A Fast Write Barrier for Generational Garbage Collectors

Urs Hölzle
Computer Systems Laboratory
Stanford University
urs@cs.stanford.edu

Introduction

Generational garbage collectors need to keep track of references from older to younger generations so that younger generations can be garbage-collected without inspecting every object in the older generation(s) [LH83], [Ung86]. The set of locations potentially containing pointers to newer objects is often called the *remembered set* [Ung86]. At every store, the system must ensure that the updated location is added to the remembered set if the store creates a reference from an older to a newer object. This mechanism is usually referred to as a *write barrier* or *store check*.

For some stores, the compiler can know statically that no store check is necessary, for example, when storing an integer (assuming that integers are implemented as immediates rather than as real heap-allocated objects). However, in the general case, a store check must be executed for every store operation. Since stores are fairly frequent in non-functional languages, an efficient write barrier implementation is essential. The write barrier implementation described here reduces the write barrier overhead in the mutator to only two extra instructions per checked store.

Card Marking

Our new write barrier implementation is based on Wilson's card marking scheme [Wil89].† In this scheme, the heap is divided into *cards* of size 2^k words (typically, k = 5..7), and every card has an associated bit in a separate bit vector. A store check simply marks the bit corresponding to the location being updated. At garbage collection time, the collector scans the bit vector and, whenever it finds a marked bit, examines all pointers in the corresponding card in the heap.

Unfortunately, card marking as just described can be fairly slow. Since the bit must be inserted into the bit vector, the corresponding word has to be read from memory, updated, and written back. In addition, bit manipulations usually require several instructions on RISC processors. Chambers and Ungar improved on Wilson's scheme by using bytes instead of bits to mark cards [Cha92]. That is, every card in the heap has one byte associated with it; a card is marked simply by storing a special value (e.g., zero) into the corresponding byte. Although this scheme uses eight times more memory, the space overhead is usually still small; for example, with a card size of 32 words = 128 bytes, the space for the byte map is less than 1% of the heap size.

---

† Similar but less efficient schemes have been proposed by Sobalvarro [Sob88] and Shaw [Sha88].

The big advantage of the byte marking scheme is its speed. In the SELF system described in [Cha92], a store check requires just 3 SPARC instructions in addition to the actual store:

```
st  [%obj + offset], %ptr              store ptr into object's field

add %obj, offset, %temp                calculate address of updated word
sll %temp, k, %temp                    divide by card size 2^k (shift left)
st  %g0, [%byte_map + %temp]           clear byte in byte map
```

This code sequence assumes that the register byte_map holds the adjusted base address of the byte map, i.e. byte_map_start_address - (first_heap_word / 2^k), thus avoiding an extra subtraction when computing the index of the byte to be cleared.

A Two-Instruction Write Barrier

Our new write barrier improves on standard card marking by relaxing the invariant maintained by the card marking scheme. The invariant maintained by standard card marking is

bit or byte *i* is marked <--> card *i* may contain a pointer from old to new

Our scheme's relaxed invariant is

bit or byte *i* is marked <--> cards $i..i+l$ may contain a pointer from old to new

where l is a small constant (1 in our current implementation). Essentially, l gives the store check some *leeway* in choosing which byte to mark—the marked byte may be up to l bytes away (in the direction of lower addresses) from the "correct" byte. The relaxed invariant allows us to omit computing the exact address of the updated word: as long as the offset of the updated word (i.e., its distance from the beginning of the object) is less than $l * 2^k$, we can mark the byte corresponding to the object's address rather than the byte corresponding to the updated field. Thus, the common-case store check is only two instructions:

```
st  [%obj + offset], %ptr              store ptr into object's field

sll %obj, k, %temp                     calculate "approximate" byte index
st  %g0, [%byte_map + %temp]           clear byte in byte map
```

Usually, a leeway of one ($l = 1$) is sufficient to cover virtually all stores (except for stores into array elements). For example, the card size is 128 bytes in the SELF system and thus all stores into any of the first 30 fields of objects can use the fast code sequence.†

If the system can determine object boundaries, the maximum leeway restriction can be lifted entirely: in this case, the relaxed variant can be stated as

bit or byte *i* is marked <--> card *i* or the last object starting in card *i* may contain a pointer from old to new

† The first 2 fields of every object are used by the system, so the first user-defined field has an offset of 8 bytes.

When scanning a card, the garbage collector simply needs to make sure that the last object in the card is scanned completely even if only part of it belongs to the card. In other words, the collector continues scanning beyond the last word of the card until encountering the next object header. However, with very large objects, this scheme could result in high scanning costs since the amount of scanning is no longer bounded by a fixed maximum leeway but only by the size of the last object. Fortunately, this problem can be mitigated by marking the exact card for stores into arrays; then, if the last object starting in a card is an array, the collector need not continue scanning into the next card(s).

Performance evaluation

We have implemented the new card marking scheme with a leeway of 1 in the SELF system and have measured its performance for the SELF programs briefly described in Table 1.

