytes per card, or 1 to 2%. The variation in card marking collection overhead was significant only in the mutation intensive benchmarks and near the extremal card sizes. It does appear reasonable to settle on a particular card size and use that for all applications in a given system, since the optimal size did not vary significantly across the benchmarks, and the curve is relatively flat near the optimum. Generation filtering is ineffective for cards since it has little impact on root processing costs, expands the size of the store check, and may incur extra run-time overhead.

The page trapping scheme performed poorly in comparison to card marking. Interestingly, this does not appear to be due to the overhead of fielding page traps, since that is included in running time, which was not significantly higher (and often lower) than in the card marking schemes. Rather, it is because pages are too large a granule so they miss the optimum card size.

Remembered sets have a strong advantage despite the extra generational check and the hash table insertions, since they allow markedly less root processing than the other schemes. If the inline space overhead is a drawback then the SSB provides a reasonable solution, except when there is a high store rate and low scavenge rate so that trapping SSB overflows has a noticeable impact on running time, as occurred for the Richards benchmark.

What was most surprising to us is how similar *all* the schemes are in performance. For example, we were surprised how close the page trapping scheme came to being tightly competitive, though that may be partly the result of the unusually good implementation of the operating system functions (i.e., in other operating systems, the running time might be more noticeably degraded). Note that this suggests that the access to hardware dirty bits discussed in [8] may not improve matters much over a decent implementation of reflecting the page trap to user code.

We were also surprised that the effects of the various schemes on the running time were not more pronounced, and that much of the difference between the schemes is in root processing. However, this could very well be because we are measuring an interpreter, so the differences tend to be obscured by the interpreter's overhead. Also, some of the run-time variation may be artifact, resulting from cache effects due to differences in the memory placement of instructions (etc.). Because we are running an interpreter, many of the store checks are in the same place. If a version of the interpreter compiled with a particular write barrier implementation happens to have a store check in a "bad" place, then the effect is magnified (as compared with compiled programs). This leads to an obvious suggestion for further work: consider the same sort of study on a compiled language. We hope to undertake such studies in our Modula-3 system before long.

We can summarize the conclusions as follows: a card size of 256 or 512 bytes appears optimal for card marking on this hardware; page trapping was surprisingly effective, but is not the best scheme because its granularity is too large; and remembered sets are best overall.

9 Acknowledgments

Amer Diwan, Rick Hudson, and Christopher Weight devised and implemented the original garbage collector toolkit. Craig Chambers provided us with Smalltalk source code for the Richards benchmark. We especially thank Digital Equipment Corporation's Western Research Laboratory, and Jeff Mogul in particular, for giving us the high resolution timing board and the software necessary to support it.

References

[1] A. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, Feb. 1989.

[2] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[4] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, Sept. 1991. Submitted for publication.

[5] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300–311, Apr. 1986.

[6] K. McCall. The Smalltalk-80 benchmarks. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 9, pages 153–173. Addison-Wesley, 1983.

[7] J. E. B. Moss. Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '86 Symposium on Interpreters and Interpretive Techniques*, pages 229–240, St. Paul Minnesota, July 1987. *ACM SIGPLAN Not. 22*, 7 (July 1987).

[8] R. A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, Mar. 1987.

[9] R. A. Shaw. *Empirical Analysis of a LISP System*. PhD thesis, Stanford University, Feb. 1988. Available as Technical Report CSL-TR-88-351.

[10] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory*, pages 235–245, Boston, Massachusetts, Apr. 1983.

[11] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3), July 1985.

[12] P. G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[13] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. *ACM SIGPLAN Not. 19*, 5 (May 1984).

[14] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–17, San Diego, California, Sept. 1988. *ACM SIGPLAN Not. 23*, 11 (Nov. 1988).

[15] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[16] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991. *ACM SIGPLAN Not. 26*, 6 (June 1991).

[17] P. R. Wilson and T. G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana, Oct. 1989. *ACM SIGPLAN Not. 24*, 10 (Oct. 1989).

[18] B. Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, Nov. 1990.

