Remembered sets can also play cards*

Antony L. Hosking
hosking@cs.umass.edu

Richard L. Hudson
hudson@cs.umass.edu

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

Abstract

Remembered sets and dirty bits have been proposed as alternative implementations of the write barrier for garbage collection. There are advantages to both approaches. Dirty bits can be efficiently maintained with minimal, bounded overhead per store operation, while remembered sets concisely, and accurately record the necessary information. Here we present evidence to show that hybrids can combine the virtues of both schemes and offer competitive performance. Moreover, we argue that a hybrid can better avoid the devils that are the downfall of the separate alternatives.

1 Introduction

Generational garbage collectors [6, 10, 11] achieve short 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 small 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 sort 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. This is intuitively explained by the fact that newly allocated objects can only be immediately initialized to point to pre-existing (i.e., older) objects. Pointers from older generations to younger generations can be created only through assignment to pre-existing objects. Detecting such assignments requires special action at every pointer assignment to see whether that pointer must now be considered by the garbage collector when collecting the younger generations. This special action constitutes an extra hurdle at every site in the program that might write a pointer into an object, hence the term *write barrier*.

A number of schemes have been suggested for generating and maintaining the older-to-younger pointer information needed by generational collectors, including special-purpose hardware support [10, 11] and generation by compilers of the necessary inline code to perform the checks in software [2] (adding to the overhead of pointer stores). Ungar [10, 11] uses *remembered sets* to maintain the necessary information on a per-generation basis, recording the objects in older generations that may contain pointers into the generation. The garbage collector examines all the objects recorded in the remembered sets of the younger generations being collected to determine the live (i.e., reachable) objects.

Alternatively, dirty bits can be maintained for older generations indicating whether the generation contains pointers to objects in younger generations. The heap is divided into aligned logical regions of size 2^k bytes—the address of the first byte in the region will have k low bits zero. These regions are called *cards* [9, 12]. Each card has a corresponding entry in a table indicating whether the card might contain a pointer of interest to the garbage collector. Mapping an address to an entry in the table involves shifting the address right by k bits and using the result to index the table.

The card table can be maintained explicitly by generating code to index and dirty the corresponding table entry at every store site in the program. Alternatively, by setting the card size to correspond to the virtual memory page size, updates to clean cards can be detected using the virtual memory hardware. All clean pages in the heap are protected from writes. When a write occurs to a protected page, the trap handler records the update in the card table and unprotects the page. Subsequent writes to the now dirty page incur no further overhead. Note that *all* writes to a clean page cause a protection trap, not just those that store pointers.

The time required to determine the relevant older-to-younger pointers for garbage collection varies with the granularity of the information recorded. In an earlier study [5]

*This work is supported by National Science Foundation Grants CCR-8658074 and CCR-9211272, Digital Equipment Corporation's Western Research Laboratory and Systems Research Center, and Sun Microsystems.

we showed that this is the dominant factor distinguishing different implementations of the write barrier, and that remembered sets offer the best performance because they compactly record just those locations that can possibly contain older-to-younger pointers. In contrast, the time to scan dirty cards is proportional to the size of the cards. While software-implemented card marking schemes are free to choose any power of two for the card size, a page trapping scheme is bound by the size of a virtual memory page. Since modern operating systems and architectures typically use a relatively large virtual memory page size (on the order of thousands of bytes), scanning overheads are very high.

Nevertheless, card marking schemes have more predictable overhead at run time. The simple write barrier imposes a small, bounded amount of additional work on each store. Meanwhile, remembered set maintenance is typically more complicated, depending on the data structures used, and a given store may incur unpredictable cost. For example, the modified slot may already be recorded in the appropriate remembered set, otherwise it must be entered. On overflow, the set must be grown to accommodate the new entry. Such conditional code at every store site is also likely to be detrimental to performance on modern pipelined architectures.

