RC24504 (W0802-087) February 22, 2008
Computer Science

IBM Research Report

Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors

Bill McCloskey
University of California at Berkeley

David F. Bacon, Perry Cheng, David Grove
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors

Bill McCloskey
U.C. Berkeley

David F. Bacon   Perry Cheng
David Grove
IBM Research

Abstract

Existing real-time garbage collectors are either unable to scale to large multiprocessors, or unable to meet hard real-time requirements even with specialized hardware support.

These limitations are rapidly becoming unacceptable: hardware improvements have brought multi-gigabyte heaps and ubiquitous multi-core parallelism; applications have increasingly stringent real-time requirements; and non-embedded, highly parallel server applications form an increasing percentage of real-time workloads.

We present Staccato, an algorithm that supports both hard- and soft-real-time garbage collection on stock hardware and both real-time and stock operating systems. The algorithm is incremental, concurrent, and parallel. Defragmentation can be performed on a per-object basis using a lock-free algorithm which requires no atomic mutator operations in the common case. On a real-time operating system it is capable of meeting hard real-time bounds.

We have implemented Staccato in IBM's J9 virtual machine and present an evaluation on IBM's real-time variant of Linux. Staccato is able to achieve maximum pause times of 753μs and an MMU of 85% over a 5ms time window, out-performing both IBM's Metronome-based product and Azul's soft real-time, hardware-assisted collector.

1. Introduction

Significant advances in the state of the art in real-time garbage collection have begun to move from research into practice. The Metronome collector [4] has been incorporated into a real-time Java virtual machine product from IBM [21] and adopted by Raytheon for use in hard real-time applications on the Navy's new DDG 1000 Destroyer [22]. Other vendors have delivered systems with various levels of soft real-time behavior [13, 7].

However, trends in both hardware technology and software systems are continuing to put pressure on the capabilities deliverable by real-time garbage collection. Financial arbitrage has reached the point where automated trading strategies can exploit differentials of a few milliseconds, telephony and video-conferencing are being integrated into mainstream applications, and many other domains are rapidly increasing their real-time requirements. Many of these new applications are not embedded systems, as was common for most real-time systems in the past. Instead they are large, server-based systems with abundant parallelism.

At the same time, rapidly increasing multi-core parallelism and large heaps enabled by 64-but architectures are creating pressures for highly scalable garbage collector implementations.

While there has been abundant work on parallel collection, and even on parallel collection with low latencies [17, 8, 6, 14, 26], it has been primarily concerned with providing good throughput and moderate pause times (in the 50-100 ms range) with a low probability (below 1%) of outliers. However, the types of applications we are targeting require absolute worst-case pause times on the order of 1 millisecond, coupled with a high and consistent minimum mutator utilization (or MMU [12]).

In previous work on the Metronome collector [4] (originally built on top of Jikes RVM [23]), we achieved worst-case pauses of 12.2ms and an MMU of 45% at 22ms. Designed for embedded applications, we explicitly chose to built a uniprocessor collector to focus on hard real-time behavior and avoid complications due to concurrency. In the process of re-implementing the technology for a product based on IBM's J9 virtual machine [21], we extended the algorithm to allow collector/collector parallelism and execution across a multiprocessor, but collector pauses were always performed synchronously between barriers, an approach we refer to as "stop-the-worldlet". This system runs well on small-scale blade-based multiprocessors, achieving worst-case pause times of 1 millisecond with an MMU of 70% over a 10ms time window.

However, because of the technology and business trends, we need a collector that can scale to very large multiprocessors while simultaneously reducing worst-case pause times by another order of magnitude. This paper describes our work on that collector, which we call *Staccato*. The contributions of this work are:

- An algorithm for performing incremental, parallel, and concurrent garbage collection on stock multiprocessors. The algorithm is based on lock-free data structures and ragged barriers, does not depend on custom hardware support, and does not suffer from MMU-reducing trap storms.
- An algorithm that allows objects to be moved during compaction without requiring any locks, and without requiring any atomic operations (e.g. compare and swap) by the mutators except in very rare cases. The algorithm is suitable for both strongly and weakly ordered multiprocessors.
- A single implementation methodology for hard- and soft- real-time systems, based on mutator-driven self-scheduling, that eliminates context switches, removes dependencies on RTOS features, enables collector quanta smaller than the operating system scheduling limits, and improves multiprocessor scaling.
- A scheduler that allows minimum mutator utilization (MMU) to be specified on a per-thread basis. This makes it possible to have high-priority, high-MMU threads that nevertheless have high allocation rates; the extra collection load can be absorbed by lowering the MMU of lower-priority threads and/or performing collection on otherwise idle processors.
- An implementation in IBM's J9 virtual machine. The major phases of collection are lock-free. Currently, some phases

(about 9-17% of collection time) have not been converted to lock-free data structures and still use barrier synchronization.

- An evaluation of our implementation on IBM's real-time Linux kernel, and a direct comparison against the Metronome collector in IBM's real-time Java product. Despite the limitations mentioned above, Staccato has much better real-time performance than the Metronome collector and has worst-case pause times that are more than an order of magnitude less than the best reported results for other low-latency compacting collectors.

Section 2 describes how Metronome's data structures and collection algorithm were extended for Staccato. Sections 3 and 4 present the two central innovations in Staccato: lock-free concurrent compaction and per-thread MMU-based scheduling. Section 5 describes shortcomings in our current implementation of Staccato in the J9 virtual machine. The performance and latency characteristics of Staccato are empirically compared to those of Metronome in Section 6. Finally we discuss related work and conclude.

2. Collection Algorithm

Like the Metronome collector [4], Staccato is a mostly non-moving mark and sweep garbage collector. The overall structure of the two collectors are similar; therefore in this paper we focus on those aspects that are unique to Staccato. In both algorithms, the collector's work is divided into three main phases: marking reachable objects, sweeping away unmarked objects, and compacting fragmented objects into contiguous pages. Staccato occasionally interrupts the mutator for short periods of time to incrementally perform these collection phases. Aside from allocation costs, the only other effects on the mutator are a write barrier to track pointer updates during the mark phase as well as a read barrier to forward pointers to objects that have been moved during compaction.

It is important to keep in mind that all interruptions to the mutator are either bounded in time (in the case of the read and write barriers) or they can be scheduled at arbitrary times, for arbitrary lengths of time (in the case of the collection phases), as long as the collector is able to keep up with the allocation rate. As described below, all data structures that are accessed by both collector and mutator threads are lock-free, therefore no mutator doing allocation or read/write barriering will ever have to wait for a garbage collecting thread. These properties guarantee that the Staccato algorithm has almost arbitrarily low pause times (bounded by the costs of allocation and the barriers).

