An Emprical Evaluation of Memory Management Alternatives for Real-time Java

Filip Pizlo Jan Vitek
Purdue University

Abstract

Memory management is a critical issue for correctness and performance of hard-real time systems. Java environments usually incorporate high-throughput garbage collection algorithms, but these algorithms can induce pause times in excess of 100 milliseconds. This is not acceptable for a real-time system. Two approaches for remedying this problem are being investigated. The pauses times can be bounded with a real-time garbage collector, or means to program around the collector entirely can be provided, as it done in the Real-time Specification for Java with its Scoped Memory interface.

This paper presents the first side-by-side empirical evaluation of the impact of memory management regimes on realistic real-time applications. We present usability arguments as well as a detailed performance comparison of scoped memory and real-time garbage collection. Experience with medium sized systems suggests that while programming with Scoped Memory is error prone, it provides substantially better throughput. We have observed a throughput reduction of up to 37% and, in the worst-case, an 80% latency penalty for real-time garbage collection.

1. Introduction

Multi-million line systems are being developed in Java for avionics, shipboard computing and simulation. A key attraction of the Real-time Specification for Java (RTSJ) [8] for such systems is that it makes it possible to develop applications that mix hard-, soft- and non-real-time tasks in the same environment and in a memory-safe way. Unfortunately, these advantages come at the expense of predictability. The garbage collection algorithms, or *garbage collectors* (GC), used for automatic memory management typically require applications to be interrupted for hundreds of milliseconds. Such pauses are not acceptable in real-time systems. There are two approaches to address this critical problem. One is to provide worst-case bounds on pause times and utilization of the garbage collection algorithm with so-called *real-time garbage collectors* (RTGC). The other alternative is to program around the garbage collector with the RTSJ's *scoped memory* API. Scoped memory allows real-time tasks to allocate data in regions of memory that can be reclaimed in bounded time. To guarantee memory safety, the virtual machine monitors every access to memory (loads and stores of object references) performed by programs and throws an exception if a program performs a memory operation that may lead to an error.

The goal of this paper is to provide the first detailed analysis and discussion of the tradeoffs between real-time garbage collection and scoped memory for real-time Java. Such an analysis is crucial for practitioners to make an informed choice as memory management regimes have profound implications on the design and architecture of applications as well as their performance profile. To date, the debate has been polarized and based on folklore rather than on hard data. The contributions of this paper are thus: (I) The first *open source* implementation of scoped memory *and* real-time garbage collection in the same Java virtual machine. Our real-time garbage collector is a time-based collector based on the state-of-the-art Metronome collector [2]. We also implemented a traditional copying collector to provide a throughput baseline [5]. (II) A discussion of the software engineering benefits and dangers of real-time garbage collection and scoped memory. This extends our previous work on design patterns and idioms for scoped memory [2]. (III) An empirical evaluation of memory management based on two realistic RTSJ applications: a collision detector (10KLoc) and a real-time CORBA ORB (202KLoc). We also use the industry standard SpecJVM98 benchmark suite to study throughput of RTGC. We have obtained data on pause time, latency and memory usage of the competing techniques.

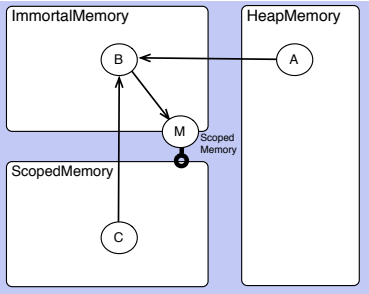
This work has been performed within the context of Ovm, a high-performance customizable virtual machine[1] which has been successfully deployed in on a ScanEagle UAV in a collaboration with the Boeing Company [4].

2. Background

2.1. Scoped Memory

In the RTSJ, storage for an allocation request (i.e. new in Java) performed by a real-time task is serviced differently from standard Java allocation. The RTSJ extends the Java memory management model to include dynamically checked regions known as *scoped memory areas* represented by subclasses of ScopedMemory. A memory area is an allocation context that provides a pool of memory shared by threads executing within it. Individual objects allocated in a memory area cannot be deallocated; instead, an entire area is torn down as soon as all threads exit it. The RTSJ defines two distinguished scopes: *immortal* and *heap* mem-

1 Measurements of the latest version of Ovm show that its mean performance in the SpecJVM98 suite is 16% better than Hotspot 1.5 server.



1. HeapMemory is garbage collected memory with no timeliness guarantees.
2. ImmortalMemory is not subject to reclamation.
3. ScopedMemory can be reclaimed in a single step if no thread is active in the area.
4. Immortal data can be referenced from any region. Scoped data can only be referenced from same scope or a nested scope. Violations lead to dynamic IllegalAssignmentErrors.
5. NoHeapRealtimeThread cannot load heap references.

Figure 1. *Memory Management in the Real-time Specification for Java.*

ory, respectively for objects with unbounded lifetimes and objects that must be garbage collected. Two new kinds of threads are also introduced: *real-time* threads that may access scoped memory areas (RealtimeThread); and *no heap real-time* threads (NoHeapRealtimeThread), which are real-time threads protected from garbage collection pauses. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory, nor can a scoped memory area hold a reference to an object allocated in an inner (more deeply nested) scope. The memory model is illustrated in Fig. 1. Memory areas provide methods like enter(Runnable) that permit application code to execute within a scope. Using nested calls, a thread may enter multiple scopes, dynamically building up a scope hierarchy. Misuse of these methods is punished by dynamic errors; e.g. a ScopedCycleException is thrown when a user tries to enter() a ScopedMemory that is already accessible. IllegalAssignmentError and MemoryAccessError are thrown on attempted violations of the memory access rules. Reference counting on enters ensures that all the objects allocated in a scope are reclaimed (and *finalized*) when the last thread leaves that scope.

