# A Generational Mostly-concurrent Garbage Collector

Tony Printezis[*]
Department of Computing Science
University of Glasgow
17 Lilybank Gardens
Glasgow G12 8RZ, Scotland
tony@dcs.gla.ac.uk

David Detlefs
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803-0902
USA
david.detlefs@sun.com

## ABSTRACT

This paper reports our experiences with a mostly-concurrent incremental garbage collector, implemented in the context of a high performance virtual machine for the Java™ programming language. The garbage collector is based on the "mostly parallel" collection algorithm of Boehm *et al.* and can be used as the old generation of a generational memory system. It overloads efficient write-barrier code already generated to support generational garbage collection to also identify objects that were modified during concurrent marking. These objects must be rescanned to ensure that the concurrent marking phase marks all live objects. This algorithm minimises maximum garbage collection pause times, while having only a small impact on the average garbage collection pause time and overall execution time. We support our claims with experimental results, for both a synthetic benchmark and real programs.

## 1. INTRODUCTION

Programming languages that rely on garbage collection have existed since the late 1950's [25]. Though the benefits of garbage collection for program simplicity and robustness are well-known and accepted, most software developers have continued to rely on traditional explicit memory management, largely because of performance concerns. Only recently has the wide acceptance of the Java™ programming language [13] allowed garbage collection to enter the mainstream and be used in large systems.

Developers have been skeptical about garbage collection for two reasons: throughput and latency. That is, they fear that collection will either slow down the end-to-end performance of their systems, or induce long collection pauses, or both. Large increases in computing power have not eliminated these concerns, since they are typically offset by corresponding increases in memory requirements.

Generational garbage collection techniques [21, 26] can address both performance concerns. They split the heap into *generations* according to object age. Concentrating collection activity on the "young" generation increases throughput, because (in most programs) young objects are more likely to be garbage, so more free space is recovered per unit of collection work. Since the young generation is typically small relative to the total heap size, young-generation collections are usually brief, addressing the latency concern. However, objects that survive a sufficiently large number of young-generation collections are considered long-lived, and are "promoted" into an older generation. Even though the older generation is typically larger, it will eventually be filled and require collection. Old-generation collection has latency and throughput similar to full-heap collection; thus, generational techniques only postpone, but do not solve, the problem.

In this paper we present a garbage collection algorithm that has been designed to serve as the oldest generation of a generational memory system. It attempts to decrease the worst-case garbage collection pause time, while taking advantage of the benefits of a generational system. It is an adaptation of the "mostly parallel" algorithm of Boehm *et al.* [5]. It usually operates concurrently with the mutator, only occasionally suspending the mutator for short periods.

### 1.1 A Note on Terminology

In this paper, we call a collector *concurrent* if it can operate interleaved with the mutator, either truly concurrently, or by working in small increments, i.e., piggy-backed on a frequent operation (such as object allocation). We propose to contrast this with *parallel* collectors, which accomplish collection using multiple cooperating threads, and can therefore achieve parallel speed-ups on shared-memory multiprocessors.

It is unfortunate that this terminology clashes with that used by Boehm *et al.* [5], since they use "parallel" for the concept we have named "concurrent." The choice is arbitrary; a local tradition led to the choice we make. Thus, we use "mostly concurrent" to mean what Boehm *et al.* termed "mostly parallel."

### 1.2 Paper Overview

Section 2 briefly describes the platform on which we based our implementation and experiments, and Section 3 describes the original mostly-concurrent algorithm. Our adaptation of

[*]Most of this work took place during the author's summer internship at Sun Microsystems Laboratories East, MA, USA, summer of 1998.

this algorithm is described in Section 4, with Section 5 containing the results of the experiments that we undertook in order to evaluate our implementation. Finally, related work on incremental garbage collectors is given in Section 6 and the conclusions and future work in Section 7.

## 2. EXPERIMENTAL PLATFORM

The Sun Microsystems Laboratories Virtual Machine for Research, henceforth *ResearchVM*, is a high performance Java virtual machine developed by Sun Microsystems. This virtual machine has been previously known as the "Exact VM", and has been incorporated into products; for example, the Java™ 2 SDK (1.2.1_05) Production Release, for the Solaris™ operating environment.[1] It employs an optimising just-in-time compiler [8] and a fast synchronisation mechanism [2].

More relevantly, it features high-performance *exact* (i.e., non-*conservative* [6], also called *precise*) memory management [1]. The memory system is separated from the rest of the virtual machine by a well-defined *GC Interface* [27]. 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 [26, 19, 28].

One of the authors learned these interfaces and implemented a garbage collector in a matter of weeks, so there is some evidence that it is relatively easy to implement to the interfaces described above.

## 3. MOSTLY-CONCURRENT COLLECTION

The original mostly-concurrent algorithm, proposed by Boehm *et al.* [5], is a concurrent "tricolor" collector [9]. It uses a write barrier to cause updates of fields of heap objects to shade the containing object gray. Its main innovation is that it trades off complete concurrency for better throughput, by allowing *root* locations (globals, stacks, registers), which are usually updated more frequently than heap locations, to be written without using a barrier to maintain the tricolor invariant. The algorithm suspends the mutator to deal properly with the roots, but usually only for short periods. In more detail, the algorithm is comprised of four phases:

- **Initial marking pause.** Suspend all mutators and record all objects directly reachable from the *roots* (globals, stacks, registers) of the system.
- **Concurrent marking phase.** Resume mutator operation. At the same time, initiate a concurrent marking phase, which marks a transitive closure of reachable objects. This closure is *not* guaranteed to contain all objects reachable at the end of marking, since concurrent updates of reference fields by the mutator may have prevented the marking phase from reaching some live objects.

  To deal with this complication, the algorithm also arranges to keep track of updates to reference fields in heap objects. This is the only interaction between the mutator and the collector.
- **Final marking pause.** Suspend the mutators once again, and complete the marking phase by marking from the roots, considering modified reference fields in marked objects as additional roots. Since such fields contain the only references that the concurrent marking phase may not have observed, this ensures that the final transitive closure includes all objects reachable at the start of the final marking phase. It may also include some objects that became unreachable after they were marked. These will be collected during the next garbage collection cycle.
- **Concurrent sweeping phase.** Resume the mutators once again, and sweep concurrently over the heap, deallocating unmarked objects. Care must be taken not to deallocate newly-allocated objects. This can be accomplished by allocating objects "live" (i.e., marked), at least during this phase.

This description abstracts somewhat from that given by Boehm *et al.* Tracking individual modified fields is the finest possible tracking granularity; note that this granularity can be coarsened, possibly trading off decreased accuracy for more efficient (or convenient) modification tracking. In fact, Boehm *et al.* use quite a coarse grain, as discussed in Section 4.2.

