Schism: Fragmentation-Tolerant Real-Time Garbage Collection

Filip Pizlo[†] Lukasz Ziarek[†] Petr Maj[‡] Antony L. Hosking[‡] Ethan Blanton[†] Jan Vitek[†,‡]

[†]Fiji Systems Inc., Indianapolis, IN 46202.
[‡]Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA
{fil,luke,elb}@fiji-systems.com {pmaj,hosking,jv}@cs.purdue.edu

Abstract

Managed languages such as Java and C# are being considered for use in hard real-time systems. A hurdle to their widespread adoption is the lack of garbage collection algorithms that offer predictable space-and-time performance in the face of fragmentation. We introduce SCHISM/CMR, a new concurrent and real-time garbage collector that is fragmentation tolerant and guarantees time-and-space worst-case bounds while providing good throughput. SCHISM/CMR combines mark-region collection of fragmented objects and arrays (arraylets) with separate replication-copying collection of immutable arraylet spines, so as to cope with external fragmentation when running in small heaps. We present an implementation of SCHISM/CMR in the Fiji VM, a high-performance Java virtual machine for mission-critical systems, along with a thorough experimental evaluation on a wide variety of architectures, including server-class and embedded systems. The results show that SCHISM/CMR tolerates fragmentation better than previous schemes, with a much more acceptable throughput penalty.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—dynamic storage management; D.3.4 [*Programming Languages*]: Processors—memory management (garbage collection); D.4.2 [*Operating Systems*]: Storage Management—garbage collection; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems; D.4.8 [*Operating Systems*]: Performance—measurements

General Terms Algorithms, Experimentation, Languages, Measurement, Performance, Reliability

Keywords fragmentation, real-time, mark-sweep, mark-region, replication-copying

1. Introduction

Real-time systems span application domains that range from financial systems to aerospace, each with its own domain-specific requirements and constraints. Nevertheless, they share common characteristics. Real-time systems are resource-constrained and long-lived; they must be extremely predictable and exhibit good throughput. Real-time developers emphasize knowing and limiting the worst-case execution time of code to manage predictability. Pressed to provide such strict guarantees for exponentially-increasing code bases, the real-time community is steadily moving towards higher-level programming languages. Managed languages, and in particular Java, have already been used in a handful of high-visibility projects with encouraging results. Developers report improved productivity and appreciate the benefits of automatic memory management. To gain widespread acceptance, however, managed languages must demonstrate predictability, in both time *and* space, that is comparable to handwritten low-level code.

Memory management is one of the key technical challenges. Real-time programmers are used to managing memory by hand, because they do not trust off-the-shelf memory allocators to be sufficiently predictable and because they fear that memory fragmentation will cause long running applications to eventually fail. To compete, managed languages must offer highly predictable garbage collection (GC) algorithms that preserve overall application throughput. It is crucial that the space bounds of GC be clearly established. Given a memory budget and knowledge of allocation patterns, developers must be confident that sufficient memory is available for the application to operate. For long running applications, GC must be able to cope with fragmentation without introducing severe overheads and restrictions on the application. Moreover, though most real-time applications still run on embedded uniprocessors, some are now moving towards careful use of multiprocessors. Thus, GC must also permit applications to scale to multiprocessors.

We argue that garbage collection *can* be used in all real-time applications, including safety-critical hard real-time systems that have stringent resource constraints. To cope with the requirements of such systems, we introduce a new real-time garbage collection (RTGC) algorithm called SCHISM/CMR that tolerates external fragmentation while providing good throughput and scalability on modern multi-cores. Table 1 illustrates the benefits of our algorithm. It is concurrent, meaning that it can operate concurrently with mutator threads (i.e., the application). This is essential for real-time applications, as high-priority tasks must be able to preempt *everything* in the system, including the garbage collector. While there are other RTGCs that operate concurrently, they typically do so at the expense of other guarantees. Consider the following properties that one takes for granted when using a standard Java virtual machine (JVM): (*i*) progress for heap accesses is guaranteed (i.e., they acquire no locks, and they never spin), (*ii*) a heap access never costs more than a handful of instructions, (*iii*) end-to-end throughput is good, and (*iv*) fragmentation is never an issue. SCHISM/CMR supports concurrency without sacrificing any of these properties. Prior approaches either impose an $O(\log(\textit{heap size}))$ cost on heap accesses, require the mutator to spin on some heap accesses, fail to handle fragmentation, or severely degrade overall throughput.

The main contribution of this work is an approach to allocation that can be thought of as *embracing* fragmentation by combining fragmented allocation with concurrent replication. We refer to this approach as Schism and its implementation in Fiji VM as SCHIS-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'10 June 5–10, Toronto, Ontario, Canada.

Copyright © 2010 ACM 978-1-4503-0019-3/10/06. . . $10.00

	Sun JDK	Jamaica	Java RTS	WebSphere SRT	Clover	Chicken	CMR	SCHISM/CMR
Concurrent	no	yes	yes	yes	yes	yes	yes	yes
Heap access progress	wait-free	wait-free	wait-free	wait-free	lock-free	wait-free	wait-free	wait-free
Heap access cost	O(1)	O(log(H))	O(log(H))	O(1)	O(1)	O(1)	O(1)	O(1)
Throughput relative to JDK	100%	*unknown*	37%	62%	*unknown*	*unknown*	84%	65%
Fragmentation tolerant	yes	yes	yes	no	yes	no	no	yes

Table 1: Comparing features of collectors. SCHISM/CMR is the only collector that supports all the features; other collectors either can not deal with fragmentation or increasing the cost of heap accesses. H is the heap size.

M/CMR. To quantify the effect of the Schism approach on throughput, we compare it with a state-of-the-art throughput-oriented concurrent mark-region (CMR) collector implemented in the same execution environment. We use two real-time benchmarks in this comparison (the CD_x collision detection algorithm and worst-case transaction times for SPECjbb2000). We also run in a mode where all slow-paths are taken to get an upper bound on worst-case execution time. We run these benchmarks on a multi-core server-class machine and on a representative embedded platform: the 40MHz LEON3 SPARC architecture used by the European Space Agency and NASA. We complement the real-time benchmarks with a synthetic benchmark designed to trigger worst-case fragmentation, plus the standard SPECjvm98 and SPECjbb2000 benchmarks for measuring throughput. To keep ourselves honest, we calibrate our execution environment against two production real-time JVMs, and best-of-breed throughput JVMs.

2. Schism and Fragmentation in RTGC

When fragmentation and space usage is not a concern, RTGC is reasonably well-understood. Unfortunately, without a solution for fragmentation, tight bounds on worst-case space utilization cannot be guaranteed. But space is just as important as time: real-time developers must provide bounds on both to ensure robustness. There are three broad approaches for handling fragmentation:

1. *Fragmented allocation:* Allocate objects in fragments of fixed size; larger objects span multiple fragments which may not be contiguous. A standard non-moving concurrent mark-sweep collector can then be used without concern for fragmentation.
2. *Replication:* Split the heap into semi-spaces, and have the concurrent collector copy live objects from one space to the other, even as mutator threads are operating on the heap. Mutator writes to the heap are captured by a write barrier that updates both the original and its replica, if one exists.
3. *Defragment on-demand:* When necessary, activate a separate defragmentation phase, requiring synchronization with the mutator. This may be combined with replication or other copying collection, but usually employs slower and more expensive techniques that try to make heap mutations atomic and lock-free.

Fragmented allocation is attractive as it can be combined with a simple non-moving collector and completely side-steps fragmentation. Implementations of this approach have had two main drawbacks [18]. Larger objects incur significantly higher overhead to access, and arrays are indexed using a trie data structure which, for a 32-bit address space, may be 10 levels deep, causing every array access to go through 10 indirections. The worst-case performance of this approach has not previously been comprehensively studied.

Replication, as proposed by Cheng and Blelloch [6], is compelling, especially for languages in which writes are very rare, or if writes need not be coherent. To ensure coherence of concurrent writes, expensive protocols (such as locking) must be used. Like any copying collector, replication entails a 2× space overhead.

On-demand defragmentation was introduced in the original Metronome [3]. That algorithm required stopping all mutator threads (though only briefly) during defragmentation. Stopping the world can cause a real-time thread to miss a hard deadline. We argue that high-priority tasks must be able to preempt the collector whenever necessary. Previously proposed techniques for concurrent on-demand defragmentation [11, 13, 14] either impede the progress of defragmentation (thus preventing space bounds) or impose prohibitive throughput overheads on the mutator, up to a 5× slowdown [14]. Defragmenting-on-demand approaches typically assume that fragmentation happens rarely and affects only a small fraction of the heap. While this is certainly true for typical programs, the approach degenerates in the worst case to copying the majority of the heap, resulting in 2× space overhead.

2.1 Schism: Fragmentation With a Touch of Replication

We propose combining fragmented allocation for objects and arrays with replication for array meta-data. In Schism, object and array fragments are always of fixed size and never move. A concurrent mark-sweep collector manages the non-moving fragments. A separate replicated semi-space manages array meta-data. This approach exploits the best of both worlds: (*i*) constant-time heap access, (*ii*) constant space-bounds for internal fragmentation with no external fragmentation, and (*iii*) coherence of heap accesses.

The key insight to getting constant-time heap access is to replace tries with *arraylets*. Arraylets represent arrays as a *spine* (a contiguous array) containing pointers to a set of array fragments. While objects can also be represented as arraylets, the Schism approach follows Siebert [18] and allocates these as linked lists of fixed-size fragments. This choice is motivated by the fact that most objects are small and are always of statically known size. Thus even with a linked representation it is possible to have a precise cost of any field access, and this cost does not vary.

In Schism, a large array is split into multiple arraylet fragments carrying the array payload. Arraylets have a worst-case performance that is roughly twice that of contiguous arrays. Whereas addressing an element in a contiguous array simply requires array + index × elementSize, with a fragmented array the address computation becomes:

```
offset = index * elementSize
fragmentIndex = offset / fragmentSize
fragmentOffset = offset % fragmentSize
address = spine[fragmentIndex] + fragmentOffset
```

Array payload fragments have fixed size and never move. Using a fixed (and small) size for array and object fragments ensures that external fragmentation is never an issue and internal fragmentation is bounded. However, the spines themselves vary with the size of the array so they may still cause fragmentation. In Schism the spines are managed by a separate replicating semi-space collector. Recall that a complication with concurrent replication is ensuring the coherence of writes to replicas. Luckily, this is not an issue for

spines as their contents (pointers to non-moving array fragments) are never updated. This allows us to use wait-free barriers for reading and writing the spine. Allocation of spines can be made lock-free. Moreover, the space overhead of copying is reduced since the copy reserve is only needed for the spines.