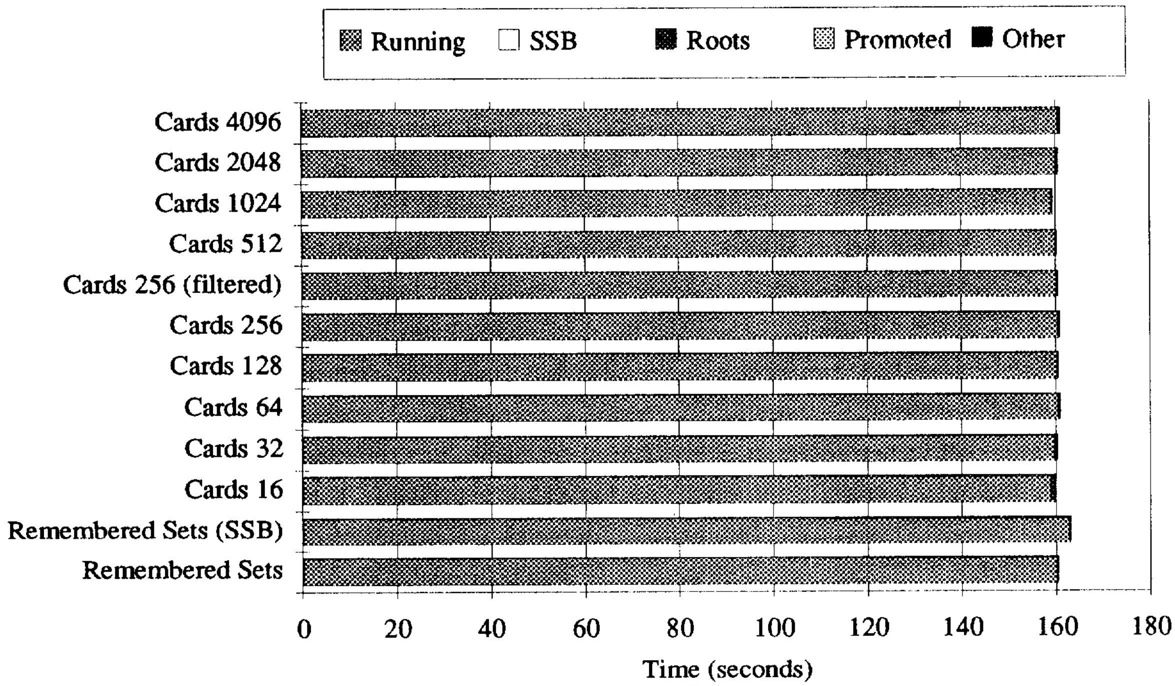

Figure 8: Elapsed time for Richards (remembering objects)

