

Concurrent Remembered Set Refinement in Generational Garbage Collection

David Detlefs
Ross Knippel
Sun Microsystems¹
david.detlefs@sun.com
ross.knippel@sun.com

William D. Clinger
Northeastern University
will@ccs.neu.edu

Matthias Jacob
Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544
mjacob@cs.princeton.edu

Abstract

Generational garbage collection divides a heap up into two or more generations, and usually collects a *youngest* generation most frequently. Collection of the youngest generation requires identification of pointers into that generation from older generations; a data structure that supports such identification is called a *remembered set*. Various remembered set mechanisms have been proposed; these generally require mutator code to execute a *write barrier* when modifying pointer fields. Remembered set data structures can vary in their *precision*: an imprecise structure requires the garbage collector to do more work to find old-to-young pointers. Generally there is a tradeoff between remembered set precision and barrier cost: a more precise remembered set requires a more elaborate barrier. Many current systems tend to favor more efficient barriers in this tradeoff, as shown by the widespread popularity of relatively imprecise *card marking* techniques. This imprecision becomes increasingly costly as the ratio between old- and young-generation sizes grows. We propose a technique that maintains more precise remembered sets that scale with old-generation size, using a barrier whose cost is not significantly greater than card marking.

¹Solaris and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

1 Introduction

Generational garbage collection [24] is a widespread and popular technique [30, 21, 32, 6, 3]. Generational collection usually both decreases average GC pause times (since most collections target just the youngest generation) and also increases GC efficiency (by concentrating collection work on the youngest generation, whose objects often die young). In collecting the youngest generation, objects in older generations are considered as roots; young objects reachable from old objects are considered live. Therefore, young objects reachable from older generations must be identified in order to have a correct collection. Further, many systems use some form of relocating garbage collection. For example, compaction enables fast linear allocation in a contiguous free space. This imposes a further requirement, that *all* pointers to young objects from other generations be identified, since those pointers must be updated.

Data structures that support iteration over old-to-young pointers (or pointers that cross other, more general boundaries) are often called *remembered sets*. Since mutator updates of pointer fields in objects may create new pointers that must be remembered, generational systems usually have an associated *write barrier* that is executed along with such updates to maintain the remembered set.

Remembered set implementations differ in their *precision*: how precisely they describe the locations of the pointer fields in old-generation objects that must be scanned. A highly precise remembered set leads to faster young-generation collection. A remembered set may be imprecise in two ways:

first, entries may only approximately describe the locations of cross-generational pointers, and second, some entries may not actually denote cross-generational pointers. Such imprecision leads to extra work during collection.

Unfortunately, there is a tradeoff between remembered set precision and the cost of the write barrier code executed by the mutator threads: a more precise remembered set generally requires a more elaborate, and therefore more expensive, write barrier. At one extreme, the “null” remembered set implementation would be extremely imprecise (scan the entire old generation to find cross-generational pointers), but impose no overhead on the mutator, since no barrier is necessary. At the other extreme, every pointer update could execute code to either insert or remove, as necessary, the updated location from a hash table representing the remembered set. Such a scheme would be very precise, but also very expensive.

Many remembered set/write barrier combinations have been proposed [33, 3]; these populate many points on this tradeoff curve. *Card marking*, in particular, has become a popular technique. In this technique, the heap is partitioned into equal-sized cards, and a *card table* array is allocated, with an entry for each card of the heap. Card table entries are initially *clean*; the mutator write barrier marks the card containing the updated field (or the head of the object containing the field, in a variant) *dirty*. The collector must scan the card table to find dirty entries, and then scan the corresponding dirty cards to find the cross-generational pointers, if any, created by the writes. This technique has a quite inexpensive write barrier (as few as 2 extra instructions per pointer update) and small memory overhead (a typical configuration has a one-byte card table entry for every 512-byte card). However, card marking is not particularly precise: the collector must scan the entire card table to find marked cards; cards may be marked by “false positive” pointer updates that create only intra-old-generation pointers; and even for dirty cards the collector may scan an entire card to find just one cross-generational pointer.² Many applications are requiring increasingly large heaps, while requiring pause times to remain small. Since the cost of card marking increases as a function of old-generation size and mutator pointer update rate, card marking may not scale to these larger

²Some card marking variants increase precision by increasing write barrier cost, for example, by filtering out old-to-old pointers in the write barrier.

heaps.

One positive trend is that applications with large heaps are increasingly multithreaded and are run on multiprocessors. However, tuning an application (and run-time system, and operating-system kernel) to scale with additional processors is often difficult. In this paper we propose that an available underutilized processor can be put to productive use by concurrent *refinement* of the remembered set to increase its precision. We investigate several strategies for remembered set representation, several alternative write barriers (two of them novel), and different concurrent processing methods. By using concurrent refinement, we are able to keep remembered set scanning in young-generation collection more nearly dependent only on the number of old-to-young pointers at the time of collection, and relatively independent of the size of the old generation or the application’s pointer mutation rate. For one customer’s application, concurrent refinement of the remembered set reduces both the average pause time and the total cost of garbage collection.

The rest of the paper is organized as follows. Section 2 discusses related work. In section 3 we describe the remembered sets we used. Section 4 describes the write barriers we considered. Section 5 describes the concurrent processing that uses the output of those barriers to refine the remembered set. Section 6 describes our experimental results. We close the paper with conclusions and future work.

2 Related work

In this section we describe related previous work. First we consider explorations of remembered set implementations and write barrier code sequences, then the application of concurrency to garbage collection.

