Replicating Real-Time Garbage Collector for Java

Tomas Kalibera
Purdue University

ABSTRACT

Real-time Java is becoming a viable platform for real-time applications, bringing new challenges to a garbage collector. A real-time collector has to be incremental as not to cause deadline misses by suspending an application for too long. In particular, a real-time collector has to relocate objects in the heap, incrementally and transparently to the application. This is usually achieved via an indirection that has to be followed on every read and write to the heap.

We present an alternative solution, based on object replication, which does not need any special handling for memory reads, but writes are more expensive: every value is written twice. As writes are less frequent than reads, the total overhead is reduced. With our implementation in a research real-time Java VM and DaCapo, pseudo-jbb, and SPEC JVM 98 benchmarks, we observe an average speed-up of 3%.

A similar technique was implemented in Sapphire, a copying concurrent collector targeting highly parallel systems. Sapphire requires that all accesses to non-volatile shared variables in applications are protected by locks. Our uniprocessor non-concurrent mostly non-copying collector, targeting green-threading embedded systems, does not have this requirement. The mutator barriers supporting our collector are simpler and more predictable.

1. INTRODUCTION

Java is on the rise as a platform for real-time applications. Several real-time Java VMs implementing Real-time Specification for Java (RTSJ) [8] are available [28, 17, 1, 2, 23], and real-time Java has been used to implement applications in avionics [3], shipboard computing [18], industrial control [12] and music synthesis [4, 19]. Although RTSJ provides memory abstractions that can bypass a garbage collector, immortal and scoped memory, these abstractions are very hard to use, and thus commonly applications instead rely on a real-time garbage collector (RTGC), which is not part of the RTSJ.

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.
JTRES'09 September 23-25, 2009 Madrid, Spain
Copyright 2009 ACM 978-1-60558-732-5/09/09 ...$10.00.

Requirements on an RTGC differ for different domains (i.e. hard real-time avionics, soft real-time music synthesis or interactive graphic environments). Every RTGC however must be incremental, not preventing the mutator from running for more than a very small and bounded amount of time. Any collector should control the amount of memory fragmentation, either preventing it or dynamically reducing it, such that a long-running system does not run out of memory due to excessive fragmentation. Known solutions to controlling the fragmentation are preventing it by splitting objects into equally-sized blocks, reducing it by relocating objects, or fighting it by splitting objects into arbitrary-sized blocks. Efficient object relocation is still a challenging task, as demonstrated by the fact that many current real-time Java production VMs do not implement relocation in their incremental RTGCs.

We present a new variant of replication, which allows to ensure heap consistency in the presence of incremental object relocation. The platform for our work is an open-source VM implementing RTSJ and RTGC [23], which targets real-time embedded systems, in particular the RTEMS and LEON architecture used by European Space Agency [20]. The solution is bound to uni-processors, as they still represent the majority of embedded systems. It is also bound to green threading (or some other mechanism allowing fast implementation of atomic barriers).

The contributions of this paper are:

- **Simple replication**: We propose a new variant of replication that improves performance of embedded real-time Java applications. It reduces the overhead imposed on the mutator due to object relocation. Being bound to uni-processors with green threading, our variant has simpler and more predictable barriers than an earlier variant proposed in [16, 14]. In addition, we do not require applications to strictly synchronize accesses to non-volatile fields.
- **Implementation**: We implement an RTGC with replication in Minuteman RTGC framework, allowing further comparisons against additional RTGC configurations supported by the framework.
- **Evaluation**: We empirically compare our solution to Brooks forwarding pointers [9], both with and without arraylets. We use a set of non-trivial application benchmarks: DaCapo [7], pseudo-jbb, and SPEC JVM 98 [26]. The performance speedup is 3%.

The source code of the implementation and scripts used to run the benchmarks can be downloaded from http://www.ovmj.net/replication

1.1 Brooks Forwarding Pointers

A common approach for achieving consistency during object relocation in an incremental collector is via Brooks forwarding pointers [9, 13, 5]. With Brooks forwarding pointers, all memory operations are executed exactly once and always in to-space (Figure 1(a)).