We begin by describing the lock-free page list that is the central data structure shared between mutator and collector threads. The next two subsections describe the marking and sweeping phases of collection. Because efficient concurrent compaction is one of the main innovations of the Staccato algorithm it is described separately in Section 3.

To simplify the exposition we first present the algorithm as it should ideally be implemented. We then highlight in Section 5 those areas in which our initial implementation of Staccato in the J9 virtual machine fails to meet this ideal.

2.1 Page Lists

Like Metronome, Staccato organizes the heap into fixed-size pages (16 KB in the J9 implementation) and each page is divided into blocks of a particular size. Objects are allocated from the smallest size class that can contain the object. Associated with every page in the heap is a `PageInfo` object that contains collector metadata describing the state of the page. `PageInfos` are organized in doubly-linked lists. Each `PageInfo` belongs to exactly one list at any time (except while a `PageInfo` is being moved from one list to another during which it is momentarily only in a local variable). The collector maintains several `PageInfo` lists for each size class, representing the pages assigned to that size class that are available for allocation, waiting to be swept, or waiting to be compacted.

Mutator threads move `PageInfos` from the available list for a size class to thread-local allocation contexts in the off-branch of the allocation sequence. Collector threads move `PageInfos` among the various lists, including the available list, during the GC cycle. Because both mutator and collector threads may need to operate on the same list, certain basic operations must be lock-free. The fundamental `PageInfo` list operations are as follows: (i) pushing `PageInfos` onto the front or back of a `PageInfo` list, (ii) removing an arbitrary `PageInfo` from the middle of a `PageInfo` list, and (iii) iterating over `PageInfos` in a `PageInfo` list. Push and remove must be atomic, efficient, and lock-free. Iteration need not be atomic, but it must be lock-free.

The basic structure of the `PageInfo` list is similar to the `ConcurrentLinkedDeque` of the `java.util.concurrent` package. Each `PageInfo` contains next and prev pointers to its adjacent `PageInfos`. There are a number of difficulties in maintaining such a structure concurrently without using locks, which we treat here.

Previous pointers. It is impossible to atomically update the next and prev pointers simultaneously when adding or removing a `PageInfo`. Thus, we take the approach that next links are considered authoritative, while the prev pointers may at times contain out-of-date information. However, out-of-date information in the back pointers can be consistently detected (by verifying that `p->prev->next == p`) and corrected. When `PageInfos` are added or removed, a single atomic compare-and-swap operation is used to update the forward pointer of the preceding `PageInfo` in the list. A non-atomic write is used to update the back pointer. Between these two operations, the back pointer will be momentarily incorrect. Also, since other list operations may be interleaved between these two operations, the write of the back pointer may be writing stale information by the time it occurs.

To counter this problem, we correct invalid back pointers whenever they are discovered by traversing through forward pointers in the list to find the correct predecessor element and updating the back pointer. This algorithm has poor worst-case complexity, but since conflicts are very rare, back pointers are almost always correct. The worst-case complexity of the allocator, which must remove `PageInfos` from the free list, remains the same, since it already must perform a linear scan to find a free `PageInfo`. The effect of the complexity on the collector is unimportant, since its pause times are controlled by the scheduler.

Removal bits. When a `PageInfo` is removed, it is insufficient simply to swing the predecessor's next pointer to the `PageInfo` following the removed `PageInfo`. In between reading the successor pointer and writing to the successor element, another thread may add a `PageInfo` directly after the one being removed. To counter this problem, we use the technique of Harris [18], which reserves an extra bit in each next pointer to denote a pending removal and performs the operation using two compare-and-swaps.

Page list lattice. A common problem with lock-free data structures is knowing when elements are no longer referenced by any other threads. Without this knowledge, it is impossible to reuse the memory associated with an element. In the case of Staccato, this problem is particularly acute since a `PageInfo` may need to be moved from one `PageInfo` list to another without any pause in between. If this problem is not addressed, a thread iterating over the list of free `PageInfos` might end up iterating instead over a different list if the `PageInfo` it is currently processing is suddenly moved from one list to another.

To solve this problem, we reserve additional bits in the next pointers to denote the list to which a `PageInfo` currently belongs. These bits can be updated atomically with the next pointer, so they do not affect the atomicity of any operation. Since `PageInfos` are fairly large, many of the low-order bits are available for this purpose. When an iterating thread is traversing a list, it may find that the current `PageInfo` no longer belongs to the list it is interested in. In this case, it must restart iteration from the beginning. Given the improbability of such conflicts, this situation in quite rare.

However, it is very important that a `PageInfo` never move through a sequence of `PageInfo` lists and end up where it started (the 'ABA' problem). In this case, an iterating thread that was preempted for a very long time would restart its iteration at a different part of the list, possibly skipping some elements or processing them twice. To avoid such problems, we require that `PageInfo` lists be arranged in an acyclic lattice and that all `PageInfos` move downwards in this lattice. At the end of an entire collection, we use a ragged barrier over all threads to ensure that all list operations have terminated. After this barrier, `PageInfos` can be moved back to the top of the lattice.

2.2 Marking

The mark phase begins by scanning the root set for references to heap objects and pushing them on a mark stack. The collector threads then incrementally traverse the heap to find and mark all reachable objects. This is done by iteratively removing an object from the mark stack, scanning it for references to unmarked objects, and pushing any such references onto the mark stack for later processing. Throughout the mark phase, a write barrier is used to ensure that no objects are lost due to an unlucky interleaving of the mutator and the collector [24]. Like Metronome, Staccato uses a Yuasa-style write barrier [29]. For each pointer write, the previous pointer value that is being overwritten is pushed onto a thread-local write buffer. These references essentially act as an additional set of roots, and the write buffer has an identical purpose to the mark stack. During the mark phase, processing of the write buffer and mark stack is interleaved and both must be empty before the mark phase can terminate.

The main challenge in making root scanning concurrent and parallel is handling the scanning of thread stacks. The original Metronome collector [4] required that all thread stacks be scanned in a single atomic step to obtain a consistent snapshot of the root set. To maintain real-time bounds in the presence of a large number of threads, the Metronome implementation in J9 [20] instead introduced a per-thread "fuzzy" snapshot to allow thread stack scanning to be done incrementally. During the period while thread stacks are being scanned, write barrier operations are modified to record both the old (overwritten) and new value on every pointer store. This double barrier is enabled for all mutator threads by flipping a per-thread barrier state flag before any thread stacks are scanned. It is disabled (reverts to the normal Yuasa barrier) on a thread-by-thread basis after each thread's stack is scanned. By utilizing this modified snapshot, the atomic unit of work during root scanning is reduced to the scanning of a single thread stack. To some extent, a programmer can control the extent of this interruption by structuring their program such that threads with harder real-time needs have short thread stacks. One could also apply stacklets [11] to bound the interruption for threads with arbitrarily deep stacks.