2.2. Real-time Garbage Collection

For garbage collection (GC) to be used in hard real-time applications, reasonable worst-case bounds must be provided for: (a) GC pause times, (b) throughput, and (c) memory usage. Because events that arrive while the collector is operating cannot be handled until the collector yields, pause times must be bounded. Further, the mutator (i.e. user code) must not be interrupted too frequently. In particular, the effect on throughput must not be so severe as to prevent the application from handling events in a timely fashion. Finally, all memory allocation requests must succeed. This requires finding a rate at which to run the collector to make it keep up with allocation. Work on real-time collection can be traced back to Baker's incremental copying collector [3]. The central idea behind Baker's work is decreasing the intrusiveness of a collector by piggy-backing work onto mutator operations. To ensure consistency, a small piece of code, called a read barrier, is inserted by the compiler before every memory read to perform copying, and the allocation code is modified to perform a bounded amount of collection work. The worst-case in a program using Baker's collector involves a copy operation upon every read, and a (large) unit of collection work on every allocation. Hence, even though individual pauses are small, the worst case execution time of an allocation makes Baker's collector unsuitable for hard real-time. Put another way, the GC fails to bound its impact on throughput. Baker's collector is said to be *work-based*, in the sense that work done by the mutator leads to work by the collector. Bacon et al. [2], Henriksson [13], and Detlefs [12] investigate a different approach to real-time collection. In their *time-based* systems, the collector interleaves with the mutator at regular intervals. Constant time read (or write) barriers are still needed to maintain consistency, but allocation can be made constant-time. The worst-case bounds on execution time in the mutator become more realistic, allowing the collector to be used in hard real-time systems.

3. Programming model

This section overviews memory management issues in real-time Java applications.

3.1. Software Architecture

The software architecture of scoped memory programs tend to reflect the scoped memory hierarchy in different ways. A memory area is typically used to contain objects with roughly equivalent lifetimes. The use of nested scopes is needed if within a particular task, there is a subtask that repeatedly allocates data with non-overlapping lifetimes. Nested scopes complicate reasoning about correctness – thus when possible they are best avoided. In general, it is true that the code that has to run in scoped memory requires

greater care to prevent errors. Thus it is not unusual to find versions of a class specialized for use in scoped memory – and possibly specialized for a particular arrangement of scoped memory areas.

Programming with scoped memory entails a loss of compositionality. By this we mean that components, when tested independently, may work just fine, but they will break when put in a particular scoped memory context. This is because scoped memory adds an extra dimension – *where* each object has been allocated – that complicates reasoning about program correctness. In our previous work, we outlined a number of design patterns and idioms for programming effectively with scoped memory [23]. Following these guidelines simplifies RTSJ development. The RTZen system is a good example of a large open source real-time Java program entirely written with scoped memory [16].

Real-time garbage collection, on the other hand, does not require significant changes to the programming development methodology. Just like in standard Java, it is important to make sure that programs don't leak memory, but besides this, no particular care needs be taken and there is thus no need to develop specialized versions of classes.

3.2. Scoped Memory Hazards

We overview the dangers of programming with scoped memory and give recommendations for improved safety.

Hidden Allocation. A surprisingly frequent source of memory errors is unexpected allocation. For instance, calling the put() method of a hash table may allocate a bucket in the current memory area. If the current area is a child of the hash table's area, an exception will be thrown. Preventing such phenomena is difficult if the programmer is not aware that a method is allocating. There are two solutions: one is to *harden allocation* at the callee or delegate memory area selection to the caller. The former implies allocation be safe irrespective of the current allocation context. For the put() method this means discovering where the table was allocated and requesting, reflectively, allocation in the same context:

```
public void put(Object key, Object value){
  ...
  MemoryArea ma = MemorArea.areaof(this);
  Bucket b =
    (Bucket) ma.newInstance(Bucket.class);
```

Of course, this style of programming quickly becomes cumbersome – especially if the objects being created are themselves allocating. The alternative is to delegate the selection of allocation context to the caller, but this requires that allocation be made part of the interface of all components.

Store. Every assignment tgt.f=src is checked by the VM to ensure that the lifetime of tgt is shorter or equal to that of src. Typically, errors occur when the program is not fully aware of the allocation context of all the objects it is manipulating.

Load. The RTSJ tries to protect NoHeapRealtimeThreads from GC interference. The family of copying GC algorithms [15] often assumes that objects in the process of being copied are not accessible to the mutator. Thus the GC is free to temporarily corrupt the state of the heap, as long as everything is patched up when the program restarts. Since a NoHeapRealtimeThread must be able to preempt the GC at any time, the real-time task must not manipulate objects allocated in the GCed heap. Any read of a reference will, if the current thread is a NoHeapRealtimeThread, be checked to ensure that the referenced object does not reside in the heap. Errors can occur for two reasons: either (a) because the program reads a field of an immortal object that happens to hold a heap reference or (b) because a memory area is used by a mixture of NoHeapRealtimeThreads and RealtimeThreads and one of the RealtimeThreads stored heap reference in one of the shared objects.