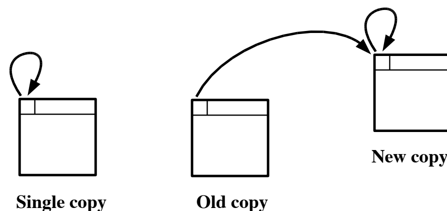
The header of each object is extended by a forwarding pointer, which always points to an up-to-date representation of the object. If an object only exists in a single copy, the forwarding pointer points to itself. Thus, a non-null pointer can be always translated to its up-to-date version unconditionally by following the forwarding pointer. This translation has to be performed before each memory read and write. An example pseudo-code for read and writes is shown in Table 1.

1.2 Our Replication

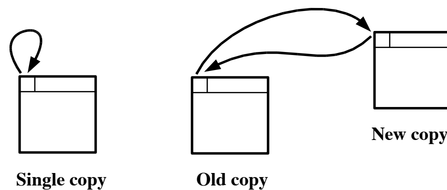
In our version of replication, which is similar to that of [16, 14], a memory read operation does not need any special handling by the mutator. The mutator performs memory read using the pointer it has, ignoring the forwarding pointers. In other words, whenever the mutator can have pointers both to the old and the new location of a relocated object, the two copies are always in sync.

If there are two copies of an object, the forwarding pointer of each copy points to the other copy. If there is only a single copy, the forwarding pointer points to itself (Figure 1(b)).

The mutator has to perform every write twice. First using the pointer it has, and second using the forwarding pointer. The write operation is again unconditional once the pointer is known not to be null: the two writes are always executed, even if to the same location. Note that memory caches typically present in current processors can execute two successive writes of the same value to the same location very quickly. An example pseudo-code for read and writes is shown in Table 1.



(a) Brooks forwarding



(b) Replication

Figure 1: Forwarding pointers.

1.3 Related Work

Replication

Sapphire [16, 14] is a non-parallel concurrent copying collector which implements a replication scheme similar to ours. An old copy of an object has a forwarding pointer to the new copy, but the reverse mapping has to be stored in a hash table.

A read operation of a non-volatile field does not require a barrier. A write operation requires a different barrier at different collection phases. The most complex one is needed during the flip phase, when pointers both to new and old copies are reachable by the mutator. The write barrier will first write using the pointer it has, and then it would enqueue the write operation for the other copy; this involves a branch checking if the pointer points to new or old copy, and then either a pointer dereference or a hash-table lookup. The enqueued write operation is executed before the next synchronization point (acquisition or release of a lock, read or write of a volatile field). Although the fact that the memory is consistent only at synchronization points poses a clear difficulty for applications, the authors argue that their GC only requires applications to conform to Java Memory Model (JMM). Sapphire is targeting highly parallel systems. The main motivation for replication in Sapphire is incrementality of object copying, and thus short collector pauses.

Our collector targets embedded systems with uniprocessors and green threading. The main motivation for replication is speed-up. Our collector thus does not have the synchronization issues: barriers are always atomic, and writes are executed instantly. Also, our barriers (described later in the text) are much simpler than those of Sapphire. In particular, the non-pointer write operation using a non-null pointer has no branch at all. The barriers are designed to be the same during all phases of a GC cycle (including when GC is not running), so they have a more predictable overhead. The fixing of pointers (making them point to new copies of objects) in thread stacks is also simpler: in Sapphire, a single thread's stack may have to be fixed multiple times in each cycle to assure convergence, depending on the mutator activity. Our collector avoids this problem by using a Yuasa barrier for the heap (as we describe later in the text), thus each thread only has to be fixed once regardless of the mutator.

A replication scheme proposed for ML language in [21] requires all reads to be executed in from-space. Writes are also executed in from-space and are logged into a mutation log. The mutation log is later read by the collector, which applies the modifications also to to-space. The solution is proposed for a two-space copying collector and implemented in a three-space generational collector in SML/NJ. By forcing the mutator to use from-space while the collector is copying objects, the solution removes the object copy overhead from mutator write operations and also removes the need for special handling of read operations. The design was highly influenced by the specifics of ML and the SML/NJ garbage collector. In particular, the mutation log was easy to incor-

`f.x=g.y`	
Brooks Forwarding	Replication
`f.fwd.x=g.fwd.y`	`f.x=g.y` `f.fwd.x=g.y`

Table 1: Pseudo-code for memory reads and writes.

porate as the generational collector already had one. The log was processed atomically, which worked well as ML programs had many immutable objects and as soft real-time guarantees were only required. Although the authors claim that making the log processing incremental would be easy, there are finalization issues: the processing may never stop if the mutator is updating frequently [13]. The solution is also different from ours in implementation details related to the language, such as that ML does not have stack and ML programs have a lot of small objects (few bytes).