During the mark phase, Staccato pushes references onto a mark stack. Multiple collecting threads may access the mark stack simultaneously, so their accesses must be synchronized. The mark stack need not be lock-free, since it is never accessed by the mutator. However, to ensure that garbage collection pause times can be scheduled precisely, it is important that collecting threads do not block indefinitely waiting to push onto the mark stack. Acquiring a lock with a timeout is sufficient to guarantee this. In practice, Staccato uses a lock-free data structure anyway for efficient load-balancing: each thread maintains its own thread-local fragment of the stack; when a popping thread runs out of thread-local fragments, it steals a fragment from another thread using atomic memory instructions.

After the roots are scanned, Staccato must process the references on the mark stack. For each reference it pops, Staccato marks the referenced object and scans its fields, pushing any new references to unmarked objects onto the stack. Any references to objects that have been moved during a previous compaction phase are forwarded at this time.

The one factor that differentiates the write buffer from the mark stack is that it is accessed by the mutator during the write barrier, so it must be implemented with a lock-free data structure. Otherwise, a mutator thread doing a pointer write could be held up by a collector for an unbounded length of time. Since our mark stack is lock-free already, we can re-use the underlying data structure to implement the write buffer.

2.3 Sweeping

During the sweep phase, Staccato scans through all non-empty pages in the heap, using the mark bits set by the mark phase to update a tally of the number of used and free cells on the page. Pages that are swept are first moved to a separate list so that concurrent allocation does not occur as this would confuse the tallies. Pages with a zero mark-count are moved to a free list, while non-empty pages are grouped into bins according to their occupancy rate. All collector-collector and collector-mutator interactions during the sweep phase occur through the page list data structure described above. Thus, there is no additional complexity in making the sweep phase concurrent.

3. Lock-free Compaction

A garbage collector can not properly be called a "real-time" collector unless it can tightly bound both its time *and* its space consumption. A real-time collector must therefore perform some form of compaction or defragmentation (although Siebert [27] has attempted to avoid this by building all objects out of linked structures of small, identically sized objects, the result is access time that is logarithmic in the object size).

Efficiently relocating objects while mutators are running concurrently is quite difficult. Entire objects must appear to move atomically so that no concurrent updates to the object are lost or appear inconsistently ordered to different threads. This problem is exacerbated by the fact that many multiprocessor systems provide only weak memory consistency. There is no guarantee that causal dependence is preserved across processors except in the presence of very expensive synchronization barriers.

Previous multiprocessor concurrent copying collectors have all relied on either per-object locking [28], per-page synchronization via page traps [1, 13], or a write barrier that writes to both the original and the copied object [12, 25].

Per-object locking is too inefficient and does not scale. Per-page locking via hardware traps causes low MMU both because the mutator may be forced to do collection work and because of the cost of the traps themselves. Double write barriers depend on a strongly consistent memory model for correctness, significantly increase write-back traffic, and require a synchronous final commit phase which is not constant time.

In this section we present an algorithm that overcomes the limitations of previous techniques. In the common case mutators access objects without any atomic operations even during defragmentation. In the uncommon case at most one atomic (compare and swap) operation will be required – mutators are never blocked by the collector's defragmentation activity. Additionally, "storms" of atomic operations are prevented by moving few objects and by randomizing their selection.

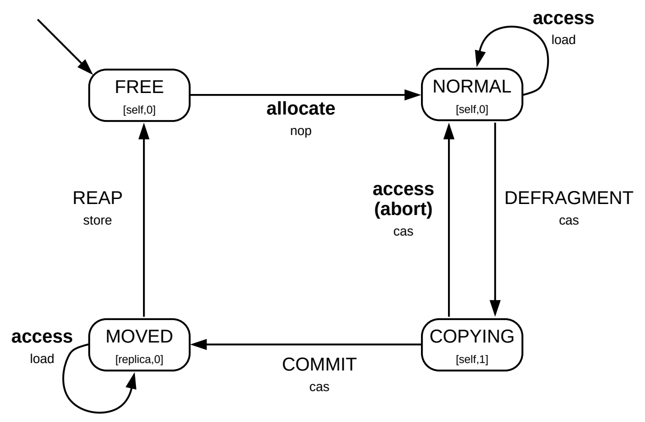


Figure 1. State transitions of the forwarding pointer for a single copy of an object.

3.1 Algorithmic Properties

Like most compacting collectors, Staccato stores a *forwarding pointer* in each object's header. When an object is moved, but before all pointers to it are redirected, the forwarding pointer slot in the original object points to the copy. Staccato uses a Brooks-style read barrier [10] to forward every pointer read. Previous work described techniques for reducing the cost of read barriers to an acceptable level [4].

Staccato's defragmentation algorithm makes use of the following techniques and properties of the system:

- *Asymmetric Abort Protocol*, in which the collector tries to copy the object but the copy is aborted by any mutator which touches the object before the copy has been committed. In the normal case the protocol requires no atomic operations for object access; during defragmentation it will cause at worst one compare-and-swap operation per processor.
- *Ragged Synchronization*, which allows global propagation of information without global synchronization [2]. The mutator threads are instrumented to perform memory barrier operations at regular intervals; to propagate information to the mutators, the collector need only wait until all running threads have passed a ragged barrier (ragged barriers are also used in the Azul collector [13]; they call them "checkpoints"). On weakly ordered machines, the ragged barriers must perform memory-fence operations (e.g. `sync` on the PowerPC).
- *Mostly Non-copying* collection [4], which only moves a small fraction of the total objects to eliminate fragmentation.
- *Arraylets*, which break large objects (arrays) into chunks which form the largest contiguous units allocated by the system [3]. This means that there is no external fragmentation beyond the arraylet granularity, and bounds the atomic work required to move an object by the arraylet size (2 KB in the J9 implementation).

The combination of arraylets with a mostly non-copying approach greatly reduces the number of objects that must be copied in order to keep fragmentation levels low

The combination of a small set of objects to move, a small time window in which they are moved, and randomization makes the algorithm highly robust.

3.2 Moving an Object

We now describe the protocols used by the collector and mutator to achieve lock-free object movement on a weakly ordered multiprocessor. Figures 2 and 3 contains the pseudo-code for the algorithm; the numbers at the far right indicate the steps described in detail below. Figure 1 shows the possible state transitions of the redirect pointer of the original copy of an object during the stages of this protocol.

3.2.1 Collector Protocol

The collector reserves a bit in the forwarding pointer to denote that the object is being copied (Java objects are always word-aligned, so stealing a bit poses no problem). In this way, the collector can use a single compare-and-swap operation to atomically change both the copying flag and the forwarding pointer.