Schism yields a simple algorithm composed from well-understood building blocks: spine-indexed arraylets, fragmented allocation, concurrent mark-sweep collection of fixed-size fragments, and concurrent replicating semi-space collection of spines. The key insight is simply that the array spines can be copied concurrently without sacrificing mutator performance or coherence. Because of its simplicity, Schism can be implemented in a variety of ways — all that is needed is a concurrent semi-space framework for the immutable spines, and a concurrent mark-sweep framework for object and array fragments, both of which are well-understood. In this study we implement Schism for the existing concurrent mark-region collector in Fiji VM [15, 16]. In this scheme, fragmented allocation is only used when fragmentation is observed allowing for hard bounds with good throughput.

3. The Fiji VM Experimental Platform

Fiji VM is a new JVM that aims to provide developers with an automated tool for converting high-level, memory safe Java applications into small, efficient, and predictable executables for a wide range of embedded devices. This section introduces the characteristics of Fiji VM that are relevant to the implementation of SCHISM/CMR.

Fiji VM is an ahead-of-time compiler that transforms Java bytecode into fast ANSI C code. It runs on any platform that has a C compiler, threads, and locks. It does not require a memory management unit since null-checks and safepoints do not rely on protection-based mechanisms. Supported platforms include Linux, Darwin, NetBSD, RTEMS, x86, PowerPC, ERC32, LEON2, and LEON3. Both 32-bit and 64-bit architectures are supported; SCHISM/CMR does run in 64-bit mode though only 32-bit mode is examined in this paper. A noteworthy feature of Fiji VM is its ability to run on very restricted real-time embedded micro-kernels such as the Real Time Executive for Multiprocessor Systems (RTEMS).[1]

Fiji VM supports priority-aware locking and real-time priorities for threads, and takes care not to use any unpredictable operating system facilities, like OS-provided dynamic memory allocation, except inside library calls (e.g., creating a direct NIO byte buffer requires calling some variant of malloc). Additionally, Fiji VM employs a variety of techniques [15, 16] for ensuring that the generated C code obeys Java semantics, and so that accurate stack maps can be generated for GC stack scanning.

Fiji VM was designed specifically to support RTGC. At present, Fiji VM has no stop-the-world GC; it does not even have a stop-the-world capability. Instead we offer the following facilities to make the implementation of RTGCs simple:

Safepoints: A safepoint is a lightweight check to see if a thread should perform any task on behalf of the collector. Calls to native methods are "effective" safepoints, in that we mark the thread as running without JVM-access. This allows the collector to make changes to the thread's Java state without synchronizing with the thread. The compiler ensures that there is a worst-case bound on the time between safepoints.

Ragged safepoints: A ragged safepoint is an asynchronous request to all threads to perform an action exactly once. Unlike a stop-the-world barrier that stops all mutator threads, a ragged safepoint does not block any threads. Threads simply acknowledge having performed the desired action at the safepoint. Priority-boosting is used to ensure that threads perform the requested action in a timely fashion. Ragged safepoints are essential for implementing on-the-fly concurrent collectors.

The safepoint functionality is built around the observation that a high-priority task that always preempts the collector and only yields to it explicitly can have its roots scanned in a non-blocking fashion [17]. The result is that when running on a real-time operating system, the collector infrastructure never pauses the mutator for root scanning. Indeed, the only pauses that we have are: slow-path executions of barriers and allocations, and synchronous collections in the event that the collector is outpaced by the mutator. Whether the former is actually a pause is debatable, as it is only a call to a procedure that takes slightly longer than the inlined fast-path. However, we count this as a pause because it may disrupt the timeliness of a real-time task.

4. The Schism Concurrent Mark-Region RTGC

We now describe SCHISM/CMR, our implementation of the Schism approach on top of a *concurrent mark-region* (CMR) garbage collector. A CMR collector extends the mark-region collector of Blackburn and McKinley [4] to be concurrent (mutator and collector threads interleave heap accesses) and *on-the-fly* (no stop-the-world phase). We start with an overview of the base CMR collector.

4.1 Concurrent Mark-Region (CMR) GC

Mark-region garbage collectors like Immix [4] partition the heap into fixed-size regions that form the units of space management for a traditional mark-sweep garbage collector. The key insight of mark-region collectors is to allocate and reclaim memory in contiguous regions, at a coarse *page* granularity when possible, and otherwise at the level of fine-grained *lines*. Objects are allocated within and spanning lines. Marking notes the live lines holding live objects, and sweeping proceeds to discover entirely free pages, as well as the free lines within pages. Unlike Immix, CMR does not perform opportunistic defragmentation. CMR implements Immix's mark-region approach as follows.

Hybrid slack-based, concurrent, and time-based scheduling. The CMR collector can run concurrently on a separate processor core so as to minimize interference with mutator threads. In *slack-based* mode the collector runs at a priority that is *lower* than the mutator's critical real-time threads so it never preempts them. On uniprocessor systems, CMR's scheduling is identical to the slack-based Minuteman [10]; as such all of the schedulability tests and analytical results from Minuteman directly apply. CMR has a Java API for changing the priority of the collector thread allowing it to be run in a time-based mode like the Metronome. However, for the purposes of this study we run CMR in a purely slack-based mode.

Hybrid bump-pointer, best-fit, and first-fit allocation. The CMR collector allocates objects in bump-pointer fashion, similarly to Immix [4]. The sweep phase notes lines of contiguous free bytes within partially-occupied pages, as well as pages that are completely free. An allocation request finds the first page with a free line big enough to satisfy the request (first-fit), and then chooses the smallest line within that page from which to allocate (best-fit). Bump-pointer allocation proceeds within the last line and page from which allocation occurred, until a new line is needed. Bump-pointer allocation is used most of the time, given sufficient memory. In practice, most free memory is in the form of free pages, not free lines, but free line allocation is preferred — free pages are used only when free lines are exhausted.

The base CMR collector is concurrent, mostly lock-free, on-the-fly, exhibits throughput that is comparable to production generational collectors, and has very short pauses. In fact, the only pauses

[1] http://www.rtems.org

are due to stack scanning, which affects only low-priority threads. CMR's main limitation is its lack of a strategy for coping with fragmentation, since it does not perform opportunistic defragmentation.

4.2 Adding Schism to CMR

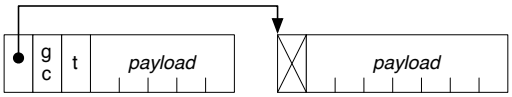
SCHISM/CMR applies the fragmented-allocation Schism approach to CMR. Both the collector, and the part of the compiler that deals with representation of objects (the object model) were modified to cope with fragmented allocation. The heap is split into two spaces: a CMR space and a pair of spine semi-spaces. The semi-spaces are 30% of the size of the CMR space. The allocator can allocate objects either contiguously (unfragmented) or fragmented into 32-byte fragments. If opportunistic contiguous array allocation fails, a 32-byte *array sentinel* fragment is allocated in the CMR space. The sentinel holds the array's object header and contains a pointer to the spine, which is allocated in the currently active spine semi-space. The spine is populated with pointers to 32-byte arraylet fragments in the CMR space. Every collection cycle completely evacuates the old semi-space, and populates the other with survivors. This process is entirely lock-free because the semi-space holds only array spines, which are immutable.

Array accesses have a guarded fast path that assumes that the array is contiguous, and a slow path for fragmented arrays that indirects via the spine to access the appropriate arraylet, incurring one indirection to get to the spine and one more indirection to get to the payload. All non-array object accesses require n hops, where n is the offset of the field divided by 32 bytes. As observed in [18], very few objects are larger than 64 bytes. It would be easy to convert the collector to use arraylets for large non-array objects, but we have not done this yet.

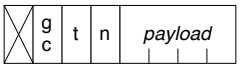
The following sections give a more detailed look inside SCHISM/CMR. Section 4.3 discusses our object model. Section 4.4 describes our replicating semi-space framework along with the barrier used. Opportunistic optimizations and the associated configuration parameters are shown in Section 4.5. SCHISM/CMR provides hard bounds on space usage; an overview of those bounds is given in Section 4.6 with further details in Appendix A. Section 5 gives a qualitative comparison to other RTGCs.

4.3 Fragmented Allocation

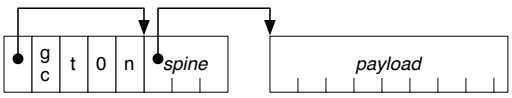
The structure of objects in SCHISM/CMR is shown in Figure 1. The first object fragment has three header words: a *fragmentation header* that either points to the next fragment of the object (Figure 1(a)) or to the arraylet spine (Figure 1(c) and 1(d)), and a *GC word* used for marking and a *type header* that holds both type and locking information. Arrays have an additional *pseudo-length* header holding either the actual array length (for contiguous arrays) or zero. Array accesses first perform an array bounds check on the pseudo-length; it will always fail for fragmented arrays causing the slow path to be taken. For normal objects, subsequent fragments only have a fragmentation header; the rest of the space is devoted to the payload. For arrays, the first fragment (the *sentinel*) may point to a spine or may have the entire array inlined (if it is small enough to fit in the remaining 16 bytes or if opportunistic contiguous array allocation succeeded). The *inline payload* case is shown in Figure 1(b). The spine itself may live inside the sentinel if it can fit in 16 bytes. In that case the spine uses one word for the length and the remaining 12 bytes for pointers to payload fragments (Figure 1(c)). If the array payload requires more than three fragments (i.e., is more than 96 bytes) then the spine will be allocated in the spine space (Figure 1(d)). In this case the sentinel has just four words in it and the remaining 16 bytes are wasted to achieve 32-byte alignment. An out-of-line spine allocated in the spine space requires a two word header: a *forwarding pointer* to support replication and the length.



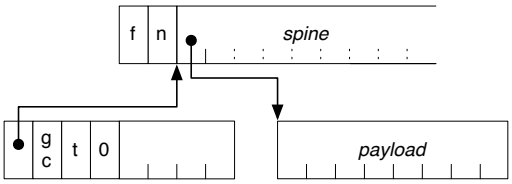
(a) A 2-fragment object. The first fragment has three header words: a fragmentation header, a GC header, and a type header.