The algorithm assumes a low mutation rate of reference-containing fields in heap objects; otherwise, the final marking phase will have to rescan many dirty reference-containing fields, leading to a long, possibly disruptive, pause. Even though some programs will break this assumption, Boehm *et al.* report that in practice this technique performs well, especially for interactive applications [5].

### 3.1 A Concrete Example

Figure 1 illustrates the operation of the mostly-concurrent algorithm. In this simple example, the heap contains 7 objects and is split into 4 pages. During the initial marking pause (not illustrated), all 4 pages are marked as clean and object **a** is marked live, since it is reachable from a thread stack.

Figure 1a shows the heap halfway through the concurrent marking phase. Objects **b**, **c**, and **e** have been marked. At this point, the mutator performs two updates: object **g** drops its reference to **d**, and object **b** has its reference field, which pointed to **c**, overwritten with a reference to **d**. The result of these updates is illustrated in Figure 1b. Also note that the updates caused pages 1 and 3 to be dirtied.

Figure 1c shows the heap at the end of the concurrent marking phase. Clearly, the marking is incomplete, since a marked object **b** points to an unmarked object **d**. This is dealt with during the final marking pause: all marked objects on dirty pages (pages 1 and 3) are rescanned. This causes **b** to be scanned, and thus object **d** to be marked. Figure 1d shows the state of the heap after the final marking pause, with marking now complete. A concurrent sweeping phase will follow, and will reclaim the unmarked object **f**.

Objects such as **f** that are unreachable at the beginning of a garbage collection cycle are guaranteed to be reclaimed. Objects such as **c**, however, that become unreachable during a collection cycle, are not guaranteed to be collected in that cycle, but will be collected in the next.

[1]Currently available at `http://www.sun.com/solaris/java`.

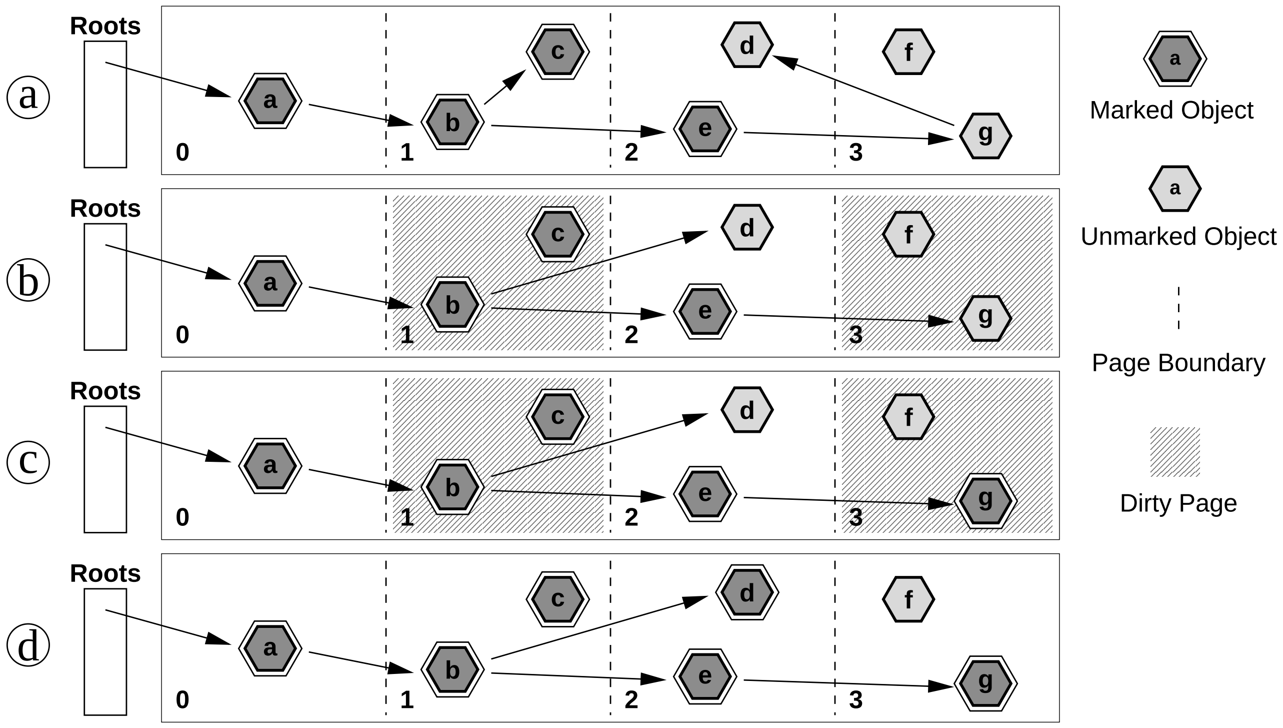

**Figure 1: Concrete example of the operation of the original mostly-concurrent garbage collector.**

## 4. MOSTLY-CONCURRENT COLLECTION IN A GENERATIONAL SYSTEM

This section describes our generational mostly-concurrent garbage collector in detail, and records some of the decisions we took during its design and implementation. Whenever possible, we also present alternative solutions that might be more appropriate for different systems.

Most aspects of the design are independent of the use of the collector as a generation in the generational framework, and we will describe these first. Later, we will describe complications specific to the use of the collector in a generational context.

### 4.1 The Allocator

The default configuration of the ResearchVM uses an older generation that performs mark-sweep collection, with a compaction pass [19, 28] to enable efficient allocation later. We will refer to this collector implementation as *mark-compact*. The object relocation implied by compaction requires updating of references to the relocated objects; this reference updating is difficult to perform concurrently. Therefore, mostly-concurrent collection does not attempt relocation. Thus, its allocator uses free lists, segregated by object size, with one free list per size for small objects (up to 100 4-byte words) and one free list per group of sizes for larger objects (these groups were chosen using a Fibonacci-like sequence).

It can be argued that a smarter allocator with better speed / fragmentation trade-offs could have been used. In fact, Johnstone and Wilson claim that the segregated free-list allocation policy is one of the policies that cause the worst fragmentation [18]. However, this work assumed explicit "on-line" deallocation, as represented by C's `malloc/free` interface. It is easier and more efficient to coalesce contiguous free areas to decrease fragmentation in an "off-line" garbage collector with a sweeping phase that iterates over the entire heap.

### 4.2 Using the Card Table

Generational garbage collection requires tracking of references from objects in older generations to objects in younger generations. This is necessary for correctness, since some young-generation objects may be unreachable except through such references. A better scheme than simply traversing the entire older generation is required, since that would make the work of a young-generation collection similar to the work of a collection of the entire heap.