To move an object, the collector performs the following steps:

1. Set the COPYING bit in the forwarding pointer of the object using a compare and swap operation. However, since mutators access the redirect pointer without atomic operations, it will take some time for this update to propagate to the mutators.
2. Wait for a ragged synchronization. At this point, all mutators will have performed a memory read synchronization and the change to the COPYING bit is guaranteed to be globally visible.
3. Perform a memory write synchronization (weakly ordered machines only). This ensures that all modifications made by the mutators before they saw the change to the COPYING bit are seen by the collector.
4. Copy the data fields from the original object to the relocated object.
5. Perform a memory write synchronization (weakly ordered machines only). This pushes the newly written data fields in the relocated object so that they are globally visible.
6. Wait for a ragged synchronization. When complete, all mutators will see the values written into the relocated object.
7. Perform a compare and swap which simultaneously changes the forwarding pointer to point to the relocated object and clears the COPYING bit. This commits the move of the object. If the compare-and-swap fails, then the mutator read or wrote the object at some point, and the move is aborted.

When the collector "waits" for a ragged synchronization, it does so either by performing other work or by allowing the mutator to run.

3.2.2 Mutator Protocol

Meanwhile, when the mutator accesses an object (to read, write, lock, unlock, or otherwise examine or modify its state) it performs the following steps:

1. Load the forwarding pointer.
2. If the COPYING bit of the forwarding pointer is clear, use the forwarding pointer field as the object pointer.
3. Otherwise, try to abort the copy by using a compare-and-swap to clear the COPYING bit (which is the same as storing the object pointer itself).
4. If the compare-and-swap succeeds, use the resulting forwarding pointer with the COPYING bit cleared as the object pointer.
5. Otherwise, reload the forwarding pointer using an atomic read (which is guaranteed to see the current atomically written value). The fact that the previous compare-and-swap failed indicates that either the collector committed the copy or else another mutator aborted it; in either case this atomic read will obtain the new value.

3.3 Reducing Abort Frequency

Many strategies for compaction are vulnerable to the "popular object problem." In some collectors, this manifests in long latencies

```
objects collectorMoveObjects(objectList)
  abortList = new List()
  replicaList = new List()

  for each (obj in objectList) // 1
    prepare(obj)

  raggedSynchronization() // 2
  readSync() // 3

  for each (obj in objectList) // 4
    replicaList.append(copy(obj))

  writeSync() // 5
  raggedSynchronization() // 6

  for each (obj in ObjectList, rep in replicaList) // 7
    if (! commit(obj, rep))
      free(rep)
      abortList.append(obj)
  return abortList


void prepare(obj)
  obj.forward.atomicCAS(obj, obj | COPYING)

object copy(obj)
  s = obj.size()
  replica = allocate(s)
  copyBytes(obj, replica, s)
  replica.forward.plainWrite(replica)
  return replica

bool commit(obj, rep)
  return obj.forward.atomicCAS(obj | COPYING, rep)
```

Figure 2. Collector pseudo-code for Staccato object relocation.

```
object mutatorAccessBarrier(obj)
  forward = obj.forward.plainRead() // 1
  if (forward & COPYING == 0) // 2
    return forward
  if (obj.forward.atomicCAS(forward, obj)) // 3
    return obj // 4
  return obj.forward.atomicRead() // 5
```

Figure 3. Mutator pseudo-code for Staccato object relocation.

when moving an object with many incoming pointers, because the relocation operation must atomically redirect all of those pointers.

Staccato has an analogous issue, in that objects that are being accessed at a very high frequency will prove difficult to move, because the accesses will trigger aborts. While we have not observed this problem in any program so far, it can be circumvented by detecting the popular object and using its page as a target of the compaction phase. That is, instead of trying to move a popular object off of a low-occupancy page, we simply increase the page's occupancy level. We note that this symmetry is exploitable because of our page architecture but is not possible in sliding compaction.

Another potential issue is "abort storms", which could happen if the collector chose to move a set of objects that were all accessed with temporal locality by a mutator. This would cause the mutator to execute an elevated number of compare-and-swap operations in a short time, which could reduce its MMU.

This is already unlikely because we move relatively few objects, and those objects are being moved off of low-occupancy pages – so objects allocated close in time are unlikely to be moved at the same time.

However, if necessary, the probability of correlated aborts can be further reduced in three ways: first, by breaking the defragmentation into several phases, thereby shortening the time window during which object moves may be aborted. Second, by randomizing the set of pages chosen for defragmentation in each phase. And third, by loosely aligning (but *not* synchronizing) the execution of the object relocation by different threads (but without violating MMU requirements), there will be fewer mutator threads running while objects are being moved, and the probability of aborts will be reduced even further.

4. Scheduling

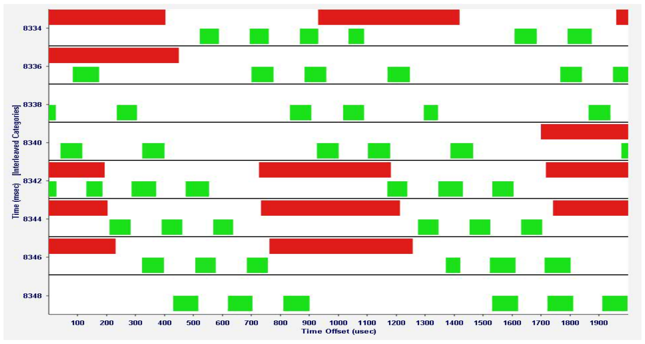
The principal motivation behind Staccato is to increase scheduler flexibility and performance, especially for multiprocessor systems. The best way to achieve these goals is to ensure that all decisions about when to collect are made *locally*. Each thread must be free to choose when to collect and how much collection to do. Since communications between processors is expensive in contemporary systems, local decision making increases performance. Because each thread can make different choices about the work to perform, we also improve the flexibility of the algorithm.

Programmers can tune the Staccato collector in two dimensions:

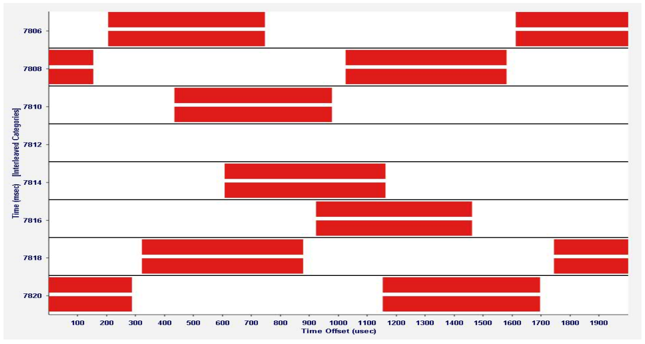
- Garbage collection work may be performed by the mutator threads or by special garbage collection threads. Like mutator threads, garbage collection threads may run at low priority, so that they "soak up" unused CPU cycles, or at high priority on a dedicated CPU.
- Threads may perform garbage collection at varying rates. High priority threads might perform no garbage collection so that they are never interrupted. Less important threads could collect more frequently so that the overall collection rate of the system is suffient to keep up with the overall allocation rate.