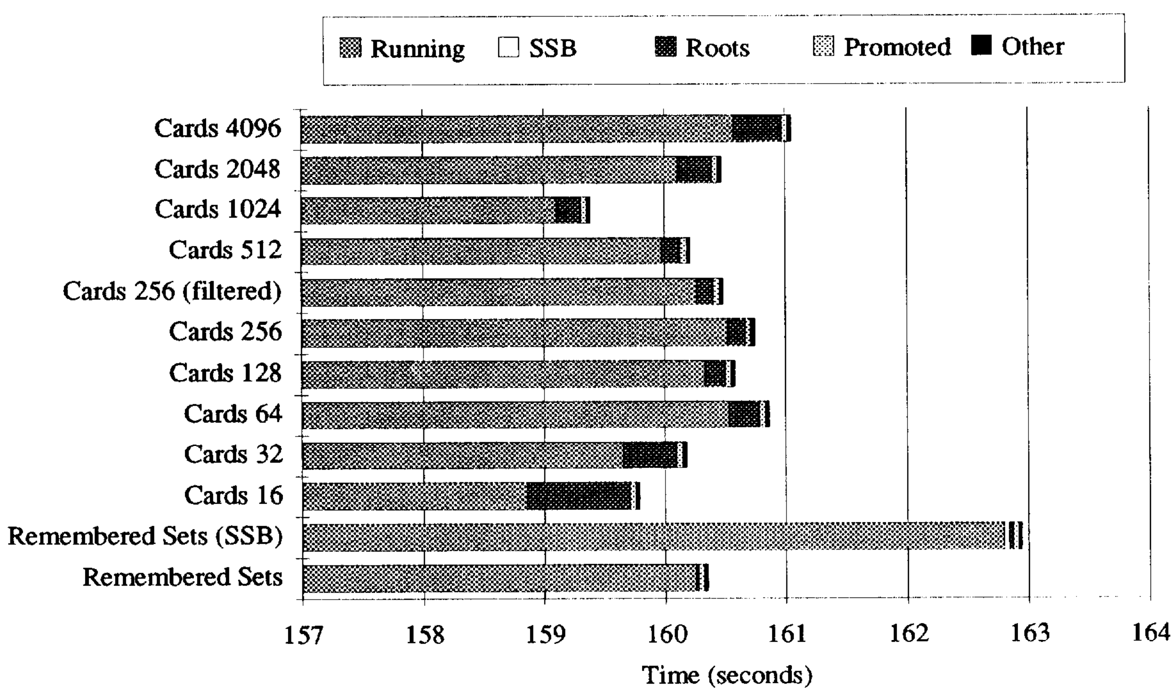

Figure 9: Elapsed time for Richards (remembering objects, expanded scale)

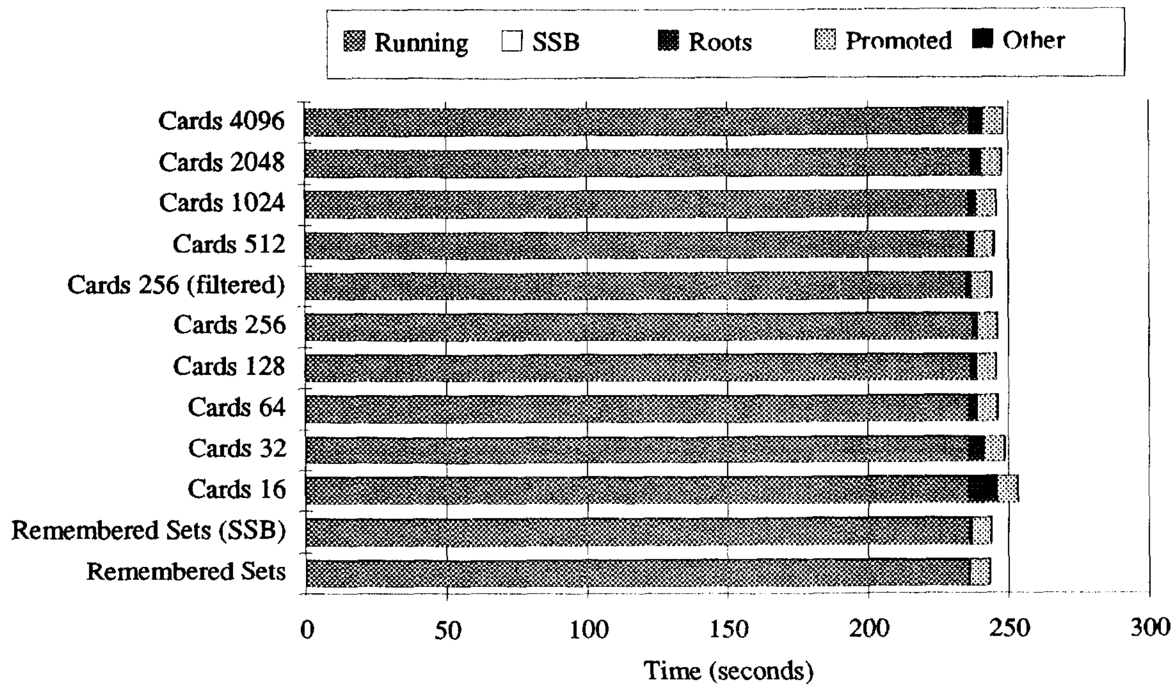

Figure 10: Elapsed time for Lambda (remembering objects)