Several schemes for tracking such old-to-young references have been used, with different cost/accuracy tradeoffs. The generational framework of the ResearchVM (see Section 2) uses a *card table* for this tracking [31, 15, 30]. A card table is an array of values, each entry corresponding to a subregion of the heap called a *card*. The system is arranged so that each update of a reference field within a heap object by mutator code executes a *write barrier* that sets the card table entry corresponding to the card containing the reference field to a **dirty** value.[2] In compiled mutator code, the extra code for card table update can be quite efficient: a two-instruction write barrier proposed by Hölzle [14] is used.

One of the fundamental decisions of our design is to exploit the happy coincidence that this efficient card-table-

[2]For efficiency, this is done without checking whether the reference is actually to an object into the young generation – a dirty card table entry indicates the *possibility* of an old-to-young pointer on the corresponding card.

based write barrier can be used, almost without modification, to perform the reference update tracking required for mostly-concurrent collection. Thus, using mostly-concurrent collection for the old generation will add no extra mutator overhead beyond that already incurred for the generational write barrier.

Boehm *et al.* used virtual memory protection techniques to track pointer updates at virtual memory page granularity: a "dirty" page contains one or more modified reference fields. Using a card-table-based write barrier has several advantages over this approach.

- **Less overhead.** The cost of invoking a custom handler for memory protection traps is quite high in most operating systems. Hosking and Moss [16] found a five-instruction card-marking barrier to be more efficient than a page-protection-based barrier; the two- or three-instruction implementation used in ResearchVM will be more efficient still.
- **Finer-grained information.** The granularity of a card table can be chosen according to an accuracy/space overhead tradeoff. The "card size" in a virtual memory protection scheme is the page size, which is chosen to optimize properties, such as efficiency of disk transfer, that are completely unconnected with the concerns of garbage collection. Generally, these concerns lead to pages that are larger than optimal for reference update tracking, typically at least 4 Kbytes. In contrast, the card size of ResearchVM is 512 bytes.[3]
- **More accurate type information.** The ResearchVM dirties a card only when a field of a reference type on that card is updated. A virtual-memory-based system cannot distinguish between updates of scalar and reference fields, and thus may dirty more pages than are necessary to track modified pointers. Furthermore, their approach was conservative elsewhere, as well: it assumed all words were potential pointers.

Hosking, Moss, and Stefanovic [15] present a detailed discussion of the tradeoffs between software and page-protection-based barrier implementations. Their basic conclusion is that software mechanisms are more efficient than those using virtual memory protection.

In fairness, we should note that the system of Boehm *et al.* was attempting to satisfy a further constraint not present in our system: accomplishing garbage collection for uncooperative languages (C and C++) without compiler support. This constraint led to the conservative collection scheme [6] on which the mostly-concurrent extension is based, and also favored the use of the virtual-memory technique for reference update tracking, since this technique required no modification of the mutator code.

Adapting the card table for the needs of the generational mostly-concurrent algorithm was straightforward. In fact, as discussed above, the write barrier and card table data structure were left unchanged. However, we took careful note of the fact that the card table is used in subtly different ways by two garbage collection algorithms that may be running simultaneously. The mostly-concurrent algorithm

[3]Other systems for tracking modified references, such as *remembered sets*, can give even more accurate information, but usually at the cost of greater space overhead and a more expensive write barrier.

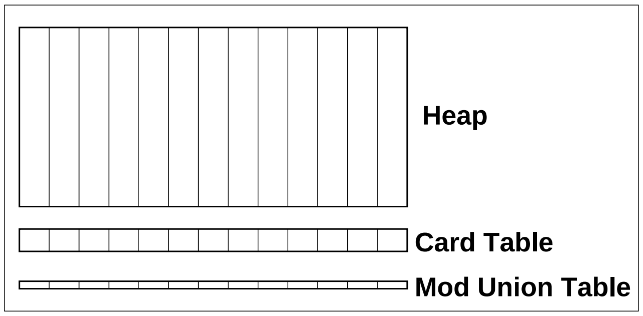

**Figure 2: The mod union table**

requires tracking of all references updated since the beginning of the current marking phase. Young-generation collection requires identification of all old-to-young pointers. In the base generational system, a young-generation collection scans all dirty old-space cards, searching for pointers into the young generation. If none are found, there is no need to scan this card in the next collection, so the card is marked as **clean**. Before a young-generation collection cleans a dirty card, the information that the card has been modified must be recorded for the mostly-concurrent collector.

This is accomplished by adding a new data structure, the *mod union* table, shown in Figure 2, which is so-named because it represents the union of the sets of cards modified between each of the young-generation collections that occur during concurrent marking. The card table itself contains a byte per card in the ResearchVM; this allows a fast write-barrier implementation using a byte store. The mod-union table, on the other hand, is a bit vector with one bit per card. It therefore adds little space overhead beyond the card table, and also enables fast traversal to find modified cards when the table is sparsely populated. We maintain an invariant on the mod union and card tables: any card containing a reference modified since the beginning of the current concurrent marking phase either has its bit set in the mod union table, or is marked **dirty** in the card table, or both. This invariant is maintained by young-generation collections, which set the mod union bits for all cards dirty in the card table before scanning those dirty cards.

## 4.3 Marking Objects

Our concurrent garbage collector uses an array of *external* mark bits. This bitmap contains one bit for every four-byte word in the heap. This use of external mark bits, rather than *internal* mark bits in object headers, prevents interference between mutator and collector use of the object headers.

Root scanning presents an interesting design choice, since it is influenced by two competing concerns. As we described in Section 3, the mostly-concurrent algorithm scans roots while the mutator is suspended. Therefore, we would like this process to be as fast as possible. On the other hand, any marking process requires some representation of the set of objects that have been marked but not yet scanned (henceforth the *to-be-scanned* set). Often this set is represented using some data structure external to the heap, such as a stack or queue. A strategy that minimizes stop-the-world time is simply to put all objects reachable from the roots in this external data structure. However, since garbage collection is intended to recover memory when that is a scarce resource, the sizes of such external data structures are always important concerns. Since the Java language is multithreaded, the root set may include the registers and stack frames of many threads. In our generational system, objects in generations other than the one being collected are also considered roots.[4] So the root set may indeed be quite large, arguing against this simple strategy.

An alternative strategy that minimizes space cost is one that marks all objects reachable from a root immediately on considering the root. Many objects may be reachable from roots, but we place such objects in the to-be-scanned set one at a time, minimising the space needed in this data structure (because of roots) at any given time. While suitable for non-concurrent collection, this strategy is incompatible with the mostly-concurrent algorithm, since it accomplishes *all* marking as part of the root scan.