In this paper we consider a hybrid implementation which combines the precision of remembered sets with the simplicity of the card marking write barrier. As the dirty cards are scanned prior to each scavenge, the older-to-younger pointers in those cards are summarized to the appropriate remembered sets, which are then used as the basis of the scavenge. The cards are thenceforth treated as clean. Subsequent scavenges need only update the remembered sets by rescanning just those cards that have been dirtied since the previous scavenge. We compare the performance of several implementations of this hybrid scheme (varying the card size, and using page traps to note updates) with the pure remembered set approach.

2 Implementation

Our experiments are based on a high-performance Smalltalk interpreter of our own design, using the abstract definition of Goldberg and Robson [4]. The implementation consists of two components: the *virtual machine* and the *virtual image*. The virtual machine implements a bytecode instruction set to which Smalltalk source code is compiled, as well as other primitive functionality. While we have retained the standard bytecode instruction set of Goldberg and Robson [4], our implementation of the virtual machine differs somewhat from their original definition to allow for more efficient execution. Our virtual machine running on the DECstation 3100 performs around three times faster than a microcoded implementation on the Xerox Dorado.

We compare several write barrier implementations: a pure remembered set approach, the hybrid card marking schemes (for various card sizes), and a similar page trapping scheme. We also measure the performance of an implementation that assumes an *oracle* to discover which pages of the heap are dirty at each garbage collection. This allows us to determine the optimal performance that could be expected if operating systems were to provide user-level dirty bits (as suggested by Shaw [8], and Appel and Li [3]).

2.1 Remembered sets

To avoid making the remembered sets too large we record only those stores that create pointers from older objects to younger objects. This involves extra conditional overhead at every store site to perform the check, in addition to a subroutine call to update the remembered set if the condition is true. Smalltalk object references are tagged to allow direct encoding of non-pointer immediate values such as integers. Since many object references are immediate, the first action performed by the check is to filter out non-pointer stores. This is followed by a generation test to filter out "initializing" stores to objects in the youngest generation (such stores cannot create older-to-younger pointers).[1] Finally, if the store creates a pointer from an older object to a younger object the remembered set is updated with a subroutine call.

In contrast to Ungar's remembered sets, ours record *locations* that might contain older-to-younger pointers, as opposed to objects. This saves unnecessary scanning upon garbage collection to determine the interesting pointers.

On the MIPS R2000 non-pointers are filtered in 2 cycles. Filtering initializing stores requires another 8 cycles, while filtering the remaining uninteresting stores consumes a further 8 cycles. The entire inline sequence for a store typically comes to 22 instructions, including the store itself, filtering of uninteresting stores, and the call to update the remembered set.

2.2 Card marking

For the card schemes we implement the card table as a contiguous byte array, one byte per card, so as to simplify the store check.[2] 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. Since the most attractive feature of card marking is the simplicity of the write barrier, we omit the checks used in the pure remembered set scheme to filter uninteresting stores.

Thus, on the MIPS R2000 a store can be recorded with just 5 instructions: 2 to load the base of the card table, a shift to determine the index, an add to index the table, and a byte store of zero. Including the store, the entire inline

[1] Note that we do not mean that the stores occur at object allocation. Such stores never need to be recorded since objects are always allocated in the youngest generation. Moreover, Smalltalk objects are always initialized to contain only nil pointers.

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

sequence comes to 6 instructions.[3] If we kept the card table base in a register this sequence would shrink to 4 instructions (registers are at a premium in the interpreter).

2.3 Page traps

The page trap scheme requires no inline code at store sites to detect pointer stores, relying instead on the page protection hardware to trap updates to protected pages. Thus there is no longer any advantage in using a byte table to simplify the store check. Rather, it is more important that the dirty page table consume the smallest possible space. For this reason we use a bit table; setting a bit indicates that the corresponding page is dirty. When a protection trap occurs the bit in the table corresponding to the modified page is set and the page unprotected.

2.3.1 User-level dirty bits

If operating systems were to provide user-level dirty bits (as suggested by Shaw [8], and Appel and Li [3]), the overhead to reflect page traps through to the user-level protection violation handler can be avoided. Presumably, an extra user-level dirty bit would be added to each page table entry, and a system call (`dirty`) provided to return a list of pages dirtied in a given address range since the last time it was called. The system call would clear the user-level dirty bits and enable traps on the specified pages. Traps could then be handled directly in the operating system. This can have substantial savings.