Benchmark	Size (lines)	Description
DeltaBlue	544	two-way constraint solver
Parser	522	parser for a version of SELF
PrimMaker	1097	program generating “glue” stubs for external SELF primitives
Richards	413	simple operating system simulator
Cecil	10,600	interpreter for the Cecil language

Table 1: Benchmark programs

All programs were run on a lightly loaded SPARCstation-2. The time consumed by store checks was determined by running the programs under an instruction-level simulator modelling the exact hardware behavior of the workstation, including its caching behavior. The scavenging overhead was measured by running the programs 100 times with a version of the virtual machine instrumented with gprof. Figure 1 and Table 2 show the results.

Benchmark	execution time (ms)	number of scavenges	store checks	card scanning	total scavenging	total overhead
			(as percentage of total execution time)			
DeltaBlue	355	12	4.4%	0.86%	5.5%	9.9%
Parser	204	5	0.4%	0.58%	5.0%	5.4%
PrimMaker	448	12	1.0%	0.42%	9.4%	10.4%
Richards	860	0	7.8%	0.00%	0.0%	7.8%
Cecil	2,228	44	0.5%	0.65%	9.7%	10.2%

Table 2: Scavenging overhead

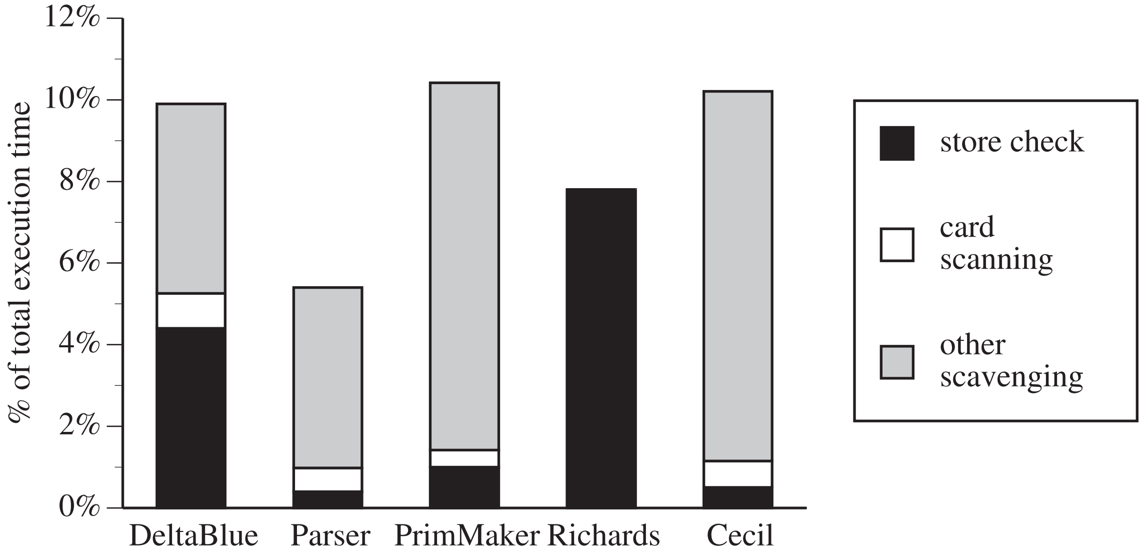

Figure 1. Distribution of Scavenging Overhead

The system we measured optimizes store checks in two ways. First, no store check is performed for initializing stores (i.e., by the _Clone primitive), unless the new object is allocated in OldSpace (which happens extremely rarely, if no space is available anywhere else). Second, the compiler omits store checks if the stored value is known to be an integer or float, or if it is known to be an "old" object.†

All of the programs show scavenging overheads of between five and ten percent of total runtime, and store check overheads between roughly 0.5% and 8% (which agrees well with measurements of Lisp programs taken by Zorn [Zor90]). Even with the fast store check code, the card marking overhead significantly exceeds the card scanning time for three of the benchmarks (Richards, DeltaBlue, and PrimMaker). For the other two benchmarks, the card marking and card scanning overheads are about equal.‡ For the programs we measured, card marking consumed between 40% and 100% of the total write barrier cost.

Discussion

Our numbers differ somewhat from those obtained in Chambers' study of garbage collection overhead in the SELF system [Cha91]. Generally, Chambers' measurements of store checking overhead showed even higher overheads than our study, ranging from 3% to 24% for a similar set of programs. However, these measurements cannot be directly compared to ours for several reasons. First, too many factors are different: hardware, compiler, store-check-implementation, and the SELF programs all have changed. Second, Chambers estimated store-checking time by running two versions of the system, one completely without store checks and one with the standard store checking code, with an "eden" space large enough to prevent any scavenging

† Old object never become new again, so if the constant being stored is old, the compiler can safely omit the store check.