Synchronization. Another source of unbounded blocking for hard real-time tasks comes from use of the Java synchronized statement. If a NoHeapRealtimeThread ever tries to acquire a monitor held by a non-real-time Java thread, two things can occur (a) the NoHeapRealtimeThread may experience unbounded locking because, the non-real-time Java not being designed with real-time requirements in mind, the Java code does not release the lock (there are documented cases of Java programs holding on locks forever). (b) The non-real-time thread may trigger GC, thus causing the NoHeapRealtimeThread to block for the entire duration of the GC. This kind of situation may be due to an object allocated in immortal memory and accessed from Java threads and NoHeapRealtimeThreads. If a scoped area is used by a mixture of threads, transitive locking can occur because of a NoHeapRealtimeThread blocking on a RealtimeThread that then blocks on a plain Java thread.

Finalization. Java supports automatic object cleanup through the use of finalize() methods. In the case of scoped memory, finalization occurs when the last thread exits a scope. The danger with finalization is that it increases the latency of scope exit. There is also a more subtle problem with finalization. If a scope is used by a mixture of NoHeapRealtimeThread and RealtimeThread, there can be objects with heap references. The implementation faces a conundrum if the thread performing finalization is a RealtimeThread, then scope exit may potentially be blocked by GC. On the other hand if the finalization thread is a NoHeapRealtimeThread, it can experience a heap memory access exception. Neither is appeal-

ing, the solution selected by the RTSJ is to risk blocking for GC.

Programming Guidelines. Some simple coding guidelines can prevent many of the above errors. Programmers should avoid mixed NoHeapRealtimeThread and RealtimeThread scopes. RealtimeThreads and finalize() methods should be avoided altogether. Real-time code should avoid, as much as possible, the use of static variables that may contain, or be accessed by Java threads. Components should document possible allocation patterns and allocation should be hardened.

3.3. Memory Usage Estimates

Memory usage needs to be bounded to ensure that Java processes do not run out of memory. While the constants are highly implementation dependent, we can describe informally the methodology to be used to estimate space requirements. For scoped memory, we need to be able to compute for each real-time thread and each scope, *MaxAlloc(R in S)*, the maximum allocation performed by the thread in that scope. This is used to size scopes. For the overall VM memory requirements we need to know *MaxLive*, the maximum number of heap allocated objects. To this number we need to add the overhead of the garbage collector. In the case of a copying collector, for example, the GC overhead is roughly equal to *MaxLive*. For a time-based real-time garbage collector we need to compute the maximum instantaneous allocation rate, *MaxAllocRate*. Together with the garbage collection rate, the max allocation rate is used to compute memory requirements of the RTGC as demonstrated in [2]. It is worth mentioning that computing the maximum number of allocated objects is easier than computing allocation rates as these depend on the relative speed of threads. Also, it is not clear what is the impact of bounding allocation rate on programming style.

4. Implementation

This section describes the implementation of scoped memory and real-time garbage collection within the Ovm real-time Java virtual machine.

For applications requiring high throughput, Ovm provides a mostly copying garbage collector that implements Bartlett's algorithm [5]. In this collector, stacks are scanned conservatively (as in the Boehm collector [7]) but pointers in the heap are traced accurately. This is the most efficient choice for any system that, like Ovm, compiles to C++ – as the C++ compiler does not support accurate stack scanning. The GC emulates a two-space design that has the added benefit of partially defragmenting memory. Conservative scanning limits defragmentation as it may require the pinning (i.e. prevent the copy) of objects referenced by ambiguous pointers on the stack.

4.1. Ovm Scoped Memory

The Ovm scoped memory implementation provides strong real-time guarantees for all scope operations. Scopes are implemented as contiguous sequences of bytes. An auxiliary data structure scopeOwner maps blocks to memory area objects. This allows for constant-time recovery of the memory area in which an object was allocated in, and *does not* require additional overhead in the object header. In this we differ from [11] and our previous work [21], which required one additional word per object. We use the algorithm of [21] to implement the constant-time write barriers needed to prevent establishing illegal cross-scope references. Constant time read-barriers are implemented by comparing the base address of the object against the start of the heap. In Ovm all barriers are implemented in Java. Code for these barriers is given in Fig. 2. The worst case execution of scope entry is linear in the number of scopes in the program – the first time a scope is entered data used for scope checks must be computed as in [21]. Object allocation is linear in ob-

```
storeCheck(VM_Address src,VM_Address tgt) {
   int sb = src.asInt() >>> blockShift;
   int tb = tgt.asInt() >>> blockShift;
   if (sb != tb) storeCheckSlow(sb, tb);
}

storeCheckSlow(int sb, int tb) {
  tidx = VM_Word.fromInt(tb - scopeBaseIdx);
  if (!tidx.uLessThan(scopeBlocks)) return;
  ta = scopeOwner[ tidx.asInt() ];
  sidx = VM_Word.fromInt(sb - scopeBaseIdx);
  if (!sidx.uLessThan(scopeBlocks)) fail();
  sa = scopeOwner[sidx.asInt()];
  if (sa == ta)  return;
  if ((ta.prange - sa.crange) & MASK)!=RES)
    fail();
}

readBarrier(VM_Address src) {
 if (!doLoadCheck)  return;
 if (src.diff(heapBase).uLessThan(heapSize))
    fail();
}
```

Figure 2. *The store and read barriers in the Ovm implementation of scoped memory. The* `storeCheck()` *checks for objects allocated in the same block. If the* `storeCheckSlow()` *has to be invoked, the memory area of each object is retrieved and a range inclusion check is evaluated. The* `readBarrier()` *will fail only if the current thread is a* NoHeapRealtimeThread *and the target of the reference is a heap location.*

ject size because our allocator zeros memory lazily, at allocation time. Finally, scope exit is constant time (excluding finalization and synchronization costs).