We use a compromise between these two approaches. The compromise takes advantage of the use of an external marking bitmap. The root scan simply marks objects directly reachable from the roots. This minimizes the duration of the stop-the-world root scan, and imposes no additional space cost, by using the mark bit vector to represent the to-be-scanned set. The concurrent marking phase, then, consists of a linear traversal of the generation, searching the mark bit vector for live objects. (This process has cost proportional to the heap size rather than amount of live data, but the overall algorithm already has that complexity because of the sweeping phase). For every live object *cur* found, we push *cur* on a to-be-scanned stack, and then enter a loop that pops objects from this stack and scans their references, until the stack is empty. The scanning process for a reference value *ref* (into the mostly-concurrent generation) works as follows:

- if *ref* points ahead of *cur*, the corresponding object is simply marked, without being pushed on the stack; it will be visited later in the linear traversal.
- if *ref* points behind *cur*, the corresponding object is both marked and pushed on the stack.

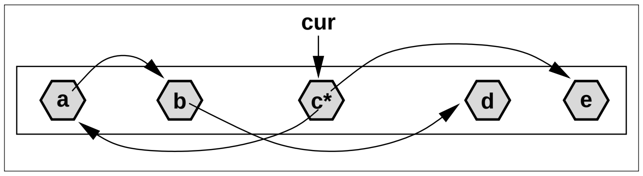

**Figure 3: Marking traversal example**

Figure 3 illustrates this process. The marking traversal has just discovered a marked object **c***, whose address becomes the value of **cur**. Scanning **c*** finds two outgoing references, to **a** and **e**. Object **e** is simply marked, since its address follows **cur**. Object **a** is before **cur**, so it is both marked and scanned. This leads to **b**, which is also before **cur**, so it too is marked and scanned. Object **b**'s reference to **d**, however, only causes **d** to be marked, since it follows **cur**, and will therefore be scanned later in the traversal.

[4]Unidirectional promotion from younger into older generations ensures that garbage cycles are eventually entirely within the the oldest generation, at which point they may be collected.

This technique reduces demand on the to-be-scanned stack, since no more than one object directly reachable from the root set is ever on the stack. A potential disadvantage of this approach is the linear traversal searching for live objects, which makes the algorithmic complexity of marking contain a component proportional to the size of the generation, rather than just the number of nodes and edges in the pointer graph. This is a practical difficulty only if the cost of searching for marked objects outweighs the cost of scanning them if when found, which will occur only if live objects are sparse. Note that if live objects are sparse, the use of a bitmap allows large regions without live objects to be skipped efficiently, by detecting zero words in the bit vector. A similar technique was used by Printezis [23] in the context of a disk garbage collector.

## 4.4 The Sweeping Phase

When the concurrent marking phase is complete, a *sweeping* process must identify all objects that are not marked as reachable, and return their storage to the pool of available storage. The allocation process will often “split” a free block, creating an allocated block and a remaining free block, both smaller than the original free block. Therefore, to prevent the average block size from continuously decreasing, the sweeping process must also perform some form of *coalescing*, that is, combination of consecutive free chunks into one larger free chunk.

In a non-concurrent free-list-based collector, sweeping and coalescing are most easily accomplished by throwing away the existing free lists and reconstructing them “from scratch” during the sweeping process. This will not work, however, in a concurrent collector, which must be prepared to satisfy allocation requests during sweeping.

Concurrent allocation complicates sweeping in two ways. First, a mutator thread could be attempting to allocate from a free list while the sweeping process is attempting to add to that free list. This contention is handled fairly easily with mutual exclusion locks. More subtly, the sweeping process could also be competing with mutator threads to *remove* blocks from free lists. Consider a situation where blocks *a, b,* and *c* are contiguous. Block *b* is on a free list; blocks *a* and *c* had been allocated to contain objects, but both have been found to be unreachable. We wish to put the coalesced block *abc* onto a free list. To do so, however, we must first remove block *b* from its free list, or else we risk that storage being allocated for two purposes.

Mutual exclusion locks can still manage this competition. However, note that this scenario places a new requirement on the free list data structures of the heap: we must be able to delete an arbitrary block from its free list. Allocation removes objects from free lists, but only at the heads of the lists. While singly-linked free lists are efficient when free blocks are deleted only from the head, deletion of arbitrary blocks favors doubly-linked free lists, which allow this operation to be done in constant, rather than linear, time. Note that this adds no space overhead, since the same memory is used to contain object information when a block is allocated and free-list links when it is not.[5]

[5]It does constrain the minimum block size to be at least three words, but this constraint was already present in the system.

## 4.5 The Garbage Collector Thread

Our system uses a dedicated garbage collection thread. This approach allows us to take advantage of multiple CPUs. For example, a single-threaded program can run on a dual processor machine and have most garbage collection work accomplished on the second processor. Similarly, collection activity can proceed while the mutator is inactive, for example, while performing I/O. In contrast, Boehm *et al.* perform collection functions incrementally, "piggy-backed" on frequent operations performed by mutator threads, such as object allocation. We believe this approach was chosen to increase portability.

We also decided to label the garbage collector thread as a "fake" mutator thread, which means that it is suspended during young-generation collections. This has two advantages: it does not slow down young-generation collections, which need to be fast, and it minimises any synchronisation with the rest of the system (see Section 4.8).

## 4.6 Interaction with Young-Generation Collection

There are some ways in which our mostly-concurrent collector has been optimized or modified to work as the older generation in a generational collector. First, we recognize that, for most programs, a large majority of allocation in the older generation will be done via promotion from the young generation. (The remainder is "direct" allocation by the mutator in the older generation, which usually occurs only for objects too large to be allocated in the young generation). Promotion occurs while mutator threads and the concurrent garbage collector thread are suspended, which simplifies matters. We take advantage of this simplification by supporting a *linear allocation mode* during young-generation collection. Linear allocation can be considerably faster than free-list-based allocation (especially when doubly-linked free lists are used), since fewer pointers are compared and modified. When linear allocation mode is in force, we maintain linear allocation for small allocation requests as long as there exist sufficiently large free chunks from which to allocate. This significantly speeds up allocation for promotion, which can be a major component of the cost of young-generation collection.

In the default configuration of the ResearchVM, the use of a compacting older generation simplifies the implementation of one function required for young-generation collection. One of the most elegant aspects of Cheney-style copying collection [7] is that the set of objects still to be scanned are contiguous. In a generational system, where some from-space objects may be copied to to-space and others may be promoted to the older generation, the promoted objects are also part of the to-be-scanned set. When the older generation uses compaction, and thus linear allocation, the set of promoted-but-not-yet-scanned objects is contiguous. However, in a non-compacting collector, the promoted objects may not be contiguous. This complicates the problem of locating them so that they may be scanned.