2.1 Remembered sets and write barriers

Many remembered set representations have been proposed in the literature, with associated write barriers. Chapter 7.5 of Jones and Lins’ garbage collection reference [21] contains an excellent

overview.³ The *sequential store buffer* scheme of Hosking, Moss, and Stefanović [19] is similar to our log-based barriers in that it separates data structures updated by mutator barriers from remembered set representations. However, the details of the mutator barrier code are different, and, while they processed logs incrementally as they overflowed, they did not process logs concurrently. The summary table mechanism used in our card-table-based remembered set implementation is similar to the the hybrid card marking scheme of Hosking and Hudson [20].

Sobalvarro [28] described a form of 2-level card table, but it required hardware support, and he did not address concurrent processing.

Recently, Fitzgerald and Tarditi [15] compared a number of different barrier and remembered set implementations including card marking, sequential store buffers, and a 2-level card table. Their conclusion was that, for their benchmarks, choice of write barrier did not greatly influence the performance of their system. They did not consider offloading work to a concurrent thread, as is done in this paper.

2.2 Concurrency

Concurrent garbage collection is not new. Steele [29] had an early algorithm. Dijkstra, Lamport, *et al.* introduced *on-the-fly* collection [10], a form of concurrent mark-and-sweep. This was extended by Kung and Song [22]. More recently, these ideas have been revisited for ML [11, 12] and for the JavaTM programming language [14, 13]. Baker invented an incremental copying collector [5], which was implemented in hardware on Lisp machines [25]. Ellis, Li, and Appel implemented this idea on stock hardware with a virtual-memory-based barrier, and added true concurrency [23]. However, most of these have little relevance to the present work beyond the fact they involve garbage collection and concurrency.

Two other collection approaches are similar to the current work in that they use log-based write barriers whose output is processed by a concurrent thread devoted to GC. The first of these is *concurrent reference counting*. In this approach, the

write barrier logs the address of the modified field, and its value before the modification. DeTreville [9] described such a collector for Modula-2+ (with a backup mark-sweep collector to detect garbage cycles). More recently, Bacon *et al.* [4] described another such system (with a concurrent local cycle-detection algorithm.) Logging is especially useful when this style of collection is applied to multi-threaded systems, since write barriers can write only to thread-local logs, and all modifications to object reference counts are done by a concurrent thread.

The other log-based collector we will mention is the *concurrent replicating collection* technique of O’Toole and Nettles [26]. In this approach all mutator updates (including those to non-pointer fields) are logged. A collector thread performs a concurrent copying collection. During collection, the mutator observes only from-space pointers. The GC thread ensures that logged updates are applied to to-space versions of already-copied objects, with pointers translated appropriately, and that updates that modify the pointer graph are handled correctly. This has little in common with the present paper beyond the use of logs and concurrency.

Another family of concurrent collectors with some relevance to the current work starts with the “mostly-parallel” collector of Boehm, Demers and Shenker [7]. Variations on this theme have been explored by Printezis and Detlefs [27] and by Heil and Smith [17]. The *process M* presented by Boehm *et al.* (termed *concurrent precleaning* in Printezis and Detlefs) could be considered a kind of concurrent remembered set refinement: in this application, the remembered set records pointers that have been modified during a concurrent marking phase, and whose referents therefore may not be marked. The concurrent process attempts to ensure that a necessary “stop-world” phase to complete the marking is short, much as the concurrent work in the current paper tries to make “stop-world” young-generation collections shorter.

While previous efforts bear some relation to the current ideas, none have explicitly had concurrent refinement of remembered set precision as a goal.

³Note, though, that we are departing somewhat from their terminology in this paper, and using *remembered set* to refer to the general class of data structures for recording cross-generational pointers, rather than a specific such data structure.

3 Remembered sets

We consider two remembered set organizations in this paper. The first is the default remembered set representation of the system in which we perform our measurements. This is a card table, augmented with a *summary table*. As described so far, a card table entry is either *clean*, indicating the absence of cross-generational pointers, or *dirty*, indicating their possible presence. When a young-generation collection scans a dirty card and finds a cross-generational pointer, and the collection does not *promote* the referent of that pointer out of the young generation, it leaves the card dirty to ensure that this pointer is also scanned in the next collection. The summary table makes this more efficient: the summary table can represent the positions of up to a maximum of k pointers within the card. If scanning of a dirty card finds k or fewer (but not zero) cross-generational pointers, then the card table entry is set to a new value *summarized* and the number and positions of those pointers are recorded in the summary table. Dirty cards containing more than k cards remain dirty; we refer to these as *overflow* cards.

The value k is a compile-time constant; how large must it be for summarization to be effective? Figure 1 contains histograms classifying cards by how many old-to-young pointers they contain at the time of collection, for each of the benchmarks described in section 6.2, summing over all collections. This is a “busy” figure; we don’t intend for the reader to note fine details. But we will note that the great majority of cards are “clean,” that is, contain no cross-generational pointers. Cards containing more than 16 cross-generational pointers are quite rare (note that the y-axis is logarithmic). Treating cards with more than 16 cross generational pointers as overflow cards has negligible cost.

One more technique can be used to decrease space overhead. As described so far, each card has a corresponding summary table able to hold k pointer offsets; in the implementation, each offset occupies a byte. In all but two of the benchmarks, no individual histogram bucket above the one for 2 pointers contains more than 100 cards (summed over all the collections that occur in the benchmark.) The other two benchmarks have quite large heaps, so the relative sizes of the histogram buckets are still small. Thus we could set k to 2, and use an encoding scheme in which a 2-byte summary table entry is identifiable as either a sequence of pointer offsets,

or else an index into a separate *mid-size table*, which is able to represent 16 offsets. Few such entries will be necessary, and using $k = 3$ would ensure that more than 2 million would be addressable. We have not yet implemented this variant.

Note that the use of a card table as a remembered set does not require the use of a write barrier that updates the card table directly; our log-based barriers, for example, will use this representation also.