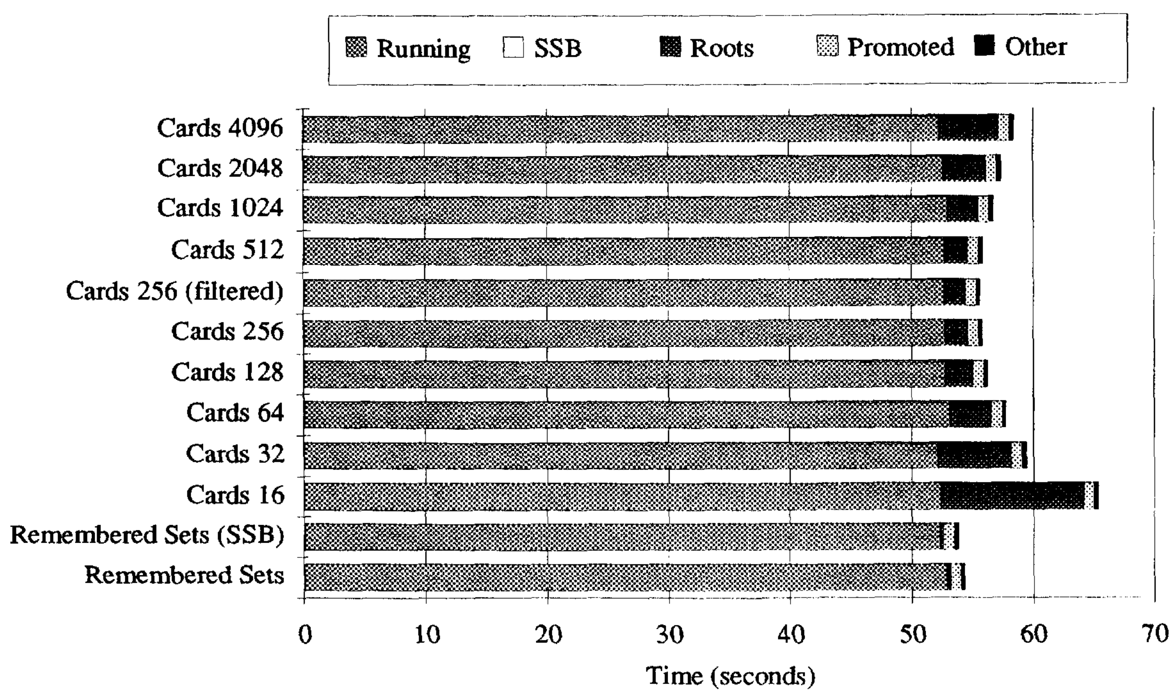

Figure 11: Elapsed time for Swap (remembering objects)