4.2. Ovm Real-time Garbage Collection

The Ovm real-time collector is a mark-sweep snapshot-at-the-beginning non-copying incremental garbage collector. The design is inspired by [2]. As the collector is written in Java, we can utilize RTSJ features in the implementation. The collector thread is a real-time thread with a priority high enough that unless it yields, it will not be interrupted by application threads. The timer that causes the context switches between the collector and mutator is simply a periodic task with higher priority than the collector. A second timer tick causes the collector to pause and allow the mutator to resume. When memory usage increases beyond a user-specified threshold, the collector thread is scheduled. Because of its priority, it immediately preempts any application threads. The collector then proceeds as follows:

1. Accurately scan the stack.
2. Scan the virtual machine's internal (immortal) data structures.
3. Walk the heap starting with the roots found in (1 - 2).
4. Unmarked objects are reclaimed.

The only part of the RTGC that is not incremental is stack scanning (we leave this to future work). In the remaining stages, the collector periodically polls to see if it should yield to the mutator threads. The time between polls is small. During the marking phase, we poll after touching each pointer. Thus, the scanning of a large array can be interrupted at any time. In the sweep phase we poll in between pages. The amount of work to sweep a page is linear in the page size (the default setting is 2048 byte pages).

Fig. 3 gives the Java code for the store barrier inserted by the compiler every time the program writes a reference to memory. While we could improve the average case by adding a test to turn off the barrier when the RTGC is *not* running, we chose instead to have an unconditional barrier. The motivation is that we want stores to be predictable. This makes empiric evaluation of program execution times easier. The cost of any sequence of instructions will not be affected by the state of the garbage collector.

```
storeBarrier(VM_Address tgt) {
  VM_Address old = tgt.getAddress();
  if (old!=null && old.getColor()==WHITE) {
    old.setColor( allocationColor);
    worklist.enqueue(old);
  }
}
```

Figure 3. *The store barrier of the Ovm RTGC is a shade-old-referent-on-write barrier [22] that, so as to ensure predictability, runs even when not collecting. Every time a non-null, unmarked target is encountered the object is colored (by changing the lower order bits of the class pointer) and added to a worklist.*

As mentioned above, the C++ backend used by Ovm does not support accurate stack scanning. To overcome this problem the Ovm bytecode-to-C++ compiler generates code to keep track of stack pointers explicitly, as in [24].

5. Experimental evaluation

In our experimental evaluation we have the following goals. First, we compare the performance of running in our RTGC to running in Java mode optimized strictly for throughput. Second, we compare the performance of running in an RTGC directly to using scopes.

5.1. Methodology

We have two full-fledged applications – one written by students at Purdue, the collision detector, and the other, a real-time CORBA server written independently by researchers at UC Irvine [16]. All programs have been designed for and written with the RTSJ APIs and scoped memory areas.

In order to obtain a comparison we have implemented two compatibility layers that let us run the same program either with a Real-time Garbage Collector, our copying non-Real-time collector, or with ScopedMemory. The simplest compatibility layer require nothing more than stubbing out the RTSJ API. All allocation is performed in the heap. Furthermore, NoHeapRealtimeThreads can be preempted by the garbage collector and scope checks are disabled. We also support a slightly more complex compatibility layer for RTSJ programs that make use of the getMemoryArea(Object) method. For these programs we do retain scoped memory area objects, and extend each Java object with an extra field that contains a reference to the scope in which the object would have been allocated if the program was running in scoped memory. This allows us to achieve a complete semantic illusion of scoped memory support, while submitting the application to the performance impact of garbage collection.

Benchmarks were run on a Pentium IV 1600 MHz with 512 MB of RAM, running Linux 2.6.14. We compiled with GCC 3.4.4.

5.2. Application Throughput

We use the industry standard SpecJVM98 benchmark suite to estimate the impact of RTGC on throughput. Results are reported for the second run with the -s100 option and a fixed heap size of 256MB.

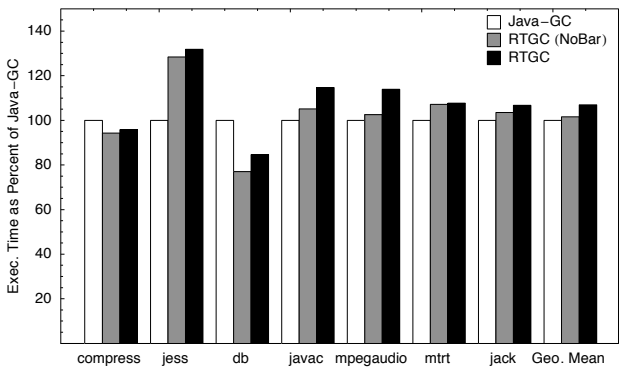


Figure 4. *Throughput overhead of RTGC for SpecJVM98. We measure the throughput with three Ovm configurations: Java-GC, RTGC with write barriers disabled (RTGC-NoBar), and RTGC. Results presented are normalized to Java-GC.*

Fig. 4 is normalized to Ovm's Java-GC algorithm. The difference is the overhead of using a real-time collector. RTGC-NoBar indicates the cost of real-time collection *without* write barriers and RTGC is the throughput overhead of the real-time collector. The geometric mean for RTGC gives a 7% slowdown with a worst case of 32% for jess. This overhead comes from (*a*) write barriers, (*b*) allocation, and (*c*) collection as each of these components has been designed to tradeoff throughput for predictability. The cost of write barriers is no more than 11% as shown by RTGC-NoBar. Several benchmarks, compress, db and mpegaudio, do not GC, and have little or no overhead. Thus we conclude that allocation costs are a small part of the overall costs. We used an allocation profiler on jess to confirm that the 32% overhead is mostly due to the time spent in the collector.