A technique related to [21] is proposed in [15]. The technique aims at allowing incremental copying of a single object, which is also possible by replication from [21] or by our replication. The technique is however much closer to Brooks forwarding pointers. Both reads and writes are always executed in to-space. It is the responsibility of the mutator to restart an operation (read or write) if the target object has just been copied by the collector, potentially interfering with the operation. The mutator may thus end up both reading twice as well as writing twice, and both the read and write code includes branches. The work is in the context of ML.

Yet another ML collector related to replication is proposed in [11]. The collector uses private thread-local heaps for immutable objects, which are common to ML. No pointer from outside a private heap can point into it, simplifying collection of the private heaps. To enforce this restriction, whenever a write operation would introduce such a pointer, the immutable object is copied into a shared heap. The shared heap also includes mutable objects. This copy operation does not require pointer updating: the local copy of the immutable object is still being used by the particular thread. Forwarding pointers are only needed in the private heaps. The shared heap uses a non-moving mark-sweep collector. The fundamental difference from our replication scheme is that only immutable objects are being copied.

Incremental Object Copying

A common motivation for replication is that it is suitable for incremental copying of a single object. Incremental object copy was however also implemented in specialized hardware [24, 30]. A hardware memory controller that relocates the objects then also directs memory reads and writes to the correct locations.

Our replication barrier supports incremental object copying, which we demonstrate by implementing it. However, in our GC, incremental object copying is not necessary as the GC only relocates small objects. Our GC benefits from the replication solely on performance grounds.

Fighting Fragmentation

The RTGC of Jamaica VM uses fixed-sized blocks of 32 bytes for all allocated objects, including arrays [25]. Larger non-array objects are split into lists of blocks, arrays are represented by blocks organized as trees. This solution trades the external fragmentation for internal fragmentation and execution overhead due to access to the split structures. If the allocator is able to find a contiguous region for array data, the internal tree nodes are not needed. The support for contiguous arrays does not impose any slow-down on the array access barriers. However, a worst-case analysis of a system must assume that the memory is fragmented, and thus arrays have to be allocated as trees.

The RTGC of Sun's RTS [10] uses a split representation for an array or an object only if contiguous representation is not possible due to fragmentation. The splitting does not necessarily have to be into blocks of equal size. It seems that the worst case run-time overhead of this solution can be high, but we are not aware of a study that would analyze it.

In Metronome RTGC [5], regular objects are allocated in contiguous blocks and relocated. Arrays have split representation. Array data are stored in 2K blocks called *arraylets*, which are not relocated. Array meta-data and pointers to arraylets are stored in a contiguous array *spine*. Regular objects larger than 16K are not relocated, but such objects do not really exist in applications in the first place. Objects smaller than 16K are relocated, protected by the Brooks forwarding pointers. Our GC in the configuration we use for this study is similar to Metronome and implements both the Brooks barrier and replication. The commercial implementation of Metronome in IBM WebSphere Real-Time however does not have defragmentation.

Aonix's PERC Java Virtual Machine [2, 22] has a collector with object relocation.

2. MINUTEMAN RTGC FRAMEWORK

We use the Minuteman RTGC framework in the Ovm Java Virtual Machine, which is an open-source RTSJ implementation from Purdue University [23]. Ovm is an ahead-of-time compiling VM, written mostly in Java. At build time, it compiles all Java bytecode of the application, the VM itself, and all needed Java libraries into C, which can be then compiled by a C compiler (GCC). All reflective calls an application could make thus have to be specified at compile time. The main target platform is real-time Linux with x86 processors, but Ovm has been ported to several other platforms, including RTEMS/LEON. The version of Ovm we use for this study has green threads scheduler, which means that all Java threads are executed using a single native thread. Preemption points are inserted at every back-branch. In particular, they are not inserted into barriers, and thus barriers are atomic. An earlier study reported that the latency (time between occurrence of an event and preemption of current Java thread) is below 6 μs [3].

The Minuteman RTGC framework can be configured to implement RTGCs of many different kinds with a range of profiling options. The key configurable features are RTGC scheduling, incrementality, defragmentation, predictability of barriers and representation of arrays. Multiple time-based scheduling modes are supported: slack-based scheduling such as [13], periodic scheduling as in Metronome [5], and a combination of both. The collector can be stop-the-world or incremental. Incrementality can be enabled selectively for individual phases of the collection. Defragmentation is supported using Brooks barrier with atomic object copy and using replication both with atomic and incremental copy. Also, a non-moving collector can be built. Arrays can either be contiguous, or use arraylets. Barriers can be optimized either for throughput, or for predictability.