‡ The above numbers overestimate the card scanning overhead somewhat since the scanning code is written in C rather than in assembler where it could exploit doubleword load instructions.

during execution of the benchmark. Thus, his measurements may well be less accurate than ours since they could be affected by caching effects unrelated to the actual store checking overhead.†

The only write barrier with comparably low mutator overhead that we are aware of is the sequential store buffer (SSB) described by Hosking et al. [Hos92]. The SSB records all locations being stored into; like our scheme, an SSB store barrier takes two instructions (a store and an add). At scavenge time, the SSB is scanned and the locations referenced by the pointers in the buffer are examined. Although the actual SSB write barrier is just as fast as our scheme, a significant drawback of the SSB scheme is that the processing effort at scavenging time is proportional to the number of stores performed between scavenges, rather than to the size of the used part of oldspace (as for card marking). In non-functional languages like SELF, stores can be quite frequent, and thus the store buffer can grow very large. Table 3 shows the number of stores

Benchmark	number of checked stores	“ideal” SSB overhead (estimated)	“realistic” SSB overhead (estimated)	card marking (check + scanning)	ratio of “realistic” to card marking
DeltaBlue	73,782	12.11%	17.30%	5.26%	3.29
Parser	3,810	1.04%	1.48%	0.98%	1.51
PrimMaker	22,021	2.88%	4.11%	1.42%	2.90
Richards	320,480	21.21%	30.30%	7.80%	3.88
Cecil	57,231	1.50%	2.15%	1.15%	1.87

Table 3: Estimated Overhead of Sequential Store Buffer

requiring a check for each program, and an estimate of the overhead of a SSB implementation. The column labelled “ideal SSB overhead” shows the estimated overhead of an ideal (but somewhat unrealistic) SSB implementation consuming a combined overhead (store check and buffer scanning) of 21 cycles per store.‡ The “realistic SSB overhead” column assumes a combined overhead of 30 cycles per store which still requires careful assembly programming and very low cache miss ratios; it is intended to represent a very good practical SSB implementation. Nevertheless, the estimated “realistic SSB” overhead is between 1.5 and almost 4 times higher than the card marking overhead for all of our benchmarks, and even the “ideal” SSB would not beat card marking for any of our benchmarks. It appears that stores are too frequent in SELF programs for the Sequential Store Buffer to be competitive.

---

† Machines with direct-mapped caches such as the Sun workstations used in both studies can exhibit pronounced variations in cache effectiveness. It is our experience that the execution time of programs can vary by up to 10 or 15 percent depending on the exact positioning of code or data on such machines.

‡ 9 cycles for the store check (1 cycle add, 3 cycle store, 5 cycles uncached write overhead) and 12 cycles for the scanning (2 loads at 2 cycles each, 2 cycles for a compare-and-branch, and 6 cycles cache overhead for the load of the pointer from the SSB).

Conclusions

We propose a new variant of card marking that reduces the marking overhead to just two instructions per store. Even with the two-instruction card marking code, the marking overhead consumes a significant fraction of the total cost of maintaining the write barrier. For the programs we measured, marking consumed between 40% and 100% of the total write barrier cost (the other part being the time for scanning the cards at scavenge time). For two of the programs, the marking overhead even accounted for more than 40% of the total garbage collection overhead.

Given that the marking cost represents roughly half of the total write barrier cost, it appears that write barrier implementations with a significantly higher mutator overhead than the two-instruction sequence presented here will not be competitive in terms of the total write barrier cost. The Sequential Store Buffer—as far as we know, the only other write barrier scheme with a two-instruction overhead—is likely to be significantly slower than card marking (based on the estimates shown in Table 3) because its scavenging overhead is too high. Thus, a card-marking scheme such as the one described here appears to be an excellent choice for maintaining the write barrier in efficient garbage-collected systems.

Acknowledgments

I would like to thank Bob Cmelik for *shade*, the tracing tool used for the measurements, and Gordon Irlam for the *spa* SPARC simulator. I would also like to think Benjamin Zorn and the members of the SELF group for their comments on an inital version of this paper.

References

[Cha91] Craig Chambers. “Cost of garbage collection in the Self system.” OOPSLA'91 GC Workshop, October 1991.

[Cha92] Craig Chambers. “The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.” Ph.D. Thesis, Computer Science Department, Stanford University, 1992.

[Hos92] Antony Hosking, J. Eliot B. Moss, and Darko Stefanovic. “A comparative performance evaluation of write barrier implementations.” In *OOPSLA'92 Proceedings*, pp. 92-109,

[LH83] Henry Lieberman and Carl Hewitt. “A Real-Time Garbage Collector Based on the Lifetime of Objects.” Communications of the ACM 26 (6): 419-429.

[Sha88] Robert A. Shaw. “Empirical Analysis of a LISP System.” Stanford University, Computer Systems Laboratory, Technical Report CSL-TR-88-351, 1988.

[Sob88] Patrick G. Sobalvarro. “A lifetime-based collector for LISP systems on general-purpose computers.” B.S. Thesis, MIT EECS Dept., Cambridge Ma, 1988.

[Ung86] David Ungar, *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, Cambridge, MA, 1986.

[Wil89] Paul R Wilson and Thomas G Moher. “A card-marking scheme for controlling intergenerational references in generation-based GC on stock hardware.” SIGPLAN Notices 24 (5), pp. 87-92.

[Zor90] Benjamin Zorn. “Barrier Methods for Garbage Collection.” Technical Report CU-CS-494-90, University of Colorado at Boulder, November 1990.