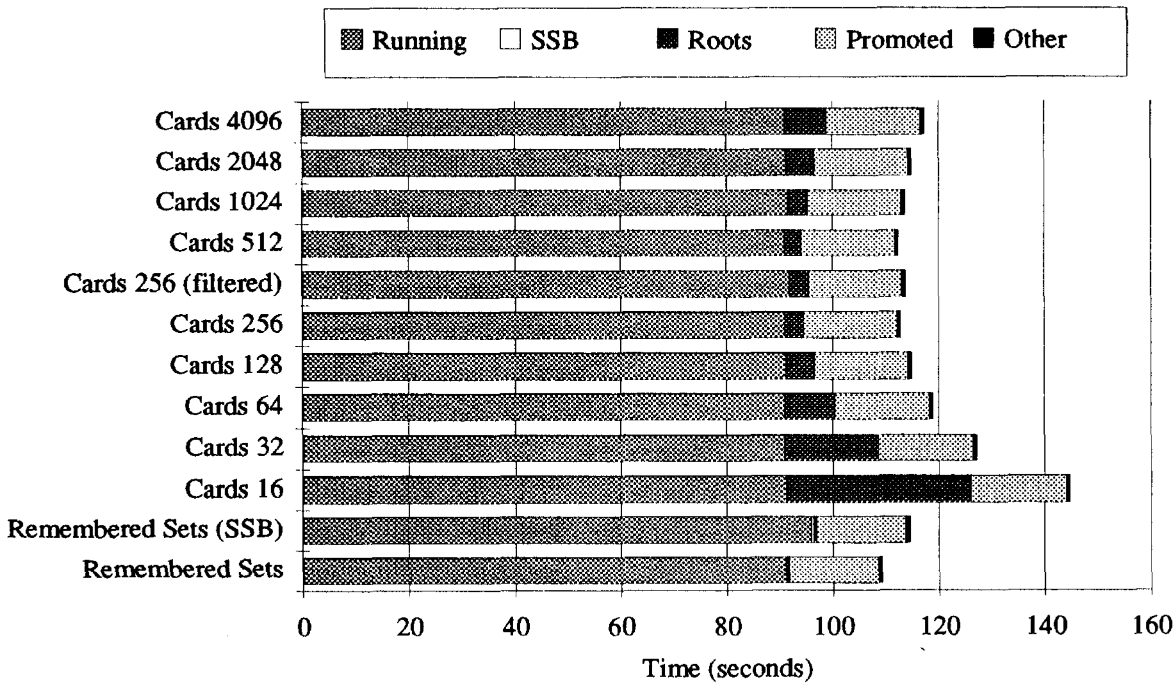

Figure 12: Elapsed time for Destroy (remembering objects)

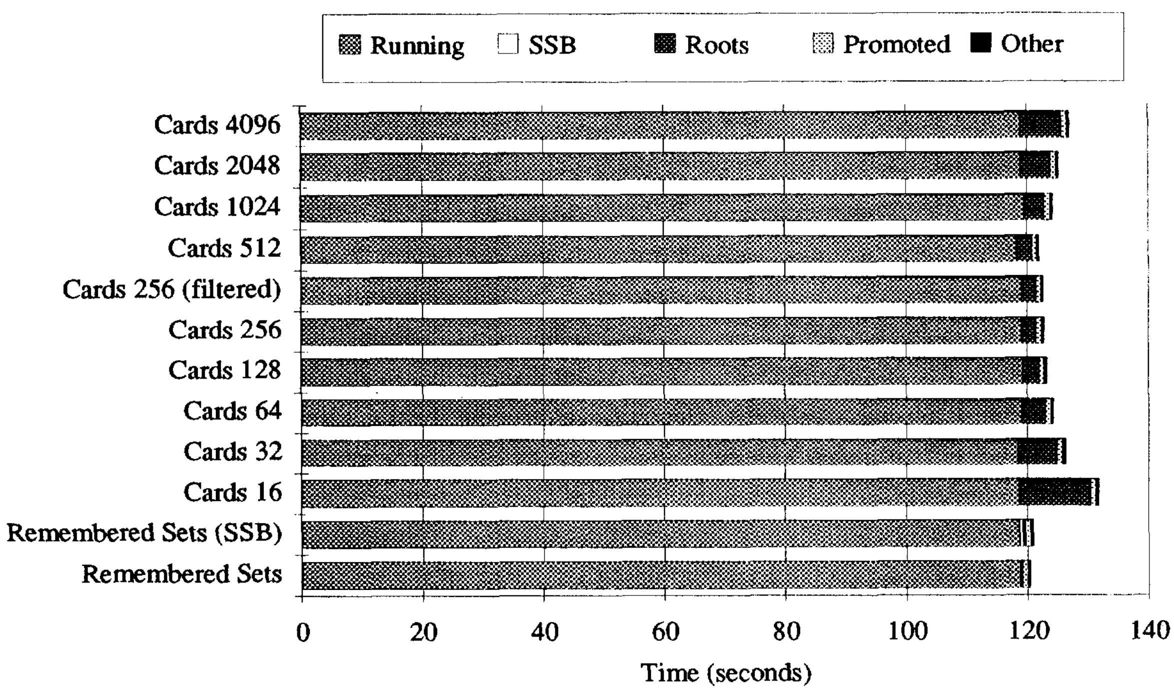

Figure 13: Elapsed time for Interactive (remembering objects)