Our second remembered set organization is really just an enhancement of the first: it is a two-level card table. Each entry in the smaller, *coarse-grained* table corresponds to some number of entries in the larger, *fine-grained* table. (In our implementation, this ratio is always of the form 2^N , for some N .) A coarse-grained entry is clean only if all the corresponding fine entries are clean.

Clearly, a 2-level table scales better with large heaps. If a young-generation collection is required to scan an entire fine-grained table to find non-clean cards, it may spend a significant amount of time just skipping clean cards. Using the coarse-grained table speeds up scanning of large clean regions by a factor of 2^N .

One of the write barriers investigated below dirties both the coarse-grained and fine-grained cards corresponding to the updated location. A novel feature of our two-level table organization is the observation that by sacrificing a relatively small amount of address space, we can arrange to have a common *card table base* value for both tables. This speeds up the barrier code; see section 4.2 for details.

4 Write barriers

In this section we describe each write barrier that we have implemented by showing the barrier code that is executed following an assignment of the form $x.f = y$ where x is a reference to an object and y is an expression of reference type. We will show the barrier code as SPARC[®] assembly language, using `%rx` to stand for a general register that contains x . We will use `%reg1`, `%reg2`, *etc.*, to stand for scratch registers, and the constant `foffset` to stand for the offset of field $x.f$ from the address of x itself.

The write barriers that we have implemented fall

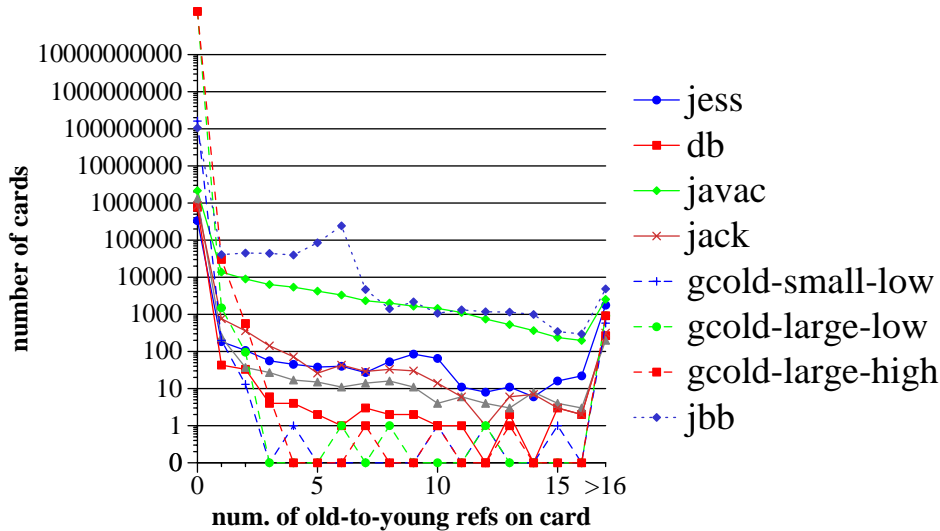


Figure 1: Histogram of number of old-to-young pointers on cards

into three categories:

1. barriers that directly update a 1-level card table
2. barriers that directly update a 2-level card table
3. barriers that adjoin an entry to a log buffer

4.1 Updating a 1-level Card Table

For our experiments, each byte of the 1-level card table represents $512 = 2^9$ bytes of the heap. Our straightforward *card-table* write barrier is

```
sethi    %reg3,%hi(base_address1) ! see below
add     %rx,foffset,%reg1    ! %reg1 = &x.f
srl     %reg1,9,%reg2
stb     %g0,[%reg2+%reg3]    ! mark card dirty
```

In the SPARC assembly code shown here, the `%g0` register always holds the constant zero; this value is used to represent a dirty card precisely because this register is available. This barrier code assumes that both the heap and card table are aligned on a 512-byte boundary, and that `base_address1` is related to the lowest heap address `H` and the lowest address of the card table `CT1` by

$$\text{base_address1} = (\text{CT1} - (\text{H} \gg 9))$$

The barrier code also assumes that `CT1` has been aligned so that the low-order bits of `base_address1` are zero.

This barrier code sequence is similar to the two-instruction sequence of Hölzle [18].⁴ We actually used a variant of this barrier in which, in any method that might execute a write barrier, a register is dedicated to holding the `base_address1` value, and the `sethi` instruction that initializes this register is executed once on method entry, not for every write barrier. The dedicated register is a SPARC local register, and is therefore preserved across calls by the the register window mechanism.

4.2 Updating a 2-level Card Table

For our 2-level card table we combined the 1-level card table with a coarse-grained card table in which each byte represents $16384 = 2^{14}$ bytes, or 32 cards in the fine-grained table. We aligned the address `CT0` of the coarse-grained table with respect to the fine-grained table so that

$$\begin{aligned} \text{base_address1} &= (\text{CT1} - (\text{H} \gg 9)) \\ &= (\text{CT0} - (\text{H} \gg 14)) \end{aligned}$$

which allows us to use a single base register for both card tables. Our straightforward write barrier for the 2-level card table, which we will identify as *card-table2*, is

⁴Except that it marks the card containing the precise location of the modified field rather than the “imprecise” location of the object head, which is used in Hölzle’s barrier. Hölzle’s optimization saves an instruction and, more importantly a register; it is more important on register-poor architectures such as x86 than on a register-rich RISC architecture such as the SPARC. Using precise marking simplifies our GC code.

```

sethi    %reg3,%hi(base_address1)
add      %rx,foffset,%reg1 ! %reg1 = &x.f
srl      %reg1,14,%reg1
srl      %reg1,9,%reg2
stb      %g0,[%reg1+%reg3] ! do coarse table
stb      %g0,[%reg2+%reg3] ! do fine table

```

As before, we actually used a variant with a dedicated local register for the `base_address1` value, initialized by the first `sethi` instruction only on entry to a method, instead of within the write barrier.