5.3. Comparing RTGC to Scopes

5.3.1. RTZen. We measure RTZen 1.1 running the DOOM application with one client. The size of the sources is 202KLoc. Client and server are on the same machine in single-user mode with almost all services disabled. In all cases, the client VM is the Ovm with scoped memory. We record the latency of processing a request in the server; barring VM jitter, this time is expected to be deterministic. Overall latencies include three terms: client-side processing, local socket processing in the OS, and server-side processing. Because the client uses scopes, it exhibits no significant jitter. Furthermore our measurements reveal no significant jitter in the sockets layer. Hence, any jitter observed is due entirely to the server.

Fig. 7(a) gives the latencies of 600 requests with a mostly copying collector. As expected there are large outliers, up to 58ms, due to GC pauses. Fig. 7(b) shows latencies of requests running on a VM with RTGC. The collector is set to run once every 4ms. Hence for every ms the collector ran, the application got 3ms. Unfortunately, our implementation does not always stop after *exactly* 1ms; thus the worst-case pause time is 1.8ms (c.f. Fig. 7(e)). This said, the results are good: the latencies range between 1.4 and 2.9ms and the difference between the best- and the worst-case latency is explained by the overhead of a single RTGC. Fig. 7(c) is the same with scoped memory. The results are nicely predictable with latencies ranging from 1.7ms to 2.1ms. As the 2.1ms outlier occurs close to startup, we believe it is either due to paging in the OS or else some lazy initialization in RTZen. Finally, Fig. 7(d) is a run without the scoped

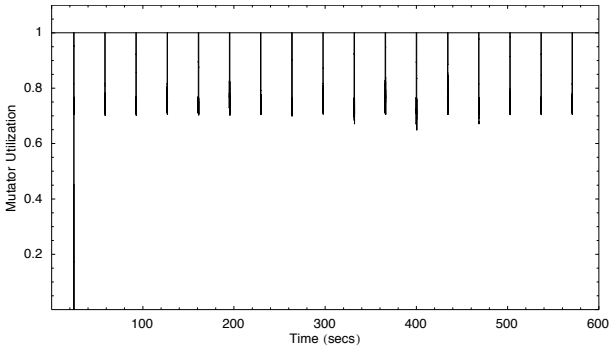


Figure 5. *Mutator utilization trace for RTGC running the RTZen benchmark measured over 10ms windows. (100% indicates the mutator is in full control of the CPU and 0% that the RTGC ran for the entire 10ms window.)*

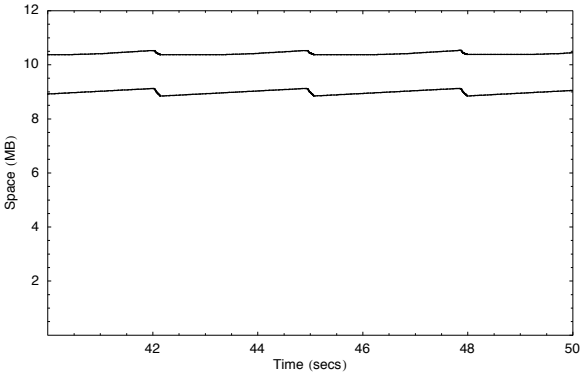
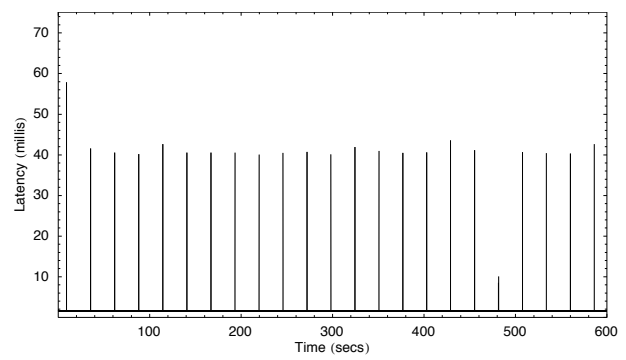
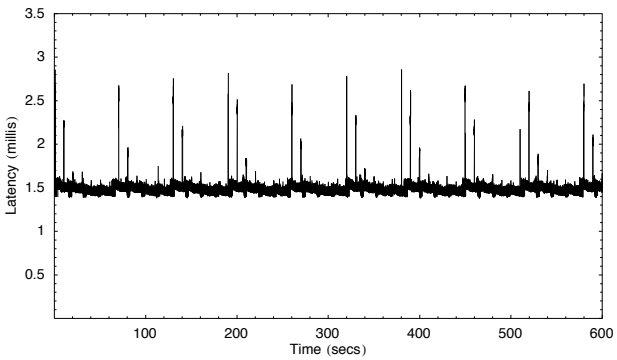