We solve this problem by representing the set of promoted-but-unscanned objects with a linked list. Every promoted object was promoted from the current from-space of the young generation, and the from-space version of the object contains a forwarding pointer to the object's new address in the older generation. These from-space copies of the promoted objects are used as the "nodes" of the linked list. The forwarding pointer indicates the element of the set, and a subsequent header word is used as a "next" field.

```
initFrac = (1 - heapOccupancyFrac) *
                          allocBeforeCycleFrac;
while (TRUE) {
  sleep(SLEEP_INTERVAL);
  if (generationOccupancy() > initFrac) {
    /* 1st stop-the-world phase */
    initialMarkingPause();
    concurrentMarkingPhase();
    concurrentPreCleaningPhase();
    if (markedPercentage() < 98%) {
      /* 2nd stop-the-world phase */
      finalMarkingPause();

      if (markedPercentage() < 98%)
        concurrentSweepingPhase();
    }
  }
}
```

**Figure 4: Pseudo-code for the GC thread.**

## 4.7 Control Heuristics

Figure 4 shows the code executed by the garbage collector thread. The first statement initializes `initFrac`, the the occupancy threshold that initiates a new collection cycle. In the ResearchVM collector framework the user specifies a desired heap occupancy (`heapOccupancyFrac`) to control heap expansion. In program's steady state, this fraction is occupied at the end of a collection cycle; we start a new cycle when the fraction `allocBeforeCycleFrac` of the free space has been allocated.

The thread wakes up periodically (`SLEEP_INTERVAL` is set to 50 ms) and checks the generation occupancy. If this has reached the `initFrac`, a new cycle starts with an initial marking pause. Then the concurrent marking phase is executed, followed by concurrent precleaning (see Section 4.9), and the final marking pause (see Section 3). Finally, the cycle is completed by the concurrent sweeping phase, which reclaims all unmarked objects. Actually, tests guard execution of the final marking pause and concurrent sweeping: if the fraction of the heap marked is already too high, sweeping will not reclaim sufficient storage to justify its cost. So neither step is performed if the fraction of the heap marked exceeds 98%.

It is important to note that "maximum pause time" is not by itself a sufficient measure of garbage collector intrusiveness. Consider an incremental system that limits GC pauses to a relatively small maximum, say 50 ms. On a uniprocessor, this may still allow garbage collection to be intrusive: if only 10 ms of mutator work is done between each of these GC pauses, the user will observe the program running at only 20% of its normal speed during garbage collection. The measurements we present later in this paper were done on multiprocessors, with an extra processor available for garbage collection work, and thus ignore this issue. However, the implementation does have a set of heuristics aimed at controlling such GC intrusion on uniprocessors. These heuristics view concurrent collection as a race in which the collector is trying to finish collection before mutator activity allocates the free space available at the start of the collection. The collector thread accomplishes marking and sweeping in series of *steps*, sleeping after each step for a period determined by the relative progress of collection and allocation. The more quickly the program fills available space, then more frequently collection activities occur.

Occasionally, despite these heuristics or the availability of an extra processor, the collector thread will "lose the race:" a mutator thread will require the completion of a concurrent collection in order to make progress. When this happens, the remainder of the collection is performed non-concurrently. This leads to longer pauses, but usually shorter than if the entire collection had been performed with the world stopped. Alternatively, we could choose to expand the heap in such situations; we are exploring heuristics to control this behavior.

## 4.8 Concurrency Issues

We have mentioned several concurrency issues already; for example, the previous section discussed the management of concurrency between old-generation allocation and sweeping. This section explores issues that remain. As discussed previously, since we expect most old-generation allocation to be done during young-generation collection, the decision to suspend old-generation collection during young-generation collection (see Section 4.5) handles many such concerns. But not all; mutator threads may still occasionally allocate objects directly in the older generation, and the old-generation collection thread must not be interrupted for young-generation collection in critical sections where its data structures are inconsistent.

Object allocation in the older generation, whether direct or by promotion, raises two issues. First, if the background collector is in a sweeping phase, we must ensure consistent access not only to free lists, but also to mark bits. If free block $b$ is allocated during sweeping, we must prevent the sweeping thread from reclaiming $b$ as an allocated but unmarked block. Thus, during sweeping we use an *allocate-live* policy: allocated blocks are marked live in the bitmap. This marking must be coordinated with the sweeping thread's examination of the mark bits.

We also allocate live during marking, but for somewhat different reasons. A viable alternative strategy would allocate unmarked objects. The final marking pause would still reach a correct transitive closure: if an object allocated during marking is reachable at the end of marking, then there exists some path from a root to the object at the end of marking. Every such path consists either entirely of unmarked objects allocated during marking, or contains at least one marked object. In the former case, the final marking pause will certainly mark the object. In the latter case, consider the last marked object in the path. It must have been modified to become part of the path after it was scanned (and thus after it was marked), or else the next object on the path would have been marked as part of that scanning. Thus, the modification that made it part of the path made the object dirty. So the last marked object in the path must be dirty, meaning that marking from live, dirty objects will lead to the object allocated during marking.

Thus, we could preserve correctness without allocating live during marking. It would even give a more accurate estimate of the set of objects reachable at the end of marking than the more conservative allocate-live strategy. This extra accuracy would come at a cost in some cases, however. If significant numbers of objects are allocated during marking, and many of these remain reachable but unmarked until the final marking pause, all of these will have to be marked in that final pause, potentially lengthening it enough to make it intrusive. On the other hand, the allocate-live policy quickly cuts off the marking process when it reaches objects allocated during marking. Any of these objects that are actually unreachable will be collected during the next collection cycle. Since eliminating intrusive collection interruptions is a more important goal in this work than maximizing overall throughput, we use the allocate-live policy during both marking and sweeping.

## 4.9 Concurrent Precleaning

We noted in section 3 that efficient mostly-concurrent collection requires a low mutation rate for reference fields of heap objects. A program with a high mutation rate will create many dirty objects that must be rescanned during the final marking pause, making this pause intrusive.

We have developed a technique called *concurrent precleaning* that partially mitigates this problem. The observation is fairly simple: much of the work done for dirty objects in the final marking phase could be done concurrently beforehand, as long as it is done in a careful manner that preserves correctness. At the end of concurrent marking, some set of objects are dirty. Without stopping the mutator, we find all such dirty objects; for each object, we mark the object clean, and then transitively mark from the object. Correctness is maintained because any further mutator updates still dirty the corresponding object and require it to be processed in the final marking phase. The hope, however, is that the concurrent cleaning process will take considerably less time than the concurrent marking phase that preceding it, allowing less time for the mutator to dirty objects. Thus, the final marking phase will have less non-concurrent work to do. Section 5.3 measures the effectiveness of this technique.