4.3 Adjoining to a Log Buffer

The design space becomes much larger when pointer writes are merely logged so that the remembered set can be updated later or by a concurrent log-processing thread. In this section we describe only two of the possible designs.

Both of the write barriers that we describe adjoin an entry to a thread-local write log buffer, which is essentially an array of heap locations that have been the left hand side of a pointer assignment. A global register `%next` is dedicated to point to the next entry in the log buffer. The main problem with this kind of barrier is that the log buffers can overflow, and explicit tests for overflow are expensive. One straightforward solution used in the past [19] is to terminate the log with a write-protected page and to detect overflow via a SIGSEGV exception. However, for overflow frequencies corresponding to reasonably-sized log buffers, we found that Unix signal handling is too expensive. Therefore, we investigated two barriers that handle log buffer overflow in other ways.

The first we call the *misalignment-utrap* barrier. The UTRAP mechanism of the SolarisTM operating system, like UNIX signals, is a mechanism for handling hardware exceptions. It essentially handles only only misaligned accesses, but is about one hundred times as fast as the UNIX signal-handling mechanism for this exception. We therefore designed a barrier that performs a misaligned store (on a non-word boundary) when the log buffer overflows, and use the UTRAP mechanism to handle the misalignment exception. Here is the barrier:

```

add      %rx,foffset,%reg1 ! %reg1 = &x.f
srl      %next,n-1,%reg2 ! %reg2 = %next>>(n-1)
st       %reg1,[%next-4]
and      %reg2,6,%reg2 ! %reg2 = 4 or 6
add      %next,%reg2,%next ! %next = %next + %reg2

```

The first instruction produces the precise address of the field modified. (An alternative version would elide the first instruction and log the object head rather than the field address, performing less mutator work at the cost of more work for concurrent refinement. This observation also applies to the *self-pointing* barrier described below. We did not implement this alternative.)

The `%next` register normally points to the next entry plus four, but when the log buffer is full `%next` points to the next entry plus six, which will cause a misalignment trap during the store instruction. We arrange this by using 2^n -byte log buffers that are each aligned on a 2^{n+1} -byte boundary but not aligned on a 2^{n+2} -byte boundary. The shift-right-logical and “and” instructions therefore generate the value 4 in `%reg2`, but generate a 6 when the buffer is full.

Our *self-pointing* barrier takes a quite different approach towards minimizing the cost of log buffer overflow: it attempts to avoid overflow altogether. Each mutator thread has an associated set of log buffers, linked in a sequence. Buffer entries are initially initialized with pointers to their own locations, except for the last entry of a buffer, which is initialized with the address of the first entry of the next buffer of the sequence (or NULL for the last entry of the last buffer). As with the misalignment-utrap barrier, a global `%next` register contains the address of the next log buffer entry to be written. The barrier stores through `%next`, then updates `%next` with the value read from the next entry:

```

add      %rx,foffset,%reg1 ! %reg1 = &x.f
st       %reg1,[%next]
ld       %next,[%next+4]

```

When NULL is read on one barrier and stored through on the next, a SIGSEGV handler adds a new log buffer to the thread’s sequence. However, a concurrent refinement thread (see section 5) will continually be processing completed log buffers at the head of the thread’s sequence. When it completes the head log buffer, it unlinks it from the thread’s sequence and relinks it at the end. It also resets the final entry of the old last buffer to contain the address of the first entry of the new one, rather than NULL. If this happens in a timely manner, the mutator thread will never observe `%next` to contain the NULL value, and will never overflow a buffer. Adding log buffers when this does occur increases the buffer space devoted to the thread, decreasing the likelihood of future occurrences.

5 Concurrent refinement

For each (compatible) combination of one of the remembered set representations described in section 3 and one of the write barriers described in section 4, we implement a concurrent thread that processes the information produced by the barrier in order to produce a more precise remembered set. First we consider aspects of concurrent refinement common to all combinations, then we consider points specific to refinement with card-table and log-based barriers, respectively.

5.1 Common considerations for all barriers

At an abstract level of description, all of the concurrent processing functions sleep for some interval, then traverse the remembered set data structure, attempting to refine it (by eliminating false positives and/or making pointer location data more precise). When a collection occurs, the “abstract” remembered set must be considered to include any write barrier data structures, such as log buffers, not yet processed by the refinement thread. The collector starts by completing any such outstanding processing. Various heuristics may be used to control the sleep interval between concurrent refinement intervals. The goal of such heuristics is to simultaneously minimize both the outstanding work necessary to bring the “concrete” remembered set up-to-date at the beginning of collection, and also the CPU time used by the refinement thread. Such heuristics will generally be based on previous program history.

Clearly, a heuristic that attempts to aggressively “throttle back” concurrent refinement because little mutator activity has been observed recently is vulnerable to sudden increases in pointer mutation rate. Some of the barriers can decrease this vulnerability somewhat. The barriers can be classified according to whether or not they produce *overflow* events; in particular, we say that the misalignment-utrap barrier produces an overflow event whenever it fills a buffer and starts to use a new buffer.⁵ The

⁵The self-pointing barrier can also be considered to produce a overflow event when it allocates a new buffer. But the whole point of the self-pointing barrier is to avoid such allocations, since they are triggered by dereferencing a null pointer, which invokes a relatively expensive signal handler. So these cannot be counted on as a reliable indicator of mutator activity.

occurrence of overflow events can be used to trigger activity by the concurrent refinement thread: instead of sleeping for a fixed amount of time, the thread waits on an operating-system condition variable, using a timeout value to cause it to be resumed when the condition variable is signaled or the timeout expires, whichever comes first.