Each thread's rate of collection is determined by a target *minimum mutator utilization* target over a given time window. This number determines the minimum percentage of CPU cycles the mutator thread should have available over the time window. Every thread in the system is assigned a target MMU; threads devoted to garbage collection always have a 0% MMU, since they perform no mutator work. The total collection rate of a system is determined by thread MMUs as well as their relative priorities. A system running too many high-priority high-MMU threads may not collect fast enough.

The Metronome garbage collector is also driven by a target MMU. However, Metronome makes all scheduling decisions globally. When the scheduler chooses to collect, it requests that all mutator threads suspend and waits for them to stop. Metronome then collects in parallel, and resumes all mutators when done. The rate of collection is based on a single, global MMU target. This MMU must be sufficiently large for all threads in the system to meet their deadlines, but not too large or else the collector will run too slowly and the system will run out of memory. Metronome's approach is also inefficient, since all the mutator threads must be suspended for each collection increment; typically there are 600 suspensions per second during a collection. In an application with many threads,



(a) Staccato GC activity



(b) Metronome GC activity

Figure 4. A depiction of GC scheduling in Staccato (a) and Metronome (b) with two mutator threads. Time advances from left to right, one row at a time. Each row (separated from the next row by a black line), contains (upper) red segments which show GC activity in the first thread and (lower) green segments which show activity in the second thread.

where some threads take longer than others to voluntarily suspend, many cycles can be wasted.

Figure 4 shows a visual depiction of the scheduling policy differences between Staccato and Metronome. These two figures show data from actual runs of a synthetic benchmark. Time moves from left to right, one row at a time. The top half of each row shows the activity of one thread and the bottom half shows the activity of another. Filled segments represent GC activity, while the remaining time is given to the mutator.

In the Staccato portion, the top thread (in red) has requested a 70% MMU over a 10 ms window. The bottom thread (in green) has asked for a 70% utilization over a 1 ms window. Staccato satisfies these requests by dividing time into beats. Ideally, the collector runs mutation work and collection work for an integral number of beats, although it may use partial beats to correct any timing errors when a collection quantum runs too long or too short. The top thread uses 500 μs beats and the bottom thread uses 100 μs beats. Each strip represents 2 ms, which is two windows for the bottom thread. The GC, which is to use 30% of the cycles, therefore runs for 600 μs, or 6 beats, per strip. The top thread runs each GC quantum longer and less frequently, as expected.

The bottom half of Figure 4 shows the same benchmark in Metronome. Metronome uses the same beat-based scheduling algorithm to achieve a given MMU. However, both threads must be run with the same target utilization, window size, and beat length. Additionally, threads are forced to run in lockstep: at a given time, all threads are either mutating or collecting.

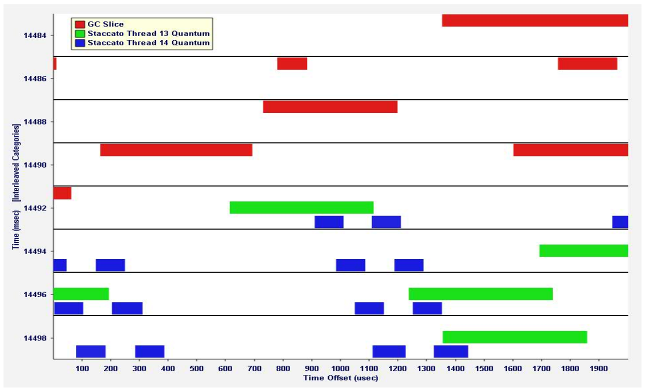


Figure 5. A view of a stop-the-worldlet collection mixed with mutator-thread collections for two threads.

The Staccato approach has clear advantages. In systems that have cycles to spare, Staccato can perform continuous garbage collection without disturbing the mutators. The collection can run either on a low priority thread or on a spare processor. In the extreme case, some mutators may not perform any collection at all. It is ill-advised to perform collection in Metronome except when memory is low, since no threads are available to respond to events that occur during the collection.

5. Reality Check

We chose to implement the Staccato algorithm in IBM's J9 virtual machine. J9 is a very efficient, production-quality JVM and, as such, is quite complex. This complexity, coupled with the fact that J9 was designed well before Staccato or Metronome, makes it somewhat impractical to implement several aspects of the Staccato algorithm. In this section, we describe the differences between the idealized Staccato and the J9 version. In the next section, we explain the impact of these differences on performance.

Certain aspects of JNI require support for allocation of large objects which cannot be represented with discontiguous arraylets. As a result, there are page list operations that cannot be accomodated as we have previously described. Such allocations (which are inherently non-realtime) require that we take a lock.

Although all of the time-consuming aspects of the collection is done in parallel and concurrently with the mutator, there are some exceptions. The processing of class loader objects, JNI global references, JDWP references, and soft, weak, and phantom references is done atomically. Furthermore, the processing of class objects (including class-static fields), interned strings, and thread-stack scanning is performed incrementally but not concurrently. Finally, compaction of the page-table data structures (related to the large object support just described) is not performed in parallel. These "stop-the-worldlet" collections are shown in red at the top of Figure 5. These occur infrequently, as we show later.

Because the stop-the-worldlet phases already introduce global (albeit infrequent) synchronizations, we have not yet implemented the ragged barrier previously described. Implementing this in any virtual machine with safe-point based precise garbage collection (such as J9) is straightforward by hooking into the periodic *safe-points*. The only caveat is that in a real-time system, one must be careful about not over-optimizing away safe-points to avoid too large of a gap between adjacent safe-points. In the current system, the ragged barrier has been replaced by the stronger synchronous barrier.

Although the Staccato algorithm is designed to work on systems with weak memory model, our current implementation runs only on the x86. Under the fairy strong x86 memory model (or the even stronger AMD model), a memory barrier is not needed to accompany the ragged barrier. Since we have replaced the weak ragged barrier with a synchronous barrier, the memory barrier is not needed.

Finally, we have not yet extended the JIT compiler to generate the more complex Staccato read barrier. Because the barrier introduces common-case control flow plus an uncommon case procedure call on all reads of heap objects, it will take significant engineering effort to implement it efficiently and to ensure that the various JIT optimizations (particularly those related to code motion) are aware of the barrier semantics.

6. Evaluation

In this section, we explore the decisions made in the design of Staccato and explain how they affect collector performance. There are several components of performance that we are interested in: (1) minimum mutator utilization over a variety of window sizes, (2) maximum pause time of the collector, (3) maximum latency as defined by the mutator application being tested, and (4) total throughput of the mutator application. We use two standard benchmarks for the mutators, SPECjbb2000 and SPECjvm98, as well as a synthetic transactional benchmark of our own design.