In further experiments (not included in those measurements) we extended concurrent precleaning in two ways. First, the original implementation of precleaning worked only on the the mod-union table, assuming that the number of modifications reflected in the card table proper between two young-generation collections would be relatively small. This turned out not to be true of real-world programs with high pointer mutation rates. Therefore, we extended the technique to also preclean the card table. This required the creation of a new card table value: **dirty** cards are changed to **precleaned**, which are considered **dirty** by generational collection, but considered clean in the final mark phase. Second, we iterate the precleaning process as long as the number of dirty cards encountered decreases by a sufficient factor (currently 1/3), or until that number is sufficiently small (currently, less than 1000). Both extensions were useful in meeting the demands of the telecommunications application discussed in section 5.4.

## 5. EXPERIMENTAL RESULTS

This collection technique is targeted at long-lived programs with large heaps. Ideally, we would present measurements of several such programs. In the real world, however, Java-technology-based implementations of such programs are difficult to obtain, and the owners of those that exist are still somewhat reluctant to have performance data quoted in public forums. Therefore, to validate our approach, we built a synthetic program (**gcold**) that could present a variety of loads to a garbage collector, including large heaps requiring significant old-generation collections. We present measurements of this application, and also a few "real" programs.

The **gcold** application allocates an array, each element of which points to the root of a binary tree about a megabyte in size. An initial phase allocates these data structures; then the program does some number of *steps*, maintaining a steady-state heap size. Each step allocates some number of bytes of short-lived data that will die in a young-generation collection, and some number of bytes of nodes in a long-lived tree structure that replaces some previously existing tree, making it garbage. Each step further simulates some amount of mutator computation by several iterations of an busy-work loop. Finally, since pointer-mutation rate is an important factor in the performance of both standard generational and mostly-concurrent collection, each step modifies some number of pointers (in a manner that preserves the amount of reachable data). Command-line parameters control the amount of live data in the steady state, the number of steps in the run, the number of bytes of short-lived and long-lived data allocated in each step, the amount of simulated work per step, and the number of pointers modified in a step.

## 5.1 Pauses as a Function of Heap Size

Table 1 compares the default non-concurrent older generation with the mostly-concurrent older generation on the **gcold** application. The mostly-concurrent collector operates in a mode in which it assumes that there is an extra processor available for garbage collection work. The runs were performed on a Sun E3500 server, with 8 336 MHz UltraSPARC™ processors sharing 2 Gbyte of memory. We show runs for various amounts of steady-state live data; each run in this table has the same number of steps (1000), ratio of short-lived to long-lived data (5:1), simulated mutator work (5 units), and pointer mutation parameter (0). We show the elapsed time, the final heap size (the maximum size is made large enough that the heap can grow to its "natural size" for the collection scheme), and the average and maxima of the young- and old-generation pause times.

Points to infer from this table include:

- Mostly-concurrent collection succeeds in dramatically decreasing old-generation pause times. The maximum and average pauses for the default old-generation collector increase with the amount of live data; in contrast, the old generation pauses are much smaller for the mostly-concurrent system, and grow less dramatically.
- The overall elapsed times are similar, with the mostly-concurrent collector generally a little faster. This is actually an interesting balancing act: more expensive allocation for promotion makes young-generation collections more expensive, but off-loading old-generation collection work onto a separate processor offsets this.
- Mostly-concurrent collection requires a somewhat larger heap. This is necessary to allow collection activity to complete before concurrent allocation exhausts the available space.

The measurements in this section were deliberately chosen to show mostly-concurrent collection in a good light, in the sense that the runs measured used parameters for the **gcold** application that did not unduly challenge the memory system. There are several parameters of mutator behavior that can make mostly-concurrent collection less effective, and the following sections explain and explore these. Note that we do not claim that our collector is the best choice for all programs; these "best-case" measurements are intended to suggest only that there exist programs with large heaps and stringent pause time for which our collector functions well. The following measurements explore the space of mutator behaviors in which the collector continues to function well.

| promotion rate *(Mbytes/sec)* | maximum heap size *(Mbytes)* | old-gen pauses avg *(ms)* | old-gen pauses max *(ms)* |
|---|---|---|---|
| 0.76 | 369 | 13 | 29 |
| 1.38 | 369 | 17 | 38 |
| 2.29 | 369 | 35 | 78 |
| 3.40 | 369 | 82 | 204 |
| 3.77 | 602 | 4199 | 10637 |
| 3.69 | 602 | 9602 | 24147 |

**Table 2: Effect of promotion rate on pause times**

## 5.2 Pauses as a Function of Promotion Rate

In order for mostly-concurrent collection to maintain small pause times, it must complete collections before allocation exhausts the available space – if this happens, the remainder of the collection is performed non-concurrently, causing a long pause. The collector is going as fast as it can; the mutator determines the rate at which old-generation allocation (i.e., promotion) occurs. The measurements summarized in table 1 performed enough simulated mutator work between allocations to slow the rate of old-generation allocation sufficiently so that this "race" was never lost. Still, these runs all promoted at least 2.1 Mbytes per second.

Table 2 shows how old generation pauses and maximum heap size are effected by the promotion rate (which was controlled by varying the amount of simulated mutator work per allocation). The live data parameter is kept constant at 200 Mbyte, the remaining parameters are as in the previous measurements. This table shows that there is a sharp cutoff point: pauses stay roughly constant as long as the collector can finish before allocation requires a collection to occur, but when the promotion rate is high enough, it must do non-concurrent collection, with correspondingly high pause times. Losing the race also, with current heuristics, causes the collector to grow the heap size.

| pointer mutation rate *Kptrs/sec* | final mark pauses no precleaning avg *(ms)* | final mark pauses no precleaning max *(ms)* | final mark pauses with precleaning avg *(ms)* | final mark pauses with precleaning max *(ms)* |
|---|---|---|---|---|
| 0 | 672 | 710 | 62 | 76 |
| 0.7 | 700 | 739 | 68 | 83 |
| 2.7/2.8 | 760 | 802 | 76 | 83 |
| 5.3/5.4 | 830 | 879 | 89 | 101 |
| 19.6/20.0 | 1064 | 1140 | 153 | 166 |
| 61.6/62.8 | 1450 | 1526 | 312 | 337 |
| 146.8/149.5 | 1776 | 1949 | 556 | 619 |