5.2 Barrier/remembered set compatibility

Which combinations are compatible? Each of the card table barriers requires the corresponding (1- or 2-level) remembered set representation, since they write directly to that representation. The log-based barriers, on the other hand, can be used with either remembered set representation, and, indeed, would be compatible with many others, since the barrier does not write directly to that representation.

5.3 Refinement with card-table barriers

Consider first concurrent refinement with the single-level card-table write barrier. The mutator will set some card table entries to *dirty*. The concurrent refinement thread traverses the card table, searching for dirty entries. When one is found, the refinement thread sets the entry to a new value, *refining*. It then scans the card for cross-generational pointers, computing a new value for the card: *clean, summarized* with some number of pointer offsets, or *overflow* (which we now distinguish from *dirty* to prevent repeated consideration of unmodified overflow cards in successive traversals). The thread then attempts to write the new value into the card table. To do so, it reads the current value, verifies that it is still *refining*, and uses a compare-and-swap (CAS) instruction to atomically change to the new value. A concurrent mutator operation may set the entry back to *dirty*; if so, the refinement is invalid, since it may have missed a pointer update, and is abandoned. We currently leave the card *dirty* and proceed to the next dirty card; we could also repeat the refinement process. We expect such contention for dirty cards to be sufficiently rare to make this choice irrelevant.

Concurrent refinement with a two-level card table barrier is very similar. The refinement thread searches the coarse-grained table for dirty entries.

When one is found it sets the coarse-grained entry to *refining* and searches the corresponding fine-grained entries. For each of those that are dirty, it goes through the process above. If all the fine-grained entries are or become *clean*, and the coarse-grained entry is still *refining*, then the coarse-grained entry is atomically reset to *clean*, otherwise it is reset (not atomically) to *dirty*.

There is a subtle concurrency issue involving the order in which the tables are updated by the barrier. The whole point of using a coarse-grained table is so that during a GC we can traverse the smaller coarse-grained table, and skip all the fine-grained cards corresponding to a clean coarse-grained card. In our system, garbage collections happen only at discrete *gc points*, which never occur during write barriers, so barriers (and their associated pointer updates) are atomic with respect to collection [2]. Thus a collection will never observe a partially completed 2-level barrier. Barriers and updates are not, however, atomic with respect to the actions of the concurrent refinement thread. It turns out that the barrier must update the fine-grained table before the coarse-grained table. With this order, the concurrent refinement thread may skip a clean coarse-grained card, where the mutator has just dirtied one of the corresponding fine-grained cards and is about to dirty the coarse-grained card. But, while this fails to achieve the maximum possible benefit of concurrent processing, it is still perfectly correct: the coarse-grained card will (correctly) be dirtied before the next GC. (And this situation is probably quite rare.) If the barrier is performed in the other order, an unfortunate scenario can result. Consider a clean coarse-grained card, all of whose covered fine-grained cards are also clean. Suppose a pointer update to a field in the last of these fine-grained cards does not create an old-to-young pointer, but nevertheless dirties the relevant fine-grained and coarse-grained cards. A second pointer update to a field in the first covered fine-grained card *does* create an old-to-young pointer, and the write barrier (redundantly) dirties the coarse-grained card. Now, before the second write barrier completes, the concurrent refinement thread observes the dirty coarse-grained card, scans all the covered fine-grained cards, and finds only the last dirty. It scans that card, finds that it does not contain an old-to-young pointer, and therefore resets both the fine-grained and coarse-grained cards to clean. At this point, the partially-completed second barrier completes, dirtying the first fine-grained card. At this point, the invariant is violated, and

the barrier is complete, so a GC could occur and observe this violation. Therefore, the 2-level barrier must dirty the fine-grained card first.

5.4 Refinement with log-based barriers

We now consider concurrent refinement with log-based barriers. Each thread has an associated *thread log set*, which contains log buffers associated with that thread. A thread log set is created and initialized as part of thread creation, before the thread can execute any write barriers. We also maintain a global set of all the thread log sets; initialization of a thread log set includes insertion of the new set into that global set. The refinement thread can iterate over this global set of thread log sets. Concurrent insertion of new thread log sets may cause those to be skipped, but the initial log processing at the start of garbage collection uses sufficient synchronization to ensure that no non-empty logs are skipped. Thread logs may contain unprocessed entries when the thread completes. Therefore, the thread log set is not deleted when the corresponding thread is dead; rather, it is marked as “dead,” implying that no more entries will be written to its log buffers. When the refinement thread completes processing of a dead log set, it deletes it from the global set, and frees its storage.

In the misalignment-utrap barrier, there is also a global list of completed log buffers. The refinement thread processes all completed buffers, and may also traverse the thread log sets, processing partially completed buffers.

As described in section 4, log entries are addresses within the heap. These can be distinguished from non-entries in both logging schemes, so the refinement thread can tell when it has read all the currently-valid entries in a log buffer. A valid entry may or may not represent an old-to-young pointer. For each entry, the refinement thread first considers the address of the field. Write barriers are executed for all objects, so some of the logged addresses may be in young-generation objects; these can be ignored.

The refinement thread next reads the current value of the field. It is important to note that in all our barriers, the barrier must be executed *after* the write it covers, or else the refinement thread might observe the log entry or modified card, but read the

field value *before* the write, and take an improper action.⁶ There is no guarantee that the value read by the refinement thread when it processes a log entry will be the value written by the mutator thread that added that entry, both because a thread may update a location several times, and because a location may be updated by several distinct threads. However, since we require the refinement thread to process all log entries for a location, and the last write to a location happens before the last entry for that location is logged, we are guaranteed that the last entry processed for a given location (either by the refinement thread, or by the collector during its initial log processing) will observe the *final* write to a given pointer field.