(b) An array with $\leq$ 16-byte payload. The sentinel fragment has four header words: fragmentation header, GC header, type header, and pseudo-length. The payload is inlined.



(c) An array with a (17,96)-byte payload. The sentinel fragment has five header words: fragmentation header, GC header, type header, pseudo-length 0 to indicate fragmentation, and the length. The remainder of the sentinel contains an inlined spine.



(d) An array with a payload $>$ 96 bytes. The sentinel has four header words: fragmentation header, GC header, type header, and pseudo-length. The remainder of the sentinel is unused. The spine has a two-word header: the length and a forwarding pointer at negative offsets. The payloads have no headers.

Figure 1: Fragmented allocation in SCHISM/CMR.

4.4 Updating Pointers to Arraylet Spines

The SCHISM/CMR semi-space spine collector is similar to earlier replicating collectors [6, 12] with one major restriction: we do not have a strong *from-space invariant*. These replicating collectors preserve the invariant that pointers into from-space all originate from from-space. The mutator operates entirely in from-space until the "flip" occurs at the end of collection when the to-space and from-space roles are reversed. They never modify the from-space; they simply discard it atomically in a global stop-the-world *flip* phase. This has two attractive properties. First, a collection cycle requires only one trace of the heap. Second, the mutator never sees a to-space object before that object has been fully copied. This approach is difficult to incorporate into SCHISM/CMR, since the non-moving CMR space may have spine pointers. Even if SCHISM/CMR employed a global stop-the-world flip, we would still require some mechanism for ensuring that the CMR space's spine pointers were updated only after copying was finished. We considered a number of solutions to this problem. Performing a second trace of the CMR space after copying would solve the problem, but we feared this would increase collection times too much. We also considered doubling the size of the semi-spaces and alternating between copying and fixup (where pointers in CMR space are flipped to refer to the to-space). Unfortunately, this solution has the potential to increase space overhead and floating garbage. In the end, we chose to add an extra indirection on arraylet accesses by introducing an *arraylet sentinel* fragment in the CMR space as shown in Figure 1. This fragment holds all of the array's meta-data as well as a pointer

to the spine. The mutator never holds pointers to spines directly. The only objects in the CMR space that hold pointers to spines are the arraylet sentinels themselves. This leads to a simple, on-the-fly, and concurrent copying algorithm that has no global stop-the-world phase. Only one heap trace is required per collection cycle. Copying and fixup are performed in a separate phase, which only considers live sentinels. The steps of the algorithm are as follows.

1. *Mutator threads switch to allocating in to-space before tracing starts.*
2. *The CMR trace loop allocates to-space spines but defers copying.* When a sentinel is encountered, a spine is allocated and zeroed in to-space. Copying is deferred by placing the sentinel on a *live sentinel* list. A forwarding pointer is installed in the from-space spine.
3. *After tracing finishes, mutator threads perform a ragged safepoint to acknowledge the installation of forwarding pointers in from-space spines.* On-going array allocations will start writing arraylet pointers to both from- and to-space spines.
4. *Spines are copied.* The collector processes the live sentinel list built during the tracing phase. The copy algorithm ensures that it does not corrupt spines that are being initialized. To do this, it relies on properties of the write barrier used for array initialization:

```
oldSpine[index] = fragmentPointer
STORE_FENCE()
oldSpine.forward[index] = fragmentPointer
```

   This initializes the from-space spine first. The spine copy loop exploits this assumption:

```
oldSpine = sentinel.spine
newSpine = oldSpine.forward
for (i = 0 ; i < spineLength ; ++i)
    if (oldSpine[i] != null)
        newSpine[i] = oldSpine[i]
sentinel.spine = newSpine
```

   The from-space spine can be null or have a value. In the latter case, it will never change again. If an entry is null, it has not yet been initialized by the mutator; we also know that the mutator has yet to store anything to the to-space copy, thanks to the use of the store fence. The mutator is guaranteed to initialize the entry at some point in the future, and the replicating barrier ensures that both the from-space and to-space will be initialized. If the entry has a value, the mutator may not have stored the same value to the to-space, but when it does the value it stores will be identical. Thus, copying the value into to-space is both necessary (if the initialization of this entry happened in the past) and harmless (since at worst we will write the same value that the mutator writes). After copying finishes, the sentinel is updated to point to the to-space.
5. *The mutator threads perform another ragged safepoint to acknowledge that sentinels have been updated to to-space, so they no longer access the from-space.*
6. *The from-space is zeroed.*

4.5 Predictability Levels

Contiguous objects lead to better performance than fragmented ones. Of course, we have designed SCHISM/CMR to have good performance even if all objects are fragmented — but even in a real-time system an opportunistic throughput boost can be a good thing. Thus, SCHISM/CMR has multiple "predictability levels" which vary the heuristics for opportunistic contiguous allocation.

Predictability level C: optimize for throughput. The collector tries to allocate objects contiguously, reverting to fragmented allocation if the former fails. Field access barriers do not exploit contiguity; they always perform n hops to get to the nth object fragment. Thus, contiguity of plain objects is used solely for accelerating allocation and locality. Array access barriers are branch-predicted in favor of contiguous access, but fragmented access is still inlined to ensure good worst-case performance.

Predictability level A: optimize for predictability. Arrays are always allocated in 32-byte fragments. An array will only be contiguous if its payload is 16 bytes or smaller. Objects are allocated opportunistically contiguous as in level C. Array access barriers are branch-predicted in favor of fragmented access.

Predictability level CW: simulate worst-case. This is the worst-case execution time mode. It behaves as in level C, except that all fast paths are *poisoned*. They execute but always fail causing the mutator to exercise the out of line slow-paths. CW poisons array accesses, GC write barriers, and allocations. This mode helps users estimate how slowly the program would run if *all* of the collector's heuristics fail. Note however that CW does not trigger worst-case space usage as some contiguous arrays may require more memory than fragmented ones.

4.6 Space Bounds: Predictability Level A

We now consider bounds for the memory used by SCHISM/CMR. The collector ensures that no object will use more memory than specified by these formulas regardless of heap structure or level of fragmentation. We focus on Level A; the precise formulas as well as a discussion of level C appears in the appendix. For simplicity, we assume a 32-bit architecture with a 4096-byte page size. Similar formulas can be derived for 64-bit architectures and different page sizes.

Bounding GC space usage is important. Many real-time systems are certified empirically but the *qualitative justification* of those techniques relies on analytical results. Because space usage is completely deterministic in predictability level A, we suspect that in many settings the actual analysis of the space usage of a program will be done empirically so long as level A is used for both analysis and deployment. Regardless of predictability level the formulas can be used for an analytical proof that the collector is correctly configured, since the analyses used for proving the schedulability of a time-based or slack-based collector will need to know the precise number of bytes used by each object [10].

We provide separate formulas for plain objects and arrays. We account for the worst case assuming an adversarial program. All of the collector meta-data and space to hold spines is accounted for. Thus, it is possible to bound the memory footprint of the entire JVM by adding the size of the .text and .data segments and an OS-specific constant for each thread to the results from this section.

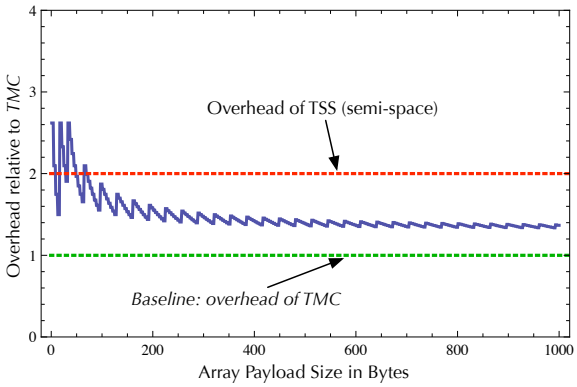
The equations in the appendix give us the following formula for computing the size in bytes of a plain object with n fields:

$$1.3104 \times 32 \lceil (2+n)/7 \rceil \tag{1}$$

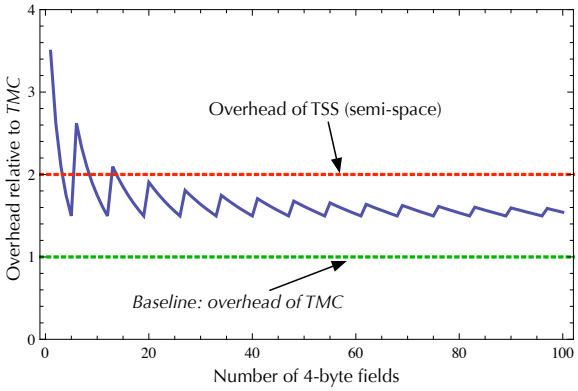
This is the exact amount of memory used by the JVM for that object, under the simplifying assumption that fields are homogeneously 4-bytes long (the appendix gives the general case that accounts for alignment). The 1.3104 coefficient accounts for the spine-space reserve and page table meta-data. For an array with a p-byte payload, the formula is

$$\text{if } p \leq 16 \text{ then } 1.3104 \times 32 \text{ else } 1.3104 \times (32 + 32 \lceil p/32 \rceil) \tag{2}$$

This accounts for the different allocation modes of arrays and is exact at level A.



(a) Arrays: SCHISM/CMR converges at an overhead of 1.4× TMC and peaks at 2.6× for small arrays.



(b) Objects: SCHISM/CMR converges at an overhead of 1.5× TMC and peaks at 3.5× for small objects.

Figure 2: Analytical Overheads. Memory overhead of SCHISM/CMR for data of varying size compared to theoretical mark-and-compact and semi-space collectors (relative to TMC).

Analytical Comparison. To illustrate the effect of space overheads, we analytically compare SCHISM/CMR's guaranteed worst-case to that of two *theoretical* baseline collectors: *TMC*, a three-phase stop-the-world mark-compact framework that requires no external meta-data, packs all object fields ignoring alignment, uses two header words for objects and three for arrays, and maintains 4-byte alignment for objects;[2] and *TSS*, a stop-the-world semi-space framework that uses an identical object model to *TMC* but requires twice as much space due to its copy reserve. The *TMC* and *TSS* object layouts are similar to what is found in Jikes RVM and Sun HotSpot. Thus, when meta-data overheads are factored in we expect those systems to use slightly more memory than *TMC* but considerably less than *TSS*. The exact overheads of objects and arrays are shown in Figure 2; the plot is relative to the TMC size and is specific to level A. For arrays, the payload size is set to range between 1 and 1000 bytes. For objects, the number of fields ranges between 1 and 100. The graphs show that SCHISM/CMR's overheads converge to roughly 1.4× TMC for large arrays, and 1.5× for large objects. The overheads peak at roughly 2.6× for small arrays and 3.5× for objects that have only one field.