We ran all benchmarks on an IBM LS20 blade (model 8850) with two dual-core 2GHz AMD Opteron 270 processors, each with 1 MB of L2 cache. The system had 5 GB of RAM. The SPECjbb2000 benchmarks were run with a 256 MB heap, while the SPECjvm98 benchmarks were given a 64 MB heap (in both cases to ensure that collections happened with reasonable frequency). All benchmarks were run in an experimental version of the IBM J9 virtual machine running in interpreted mode (the Staccato write barrier is not yet implemented in the JIT compiler). [1]

Our validation is divided into three sections. We begin by examining properties of defragmentation, stop-the-worldlet collections, and `PageInfo` lists and showing that they do not have any impractical worst-case behaviors. Then we look at the throughput of application programs and compare to the Metronome collector, ensuring that programs are not significantly slowed down by read barriers for the new defragmentation algorithm. Finally, we present a close look at MMU, pause times, and latency, comparing to Metronome and to a competing pauseless collector [13].

6.1 Algorithmic Properties

Here we examined three different components of the collector: the stop-the-worldlet quanta, defragmentation, and `PageInfo` lists.

Stop-the-worldlet collections. The major divergence between the ideal algorithm (Section 2) and the one we implemented (Section 5) is the use of occasional global pauses, where all threads are forced to simultaneously perform collection work. To ensure that these stop-the-worldlet collector quanta are not a significant portion of the total time spent collecting, we measured the ratio over our benchmark suite. The results are presented in the first column of Table 1.

The average percentage of time spent performing stop-the-worldlet (STW) collections over all benchmarks was 17.2%. However, the STW percentage for _201_compress is a major outlier. We used the TuningFork performance analysis tool [19] to track down the cause, and found a safe-point placement problem in both the Metronome and Staccato collectors. The error causes the allocation of large objects to be performed atomically and to allow multiple such allocations to occur without any intervening safe-point. In general, our instrumentation considers the time between a thread-premption request is posted to the time when all threads have reached their safe-points to be collector work to be conservative in our accounting. In practice, for a single-threaded application this charge to the collector is unnecessarily stringent. Because _201_compress is single-threaded, it is correct to count these time intervals as mutator work, thereby removing the outliers. This correction would reduce the overall STW perentage to 8.0%. However, because this strategy does not generally work (for multi-threaded applications), we can only report an overall 17.2% figure until this issue (both the collector and the VM aspects) is resolved. More generally, as we do the engineering work to reduce the number of atomic and non-concurrent phases in the J9 implementation, this number will continue to drop.

Defragmentation aborts. Our lock-free compaction algorithm is an important and novel step beyond the Metronome collector, so we studied it carefully. The next six columns of Table 1 are related to defragmentation. We tested two defragmentation policies. In the first one (shown in columns labeled [ALL]), the algorithm attempts to defragment every objects possible. Pages are sorted by occupancy (percentage of objects in the page that are allocated) and objects from low-occupancy pages are moved to high-occupancy pages. Eventually, the algorithm runs out of high-occupancy pages and stops. We tested a second policy (shown in columns labeled [15%]) in which only pages with occupancies between 0% and 15% are evacuated. The first policy is meant to stress-test the defragmentation algorithm, while the second one is more reasonable and is intended to be used in practice.

The first column of each policy (labeled "average pages moved") gives the average over all collections of the percentage of pages that were successfully evacuated. One reason for unsuccessful evacuations is the lack of high-occupancy pages, as mentioned above. Another reason is that an object on the page could not be moved because a mutator wrote to the object while it was being copied (this is called an abort). In most cases, a greater percentage of objects is moved using the ALL policy than the 15% policy, but sometimes nondeterminism (which is a significant factor in aborted page evacuations) causes this not to be true.

It is interesting that in most cases the two policy perform quite similarly. As we show, much of the similarity is due to the lack of availability of high-occupancy pages, which limits the number of pages that can be evacuated. The columns labeled "object moves" for each policy show the number of objects that the collector attempts to move over all collections. (The collector only tries to move an object if there is a high-occupancy page in which to put it.) The number in parenthesis is the percentage of moves that are aborted by the mutator. In both policies, less than 1% of moves are aborted in any benchmark.

Of course, a page cannot be evacuated if *any* one of its objects cannot be moved due to an abort. The columns labeled "page moves" show the number of pages whose objects the collector tries to move to an existing high-occupancy page. The number in parentheses is the percentage of page evacuations that fail due to aborts. In one case, as many as 17.2% of page evacuations are aborted. However, the less aggressive policy fares better, with at most a 5.1% abort rate. This policy evacuates only pages with low occupancy, so there are fewer objects to move per page, and thus a lower probability of failure.

The low failure rate of the 15% policy suggests that the similarity in portion of pages moved between the two policies is due to lack of availability of high-occupancy pages. This means that the more reasonable 15% policy is almost successful at defragmenting

[1] In addition to the changes we made to implement Staccato, we also disabled support for the Real-Time Specification for Java (RTSJ) standard [9] and enabled defragmentation. Therefore, the performance results for our Metronome configruation are not directly comparable to the WebSphere Real Time product.

Benchmark	Number of collections	Time spent in STW (%)	Avg pages moved [ALL]	Avg pages moved [15%]	Object moves [ALL] (abort%)	Object moves [15%] (abort%)	Page moves [ALL] (abort%)	Page moves [15%] (abort%)	PageList ops
SPECjbb 2000 (1thr/256M)	7	7.0%	18.0%	14.0%	132575 (0.1%)	53036 (0.5%)	4113 (10.0%)	3200 (4.3%)	403736 (0%)
SPECjbb 2000 (4thr/256M)	68	11.6%	15.6%	10.8%	3869326 (0.6%)	1524890 (0.3%)	94688 (7.2%)	56078 (4.7%)	3656855 (0.05%)
_201_compress	3	82.0%	1.5%	1.6%	2903 (0.8%)	378 (0%)	58 (17.2%)	48 (0%)	38963 (0%)
_202_jess	12	10.1%	2.9%	13.3%	14119 (0.5%)	11485 (0.0%)	305 (4.9%)	1578 (0.3%)	198882 (0%)
_209_db	4	3.8%	1.2%	0.7%	3478 (0.0%)	423 (0%)	53 (1.9%)	32 (0%)	46552 (0%)
_213_javac	12	3.7%	35.2%	21.2%	665317 (0.7%)	91549 (0.2%)	7917 (6.0%)	3999 (1.6%)	179430 (0%)
_222_mpegaudio	0	—	0	0	0	0	0	0	156 (0%)
_227_mtrt	8	6.8%	3.9%	1.9%	25267 (0.0%)	2975 (0.2%)	289 (5.9%)	157 (5.1%)	124758 (0.01%)
_228_jack	5	12.8%	13.1%	13.0%	18874 (0.3%)	7513 (0.1%)	385 (3.6%)	375 (2.7%)	77948 (0%)

Table 1. We measured a variety of collector characteristics over our benchmark suite.

the heap as the ALL policy, but it is cheaper and has a lower failure rate. To reduce the failure rate further, the collector could randomly select low-occupancy pages to evacuate in each collection.