If the pointer value is not a pointer into the young generation, then the log entry is ignored. This is a design choice; we have chosen to have the remembered set be monotonically non-decreasing between collections. We could have alternatively attempted to detect when pointer updates decrease the size of the remembered set, but we judged that this would have created significantly more work for the refinement thread for a small benefit.

6 Results

In this section we present measurements of the effectiveness of concurrent refinement. Section 6.1 describes the system in which we performed our preliminary experiments, section 6.2 describes the benchmarks, and sections 6.3 and 6.4 report the results.

Following those experiments, we transferred this technology to a product group. Section 6.5 describes their implementation of concurrent refinement and summarizes its impact on one customer’s application.

6.1 Experimental system

We implemented concurrent refinement by modifying the Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*, a

⁶As discussed previously, other mechanisms ensure that the entire field-write/write-barrier combination is atomic with respect to GC.

high performance JavaTM virtual machine⁷ developed by Sun Microsystems. This virtual machine has been previously known as the “Exact VM”, and has been incorporated into products; for example, the JavaTM 2 SDK (1.2.1_07) Production Release, for the Solaris operating environment.

The ResearchVM features high-performance *exact* (i.e., non-*conservative* [8], also called *precise*) memory management [1]. The memory system is separated from the rest of the virtual machine by a well-defined *GC Interface* [31]. This interface allows different garbage collectors to be “plugged in” without requiring changes to the rest of the system. A variety of collectors implementing this interface have been built. In addition to the GC interface, a second layer, called the *generational framework*, facilitates the implementation of generational garbage collectors. These interfaces allowed us to parameterize the implementation over the various choices of remembered sets and barriers relatively easily.

The default configuration of this system uses a two-generation collector, with a semispace-based young generation, and an older generation that uses mark-sweep-compact collection.

Measurements were performed on an otherwise idle Sun EnterpriseTM E6500 server with 16 400 MHz UltraSPARC[®] II processors sharing 16 GB of memory.

6.2 Benchmarks

We measured the performance of this first implementation on several benchmarks.

The first benchmark is a synthetic one written by the authors, called **gold**.⁸ This application takes several command-line flags that control the workload it presents to the collector. In particular, we can create workloads that require a large old-generation, and also vary the pointer mutation rate. We use this application to demonstrate the existence of workloads for which concurrent refinement shows a significant advantage. This application has been found to be predictive of real application performance in the past. We vary two parameters: heap

⁷The term “Java virtual machine” means a virtual machine for the JavaTM platform.

⁸Earlier versions of this benchmark have been used in other studies of concurrent [27] and parallel [16] collection.

size and pointer mutation rate in the old generation. The *small* heap size is 30 MB live in a 45 MB heap, and *big* is 300 MB live in a 450 MB heap. The *low* pointer mutation rate is less than 1000 old-generation pointers updated per second, and *high* is approximately 300,000 old-generation pointers updated per second of mutator operation. (For comparison, **javac** updates about 70,000 old-generation pointers per second of mutator time. A multi-threaded program with behavior similar to **javac** could easily have an aggregate pointer mutation rate this large.) We ran the following **gcold** configurations: **gcold-S-L** (small/low), **gcold-B-L** (big/low), and **gcold-B-H** (big/high).

The next benchmark is called **jbb**; this is a SPEC benchmark aimed at measuring performance of Java virtual machines executing server applications on multiprocessors. It has large heaps and requires significant GC activity. Unlike the others, which measure the time necessary to accomplish some fixed task, this is a “throughput-oriented” benchmark, measuring how many iterations of a repetitive task can be executed in a fixed amount of time. We make the measurements commensurate by restricting ourselves to the first 500 garbage collections. (The different configurations measured do not differ in the timing of allocation and young-generation collection.) This was run with a 16 MB young generation and a 300 MB old generation.

The remaining four benchmarks are the members of the SPECjvm98 suite that spent more than 3% of their elapsed time performing collection in our system with its default configuration. These are

- **javac**: a compiler that translates Java programming language source code to Java class files, compiling a given set of files;
- **jess**: an “expert systems shell” written in the Java language, solving a set of logic problems;
- **mtrt**: a “multi-threaded ray tracer,” in which two worker threads render an image; and
- **jack**: a parser generator.

These four run in heaps of at most 24 MB.

6.3 Measurements

Table 1 shows the performance of our benchmarks averaged over five runs for each of the various system configurations. The configurations are as follows:

- CT1: the default one-level card table barrier and remembered set.
- CT1+C: one-level card table barrier and remembered set with concurrent refinement.
- CT2: two-level card table barrier and remembered set, no concurrent refinement.
- CT2+C: two-level card table with and remembered set, with concurrent refinement.
- SP-CT1: self-pointing logging barrier, concurrent refinement, one-level card table remembered set.
- SP-CT2: self-pointing logging barrier, concurrent refinement, two-level card table remembered set.
- MIS-CT1: misalignment-utrap logging barrier, concurrent refinement, one-level card table remembered set.
- MIS-CT2: misalignment-utrap logging barrier, concurrent refinement, two-level card table remembered set.

All configurations allow up to 16 summary-table entries per card.⁹

6.4 Discussion

We would first direct the reader’s attention to the columns for young-generation collection time and time to find cross-generation pointers; these are the aspects of collection that we are trying to improve via concurrent refinement.

The **gcold-B-H** run shows that there exist applications (admittedly synthetic) for which the improvement can be dramatic. The **jbb** benchmark is somewhat less artificial, having been written to model a class of real programs; here we see as much as a 5%

⁹This is to allow more direct comparisons; the standard configuration of the ResearchVM allows only 2 such entries.