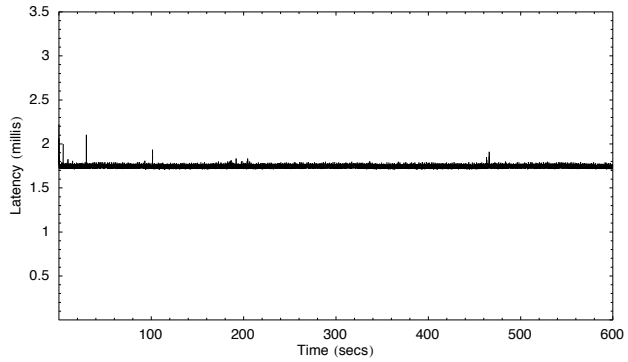
Figure 6. *Memory usage of RTGC running the RTZen benchmark. The lower line gives the size of all live objects at a given point in time and the upper line is the number of pages in use. The difference between the two is a measure of fragmentation.*



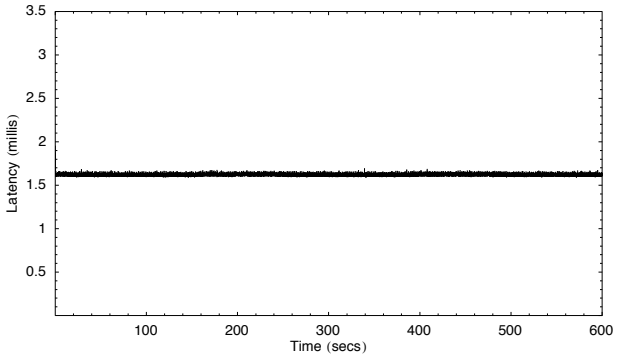
(a) Java-GC: Latency.



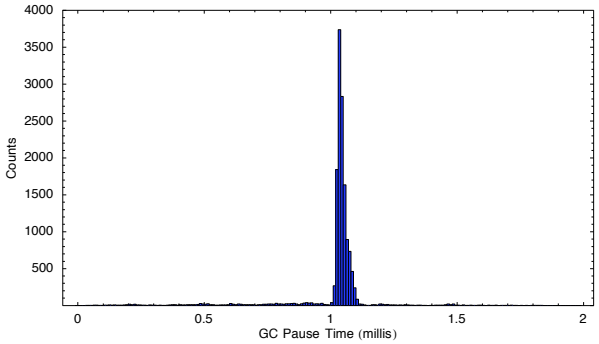
(b) RTGC: Latency.



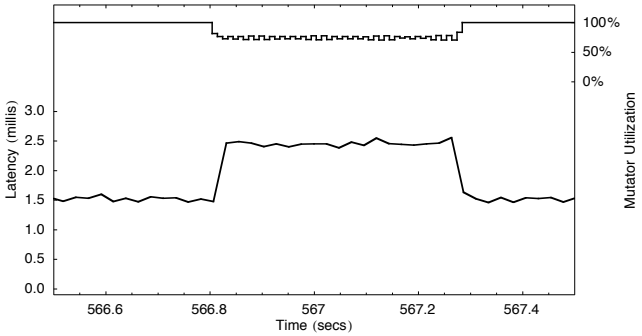
(c) ScopedMemory: Latency.



(d) ScopedMemory-NoChecks: Latency.



(e) RTGC: Pause Times.



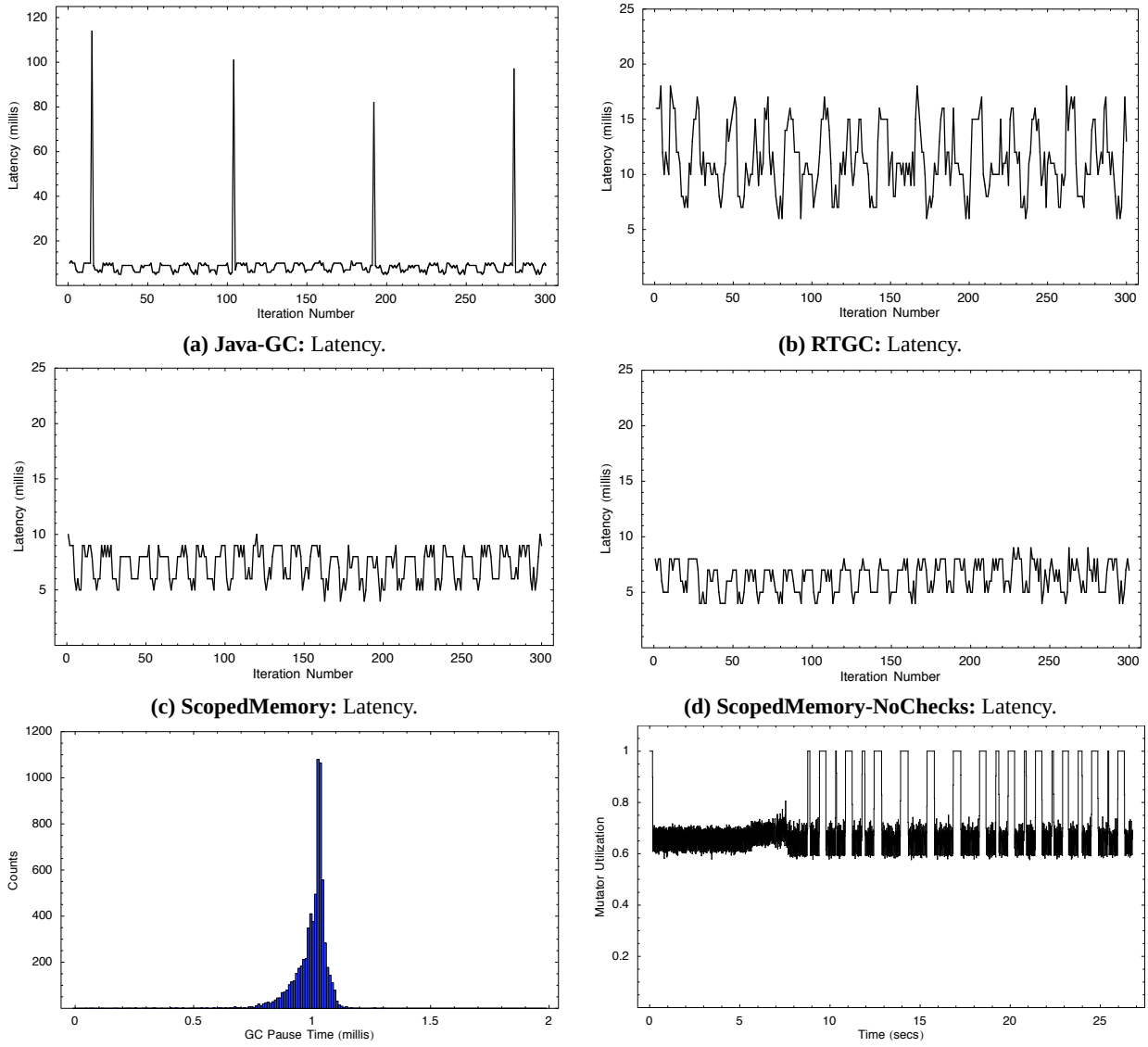
(f) RTGC: Latency and Utilization.