As reported for a MIPS R2000 [1], the time for a user program to trap to a null C routine in the *kernel* and return to the user program is 15.4μs round trip. In contrast, Appel and Li report the corresponding overhead to handle page-fault traps in *user-mode* to be 210μs on a DECstation 3100 (MIPS R2000) running Ultrix 4.1. We have confirmed this with our own measurement of page traps in a tight loop using the same hardware and operating system configuration, obtaining a round-trip time of $\sim$250μs. Note that these measurements are for a tight loop executing many repetitions, and so may tend to underestimate trap costs. Traps interspersed throughout a program's normal execution may perform less favorably, since the OS trap handling code and data structures needed to service the trap may no longer be in the hardware caches. Meanwhile, a call to `dirty` should be no more expensive than current primitives for manipulating page protections, except in copying out the dirty bit information, adding little if any extra overhead to applications that use the new primitive.

[3] We note that the byte store instruction on the R2000 is implemented in the memory hardware as a read-modify-write instruction, requiring several cycles for execution.

3 Experimental setup

We ran our experiments on a DECstation 3100 (MIPS R2000A CPU clocked at 16.67MHz) running ULTRIX 4.1.[4] The benchmarks were run with the system in single user mode and the process's address space was locked in main memory to prevent paging. We measured elapsed time on the client machine using a custom timer board[5] having a resolution of 100 ns. The fine-grained accuracy of this timer allows separate measurement of each phase of a benchmark's execution.

4 Benchmarks

We use two benchmarks to evaluate garbage collection performance. The first is a synthetic benchmark of our own devising based on tree creation. The second consists of several iterations through the standard "macro" benchmark suite that is used to compare the relative performance of Smalltalk implementations [7]. Our benchmarks have the following characteristics:

- **Destroy**—trees with destructive updates: A large initial tree ($\sim$2M bytes) is repeatedly mutated by randomly choosing a subtree to be replaced and fully recreated. The effect is to generate large amounts of garbage, since the subtree that is destroyed is no longer reachable, while retaining the rest of the tree to the next iteration. Rebuilding the subtree causes many pointer stores, some of which create older-to-younger pointers of interest to the garbage collector.

  Each run performs 160 garbage collections. Of the 59 646 stores performed, 37 403 are of non-immediate pointers. Of these, 4 974 are non-"initializing" stores into objects in older generations, of which 4 965 create interesting older-to-younger pointers.

- **Interactive**—10 iterations of the "macro" benchmarks: These measure a system's support for the programming activities that constitute typical interaction with the Smalltalk programming environment, such as keyboard activity, compilation of methods to bytecodes, and browsing.

  Each run performs 137 garbage collections. Of 654 245 stores performed, 154 795 are of non-immediate pointers. Of these, 26 637 are non-"initializing" stores into objects in older generations, of which 2 396 create interesting older-to-younger pointers.

[4] DECstation and ULTRIX are registered trademarks of Digital Equipment Corporation. MIPS and R2000 are trademarks of MIPS Computer Systems. This version of the operating system had some official patches installed that fix bugs in the `mprotect` system call.

[5] We 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.

5 Results

We report the elapsed time of each phase of execution of the benchmark, including:

- *running*: the time spent in the interpreter executing the program, as opposed to the garbage collector (note that running includes the cost of store checks or page traps);
- *roots*: the time spent scanning through remembered sets or card/page tables and copying the immediate survivors; [6]
- *promoted*: the time spent copying any remaining survivors; and
- *other*: the time spent in any remaining GC bookkeeping activities.

Figure 2 plots the results for the remembered set (rem-sets), hybrid page trap (pages), and hybrid card implementations (for card sizes of 16, 64, 256, 1024, and 4 096 bytes) on the Destroy benchmark. The performance that might be obtained using a zero-cost implementation of `dirty` is estimated by taking the *running*, *roots*, and *promoted* times for the oracle-based implementation along with the *other* overheads for the card marking scheme. This is plotted alongside the other measurements (dirty). For comparison, Figure 1 gives the results for the original non-hybrid card and page trap implementations. The results for the Interactive benchmark are similarly shown in Figures 3 and 4.