For this study, we use periodic scheduling, fully incremental collector, predictable barriers, and arraylets. Parameters that we vary are replication/Brooks forwarding, incremental/atomic object copy and arraylets/contiguous arrays.

The collector is a mostly non-copying, mark-and-sweep snapshot-at-the beginning Yuasa style [29] collector with weak tri-color invariant. Further we describe our collector

focusing on aspects important for integration of the replication and the listed parameters that we vary. The core of the algorithm is described in Section 2.3, building on concepts explained in Sections 2.1 and 2.2.

2.1 Memory Allocator

The heap is divided into 2K pages. There is a *small object allocator* for objects smaller than 2K (including object header), *large object allocator* for larger objects, and a special allocator for arrays if arraylets are enabled. Each page is used by a single allocator, or is unused. The small object allocator divides pages into slots of predefined sizes, each page containing only slots of the same size. Freed slots can be re-used, but only by objects of a given size. When the amount of free slots in pages reserved for a particular size is too large, there may not be enough memory for allocation of an object of different size (fragmentation). This is solved by defragmentation, during which mostly free pages are evacuated to mostly full pages reserved for the same size.

The large object allocator has to find a block of free pages large enough for an object it needs to allocate. This operation is very slow, which is why arraylets are supported. Also, external fragmentation may prevent allocating a large object, even if there was enough free memory.

With arraylets enabled, arrays are allocated as shown in Figure 2(a). An array is represented by a *spine*, which contains object meta-data (header), array length, pointers to external arraylets, and optionally also inline arraylets. Each external arraylet takes a whole 2K page, which is fully filled with array data. If the size of array data is not an exact multiple of 2K, there is an internal arraylet included in the spine with a size smaller than 2K. The spine is allocated as a regular object (as a small object or as a larger object, depending on its size). Thus, a very large array still would have a spine represented via a large object. With arraylets enabled, each array in the system has this structure. All array accesses include barriers which direct the access to the required arraylet.

Note that arrays smaller than 2K are always allocated as contiguous, with a single arraylet that is internal in the array spine. The compiler sometimes knows that a particular array is smaller than 2K and can bypass the arraylet pointer calculation. Also, the scheme allows even larger arrays to be contiguous by inlining all arraylets into the spine, which is important for internal VM structures, in particular for large arrays out of the heap.

As the arraylets already introduce one level of indirection to every array access, there is no need for any special barrier for supporting defragmentation of arrays. External arraylets don't move. The only array data that may move is stored in internal arraylets. As only small objects can move, if an array spine moves, it contains at most one internal arraylet (2K is both the arraylet size and the page size). When copying an array spine, we thus redirect the internal arraylet pointer to the new copy of the spine, getting Brooks forwarding with no additional runtime overhead for the mutator (Figure 2(b)). We therefore use this solution even with replication enabled.

A further optimization for allocation of small objects is bump-pointer allocation for the youngest generation. When a fresh page is allocated for objects of a particular size, it is enabled for bump-pointer allocation. Free lists are only created at sweep time if some, but not all objects, in the page die.

2.2 Barriers

The collector requires several barriers to be inserted into mutator code. They allow incremental collection, object relocation, and split representation of arrays using arraylets. In addition to the barriers, the runtime code of the VM also requires certain modifications, such as array copy functions, interface to native code, I/O, and reflective access to data structures in the heap.

In the following text, a *clean* pointer is a pointer that points to an up-to-date location of an object and a *dirty* pointer points to the old location of a relocated object. Object colors are white, grey, black. During marking, reachable scanned objects are *black*, reachable not-yet-scanned objects are *grey* (they are stored in a list of reachable not-yet-scanned objects), and not reached objects are *white*. After marking, no objects are grey, white objects are unreachable, and black objects are a superset of reachable objects. By a white (grey,black) pointer we understand a pointer to a white (grey,black) object. By "marking" we always mean marking if the object is white.

Read Barrier

With Brooks forwarding pointers, every read from an object has to follow the forwarding pointer of the object, as shown in Table 1. The same applies to arrays, if arraylets are disabled.