Benchmark	system configuration	total (sec)	mutator (sec)	old-gen gc (sec)	young-gen gc (sec)	cross-gen ptrs (sec)
gcold-S-L	CT1	101.5	56.3	33.5	11.6	0.6
	CT1+C	101.2	56.7	32.9	11.6	0.4
	SP-CT1	103.6	58.1	33.8	11.7	0.4
	MIS-CT1	102.8	58.8	33.0	10.9	0.4
	CT2	101.0	56.6	32.9	11.6	0.6
	CT2+C	101.4	56.6	33.7	11.1	0.5
	SP-CT2	103.2	58.3	32.9	12.0	0.3
	MIS-CT2	104.5	59.7	33.9	11.0	0.3
gcold-B-L	CT1	113.0	60.4	25.8	26.8	1.6
	CT1+C	115.9	64.1	25.0	26.8	1.3
	SP-CT1	117.3	64.2	26.2	26.8	1.2
	MIS-CT1	117.5	66.3	25.0	26.2	1.2
	CT2	112.5	60.8	25.1	26.7	1.2
	CT2+C	112.8	61.0	26.0	25.8	0.8
	SP-CT2	117.0	65.5	24.9	26.6	0.8
	MIS-CT2	120.2	68.3	26.2	25.8	0.8
gcold-B-H	CT1	257.4	167.6	12.2	77.6	60.5
	CT1+C	215.8	184.3	11.7	19.9	2.5
	SP-CT1	199.0	168.6	12.5	17.9	0.8
	MIS-CT1	202.0	172.4	11.7	17.9	0.9
	CT2	260.8	171.1	11.7	78.0	60.7
	CT2+C	216.6	184.8	12.3	19.5	2.4
	SP-CT2	199.5	170.2	11.6	17.7	0.6
	MIS-CT2	204.3	174.0	12.5	17.8	0.7
jbb	CT1	311.4	189.4	22.3	99.8	23.4
	CT1+C	314.3	197.1	21.7	95.5	19.1
	SP-CT1	312.1	192.7	21.8	97.6	18.6
	MIS-CT1	329.5	212.2	22.2	95.1	18.7
	CT2	334.5	211.2	23.8	99.5	22.8
	CT2+C	329.2	211.8	22.8	94.6	18.2
	SP-CT2	313.6	195.9	22.3	95.4	17.7
	MIS-CT2	343.0	226.4	23.0	93.6	17.7
javac	CT1	35.6	28.1	3.5	4.0	1.4
	CT1+C	34.5	27.5	3.4	3.6	0.9
	SP-CT1	36.4	29.1	3.5	3.7	0.9
	MIS-CT1	37.4	30.3	3.5	3.6	0.9
	CT2	35.2	27.7	3.4	4.1	1.4
	CT2+C	35.0	27.9	3.5	3.6	0.9
	SP-CT2	36.1	28.9	3.5	3.7	0.9
	MIS-CT2	37.6	30.5	3.5	3.6	0.9
jess	CT1	17.0	16.3	0.0	0.7	0.1
	CT1+C	16.4	15.6	0.0	0.7	0.1
	SP-CT1	20.5	19.6	0.0	0.9	0.1
	MIS-CT1	20.8	20.0	0.0	0.7	0.1
	CT2	16.7	16.0	0.0	0.7	0.1
	CT2+C	17.0	16.2	0.0	0.7	0.1
	SP-CT2	21.1	20.2	0.0	0.9	0.1
	MIS-CT2	21.6	20.8	0.0	0.8	0.1
mtrt	CT1	9.9	9.1	0.2	0.6	0.0
	CT1+C	9.7	9.0	0.2	0.5	0.0
	SP-CT1	10.1	9.3	0.2	0.6	0.0
	MIS-CT1	9.8	9.0	0.2	0.5	0.0
	CT2	10.0	9.3	0.2	0.5	0.0
	CT2+C	9.5	8.8	0.2	0.5	0.0
	SP-CT2	10.1	9.3	0.2	0.6	0.0
	MIS-CT2	9.8	9.0	0.2	0.5	0.0
jack	CT1	22.5	21.1	0.1	1.3	0.0
	CT1+C	22.4	21.1	0.1	1.3	0.0
	SP-CT1	23.7	22.2	0.1	1.4	0.0
	MIS-CT1	23.4	22.0	0.1	1.3	0.0
	CT2	22.1	20.7	0.1	1.3	0.0
	CT2+C	22.3	20.9	0.1	1.3	0.0
	SP-CT2	23.3	21.8	0.1	1.4	0.0
	MIS-CT2	23.5	22.1	0.1	1.3	0.0

Table 1: Comparison of elapsed, mutator, and gc times for various system configurations.

decrease in young-generation collection times. The **javac** benchmark is based on a real program, and shows up to 10% improvements in young-generation collections. The other benchmarks have few cross-generational pointers, and show little if any benefit from concurrent refinement.

The six configurations that use concurrent refinement show little variation with respect to young-generation collection times.

Improvements due to concurrent refinement do not come without cost.

The default CT1 barrier executes three instructions, including one write, per pointer write (not counting the **sethi** per method entry). The more complicated barriers (CT2, SP, and MIS) increase mutator time significantly. The SP barrier performs more memory operations, and most of its increased mutator time is due to data cache misses. The CT2 and MIS execute more instructions and have more instruction cache misses. Larger caches would reduce these costs. A more general UTRAP mechanism (that supports segmentation violations as well as misalignment exceptions) would give us a logging barrier with the same instruction count and memory operations as the default CT1 barrier.