Figure 7. *Evaluating the UCI RTZen real-time CORBA ORB.*

memory dynamic checks (read/write barriers) as one could obtain with static analysis or with the scoped type systems of [1]. The latencies are lower than before, around 1.6ms.

As mentioned, the RTGC is configured for 75% utilization; i.e. the worst-case drop in throughput should be 25%. Fig. 5 shows a trace of the mutator utilization measured over 10ms intervals. At the beginning of the trace it drops to zero; this is an artifact of the profiler used to record the utilization and is unrelated to RTGC. Subsequently the utilization holds between 65% to 70%. It is instructive to relate utilization and latency simultaneously as shown in Fig. 7(f). The graphs zooms in on one RTGC cycle. Every request clearly takes a hit when the collector is running. The increase in latency corresponds to the typical GC pause time. Because the mutator gets 3ms windows, one would expect to see some requests get processed in the best-case time of 1.5ms; however, because the client itself is periodic, the requests typically end up coming at around the time of GC activity.

Finally, Fig. 6 examines memory usage in the RTGC. The lower line indicates the sum of the sizes of all objects in the heap that have been allocated and not yet collected (live data). The top line indicates the sum of the sizes of the pages used. The difference between them is a worst-case indicator of fragmentation. In this application fragmentation



(a) Java-GC: Latency.

(b) RTGC: Latency.

(c) ScopedMemory: Latency.

(d) ScopedMemory-NoChecks: Latency.

(e) RTGC: Pause times.

(f) RTGC: Utilization trace.

Figure 8. *Evaluating the Collision Detector.*

is small, no more than 10%. Moreover we can observe how GC gradually free dead objects.

This benchmark is interesting as it has shown that RTGC meets the application's predictability requirements. In term of median latencies, Java-GC and RTGC are the fastest with a median latency of 1.5ms. Scopes are a bit slower with median latencies of 1.7ms (1.6ms without scope checks).

5.3.2. Collision Detector The collision detector is a 41KLoc RTSJ program with two real-time threads. One thread is a periodic NoHeapRealtimeThread that detects collisions in data generated by a simulator. The other thread is a Java thread that interacts with the environment. The input is a complex simulation involving over 200 aircrafts. We record the latency of processing one input frame.

As before, we begin with the mostly copying collector. Fig. 8(a) shows that the maximum latency for one input frame is 114ms. Switching to RTGC dramatically decreases the worst-case latency, it drops to 18ms, shown in Fig. 8(b). The RTGC is configured for 2ms of mutator activity for every 1ms of collector activity. The utilization trace of Fig. 8(f) shows 60% utilization most of the the time. The long period of collector activity at the beginning of the utilization trace is due to the simulation initializing itself; this is not part of the mission phase and has no corre-

sponding latency measurements. The RTGC pause times of Fig. 8(e) exhibit a worst case of 1.4ms (during the initialization phase) and 1.1ms during the mission phase. Scoped memory, as shown in Fig. 8(c), is more regular behavior with latencies ranging between 5ms and 10ms. The jitter appears to entirely due to algorithmic reasons (input frames that have more collisions are more complex). Disabling dynamic checks improves the performance by about 1ms as shown in Fig. 8(d).

In summary, the collision detector behaves as expected with Java-GC running at median 9ms, RTGC at 11ms and scoped memory 8ms, and without checks at 7ms. Here using scoped memory is a performance win over garbage collection.

6. Related Work

BeeBee and Rinard reported [6] that they found it "close to impossible" to develop error-free RTSJ programs. This difficulty of has motivated Kwon, Wellings and King to propose Ravenscar-Java [18]. The authors point out that while Java is a better programming language for high-integrity system than C, there are some features that are error prone. The profile defines a subset of the RTSJ that can decrease the likelihood of errors. Ravenscar mandates that applications be split in two phases: an initialization phase in which data structures, scopes, and threads are created and have initial values assigned to them, and a mission phase in which the real-time logic is invoked. All memory areas are created in the initialization phase and reside in immortal memory, in other words, the scope hierarchy is flat. While Ravenscar simplifies the scope structure, it does not prevent memory access violation. In [17], Kwon and Wellings propose another approach simplification to the RTSJ. They associate scoped memory areas with methods transparently avoiding the need for explicit manipulation of memory areas. Their approach is elegant and has the potential for catching many common errors, but it cannot guarantee the absence of memory violation (in general the problem is undecidable). Furthermore, their scopes cannot be multi-threaded and we see no obvious way to handle the RTSJ idioms of [23].