[2] For comparison, the space requirements of TMC for an object with n 4-byte fields is $8+4n$, and for an array with a p-byte payload TMC requires $12+4\lceil p/4 \rceil$ bytes.

5. State of the Art in RTGC

Commercially available virtual machines with real-time garbage collectors include the IBM WebSphere Realtime [1, 2], Sun Java RTS [5], Aicas Jamaica [18], Aonix's PERC, and the Azul JVM. There are also a number of academic RTGC projects including the original Jikes RVM-based uniprocessor version of Metronome [3], a suite of RTGCs for .NET [13, 14], Minuteman [10], Sapphire [8], and a parallel real-time collector for ML [6]. RTGCs differ in two main regards: first, how collection work is scheduled, and second, what object model is used to deal with fragmentation.

5.1 Scheduling Strategies

RTGCs use two different scheduling strategies: either the user chooses the collector's schedule with time-based or slack-based scheduling, or it automatically adapts to the allocation rate with work-based scheduling. SCHISM/CMR, WebSphere SRT, and Java RTS use the former while Jamaica uses the latter.

Time-based and slack-based scheduling. Time-based scheduling, pioneered in [3], runs the collector periodically for short tightly-bounded increments. This yields uniform mutator utilization and is easy to understand and configure. However, it is not ideal for low-latency hard real-time tasks — for such tasks it is better to avoid all collector interference. This is achieved by slack-based scheduling where the collector runs at a fixed priority that is lower than that of real-time tasks, which can always preempt the collector. Slack-based scheduling by itself will not work in every application: if real-time tasks run for too long the collector will starve. Neither scheme is ideal, so most JVMs support both approaches. Websphere includes an innovative scheduling scheme [2], which invokes the time-based collector only when the slack-based one is starved. Java RTS and SCHISM/CMR are slack-based by default but provide APIs for controlling the collector thread directly, which allows for time-based scheduling to be enabled if needed.

These schemes may fail if the mutator outpaces the collector by allocating too much too quickly. In this case the RTGC may have to suspend allocating threads. Ensuring that this does not happen is up to the developer, and requires computing the worst-case (i.e., highest) allocation rate, worst-case (i.e., lowest) collection rate, and object lifetimes. Given this information it is possible to determine the optimal heap size and mutator utilization target (for time-based collectors) or heap size and thread priority (for slack-based collectors) [10].

Work-based scheduling. A work-based collector will perform an increment of collection work on every allocation [18]. This scheme can be made to provide linear-time allocation, though with a larger constant factor than other schemes. Thus, work-based scheduling punishes threads that allocate a lot but rewards those that allocate little, and can be thought of as precisely matching the collection rate to the allocation rate. The main advantage of work-based scheduling is that it is easier to configure the collector: only a heap size needs to be picked. The collector can then perform exactly enough work on each allocation to ensure that a full collection cycle completes before memory is exhausted. However, this scheme requires that collector work can be uniformly partitioned into very short bursts, and that these bursts can be efficiently distributed among all threads in the system.

Concurrent collection. A concurrent collector that never offloads collector work to application threads can be used in a real-time environment without any modifications, provided that spare CPUs are available. This is the primary mechanism used in the Azul JVM [7]. The other RTGCs all support concurrent scheduling and will use that as their primary scheduling facility if unused CPU cores are available. As with time-based scheduling, a concurrent schedule

Real-time Java Virtual Machines:

WebSphere SRT	-Xgcpolicy:metronome (build 2.5, J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi3260srt-20081016_24573 (JIT and AOT enabled))
Java RTS	Java 2 Runtime Environment, Standard Edition (build 1.5.0_16_Java-RTS-2.1_fcs-b11_RTSJ-1.0.2) Java Real-Time System HotSpot Client (build 1.5.0_16-b11, mixed mode)
Fiji VM	0.0.3-r1206f3ecc7c2

Desktop Java Virtual Machines:

IBM J9	IBM J9 (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20080816_22093 (JIT and AOT enabled)
Sun JDK	Java SE Runtime Environment (build 1.6.0_12-b04) Java HotSpot Server (build 11.2-b01, mixed mode)

All JVMs were run with the options "-Xmx50M, -Xms50M" unless otherwise indicated.

Platforms:

Sharpay	Intel Zeon CPU X5460, 3.16Ghz, 8-core, 8GB of RAM. Ubuntu 7.10 Linux kernel 2.6.22-14-server.
LEON3	Gaisler GR-XC3S-1500 / Xilinx Spartan3-1500 FPGA flashed with a LEON3 configuration running at 40Mhz, 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. RTEMS 4.9.2 as the operating system.

Table 2: **Experimental Setup.**

requires knowing the collection rate, allocation rate, and object lifetimes, as well as a schedulability test, to choose a configuration that does not result in the collector being outpaced.

5.2 Object Model and Fragmentation

Except for Azul, all other real-time garbage collectors may fragment objects. WebSphere SRT fragments large arrays using arraylets. Java RTS and Jamaica may fragment any object; non-array objects may become linked lists and arrays become tries. The original Metronome [3] used on-demand defragmentation in addition to arraylets to handle fragmentation.

Even without concurrent copying, WebSphere SRT will tend to perform well for most programs thanks to its use of segregated free-list allocation and arraylets for large arrays — however, it is not completely fragmentation-tolerant and thus cannot bound space usage as aggressively as SCHISM/CMR. Like SCHISM/CMR, Jamaica and Java RTS are fragmentation-tolerant but have a worst-case array access cost of $O(log(H))$. Azul copies objects concurrently and achieves complete fragmentation tolerance; it can do so efficiently thanks to specialized hardware. Concurrent object copying requires a copy reserve. SCHISM/CMR needs only a very small copy reserve but has large (though predictable) per-object overheads, while Azul may in the worst case need a 100% copy reserve but has extremely compact objects (to our knowledge, it is the only JVM that uses one-word headers). Overall, we expect that Azul is more space-efficient than SCHISM/CMR for small objects and slightly less space-efficient for large ones. It is the only RTGC that has demonstrated scalability to hundreds of cores.

There has been extensive work in the literature on real-time garbage collection. The Cheng and Blelloch [6] collector was one of the first to offer hard real-time bounds on multi-processors. SCHISM/CMR's use of replication is largely inspired from that work's emphasis on immutability. One way to view SCHISM/CMR is that it achieves immutability in Java by "boxing" the payload and storing it in a non-moving space. Sapphire [8] is another attempt to bring replication to Java, though at the cost of some object access coherence. Unlike SCHISM/CMR, both Cheng-Blelloch and Sapphire may have to resort to locking for some object accesses if both coherence and mutability are required. Minuteman [10] is an open-source uniprocessor implementation of the Metronome segregated free-list mark-sweep collector complete with on-demand defragmentation. It can be made to use either pure time-based or pure slack-based scheduling allowing the two styles to be compared directly. Stopless, Chicken, and Clover are real-time garbage collectors for .NET [13, 14]. These collectors enable concurrent copying of objects on multiprocessors, though with higher worst-case costs than SCHISM/CMR.

6. Evaluation

This section aims to demonstrate that SCHISM/CMR can handle fragmentation (Section 6.1), has competitive throughput (Section 6.2), delivers on predictability (Section 6.3), and is able to scale (Section 6.4). To demonstrate these properties in a convincing manner, we have selected a number of benchmark programs, architectures and operating systems, and Java implementations. This broad range of experiment yields the most thorough comparison of real-time Java virtual machines to date.

Our experimental setup is summarized in Table 2. We evaluate three real-time virtual machine configurations: IBM WebSphere SRT, Sun Java RTS, and Fiji VM. WebSphere SRT is IBM's soft real-time product based on the latest variant of the Metronome. The hard real-time version of WebSphere (WRT) adds support for scoped memory and is usually substantially slower than SRT. Java RTS is a production real-time JVM with a memory management strategy that bears some similarities to Fiji VM. It uses the HotSpot client compiler. For Fiji VM, we evaluate four configurations: CMR, the base concurrent mark-region algorithm; and three predictability levels of SCHISM/CMR (C=highest throughput, A=most predictable, CW=worst-case). For the purpose of establishing a baseline on throughput, we also evaluate two JVMs that are optimized for throughput rather than for predictability. These are IBM's J9 and Sun's JDK 1.6 (HotSpot Server).

We selected two platforms for our measurements. The first (Sharpay) is a powerful server machine that we use to explore the throughput of SCHISM/CMR on a modern multi-core architecture. The second platform is a LEON3 with the RTEMS hard-real-time operating system. This single-core platform is more representative of current embedded systems. In fact, it was selected because it is used by NASA and the European Space Agency in aerospace applications.

6.1 Fragmentation

We evaluate the ability of various GCs to deal with fragmentation using a synthetic benchmark (Fragger). Fragger maximizes fragmentation by allocating small arrays until memory is exhausted, then freeing every other array. Fragger then tries to allocate as many large arrays as possible. The benchmark is run three times for four sizes of arrays (small arrays range between 200 bytes and 88KB, large from 600 to 168KB). GCs that are able to deal with fragmentation, either through relocation or fragmented allocation, can allocate all of the large arrays.