PageList operations. One concern in using our lock-free `PageInfo` list data structure is that contention between threads can cause previous pointers to be incorrect, forcing a potentially complete traversal of the entire `PageInfo` list. In the last column of Table 1, we measured the total number of `PageInfo` list operations as well as the number of re-traversals (in parentheses as a percentage). The highest contention rate was for SPECjbb2000 for which we specifically measured the cost of `PageInfo` allocation, which is the only scenario in which the `PageInfo` list data structure interacts with the mutator. In the worst case it was only 20 μs.

6.2 Throughput

Another performance concern regarding the Staccato collector is that it uses an especially complex write barrier to abort concurrent moves during the compaction phase. This barrier operates on all stores, not just pointers. Click et al. [13] benchmark SPECjbb, but they do not provide throughput results for comparison. Instead, we compared Staccato to Metronome, which uses a standard Yuasa-style write barrier on pointers. (Metronome and Staccato use similar read barriers; Staccato simply adds one additional mask instruction.)

We measured the throughput of SPECjbb2000 in transactions per second running three warehouses (three threads) with no compaction. Compaction was disabled because we were interested only in mutator throughput, not differences in the collection algorithms. For a similar reason, we tested two versions of Staccato. In one, collection was primarily performed on mutator threads as usual. In the other, all collection work was done in stop-the-worldlet mode. All three versions used identical scheduling policies (same target MMU and window size). We did this to make the collection algorithms as similar as possible. Disabling collection entirely was not possible on the machine we tested on. The results are shown relative to Metronome, since absolute SPECjbb2000 results are relatively meaningless.

Configuration	SPECjbb throughput
Metronome	1.0x
Staccato (STW mode)	1.04x
Staccato (GC mostly on mutators)	1.07x

Despite the more barriers, Staccato still performs better than Metronome, even in STW mode. The difference may be explained by the use of lock-free `PageInfo` lists in Staccato and not in Metronome. We were unable to eliminate this variable from the experiment since this data structure is an integral part of the Staccato implementation. Nevertheless, the additional cost of the Staccato write barrier does not seem to be a significant performance problem.

The difference in performance between the two Staccato configurations can be explained by the elimination of most context switches between GC threads and mutator threads. In this case, it is fairly clear that running collections on the mutator threads generates a 3% speedup in overall application performance.

6.3 Latency

We measure three interconnected values that determine an application's responsiveness: the minimum mutator utilization over a given time window, the worst-case pause time, and the maximum latency as perceived by the application. In the third case, most application have notions of deadlines or transactions that should complete in a specified time bound.

Unlike the ideal collector presented in Section 2, the collector we implemented in J9 cannot meet arbitrary latency goals. Because of the way J9 works, many of the stop-the-worldlet collection phases must be executed atomically, and so the collector must pause for at least this long to complete the phase. This pause time in turn determines the MMU and maximum latencies that the collector can support. Currently, the J9-imposed minimum pause time for both Metronome and Staccato is approximately 500 μs.

However, the performance of a collector is also constrained by its ability to keep up with the allocation rate. If the collector pauses only once in a given time window W, then it can support an MMU of $1 - 500\mu s/W$ over that window. Based on this equation, Metronome and Staccato both support an MMU of 90% over a 5 ms time window. However, Metronome is unlikely to be able to keep up with the allocation rate if it pauses only once per 5 ms. In most cases, Metronome is run with a 70% utilization over 10 ms for this reason.

Staccato has more scheduling flexible than Metronome. It might run at 90% utilization over 5 ms in its stop-the-worldlet phases while using an idle thread on a spare CPU to perform the remainder of the collection work. Mutator threads would be unaffected by the idle thread. Another alternative would be to run unimportant work with a low mutator utilization target; this thread would perform the bulk of collection work, while more important threads would benefit with better responsiveness. To demonstrate Staccato's advantages, we have tested both these scenarios.

Idle thread test. We tested SPECjbb2000 with a 256 MB heap by running it with two mutator threads (two warehouses), one idle thread, and one GC thread to perform STW collection phases. Each mutator thread was assigned a target utilization of 100%, while the STW collections were run with a target utilization of 90% over 5 ms. The maximum pause time over the test period was 753 μs, which is typical for both Metronome and Staccato. Since SPECjbb2000 is a transactional benchmark, we also measured transaction times. The mean transaction time was 563 μs and the maximum time was 2078 μs. (SPECjbb transactions are not designed to all be the same length.) The theoretically achievable MMU over 5 ms, given a 500 μs maximum pause time, was 90%. Staccato achieved an 85% MMU.

We ran Metronome on the same benchmark. Its GC was performed entirely in STW phases on the GC thread, rather than alternating between the GC thread and an idle thread as Staccato does. Metronome was able to keep up with the allocation rate, but its MMU frequently dropped to 70%, and sometimes as low as 65% over the 5 ms window. The pauseless collector of Click et al. [13], also evaluated on SPECjbb, had an MMU of 0% over 20 ms windows and 52% over 200 ms windows.

High-priority test. As a second test, we ran a synthetic transactional benchmark in parallel with SPECjbb2000. Our synthetic benchmark has more regular transaction times than SPECjbb, so the effects of the collector on it are clearer. It performs allocation and numerical computation in a tight loop. Each transaction is designed to last 7 ms. The synthetic benchmark ran with a higher thread priority and with a 100% utilization target. SPECjbb was run with a 50% target over 10 ms to ensure that the collector could keep pace with allocation. This scenario is meant to resemble a real-world situation with an important thread that should almost never be preempted by the collector. Stop-the-worldlet collections ran with a 95% target utilization over 10ms, as in the previous experiment.

In this test, the maximum collector pause time was 679.6 μs. The synthetic benchmark obtained an 88% MMU over a 5 ms window. The synthetic transactions had mean duration 7.031 ms and maximum duration 7.797 ms. This test reinforces the previous one, showing that Staccato can achieve low pause times and good MMU despite its occasional reliance on stop-the-worldlet collection phases.

7. Related Work

Multi-core and multi-processor computers are becoming ubiquitous even as garbage-collected languages (be it general-purpose Java or Web-based scripting languages) are used in areas where overall timeliness is necessary. These factors have led to modern garbage collectors that possess one or more of the following attributes: *incrementality* – the ability to break a collection into small units; *parallelism* – the ability to run multiple collector threads simultaneously; *concurrency* – the ability to run collector threads simultaneously with mutator threads; *compacting* – the ability to avoid memory exhaustion due to fragmentation; and *real-time* – the ability to provide predictable pause times, context switch latencies, and minimum mutator utilization. As explained earlier, Staccato has all of these features, with the proviso that the implementation is not yet fully concurrent.