**Table 3: Effect of pointer mutation rate on pause times**

| old-gen collector | live data *(Mbyte)* | elapsed time *(sec)* | max heap *(Mbyte)* | young-gen pauses avg *(ms)* | max *(ms)* | total *(sec)* | old-gen pauses avg *(ms)* | max *(ms)* | total *(sec)* |
|---|---|---|---|---|---|---|---|---|---|
| **default** | 50 | 370 | 69 | 18 | 39 | 51.1 | 1298 | 1959 | 42.8 |
| **mostly-concurrent** | 50 | 334 | 93 | 24 | 70 | 69.8 | 14 | 39 | 1.2 |
| **default** | 100 | 351 | 189 | 19 | 36 | 54.1 | 2593 | 3491 | 20.7 |
| **mostly-concurrent** | 100 | 342 | 189 | 26 | 178 | 76.5 | 23 | 57 | 0.9 |
| **default** | 150 | 364 | 252 | 19 | 58 | 56.1 | 3985 | 6274 | 31.9 |
| **mostly-concurrent** | 150 | 347 | 286 | 28 | 361 | 81.7 | 27 | 67 | 0.7 |
| **default** | 200 | 363 | 315 | 20 | 42 | 57.5 | 4981 | 6763 | 29.9 |
| **mostly-concurrent** | 200 | 349 | 369 | 29 | 146 | 84.6 | 32 | 69 | 0.6 |
| **default** | 250 | 370 | 415 | 21 | 38 | 59.6 | 6944 | 10368 | 34.7 |
| **mostly-concurrent** | 250 | 356 | 498 | 31 | 145 | 91.1 | 41 | 105 | 0.6 |
| **default** | 300 | 362 | 500 | 21 | 39 | 61.8 | 7900 | 9938 | 23.7 |
| **mostly-concurrent** | 300 | 382 | 566 | 35 | 136 | 102.5 | 44 | 112 | 0.6 |

**Table 1: Default vs. mostly-concurrent for different heap sizes**

## 5.3 Pauses as a Function of Pointer Mutation Rate

A second mutator parameter that can lengthen the pause times of mostly-concurrent collection is the rate at which the program updates pointers in heap objects. In our implementation, each such update creates a dirty card whose marked objects must be considered roots in the the final marking phase. Recall that each "step" of the **gcold** application performs some number of pointer writes, controlled by a command-line parameter. Table 3 shows the effect of varying this parameters. The **gcold** application counts the number of pointer writes performed because of this parameter, and reports the cumulative rate of such writes at the end. Note that this count is only a lower bound on the number of pointer writes actually performed, since the application may also perform some other non-counted pointer writes (for example, as part of object construction.) Such writes use the same write barrier, and may increase the number of dirty cards. When two numbers are shown for the pointer mutation rate, the same input parameter gave slightly different rates for the runs with and without precleaning enabled. The live data size was kept constant at 200 Mbyte, and the mutator work parameter was kept constant at a value which produced promotion rate between 0.94 and 2.31 Mbytes/sec. (The variation reflects the fact that the pointer mutation operations added additional mutator work).

The table shows that final marking pauses are indeed quite dependent on pointer mutation rate. The concurrent precleaning technique is quite effective at decreasing this dependency, at least at moderate pointer mutation rates.

## 5.4 Validation: Real Programs

Table 4 compares the default mark-compact collector with the mostly-concurrent system on several programs from the SPECjvm98 benchmark suite, in particular, those that perform old-generation collections given a 20 Mbyte maximum heap size. (This omits the **jess** and **mpegaudio** benchmarks). As with the synthetic benchmarks, we find similar elapsed times, and somewhat larger heaps and significantly smaller old-generation pause times for the concurrent system. The **javac** benchmark explicitly requests garbage collections; such requests are performed synchronously even in the concurrent system. This accounts for the large maximum pause time for **javac**.

We also constructed a much larger workload for **javac,** compiling 2,740 source files from the standard Java libraries. These files contained 776,488 lines of code and producing 4,638 class files. We used this workload to compare the concurrent and stop-world collectors. We fixed the heap size at 139 Mbyte for both systems. Both systems performed similar numbers of young-generation collections (between 1,140 and 1,150 in both cases), with the young-generation collections of the concurrent system being somewhat longer: we measured a 32.8 ms average and 117 ms maximum pause for the concurrent collector, compared with a 24.6 ms average and 72 ms maximum for the default system. This slowdown was partially offset by concurrency in old-generation collections. The default system performed two old-generation collections in this run, averaging 3.2 sec each. The concurrent system paused 90 times for old-generation collection, but the average duration of these pauses was only 22 ms, with a 59 ms maximum and total time of 2.0 sec. For this benchmark, the increased young-generation collection time outweighed decreased old-generation time, so the elasped time for the concurrent system was greater than for the default: 190.6 sec compared with 184.6. For a batch process such as compilation, only the bottom-line elapsed time matters, so the concurrent collector offers no advantage. However, these measurements show that in applications where maximum pause time is an important factor, the concurrent collector offers roughly equivalent throughput with much smaller old-generation collection pauses.

Finally, we have also validated our collector with customer code. We have worked with a telecommunications company that has a call-processing application written in the Java language. Under expected workloads, it has a steady-state live data size of several hundred megabytes, yet the customer very much wishes to avoid pause times longer than one second. Stop-world and incremental collectors already offered in Sun's JVM products were unable to meet these requirements, but our concurrent collector has been able to process the old-generation garbage in a timely manner. This application updates pointers in old-generation at a relatively high rate; our concurrent precleaning techniques were crucial in keeping remark pauses within acceptable ranges. This application has been deployed using a limited-release version of the concurrent collector.

| benchmark | old-gen collector | elapsed time *(sec)* | max heap *(Mbyte)* | old-gen pauses avg *(ms)* | max *(ms)* | total *(ms)* |
|---|---|---|---|---|---|---|
| **compress** | **default** | 46.4 | 15 | 27 | 71 | 356 |
| **compress** | **mostly-concurrent** | 42.8 | 24 | 8 | 23 | 169 |
| **db** | **default** | 69.4 | 13 | 312 | 312 | 312 |
| **db** | **mostly-concurrent** | 68.6 | 19 | 25 | 64 | 101 |
| **javac** | **default** | 37.9 | 13 | 301 | 412 | 4213 |
| **javac** | **mostly-concurrent** | 40.0 | 19 | 102 | 754 | 2862 |
| **mtrt** | **default** | 10.9 | 12 | 296 | 296 | 296 |
| **mtrt** | **mostly-concurrent** | 11.1 | 16 | 17 | 30 | 34 |
| **jack** | **default** | 25.2 | 8 | 42 | 44 | 84 |
| **jack** | **mostly-concurrent** | 24.7 | 15 | 17 | 37 | 69 |

**Table 4: Effect of pointer mutation rate on pause times**

## 6. RELATED WORK

For a good detailed introduction to garbage collection, the reader is referred to two excellent publications, which touch most of the techniques mentioned in this paper: Jones and Lins' book [19] and Wilson's survey [28]. On a related topic, Wilson, Johnstone, and others survey different dynamic allocators and discuss the problem of memory fragmentation [29, 18].

A number of different techniques for incremental garbage collection have been proposed in the past, and it would be impossible to present all of them here, therefore we will only mention the most relevant ones.