Table 3 reports the number of arrays successfully allocated and the approximate free memory utilization. Approximate free memory utilization is a measure of fragmentation tolerance; higher numbers are better. This column does not account for object layout or any meta-data overheads; thus getting exactly 100% is unlikely. As

Configuration	# Small Arrays Initially Allocated	Payload Size for Small/Large arrays	Large Arrays Allocated	Approximate Free Memory Utilization
CMR	339847	200/600	0	*0.0%*
	58290	1024/3072	0	*0.0%*
	6516	10240/30720	0	*0.0%*
	889	88064/168960	0	*0.0%*
Schicm/cmr level C	186862	200/600	32733	*105.1%*
	41305	1024/3072	7026	*102.1%*
	3608	10240/30720	715	*118.9%*
	492	88064/168960	130	*105.7%*
Schicm/cmr level A	163498	200/600	32699	*120.0%*
	41275	1024/3072	7021	*102.1%*
	4280	10240/30720	714	*100.1%*
	499	88064/168960	130	*104.2%*
Schicm/cmr level CW	163498	200/600	32699	*120.0%*
	41275	1024/3072	7021	*102.1%*
	4280	10240/30720	714	*100.1%*
	499	88064/168960	130	*104.2%*
Sun RTS	290978	200/600	34170	*70.5%*
	64394	1024/3072	9053	*84.4%*
	6667	10240/30720	970	*87.3%*
	777	88064/168960	201	*103.5%*
IBM Metronome	255006	200/600	95	*0.2%*
	58998	1024/3072	108	*1.1%*
	6450	10240/30720	710	*66.0%*
	750	88064/168960	195	*104.0%*
HotSpot 1.6	307073	200/600	53837	*105.2%*
	65859	1024/3072	11090	*101.0%*
	6724	10240/30720	1121	*100.0%*
	782	88064/168960	203	*103.8%*

Table 3: **Fragger results.** Percentage of memory different JVMs are able to reuse when the heap becomes fragmented due to either relocation or fragmented allocation. SCHISM/CMR performs as well as JDK. Java RTS performs almost as well, but WebSphere SRT performs poorly except for large arrays.

Small Array Size	Large Array Size	# of bytes used by one Small Array	# of bytes used by all Small Arrays allocated	# of bytes used after half are freed	# of bytes used for a single large array	# large arrays it should be possible to allocate	# of large arrays actually allocated
200	**600**	335.46	54847431.48	27423715.74	838.66	32699	32699
1024	**3072**	1383.78	57115618.56	28558501.17	4067.48	7021	7021
10240	**30720**	13460.43	57610635.26	28805317.63	40297.42	714	714
88064	**168960**	115441.00	57605058.20	28860249.60	221447.12	130	130

Table 4: **Analytical vs. Observed.** Comparing analytical results for fragger using memory usage formulas and the empirical results for SCHISM/CMR level A. They correspond exactly: SCHISM/CMR level A can never allocate more or less arrays than predicted.

CMR is non-compacting, it will not be able to handle fragmentation at all. At the opposite end of the spectrum JDK behaves well as it has a GC that is free to stop the world and relocate objects at will. The different predictability levels of SCHISM/CMR perfectly handle fragmentation. Level C has slightly fewer space overheads due to its ability to allocate contiguously in some cases. Java RTS is close to SCHISM/CMR while WebSphere SRT performs poorly except for large arrays, for which it is able to use arraylets.

Table 4 compares the analytical memory usage model of SCHISM/CMR level A from Section 4.6 to the observed values. The numbers match up exactly, confirming the tightness of the space bounds. This comparison further illustrates the effect shown in Figure 2: large arrays have lower per-element overheads than smaller ones; this is the reason why switching from a 200 byte payload to a 600 byte one results in 120% utilization.

Figure 3 shows the average access time of random array elements. The graph has solid lines for accesses before fragmentation occurs and dashed lines for accesses after allocating in fragmented memory. This is an indication of execution time costs incurred by

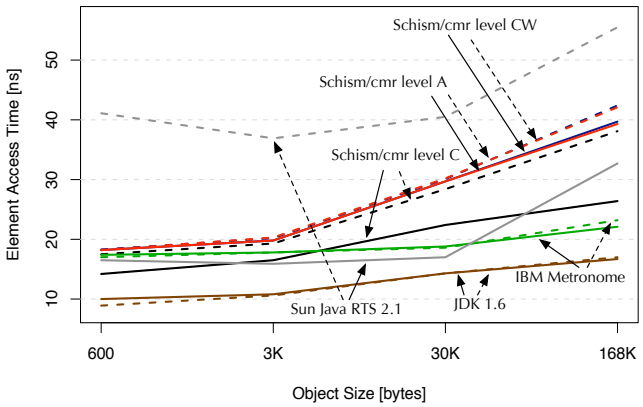


Figure 3: **Performance of fragmented array accesses.** Solid lines depict access cost prior to large array allocation and dotted lines after. Since JDK does not fragment, performance is identical. Java RTS, SCHISM/CMR, and WebSphere SRT all fragment arrays. WebSphere SRT performs the best out of the real-time collectors, while Java RTS (which uses tries) has the most extreme worst-case.

fragmentation. We can observe that JDK, which never fragments objects but can defragment the heap through stop-the-world object copying, has consistently faster access times that the other JVMs. Java RTS has good performance before fragmentation, but exhibits the worst performance once memory is fragmented due to its use of tries instead of arraylets. Java RTS improves average case access times through caching [5], but Fragger randomizes accesses to force worst-case behavior. WebSphere SRT performs just as well with fragmentation as without. SCHISM/CMR at all predictability levels provides reasonable performance but is somewhat more efficient when arrays are not fragmented.

6.2 Throughput

To evaluate the impact of SCHISM/CMR on throughput we compare performance of the different configurations on the SPECjvm98 benchmark.[3] We run SPECjvm98 experiments as follows. We invoke each JVM three times, running the given benchmark for seven iterations, and averaging the last three iterations. Those averages are again averaged. This gives a four-iteration warm-up as necessary for the JIT-based JVMs to reach steady state. We note that for individual benchmarks the execution time differences between the non-real-time JVMs (JDK and J9) and any of the other JVMs are statistically significant and almost always quite large. For brevity, our throughput overview focuses on a geometric mean comparison that takes into account all SPEC benchmarks at once. We are not aware of a statistically sound formulation of confidence intervals for the geometric mean over averages of non-independent benchmarks; thus we avoid using confidence intervals. The differences between configurations are very pronounced (typically exceeding 10%) and easy to reproduce.

Figure 4 shows a summary of SPECjvm98 performance. The two desktop JVMs are the fastest (JDK and J9). SCHISM/CMR level C runs at 65% of JDK's throughput. SCHISM/CMR level C appears to be faster than the other two commercial real-time JVMs. The figure also shows that there is approximately a 20% difference in performance between level C and level CW, which forces all fast

[3] While there are benchmarks that are more revealing than SPECjvm98, it would be difficult to run them on an embedded JVM. Fiji VM's library is tailored for JavaME applications and lacks some of the class libraries needed by larger benchmark suites. In fact, running SPECjvm98 itself requires quite a few dedicated extensions.

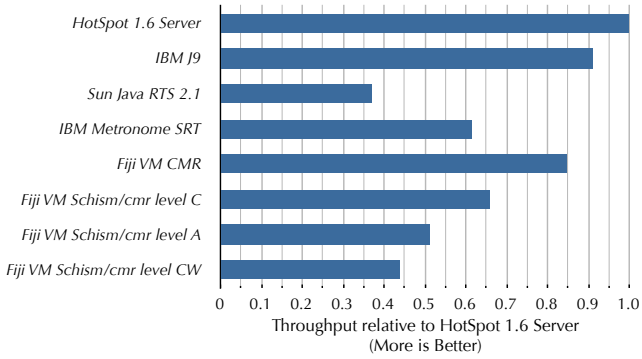
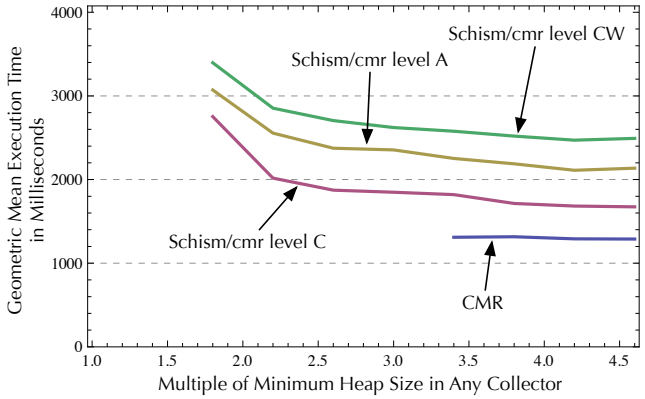


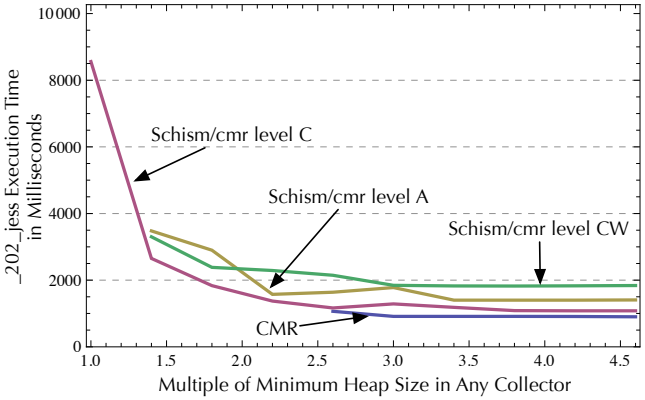
Figure 4: **SPECjvm98 Throughput.** Fiji VM with CMR runs at roughly 84% throughput relative to JDK, and SCHISM/CMR at 65%. Both appear to be faster than other real-time Java products.

	CMR min heap size (kb)	CMR max live size (kb)	CMR wastage (relative to heap size)
_201_compress	6800.000	6288.922	0.075
_202_jess	3300.000	1197.691	0.637
_209_db	9700.000	9545.891	0.016
_213_javac	16400.000	8002.316	0.512
_222_mpegaudio	300.000	257.871	0.140
_227_mtrt	8200.000	6993.922	0.147
_228_jack	1800.000	690.480	0.616
AVERAGE	6642.857	4711.013	0.306
Relative to CMR	1.000	1.000	1.000
	Schism/cmr level C min heap size (kb)	**Schism/cmr level C max live size (kb)**	**Schism/cmr level C wastage**
_201_compress	8800.000	8767.363	0.004
_202_jess	1900.000	1895.806	0.002
_209_db	16900.000	16692.447	0.012
_213_javac	14800.000	14396.647	0.027
_222_mpegaudio	400.000	391.666	0.021
_227_mtrt	12200.000	11179.350	0.084
_228_jack	1200.000	1156.716	0.036
AVERAGE	8028.571	7782.856	0.027
Relative to CMR	1.209	1.652	0.087
	Schism/cmr level A min heap size (kb)	**Schism/cmr level A max live size (kb)**	**Schism/cmr level A wastage**
_201_compress	8800.000	8606.325	0.022
_202_jess	2100.000	2001.675	0.047
_209_db	17800.000	16358.469	0.081
_213_javac	14900.000	14553.703	0.023
_222_mpegaudio	400.000	375.781	0.061
_227_mtrt	12400.000	12092.356	0.025
_228_jack	1300.000	1238.494	0.047
AVERAGE	8242.857	7889.543	0.044
Relative to CMR	1.241	1.675	0.143

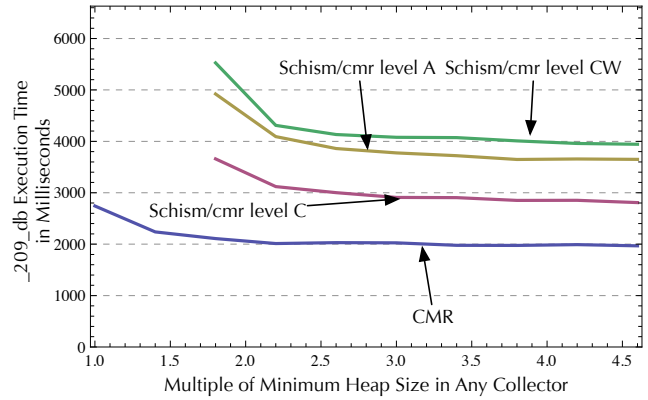
Table 5: **External Fragmentation.** On average, SCHISM/CMR requires roughly 65% more memory than in CMR. However, the average minimum heap size in SCHISM/CMR is only 20% (for level C) or 24% (for level A) larger than in CMR. This is because in CMR roughly 30% of the heap is wasted due to fragmentation and other overheads. SCHISM/CMR has almost no heap wastage.