In contrast with the original implementations, we see that summarizing interesting pointer information into remembered sets for use in subsequent scavenges can significantly reduce the scanning overhead needed to determine root objects for collection. This allows the hybrid card schemes to be competitive with the pure remembered set scheme, with the 1K byte card scheme being best overall. Nevertheless, the pure remembered set scheme still has markedly less overhead to determine the roots. We also note that using a bit table versus a byte table has little effect on root processing time (the *roots* times are very similar for dirty, which scans a bit table, and cards-4096, which scans a byte table).

The results are somewhat less conclusive for running-time overheads. The variation in running time amongst the card schemes can only be explained by hardware data cache effects, since the card schemes all execute the exact same code (barring differences in the shift value used to index the card table). Similarly, the fact that the oracle-based dirty scheme does not exhibit the best running time of the different implementations can only be explained as a result of such data cache effects. Nevertheless, dirty has running time less than pages for both benchmarks. Since these implementations use exactly the same virtual machine and garbage collection data structures, any difference is unlikely to be due to hardware cache effects. Thus we can get some idea of the overhead to field a trap from the operating system, unprotect the appropriate page, and return to normal execution, by subtracting the running time of the oracle-based dirty scheme from that of the pages scheme, and dividing by the number of page traps. This yields a per-trap overhead of 4 387μs for the Destroy benchmark (864 traps), and 1 211μs for the Interactive benchmark (1 656 traps), showing that the traps can be much more expensive than the lower bound of 250μs we obtained by measuring their cost in a tight loop. These results suggest that the frequency of traps affects their cost. Presumably, more frequent traps mean that the hardware caches are more likely to contain the operating system code and data required to service a trap, making for faster trap handling.

Given the number of store checks executed by the card schemes and the number of traps incurred by the page trap scheme for each benchmark, we can determine the tradeoff between using explicit code to maintain dirty bits and a page trapping approach. Ignoring hardware cache effects, the break-even point is determined by the formula:

$$cx = fty$$

where

c = the number of store checks executed by an explicitly coded software scheme;

x = cycles per check;

f = clock frequency (16.67 MHz for DECstation 3100);

t = the number of traps incurred by a page trapping scheme;

y = μs per trap.

For these benchmarks this yields Table 1, which gives the maximum page trap overhead such that a page trapping approach will incur less running time than an alternative explicit implementation having the given overhead per store check. Let us assume that our current 5-instruction sequence for card marking executes in no more than 10 cycles. To be competitive a page trap implementation would have to incur no more than 41μs and 237μs per trap, for the Tree and Interactive benchmarks respectively. These values are significantly lower than the estimated trap overheads for these benchmarks quoted above, and lower even than the 250μs lower bound obtained for a tight loop.

Finally, we note that the hybrid page implementation incurs more page traps than the original implementation, since all pages are considered to be clean after each scavenge, rather than just those that contain no interesting pointers. This increases the running time component for the hybrid page trapping scheme. Nevertheless, the reduction in the number of dirty pages that must be scanned to determine the garbage collection roots makes this tradeoff well worthwhile, resulting in a net reduction in total execution time.

[6] 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.