Recent development in the general area of type systems for controlled sharing of references are promising. The goal of previous works such as flexible alias protection [20] and Islands [14] was to restrict the scope of references in object-oriented programs. The idea of using these techniques for safety of region-based memory management was is due to Boyapati et al. [10]. But, they required changes to the Java syntax and has not yet been evaluated on realistic applications. One of the authors has worked on *STARS* (the acronym expends to "scoped types and aspects for real-time systems"), a proposal that uses a pluggable ownership type system to give a strong guarantee of correctness. RTSJ programs developed with the STARS methodology will never experience memory access exceptions. This is achieved without changes to the Java programming language and small changes to the programming style [1]. An added benefit of STARS is that dynamic access checks are not required.

Borg et al. [9] have made an intriguing case for an *Entropy Hypothesis* which connects memory management to information theory. The performance of a memory management system is then related to the amount of information the programmer provides about the lifetimes of objects.

Nilsson et al. [19] present an ahead-of-time compiler for Java and a suite of real-time garbage collectors. However, our experimental evaluation is larger in scope, as it includes more benchmarks as well as a comparison against scoped memory and uses a high-performance real-time Java virtual machine.

7. Conclusions

This paper is the first side-by-side empirical evaluation of the impact of memory management regimes on realistic real-time Java programs. Ovm is the only open source, high-performance, real-time Java virtual machine that supports both scoped memory and real-time garbage collection. Our benchmarks include the throughput-oriented industry standard SpecJVM98, as well as two medium sized real-time applications, RTZen (202KLoc) and CD (41KLoc). The results reported in this paper, were obtained by implementing a compatibility layer that lets users run the exact same code, with the same semantics, either with a real-time garbage collector or with scoped memory. In terms of applicability, our real-time garbage collector is most closely related to the work of Bacon et al. [2], but we believe that our results also apply to Henriksson [13] algorithm. Siebert uses a very different approach in [25], his algorithm avoid copying of objects but access to their fields is not guaranteed to be constant-time.

There is a clear software engineering advantage to the use of real-time garbage collection as it obviates the need to develop special versions of classes as is the case with scoped memory and reduces the opportunities for run-time errors. Thus the development times of applications using RTGC are likely to be substantially smaller. But this comes at a price. The median overhead of using RTGC is 37% as compared to scoped memory, and 80% in the worst case. This means that when the RTGC is running only 20% of the CPU may be available to an application. Another difference between RTGC and scopes is in the ease of analysis: scoped memory only requires knowing the maximum number of allocated bytes per scope, while for RTGC it is necessary to also predict the maximum allocation rate. Analyzing the maximum allocation rate is much harder, especially in a highly concurrent system. Overall, we recommend the use of real-time garbage collection when the overheads are

acceptable but we expect that there will be a number of applications where scoped memory areas will remain the only viable alternative.

We are currently working on improving the Ovm real-time collector to support defragmentation and to improve worst-case times for allocation of large objects. More aggressive compiler optimizations are envisioned to reduce costs of barriers in real-time garbage collection as well as for scoped memory.

Ovm is freely available from http://ovmj.org. The collision detector is available from the authors. RTZen can be obtained from http://doc.ece.uci.edu/rtzen.

References

[1] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-Time Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2006)*, volume 4067 of *Lecture Notes in Computer Science*, pages 124–147, Nantes, France, July 2006. Springer.

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, volume 38, pages 285–298, January 2003.

[3] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbuster, Edward Pla, and David Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. IEEE Computer Society, 2006.

[5] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, February 1988.

[6] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

[7] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[8] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[9] Andrew Borg, Andy Wellings, Christopher Gill, and Ron K. Cytron. Real-time memory management: Life and times. In *18th Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[10] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of Conference on Programming Languages Design and Implementation*. ACM Press, 2003.

[11] Angelo Corsaro and Ron K. Cytron. Efficient memory reference checks for real-time Java. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[12] David Detlefs. A hard look at hard real-time garbage collection. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 23–32, 2004.

[13] Roger Henriksson. *Scheduling Garbage Colection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[14] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, volume 26, pages 271–285, New York, November 1991. ACM Press.

[15] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996. Chapter on distributed collection by Lins.

[16] Arvind S. Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enhancing Real-Time CORBA via Real-Time Java Features. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 66–73, Hachioji, Tokyo, Japan, March 2004.

[17] Jagun Kwon and Andy Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops 2004, LNCS 3292, pp. 33–345*, 2004.

[18] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Joint ACM Java Grande/ISCOPE Conference*, November 2002.

[19] Anders Nilsson and Sven Gestegard Robertz. On real-time performance of ahead-of-time compiled java. In *ISORC*, pages 372–381. IEEE Computer Society, 2005.

[20] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th Eurpean Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[21] Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 378–404, Darmstadt, Germany, July 2003. Springer.

[22] Pekka P. Pirinen. Barrier techniques for incremental tracing. In *ISMM*, pages 20–25, 1998.

[23] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)*, Vienna, Austria, May 2004.

[24] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *International Conference on Compiler Construction (CC)*, pages 304–318, 2001.

[25] Fridtjof Siebert. The impact of realtime garbage collection on realtime java programming. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 33–40, 2004.