(a) Geometric mean performance of SPECjvm98. SCHISM/CMR is slightly slower than CMR, but runs in smaller heaps thanks to its ability to tolerate fragmentation.



(b) _202_jess benchmark. CMR outperforms SCHISM/CMR, slightly, but requires larger heaps to run.



(c) _209_db benchmark. CMR runs in a smaller heap because the program does not cause fragmentation and SCHISM/CMR has larger objects.

Figure 5: **Time-memory Curves.** The *x*-axis is a benchmark-dependent multiple of the minimum heap size which is measured as the minimum for all available collector configurations. The *y*-axis is the execution time.

paths to fail. It should be noted that there are reasons to take these numbers with a grain of salt. For instance, all the JVMs were given the same heap size (50MB), but it is unclear how that number is used. Some JVMs account for their meta-data separately from the heap. Fiji VM accounts for all of the meta-data as part of the heap size. Moreover, the JVMs have very different compilation strategies and optimizations. Our argument here is simply that the performance of SCHISM/CMR is competitive.

Figure 5 focuses on Fiji VM and gives time-memory curves. The curves show the execution time of benchmark programs for various heap sizes. The x-axis in these graphs represents multiples of the minimum heap size needed to run the benchmark in any of the Fiji VM collectors. If a curve does not reach 1, it means that at least one run of the benchmark failed at that heap size multiple. Figure 5(a) gives the geometric mean for the entire SPECjvm98 suite. The results clearly show that SCHISM/CMR can run in less memory than CMR (which starts at a 3.3 multiple), illustrating that fragmentation matters even in SPECjvm98.

To better explore the effects of the different collectors, we show details for two benchmarks with particularly extreme behavior: _202_jess (Figure 5(a)) and _209_db (Figure 5(b)). The conclusion that can be reached from these outliers is that some benchmarks run better in CMR, while others run better in SCHISM/CMR. For example _202_jess and _213_javac run in smaller heap sizes in SCHISM/CMR because they can fragment a small heap quite rapidly in a non-moving collector. While SCHISM/CMR can often run in a smaller heap, this is not always the case. _209_db seems to generate no fragmentation, but uses a lot of small objects (we witnessed, for example, a large number of Enumerations that are less than 16 bytes). For small objects, SCHISM/CMR has enough of a size overhead that it can, and in this case does, outweigh the benefits of fragmentation tolerance.

Table 5 reports data for external fragmentation of the different Fiji VM collectors for SPECjvm98. The minimum heap size was obtained by running each benchmark with increasing heap sizes (in 100KB steps) until the program was able to run. Then, running at the minimum heap size, we record the total memory used by live objects at each collection. The maximum is the *maximum live size*. This does not count external fragmentation but does include all meta-data as well as internal fragmentation. The external fragmentation is reported as the wasted space ("wastage"): the difference between maximum live size and minimum heap size, scaled by the heap size. For some benchmarks, CMR exhibits $> 50\%$ wastage. For example, _202_jess has the largest wastage (63.7%), which explains why SCHISM/CMR allows for smaller heap sizes than CMR. SCHISM/CMR on average requires a 20% (for level C) or 24% (for level A) larger heap size to run. Note that according to Table 5, a benchmark only runs in a smaller heap size in SCHISM/CMR if it exhibits high wastage (>50%). The reason why wastage in SCHISM/CMR is not 0% is that both our minimum heap size and our maximum live size measurements are imprecise: minimum heap size may be off by nearly 100KB, and the maximum live size is not measured on every allocation but only when the collector runs in response to heap exhaustion. This figure also shows the *typical* object size overheads of SCHISM/CMR: 65% for level C and 67% for level A. This is substantially better than the predicted worst case memory usage (i.e., the prohibitive 3.5× overhead of allocating very small objects) overheads as computed in Section 4.6. These results lead us to two conclusions. First, if *typical* memory usage is of the utmost concern, CMR will tend to outperform SCHISM/CMR. On average it will run in a 20% smaller heap. But SCHISM/CMR allows for smaller heaps in some pathological programs and always provides a hard bound on space usage.

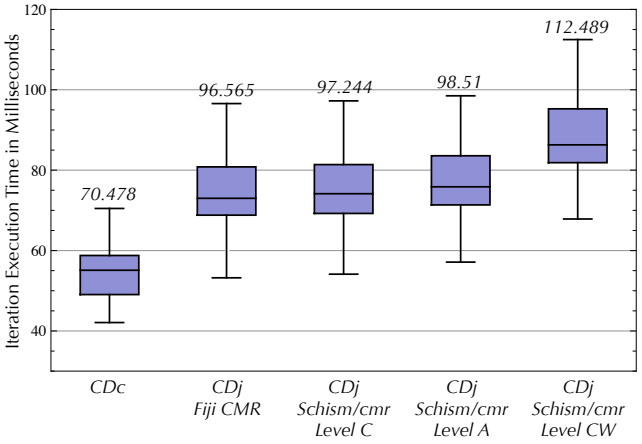


Figure 6: **Execution time of CD$_x$ compared to C.** Boxes represent the middle 50% population of data samples; the middle line is the median. The top and bottom "whiskers" represent the maximum and minimum, respectively. SCHISM/CMR level A performs just 40% worse than C. We are unaware of *any* results in the literature that show an RTGC performing this well on a non-trivial workload.

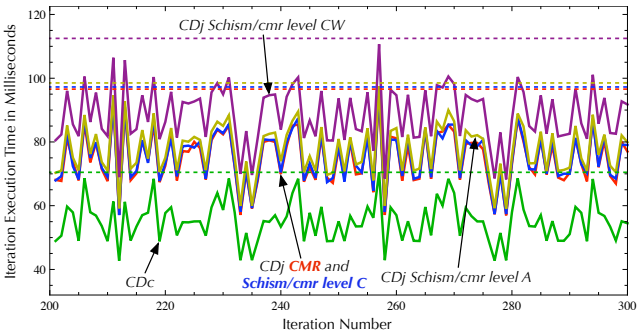


Figure 7: **Execution time of CD$_x$ compared to C.** A detailed view of a subset of execution of CD$_x$. CMR, SCHISM/CMR level A, and SCHISM/CMR level C perform almost identically. It is only SCHISM/CMR level CW that shows a performance degradation.

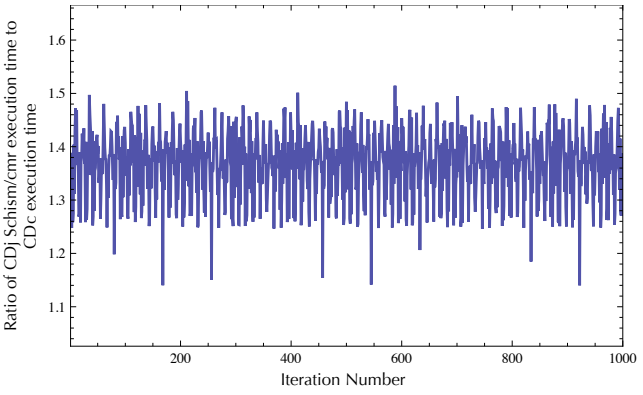


Figure 8: **CD$_x$ Ratio.** The ratio of runtime performance of SCHISM/CMR level C to C. Each data point represents the same iteration executed in C and Java.

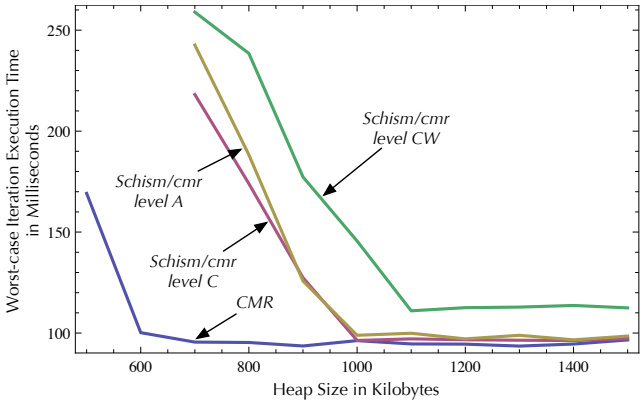


Figure 9: **Worst-case execution time as a function of heap size.** SCHISM/CMR degrades sooner, implying that this benchmark does not experience fragmentation. Levels A and C have similar performance on larger heaps, but A performs worse for small heaps because it uses more memory.

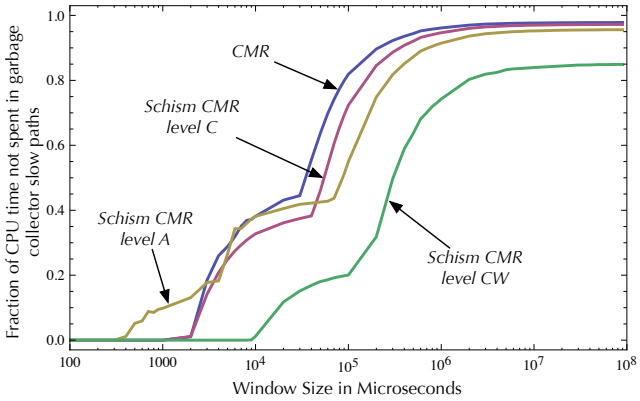


Figure 10: **Minimum mutator utilization for CD_x on LEON3.** SCHISM/CMR level A has 400 microsecond pauses, CMR and level C have roughly millisecond-level pauses, and level CW pauses for 9 milliseconds in the worst case.