In addition, we find that concurrent card table refinement can increase mutator time even with the default card table barrier, as in the CT1+C configuration on the **gcold-B-H** and **jbb** benchmarks. Both of these benchmarks have a high rate of pointer mutation, and the concurrent refinement thread runs almost continuously. We suspect this leads to data cache contention. With the two-level card table (CT2+C), the refinement thread's duty cycle on the **jbb** benchmark drops from 91% to 72%, eliminating this effect. For concurrent refinement with the logging barriers, the duty cycle ranged from 11% (SP-CT1 on **db**) to 82% (MIS-CT2 on **jbb**).

We had expected more improvement from CT2 *vs.* CT1 for large heaps. We believe the lack of such a gain is due to CT1 using an optimized assembly-language routine to recognize 64-bit blocks of clean cards. This loop is not currently used in CT2; it could be, and should result in more speedups for very sparse card tables. In any case, the advantage of the two-level card table will become important only on very large heaps.

Old-generation collection time is not affected by

concurrent refinement. The variation in old-generation collection time for the three **gcold** benchmarks is due to different heap sizes, numbers of collections, object lifetimes, and patterns of floating garbage.

6.5 Production system

We have been working with a telecommunications company that has a call-processing application written in the Java language. In its steady state, this program has several hundred megabytes of live data. When a mostly-concurrent mark/sweep collector is used to collect the old generation, pause times are dominated by the time required to collect the young generation [27]. In the terminology of this paper, this system used the CT1 barrier and remembered set. To reduce pause times still further, a product group added concurrent refinement, to get the equivalent of CT1+C, in a limited-release version of the ResearchVM.

Using the original CT1 barrier adds no mutator overhead, other than processor time taken by the concurrent refinement thread. The main problem with our experimental implementation had been that this processor time had been considerable. To reduce that overhead, the new version does not initiate concurrent refinement until there is barely enough time to complete one or two passes of concurrent refinement before the young generation is collected. For example, concurrent refinement might begin when 90% of the young generation has been allocated, but this threshold is adjusted dynamically.

When the telecommunications application is run on a machine with eight processors, the concurrent refinement thread runs less than 2% of the time, while reducing the average pause time by about 15%. All mutator threads are stopped while the young generation is collected, and those collections had accounted for about 15% of the total time, so concurrent refinement increases the mutator's utilization of the processors by about 2%:

$$.02 \approx .15 \cdot .15 - \frac{.02}{8}$$

In other words, concurrent refinement simultaneously reduces both average pause times and the total cost of garbage collection.

7 Conclusions

The desirable pause time characteristics of generational collection will not scale with ever-increasing heap sizes, at least using currently-popular remembered set techniques. Programs with very large heaps are likely to be run on machines with many processors. We have therefore suggested ways to use concurrency to retain short GC pauses, by refining the precision of remembered set representations so that cross-generational pointers can be found quickly.

We have presented two basic refinement techniques: “direct” refinement of one- and two-level card tables (which has interesting non-blocking concurrency control) and log-based refinement, in which mutator threads log updates and the refinement thread applies those, as appropriate, to a representation of the remembered set. In the latter case, we presented two novel write barrier code sequences that minimize the frequency and/or cost of log buffer overflow.

Concurrent refinement techniques allow generational collection to scale to future systems that will have extremely large heaps and high aggregate pointer mutation rates.

8 Acknowledgments

Several colleagues at Sun Microsystems have contributed to this general set of ideas. Bernd Matthiske and Ross Knippel originally proposed a version of the self-pointing barrier, using the UTRAP mechanism to trigger flushing of small thread-local log buffers to a large common one. They did not consider concurrent refinement. Dave Dice suggested the use of a self-pointing barrier with concurrent refinement to avoid overflow.

References

- [1] O. Agesen and D. Detlefs. Finding references in Java™ stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.
- [2] Ole Agesen. GC points in a threaded environment. Technical Report 98-70, Sun Microsystems Laboratories, 1998.
- [3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.
- [4] David Bacon, Clement Attanasio, Han Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 92–103, N.Y., June 20–22 2001. ACM Press.
- [5] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
- [6] Henry G. Baker. “Infant mortality” and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, April 1993.
- [7] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, ON, Canada, June 1991. ACM Press.
- [8] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.
- [9] John DeTreville. Experiences with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, 1990.
- [10] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1978.
- [11] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, New York, NY, 1993. ACM.
- [12] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–83, New York, NY, USA, January 1994. ACM Press.
- [13] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levroni.

- Implementing an on-the-fly garbage collector for Java. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.
- [14] Tamar Domani, Elliot K. Kolodner, and Erez Pe-trank. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 274–284, Vancouver, British Columbia, June 18–21, 2000.
- [15] Robert Fitzgerald and David Tarditi. The case for profile-directed selection of garbage collectors. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.
- [16] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Java™ Virtual Machine Research and Technology Symposium*, Monterey, April 2001. USENIX.
- [17] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.
- [18] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.
- [19] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.
- [20] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *ACM OOPSLA '93 Workshop on Memory Management and Garbage Collection*, Washington, DC, October 1993.
- [21] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996.
- [22] H. T. Kung and S. Song. An efficient parallel garbage collector and its correctness proof. Technical report, Carnegie Mellon University, September 1977.
- [23] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Equipment Corporation Systems Research Center, February 1988.
- [24] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.
- [25] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.
- [26] James O’Toole and Scott Nettles. Concurrent replicating garbage collection. In *Conference on Lisp and Functional programming*. ACM Press, June 1994.
- [27] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.
- [28] Patrick G. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. B.S. thesis, Massachusetts Institute of Technology EECS Department, Cambridge, Massachusetts, 1988.
- [29] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.
- [30] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.
- [31] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.
- [32] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.
- [33] Paul R. Wilson and Thomas G. Moher. A “card-marking” scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.