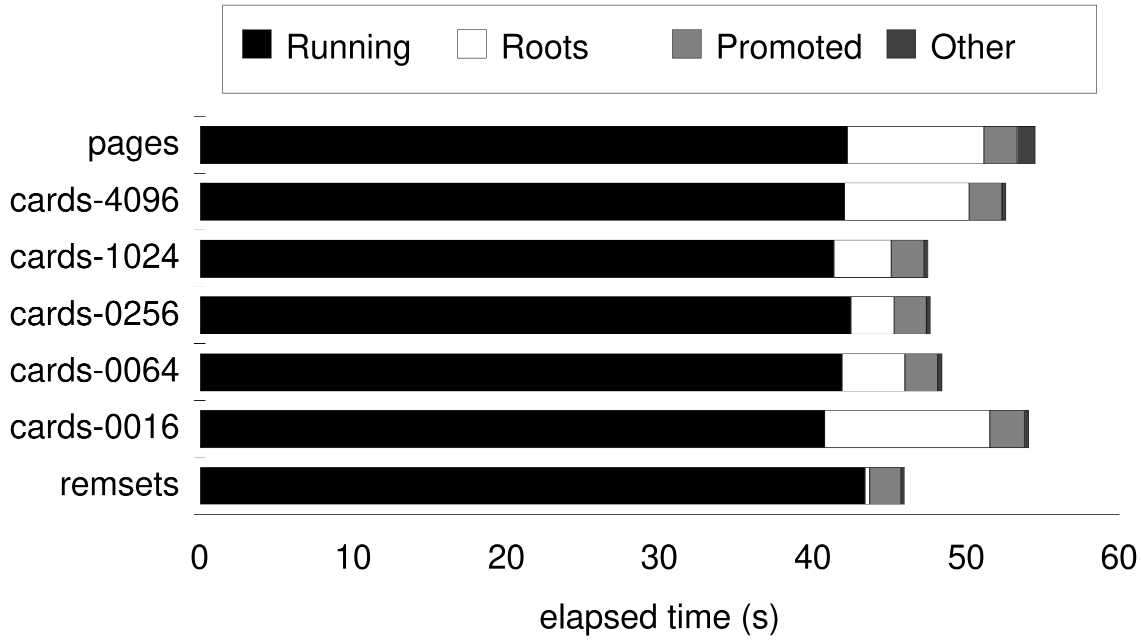

Figure 1: Destroy: old implementation

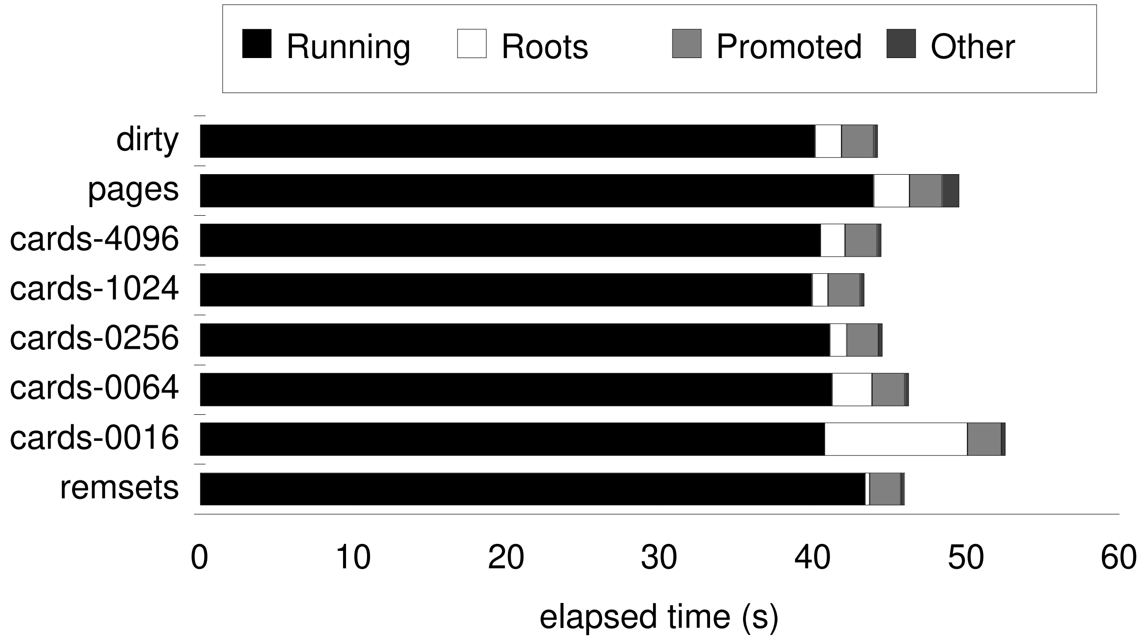

Figure 2: Destroy: new hybrid implementation

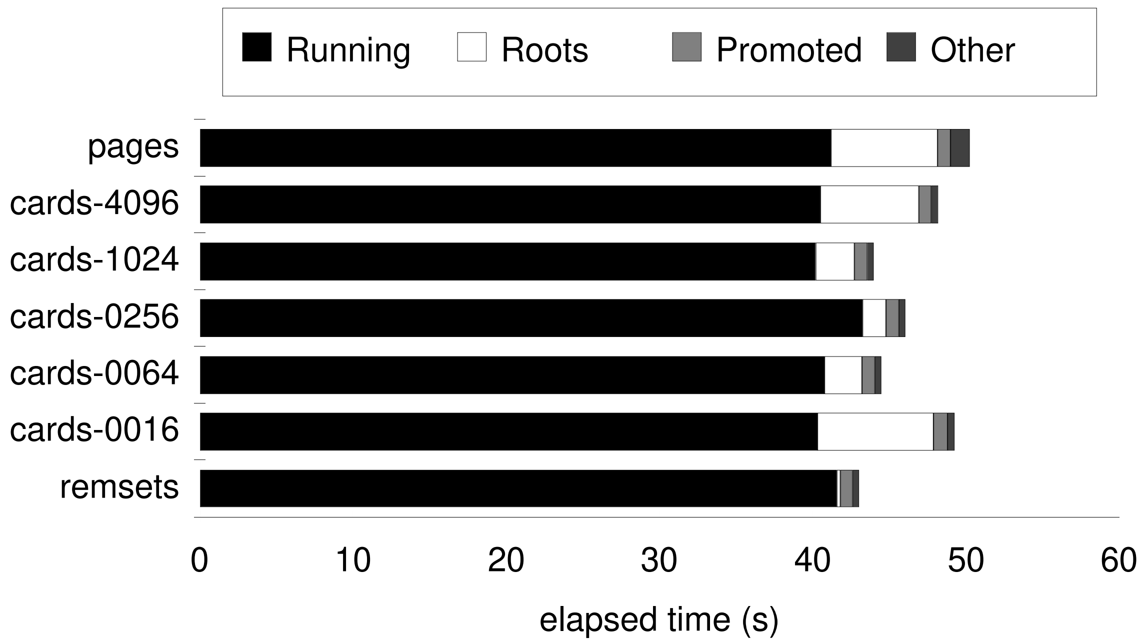

Figure 3: Interactive: old implementation

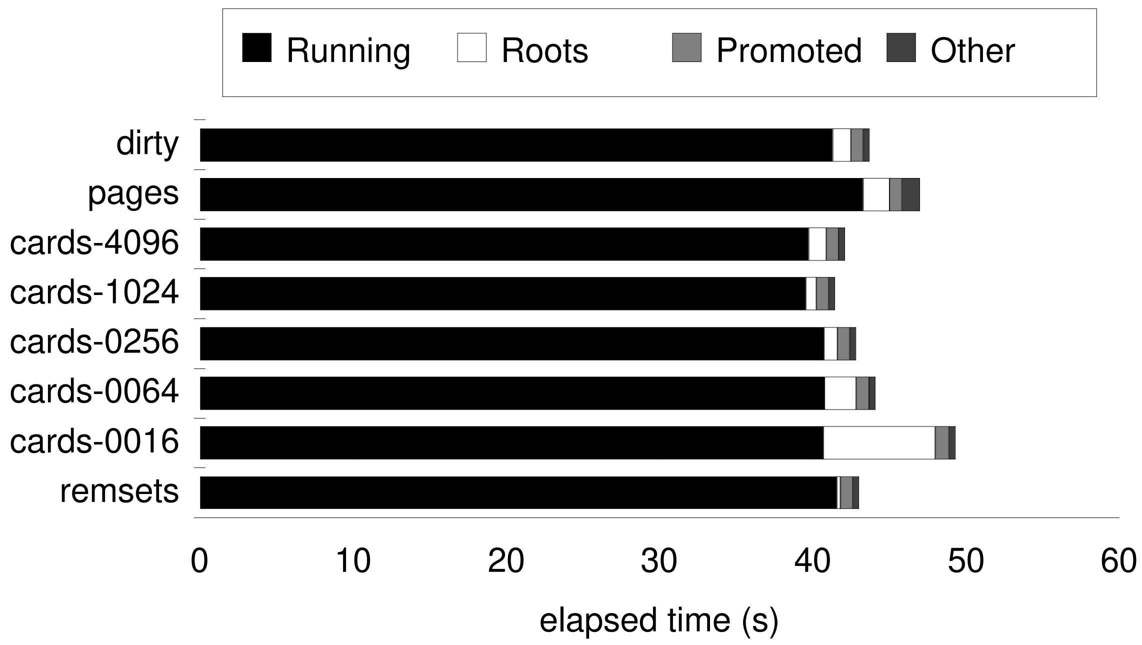

Figure 4: Interactive: new hybrid implementation

	Tree	Interactive
Store checks (c)	59 646	654 245
Page traps (t)	864	1 656
Cycles per check (x)	μs per trap (y)	
1	4	24
2	8	47
4	17	95
5	21	118
10	41	237
15	62	355
20	83	474
50	207	1 185
100	414	2 370
150	621	3 555
200	828	4 740

Table 1: Break-even points for GC implementations that use page trapping vs explicit checks

5.1 Interpreters versus compilers

The experiments and results we have presented are for an interpreted language. An obvious question is how they would change for a compiled language. The results will likely stand since compilation will tend to shrink the time actually spent in the program code, but garbage collection costs will be unaffected since they are part of the language run-time system. Thus, collection time overheads become more important, strengthening our arguments for software rather than hardware approaches in many circumstances. Reducing the incremental costs of noting updates for garbage collection will not improve the situation. Rather, it is better to invest in improving the garbage collection overheads, which are heavily impacted by the granularity of the noted updates. Nevertheless, inline checks do affect program size and possibly instruction cache behavior, so we cannot predict the situation with certainty. The size and frequency of inline checks will affect overhead with respect to a *non-garbage collected* implementation, but in the absence of contrary evidence, it is reasonable to assume that the relative ranking of the various schemes will remain the same.