6.3 Predictability

To evaluate predictability we switch to the LEON3 platform and select a representative real-time benchmark, the well-known open-source Collision Detector (CD_x) [9]. We use C as our baseline, as it is the language of choice for real-time developers.

CD_x is an idealized air traffic collision detection algorithm that iteratively attempts to detect potential collisions based on simulated radar frames. It utilizes many arrays and performs significant mathematical computations, making it ideally suited to low-level C programming idioms — but it has been deliberately implemented using Java idioms even when they may be slow. For example, CD_x uses classes in `java.util` extensively, and makes no effort to resize, preallocate, or pool any objects. We configure CD_x with a single real-time periodic task running with a period of 120 milliseconds. We have also implemented a version of the collision detector in C. The implementation is idiomatic C that tries to follow the algorithmic behavior of the Java code with some differences. For instance, the hash table used by the C code requires almost no allocation and has constant time traversal. The code size of the Java version of CD_x used in this experiment is 3859 LoC and the C version is 3371. (The C version is somewhat simpler since it does not have hooks for the multiple configurations supported by the Java version). All versions of CD_x were run for 1,000 iterations. Note that on LEON3, execution is fully deterministic: though we ran the benchmarks multiple times for sanity, we observe that for iteration i in a particular configuration, the execution time is always identical. Figure 6 compares the performance of Java and C. Java's performance is only 40% worse than that of C for SCHISM/CMR level A. For level C, the performance is 38% worse, and for CMR, the performance is 37% worse. Figure 7 shows a "zoomed in" view of just 100 iterations (out of 1,000). Observe the similarity in the shape of the plots for C and Java. Clearly the variations are algorithmic and not due to the JVM or the GC. The overhead of CW is clear but remains acceptable. Figure 8 shows the ratio between C and SCHISM/CMR level C for each iteration. This again shows that there are no outliers. The performance difference is accounted for by the various checks performed by Java (for a more detailed look at the overheads of Java see [16]). Figure 9 gives the worst-case observed behavior of the different collectors when the heap size ranges between 500 KB and 1500 KB. For SCHISM/CMR, the minimum heap size in which the program can run without missing deadlines is 1000 KB whereas for CMR it is 600 KB.

Figure 10 gives the minimum mutator utilization (MMU) of the different collectors for CD_x. MMU is often used as a metric for real-time garbage collectors [6]. It measures the minimum amount of time that the mutator was able to run in a time window of a given size. MMU is interesting because it embodies a metric that combines the length of GC pauses with their frequency. Unfortunately, the notion of collector pause is a little tricky to define. We considered two definitions: (*i*) time during which the mutator is preempted by the collector, or (*ii*) time spent by the mutator in allocation, store barrier, array access, and stack scan slow paths. Under the first definition our collectors exhibit an MMU of 100% with no pauses (provided that the heap size is $>$ 1100KB and the collector is the lowest priority thread, which is the default). But this is not particularly informative since all collectors have slow paths which may slow down execution. Thus, we chose (*ii*) and measure it conservatively as pauses include some mutator work: array access and allocation slow paths include some of the same logic as the fast paths that we charge to the mutator. The longest pauses are in SCHISM/CMR level CW, which are due to the allocation of large arrays: level CW simulates the effect of level C attempting to allocate a large array contiguously, failing, and then attempting to allocate payload fragments the quick way (bump pointer) but failing again, and having to go into a deeper slow path for each fragment. Level A exhibits the smallest pauses (roughly 0.4 ms) because almost all allocation is done 32 bytes at a time. The 0.4 ms pause corresponds to the time it takes to zero-initialize a fresh 4096 byte page, and is to our knowledge the smallest GC pause ever reported on a 40 MHz LEON3.

We have also measured the performance of CD_x against other real-time Java virtual machines on Sharpay. Because Sharpay is at least an order of magnitude faster, we have increased the workload of CD_x to 60 planes as opposed to just 6. We see that the worst-case observed execution time for one iteration of the benchmark on Java RTS is 25.4 ms, WebSphere SRT is 16.7 ms, while Fiji VM is 9.9 ms. Specifically, CMR and SCHISM/CMR level C has a worst-case of 5.2 ms, and levels A and CW are 6.4 ms and 9.9 ms respectively.

6.4 Scaling Predictability

So far we have shown that SCHISM/CMR performs respectably on mostly uniprocessor benchmarks. The predictability of collectors is good, and SCHISM/CMR does a good job of managing fragmentation. But real-time systems are slowly and steadily moving towards the adoption of multiprocessors. We evaluate scalability using the

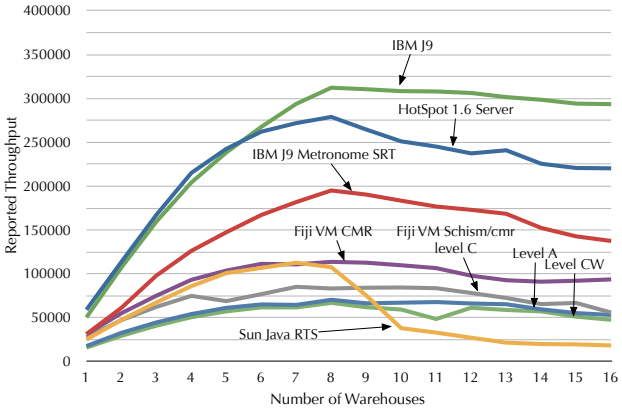


Figure 11: **SPECjbb2000 scalability.** Fiji VM and other JVMs on an 8-way multi-core.

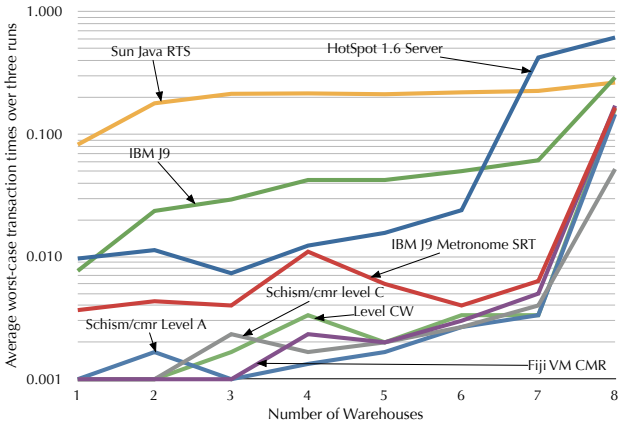


Figure 12: **Predictability.** Using the reported worst-case transaction execution times in SPECjbb2000.

SPECjbb2000 benchmark running on an 8-way machine (Sharpay). We think of SPECjbb2000 as a soft real-time workload, in that it is reasonable to assume that in a real transaction processing system worst-case processing times are important.

Figure 11 shows that our performance does scale but not as well as some of the other systems. The reason, we believe, is simple: our collectors are not parallel. While the algorithms we have presented do not preclude parallelization, we have not done this yet. At 8 warehouses on an 8-way machine, the benchmark ends up competing for CPU time against the collector itself. WebSphere SRT scales much better than Fiji VM. Both CMR and SCHISM/CMR exhibit better performance than Java RTS when we overload the processor. When the processor is not overloaded, Java RTS scales about as well as CMR.

Figure 12 gives the worst-case transaction times for all JVMs. Because we opted not to use real-time scheduling, these measurements tend to be somewhat noisy — a millisecond hiccup due to OS scheduling is common, but not common enough to be visible on every run. Thus we ran the experiment three times and report the average. For measurements up to 7 warehouses, all of Fiji VM's collectors produce better results than any other JVM. With either CMR or SCHISM/CMR at any predictability level we are able to guarantee millisecond-level worst-case transaction times. At 8 warehouses, Fiji VM performs about the same as WebSphere SRT, requiring between 52 and 171 milliseconds in the worst case (SRT requires 165 milliseconds). For 9 or more warehouses, all JVMs steadily degrade to worst-case transaction times in excess of 200 milliseconds.

These results show that SCHISM/CMR scales about as well as Java RTS while achieving significantly better predictability than any other JVM so long as the collector has a spare core on which to run.

7. Conclusion

We have introduced a novel approach to fragmentation-tolerant garbage collection, dubbed Schism, and implemented it on a hard real-time Java virtual machine. Schism employs fragmented allocation of objects and arrays, managed with a mark-region garbage collector, along with separate replication-copying of array spines, to achieve real-time bounds with good throughput. Our performance evaluation is, to our knowledge, the most thorough to date for real time garbage collectors. Our experiments show that SCHISM/CMR allows for significantly better fragmentation tolerance than any of the other RTGCs while still producing throughput to within 40% of C code. In addition SCHISM/CMR shows good performance and scalability even when compared to non real-time production JVMs on well-known benchmarks.

A. Detailed Derivation of Space Bounds

This appendix gives the detailed derivation of space bounds. For simplicity, we assume a 32-bit architecture with a 4096-byte page size. Similar formulas can be derived for 64-bit architectures and different page sizes.