Perhaps the first attempt at a real-time collector was implemented by Baker [5] for LISP in which he bound the operation of each list-manipulating primitive to constant time. Unfortunately this work-based algorithm fundamentaly does not decouple the mutator from the collector, leading to the mutator always performing a large of amount of collection work when a collection is first initiated and the copying is fault-induced. Although the pauses are predicted at an instruction level, the actual utilization experienced by the program at any reasonable granularity is very low.

Doligez et al. [15] presented a concurrent collector with a proof of correctness. However, objects are moved only from a nursery to the mature space in a synchronous manner. Depending on whether we keep the generational aspect, this algorithm is either not real-time or not compacting. More recent on-the-fly collectors [16] also cannot compact the heap to remove fragmentation and are unsuitable for long-runing applications.

Nettles and O'Toole [25] introduced replicating copying and eliminated the need for a read-barrier by performing a *double-update* for each mutator write and typically has low pauses. While this technique is suitable for functional languages where the mutation rate is low, it can be a net loss for imperative languages. Additionally, the pauses are not guaranteed because large objects and incremental log processing was not handled. was often.

Cheng and Blelloch [12] formalized the notion of MMU and presented a parallel, real-time collector that met MMU goals in the 10ms range. However, because it is a replicating collector the space overhead is large and space consumption can approach double that of a defragmenting collector.

Flood et al. [17] describe a parallel collector with a parallel sliding mark-compact phase for the old generation. Ben-Yitzhak et al. [8] (in expanding over previous work by Barabash et al.[6]) explore how the pause-time cost of compaction in the IBM product JVM can be ameliorated by what they call a parallel, incremental compaction algorithm. However, the algorithm is not incremental in the usual sense [5]; instead, a section of the heap, say $1/16^{th}$, is compacted when the collector detects fragmentation). Unfortunately both suffer from the popular object problem (in this case, popular region) so that the pause time is not bounded due to non-concurrent fixup. There are other collectors [14, 26] that perform partial heap compaction. These techniques fundamentally cannot provide real-time guarantees because the amount of work necessary for the compaction is potentially much larger than the region being compacted. Additionally, space overhead cannot be controlled and can be large (e.g. up to 30% for SPECjbb2000) in practice.

The Azul garbage collector [13] uses specialized hardware and, like Baker's collector, does not decouple the mutator from the collector. In some 50ms time windows, the program is only able to run for 10ms. Out of the lost 40ms, half of it is attributable to engineering issues related to root-scanning. Unfortunately the other 20ms is more fundamental and comes from the "trap storm" that arises from the mutator being forced to do collection work.

In comparison, Staccato's worst-case pauses of 753μs are 28 times shorter than the 21 ms pauses reported for Azul's so-called "pauseless" collector [13], and the Staccato implementation achieves an 85% MMU at 5ms, while Azul's reported MMU is only 52% at 200ms. Although their machine has as many as 384 processors, published measurements are for 8 workers CPUs and an unspecified number of additional GC CPUs. Thus their scalability results are roughly similar to ours.

The Metronome system in IBM's real-time Java product [21] is parallel, incremental, and real-time – but not concurrent. The original Metronome collector implemented in Jikes RVM [4] was incremental and real-time but neither parallel nor concurrent. We have already compared Staccato in considerable detail with Metronome [4] in section 6.

8. Conclusion

This paper presents an algorithm for concurrent real-time garbage collection. It introduces an efficient, lock-free algorithm for compacting the heap in the presence of mutator-collector parallelism. Combined with lock-free implementations of the mark and sweep phases of collection, we have demonstrated a practical collector implemented in IBM's production J9 VM. Our algorithm permits the user to specify per-thread MMU targets and opens up a wide range of collector scheduling policies that support high allocation rates even for high priority high, MMU threads. Empirically, our implementation supports an 85% MMU over a 5ms time window. These results improve significantly upon previous work as well as competing low-latency collectors.

Acknowledgements

This research was conducted during an internship at IBM Research in the summer of 2006.

References

[1] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices*, *23*, 7 (July), 11–20.

[2] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 92–103.

[3] BACON, D. F., CHENG, P., AND RAJAN, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, June 2003). *SIGPLAN Notices*, *38*, 7, 81–92.

[4] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[5] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[6] BARABASH, K., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., OSSIA, Y., OWSHANKO, A., AND PETRANK, E. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst. 27*, 6 (2005), 1097–1146.

[7] BEA. BEA WebLogic Real Time, Core Edition 1.1. URL `edocs.bea.com/wlrt/docs11/index.html`, July 2006.

[8] BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., KUIPER, K., AND LEIKEHMAN, V. An algorithm for parallel incremental compaction. In *Proc. of the Third International Symposium on Memory Management* (Berlin, Germany, June 2002), pp. 100–105.

[9] BOLLELLA, G., GOSLING, J., BROSGOL, B. M., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[10] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[11] CHENG, P. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnegie-Mellon Univ., Sept. 2001.

[12] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 125–136.

[13] CLICK, C., TENE, G., AND WOLF, M. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (New York, NY, USA, 2005), ACM Press, pp. 46–56.

[14] DETLEFS, D., FLOOD, C., HELLER, S., AND PRINTEZIS, T. Garbage-first garbage collection. In *International Symposium on Memory Management* (Oct. 2004).

[15] DOLIGEZ, D., AND GONTHIER, G. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conf. Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 70–83.

[16] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[17] FLOOD, C., DETLEFS, D., SHAVIT, N., AND ZHANG, C. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium* (Monterey, California, Apr. 2001).

[18] HARRIS, T. L. A pragmatic implementation of non-blocking linked lists. *LNCS 2180* (2001), 300–314.

[19] IBM CORP. TuningFork Visualization Tool for Real-Time Systems. URL `www.alphaworks.ibm.com/tech/tuningfork`.

[20] IBM CORP. WebSphere Real Time Java virtual machine. URL `www.ibm.com/software/webservers/realtime`, Aug. 2006.

[21] IBM CORP. *WebSphere Real-Time User's Guide*, first ed., 2006.

[22] IBM CORP. AND RAYTHEON INTEGRATED DEFENSE SYSTEMS. IBM and Raytheon deliver technology solution for DDG 1000 next generation navy destroyers. Press release, Feb. 2007.

[23] Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net.

[24] JONES, R., AND LINS, R. *Garbage Collection*. John Wiley and Sons, 1996.

[25] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[26] SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. MC^2: High-performance garbage collection for memory-constrained environments. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004).

[27] SIEBERT, F. Eliminating external fragmentation in a non-moving garbage collector for Java. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (San Jose, California, Nov. 2000), pp. 9–17.

[28] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[29] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.