6 Discussion

Our earlier results [5] showed that, despite the more complicated store checks, remembered sets gave best performance overall because of their precise representation of older-to-younger pointers, allowing the scavenge roots to be discovered quickly. The hybrid scheme used here has eliminated the need to rescan cards that have not changed since the last scavenge, thus reducing the scanning overhead to locate roots. We have also established that a trap implementation is unlikely to be competitive with our software card marking scheme unless page trap overheads can be dramatically reduced. Thus, a combination of card marking with remembered sets seems the best option.

One of the problems with any remembered set scheme is that in the worst case the remembered set for a given generation can grow to be as large as all the older generations—if every word in the older generations contained a pointer into that generation then the remembered set would contain an entry for each of those words. At some point scanning a card becomes less expensive than the corresponding remembered set processing. By arranging for the hybrid garbage collector to detect when remembered sets are becoming overly large and unwieldy, it can dynamically switch over to card scanning on a card-by-card basis. If a given card contains more interesting pointers than can be efficiently recorded with remembered sets, then the card is marked as dirty. The interesting pointers are then rediscovered while scanning at the next scavenge. The collector can re-evaluate this decision at each scavenge, and switch back to recording the interesting pointers in the remembered sets.

7 Conclusions

We have shown that a hybrid scheme which uses card marking to track pointer stores, and remembered sets to summarize the interesting pointers for garbage collection, can offer acceptable performance and attractive behavior. Remembered sets provide accurate determination of interesting pointers, but suffer from high store check costs. Cards offer a simple, predictable store check. Combining these two techniques provides a competitive scheme that has the store check overhead of cards while preserving most of the high precision of remsets, yielding better behaved, more predictable garbage collection.

References

[1] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, California, Apr. 1991. *ACM SIGPLAN Not. 26*, 4 (Apr. 1991).

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

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, California, Apr. 1991. *ACM SIGPLAN Not. 26*, 4 (Apr. 1991).

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

[5] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, Oct. 1992. *ACM SIGPLAN Not. 27*, 10 (Oct. 1992).

[6] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[7] 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.

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

[9] 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.

[10] 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).

[11] 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.

[12] 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).