*Generational* techniques attempt to visit newly-allocated objects more often than longer-lived ones, in the hope that the former are more likely to become garbage quickly. When this assumption holds, it is possible to collect most garbage objects by just looking at a small area of the heap (the young-generation) where objects are allocated, and hence minimise the garbage collection pause time. Generational garbage collection was first proposed by Lieberman and Hewitt [21], but Ungar reported the first implementation [26].

Probably one of the most famous incremental garbage collectors is *Baker*'s algorithm [3]. It is a two-space algorithm that works by copying live objects either eagerly (when they are accessed) or lazily (by a background process) from the from-space to the to-space, so that they are always manipulated in the latter. When all live objects have been copied, the two spaces are flipped. An implementation of this algorithm exists for the ResearchVM. Baker also proposed a variation on this algorithm, called the *Treadmill* [4], which removes the usually expensive two-space requirement (but requires extra per-object space).

Another incremental algorithm is *replicating garbage collection* [22], proposed by O'Toole and Nettles. This is a copying algorithm like Baker's, but where Baker's algorithm uses a read barrier to ensure that mutator threads observe only to-space references, replicating collection has mutators observe only from-space pointers during collection, and uses a write barrier to apply mutator updates (for scalar as well as reference values) to both from-space and to-space versions of objects.

There have been several other variants of concurrent mark-sweep collection. As we have mentioned, Dijkstra *et al.* presented one of the earliest forms [9]. Kung and Song added an advance in how the set of grey objects was represented [20]. The original *mostly-parallel* algorithm, on which this paper is based, was invented by Boehm *et al.* [5] as a way to introduce incrementality in the Boehm-Demers-Weiser conservative garbage collector for C and C++ [6]. This algorithm is described in Section 3. Doligez and Leroy [10], and Doligez and Gonthier [11], present different aspects of an innovative concurrent mark-sweep collector developed for the Concurrent CAML Light system. This collector scans mutator threads individually, never requiring all mutator threads to be stopped simultaneously. A group at IBM Haifa has adapted this collector design, along with modifications to support generational collection, to the Java platform [12].

Finally, an algorithm that has been gaining popularity lately is the *mature object space* algorithm, usually called the *train* algorithm, originally proposed by Hudson and Moss [17] and first implemented and analysed by Seligmann and Grarup [24]. In this algorithm, the heap is split into small regions (*train cars*), each of which can be collected independently (at the cost of maintenance of inter-car remembered sets). Garbage cycles, which can span several cars, are dealt with by the live objects being copied out of each region into other regions, in such a way that eventually the cycle will end up in one region and then be easily collected. An implementation of this algorithm for the ResearchVM is under way.

## 7. CONCLUSIONS AND FUTURE WORK

We believe that our *generational mostly-concurrent* algorithm improves both mostly-concurrent collection and generational collection. The original implementation of Boehm *et al.* was constrained by limitations of a C/C++ runtime system that effectively mandated the use of the virtual memory system for tracking of pointer modifications. Our implementation is appropriate in contexts, such as Java virtual machines, providing greater control over mutator code, allowing the use of card-table-based write barriers for tracking pointer modifications. This offers several advantages:

- it does not rely on operating system facilities, which may not be portable across platforms, and
- it is both more efficient than virtual-memory-based techniques, and yields more fine-grained, accurate results.

The use of the mostly-concurrent algorithm as the older generation in a generational system also has several advantages. While generational collection usually results in short pauses, there are still occasional long interruptions for old-generation collections. The use of a concurrent older-generation algorithm completes the generational system's "story" on GC latency. Further, the mostly-concurrent algorithm dovetails nicely with generational collection. Generational systems must implement a write barrier for tracking old-to-young pointers, and this write barrier can be easily adapted to also meet the mostly-concurrent algorithm's requirement of tracking modified old-generation pointers.

Our measurements show that for programs whose promotion rates are sufficiently low to allow a collector thread running on a separate processor to meet its deadlines, the use of a mostly-concurrent older generation dramatically decreases pauses for old-generation collection. Young-generation collection is slowed somewhat, but this slowdown can be more than offset by the offloading of collector work to the other processor.

There are several potential areas of future work. In the system of terminology we have adopted, we noted that "concurrency" (between mutator and collector) and "parallelism" (among multiple collector threads) were orthogonal collector properties. We might therefore attempt to further decrease garbage collection interruptions by making collection parallel as well as concurrent. We could parallelize the individual marking and sweeping phases; another interesting observation is that we could overlap the sweeping phase of one collection cycle with the marking phase of the next.

## 8. ACKNOWLEDGMENTS

We would like to thank Richard Jones, Andy King, Steve Heller, Brian Lewis, and anonymous reviewers for careful readings and insightful comments. Thanks also to Steffen Grarup and Y. S. Ramakrishna for recent implementation help.

## 9. TRADEMARKS

## 10. REFERENCES

[1] O. Agesen and D. Detlefs. Finding Reference in Java™ Stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of OOPSLA'99*, Denver, Colorado, USA, November 1999.

[3] H. G. Baker. List Processing in Real-Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] H. G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[5] H. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Canada, June 1991.

[6] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. In *Software – Practice and Experience*, pages 807–820, September 1988.

[7] C. J. Cheney. A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 11(13):677–678, November 1970.

[8] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proceedings of ECOOP'99*, pages 258–278, Lisbon, Portugal, June 1999.

[9] 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.

[10] 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.

[11] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 70–83, New York, NY, USA, 1994. ACM Press.

[12] Tamar Domani, Illiot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 274–284, Vancouver, British Columbia, Canada, June 2000. ACM Press.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[14] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *ACM OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, DC, October 1993.

[15] 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.

[16] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 106–119, New York, NY, USA, December 1993. ACM Press.

[17] R. L. Hudson and J. E. B. Moss. Incremental Garbage Collection of Mature Objects. In *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[18] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *First International Symposium on Memory Management*, Vancouver, Canada, October 1998. ACM Press.

[19] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996.

[20] H. T. Kung and S. Song. An efficient parallel garbage collector and its correctness proof. Technical report, Carnegie Mellon University, September 1977.

[21] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[22] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 161–174, Asheville, NC (USA), December 1993.

[23] T. Printezis. Analysing a Simple Disk Garbage Collector. In *Proceedings of the First International Workshop on Persistence and Java (PJW1)*, Drymen, Scotland, September 1996.

[24] J. Seligmann and S. Grarup. Incremental Mature Garbage Collection using the Train Algorithm. In *European Conference on Object-Oriented Programming*. Springer-Verlag, August 1995.

[25] Herbert Stoyan. Early Lisp History (1956-1959). Web version: `http://www8.informatik.uni-erlangen.de/html/lisp/histlit1.html`.

[26] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[27] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc, 1999.

[28] 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.

[29] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, Kinross, Scotland, September 1995.

[30] 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.

[31] B. G. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, August 1992.