A.1 Methodology

We begin by deriving the *base size*, denoted B, for an object. To this we add the collector's meta-data overheads. These are the *page header* overhead due to both the headers the collector adds to individual pages and the space used by the page table, and the *spine space* overhead. In our implementation, the size of the spine space is set to 30% of the size of the CMR space. That is, given heap size H, $\frac{10}{13}H$ bytes are always used for the CMR space that stores 32-byte payload fragments and $\frac{3}{13}H$ bytes are always held in reserve for arraylet spines even if no spines are ever allocated; Appendix A.5 proves that this is sufficient even for adversarial programs. We denote these overheads as $\mathbf{P}$ and $\mathbf{S}$, respectively. This allows us to compute the total memory size used by an object using:

$$M = B + \mathbf{P}(B) + \mathbf{S}(B) \quad (3)$$

A.2 Plain Objects

Non-array objects consist of an ordered collection of fields. The compiler guarantees deterministic layout of fields allowing the total object size to be derived as follows. A three word header (12 bytes) is prepended to every object.[4] Fields are laid out in program order starting with `Object` and walking down the `extends` chain. They are aligned in memory according to their size (for example an 8-byte field will always lie on an 8-byte boundary). The size of an n-field object can thus be obtained by the following recurrence relation, in which b_0 denotes the header size and b_i denotes the size after

[4] The object header comprises: a fragmentation word, used for linking the various 32-byte fragments together (accounting for this header is slightly tricky as it repeats every 32 bytes); a GC word used for marking by CMR; and a type word used to store both a Java lock and a pointer to the object's type.

adding the ith field, whose size is denoted by f_i for $1 \leq i \leq n$:

$$b_0 = 12 \quad (4)$$

$$b_i = align(b_{i-1}, f_i) + f_i \quad (5)$$

The *align* function accounts for byte-boundary padding and the fragmentation header inserted every 32 bytes to link to the next fragment. It can be computed as the recurrence relation $a_k(b_{i-1}, f_i)$ that is executed until fixpoint:

$$a_0 = b_{i-1} \quad (6)$$

$$a_k = \begin{cases} a_{k-1}+4 & \textbf{if } a_{k-1} \; mod \; 32 = 0 \\ a_{k-1}+1 & \textbf{if } a_{k-1} \; mod \; f_i \neq 0 \\ a_{k-1} & \textbf{otherwise} \end{cases} \quad (7)$$

A fixpoint is guaranteed provided that $f_i \leq 16$; in Java we are guaranteed that $f_i \leq 8$. Given n fields, we define the base size B as follows:

$$B = 32 \left\lceil \frac{b_n}{32} \right\rceil \quad (8)$$

A.3 Arrays

Arrays comprise a sentinel, a spine, and the payload fragments. For very small arrays, the sentinel may contain the entire array payload, or its spine. At level C, some arrays will be allocated contiguously, which results in a smaller size. We ignore optimization in deriving the worst case. Additionally, we do not include the spine size in the computation as it is part of **S**. The sentinel is a single 32-byte fragment which contains 16 bytes of header and pseudo-length meta-data.[5] The remaining 16 bytes may be used for the payload if the array is small. Otherwise, there will be 0 or more 32-byte fragments for storing the payload. Thus the base size B of an array is as follows. We use l to denote the array length and e to denote the element size in bytes:

$$B = \begin{cases} 32 & \textbf{if } l \times e \leq 16 \\ 32 + 32 \left\lceil \frac{l \times e}{32} \right\rceil & \textbf{if } l \times e > 16 \end{cases} \quad (9)$$

For arrays B is precise at level A, but an upper bound at level C.

A.4 Page overhead

The CMR space is a collection of pages that are contiguous in memory and separate from the spine space. The default allocation mode for level A is that one page may contain multiple objects, but that no contiguous object ever straddles multiple pages. In level C, contiguous objects are allowed, in some cases, to cross page boundaries. Each page is then devoted entirely to that one object and even if there is free space in the page it cannot be used so long as that object is alive.

Page status is maintained in a page table and in page headers. The page table has a 4-bit state per page. Page headers are 32-bytes, leaving $4096 - 32 = 4064$ bytes for data in each page. We compute page overheads such that they may be fractional: for example if an object is 10 bytes long then we say that it uses $\frac{10}{4064}$th of the page header and $\frac{10}{4064}$th of the 4-bit page table field. We compute this by first introducing a helper function $p(B)$ which gives the number of pages (which may fractional) used by the object:

$$p(B) = \frac{B}{4096 - 32} \quad (10)$$

[5] The sentinel header consists of a fragmentation word, a GC word and type word as before. The fragmentation word is used for linking to the spine. The pseudo-length is used to determine the length of the array as well as to indicate if the array is contiguous or fragmented. If the array is fragmented, this field will be 0 and the "true" array length will be stored in the spine.

Thus, if we just consider the page header then the number of bytes used is $4096 \times p(B)$, so a 4064 byte object will use exactly one page. If we just consider the page table, the number of bytes used is $\frac{1}{2}p(B)$, so for every 4064 bytes we use 4 bits. Putting this together, the page overhead for level A is as follows:

$$\mathbf{P}(B) = 4096 \; p(B) + \frac{1}{2}p(B) - B \quad (11)$$

$$= \frac{65}{8128}B \quad (12)$$

$$\simeq 0.007997 \; B \quad (13)$$

This is a precise account of the overhead at level A. For level C, arrays that are larger than the 4064 maximum size for single-page contiguous allocation can be allocated over multiple pages. In that case the first page requires 16 bytes for the CMR's *large object header* and any free space on the last page is wasted until the object dies. Thus the total page overhead considering a contiguous large object allocation is:

$$\mathbf{P}_{large}(B) = \left[4096 \left\lceil \frac{B+16}{4096} \right\rceil \left(1 + \frac{0.5}{4096}\right)\right] - B \quad (14)$$

This is only needed for $B > 4064$ on level C. In fact, depending on the object size, sometimes $\mathbf{P}(B)$ can be larger than $\mathbf{P}_{large}(B)$, so to account for the worst case we take the maximum of the two.

A.5 Provisioning Spine Space

Spines are allocated in the separate spine space which is set to 30% the size of the CMR space. We show that this is sufficient even for adversarial programs. A spine requires a 8-byte header (a forwarding pointer and the length), and a 4-byte pointer for every fragment of the payload. We define the spine size s as follows, assuming that l is the array length and e is the element size:

$$s = 8 + 4 \left\lceil \frac{l \times e}{32} \right\rceil \quad (15)$$

We conservatively assume a heap filled with arrays, and that the sizes of those arrays are chosen adversarially resulting in the largest possible spine space overheads. For payloads ≤ 16 bytes no spine is needed as the data fits in the sentinel. For payloads between 16 and 96 bytes, the spine can be inlined in the sentinel. Thus, spine space allocation can only happen for arrays with payloads larger than 96 bytes. The worst-case occurs for the smallest array that results in spine allocation. Taking $l \times e = 97$, we obtain $B = 160$ and $s = 24$. Thus we need $s/B = \frac{24}{160} = 0.15$ bytes of spine space for every byte of CMR memory, excluding page overheads. As $l \times e$ increases then s/B converges to 0.125 — thus 0.15 is indeed the worst-case. We double this amount to account for the spine space's copy reserve to get an overhead of 0.3. Thus setting aside 0.3 bytes in the spine space for every byte in the CMR space is sufficient to ensure that the spine space is never exhausted.

This analysis is slightly pessimistic. Instead of using s/B, we should use $s/(B + \mathbf{P}(B))$, which is slightly smaller; it gives us 0.29762 instead of 0.3. Using round numbers has its benefits: when the user specifies a heap size we need to slice the heap into a spine region and a CMR region; the rounder the number the more likely we are to be able to do so precisely.

Because the heap is always divided in this fixed way we always assume that the spine overhead of every object is:

$$\mathbf{S}(B) = 0.3\,[B + \mathbf{P}(B)] \quad (16)$$

A.6 Total Object Size

The total object size is the sum of B and the two sources of overheads: page allocation overhead and spine space overhead.

Thus we write the total object size as follows:

$$M = B + \mathbf{P}(B) + \mathbf{S}(B) \quad (17)$$

For predictability level A both $\mathbf{P}(B)$ and $\mathbf{S}(B)$ have a simple closed form, so this simplifies to:

$$M = 1.3104B \quad (18)$$

This formulation allows the programmer to compute exactly what heap size to pick given an analysis of the number and type of objects known to be live at the high watermark. Simply summing the sizes M and rounding up to the nearest page size yields the heap size necessary to never run out of memory. Even if the heap structure changes and the programmer suddenly decides to allocate arrays instead of objects or vice versa, it is guaranteed that an out-of-memory condition will not be reached provided that the total sizes of objects are less than or equal to the heap size.

Acknowledgments

We thank Tomas Kalibera, Gaith Haddad and Ales Plsek for their help with CD_x, and the anonymous reviewers for their detailed comments. This work is supported in part by NSF grants CCF-0702240, and CCF-0811691, and IIP-0912730.

References

[1] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time Java virtual machine. In *Conference on Embedded Software (EMSOFT)*, 2007, pages 249–258. doi: 10.1145/1289927.1289967.

[2] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: democratic scheduling for real-time garbage collection. In *Conference on Embedded Software (EMSOFT)*, October 2008, pages 245–254. doi: 10.1145/1450058.1450092.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Symposium on Principles of Programming Languages (POPL)*, January 2003, pages 285–298. doi: 10.1145/604131.604155.

[4] Steve Blackburn and Kathryn McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Programming Language Design and Implementation (PLDI)*, 2008, pages 22–32. doi: 10.1145/1375581.1375586.

[5] Eric Bruno and Greg Bollella. *Real-Time Java Programming: With Java RTS*. Addison-Wesley, 2009.

[6] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Conference on Programming Language Design and Implementation (PLDI)*, 2001, pages 125–136. doi: 10.1145/378795.378823.

[7] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In *International Conference on Virtual Execution Environments (VEE)*, 2005, pages 46–56. doi: 10.1145/1064979.1064988.

[8] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3–5):223–261, 2003. doi: 10.1002/cpe.712.

[9] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, and Jan Vitek Ben Titzer and. Cdx: A family of real-time Java benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2009, pages 110–119. doi: 10.1145/1620405.1620421.

[10] Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *Real-Time Systems Symposium (RTSS)*, December 2009, pages 81–92. doi: 10.1109/RTSS.2009.40.

[11] B. McCloskey, David Bacon, Perry Cheng, and David Grove. Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors. Technical Report RC24505, IBM Research, 2008.

[12] Scott Nettles and James O'Toole. Real-time replication-based garbage collection. In *Conference on Programming Language Design and Implementation (PLDI)*, 1993, pages 217–226. doi: 10.1145/155090.155111.

[13] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for modern platforms. In *International Symposium on Memory Managment (ISMM)*, 2007, pages 159–172. doi: 10.1145/1296907.1296927.

[14] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008, pages 33–44. doi: 10.1145/1375581.1375587.

[15] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2009, pages 110–119. doi: 10.1145/1620405.1620421.

[16] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *EuroSys Conference*, April 2010.

[17] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, 2008, pages 68–76. doi: 10.1145/1434790.1434801.

[18] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Java Technologies for Real-time and Embedded Systems (JTRES)*, September 2007, pages 277–278. doi: 10.1145/1288940.1288954.