With arraylets enabled, an array read operation has to (a) calculate the arraylet index based on array element index, (b) calculate offset within the arraylet, (c) read the arraylet pointer, (d) read the requested value from the given arraylet and offset. The arraylets are designed such that these calculations only need integer division and remainder. In particular, branches are not needed.

If the value being read is a pointer, it is not forwarded in our collector, as we use lazy forwarding.

Non-Pointer Write Barrier

With Brooks forwarding pointers, every write to an object has to be preceded by dereferencing of the forwarding pointer (Table 1). With replication, every write is executed twice, first on the current pointer location, and then on forwarded location (Table 1). When arraylets are disabled, this applies also to arrays.

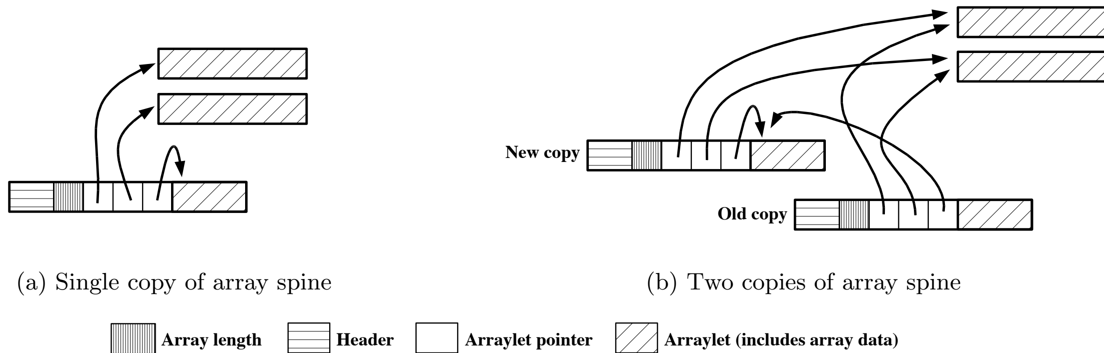
When arraylets are enabled, the write has to be redirected to the correct address within the correct arraylet. The same calculation as for the read barrier is used.

If a pointer type is being written, the barrier is more complex as we describe later.

Pointer Comparison Barrier

Sometimes, the mutator can have pointers to two different copies of the same object. Pointer comparison barrier is thus needed to hide the fact that objects can have multiple copies from the mutator. Note that the barrier is only needed when both of the compared pointers are non-null.

The barrier is optimized for the typical case when both objects being compared only have a single location. The optimization relies on an observation that if the pointers point to the same copy of an object, they indeed represent the same object and should compare as equal. The comparison is first attempted on unforwarded objects. If not equal,



(a) Single copy of array spine

(b) Two copies of array spine

Figure 2: Array representation with arraylets.

one of the pointers is forwarded and another comparison is made. If not equal, the other pointer is forwarded as well and the last comparison is made. If not equal, we know that the pointers represent different objects, both with Brooks forwarding and with replication. With replication, we can actually skip the second pointer forwarding[1]. Pseudo-code for the barrier is shown in Figure 3. It already includes optimizations explained later in this section.

Pointer Write Barrier

Pointer write barrier has to do additional work compared to the non-pointer write barrier. To let the collector do its job, it is (sometimes) needed to update pointers before they are written. Sometimes, a pointer that is being overwritten has to be marked grey (Yuasa barrier [29]). And sometimes, a new pointer being written has to also be marked grey (Dijkstra barrier). When these operations are needed depends on the current state of the GC. However, in predictable configuration and atomic object copy, these operation are always performed even when not needed, targeting predictable overhead. One exception is needed to this rule: with incremental object copy, the pointer updating on writes has to be disabled when objects are being relocated.

The Yuasa and Dijkstra barriers ensure that no reachable object is accidentaly hidden from the collector and reclaimed. To do this, the barriers arrange that a pointer modification never breaks the weak tri-color invariant, which states that if a white object is pointed to by a black object, it is also reachable from a grey object through a chain of white objects. The Yuasa barrier marks the pointer being overwritten to make sure that it is not white, and thus it does not matter if it is reachable from "a grey object through a chain of white objects" (see pseudo-code in Figure 3). The Yuasa barrier cannot however easily mark pointers that are being overwritten on the stack and in local variables. This is why we need the Dijkstra barrier. The Dijkstra barrier, by marking a new pointer value when written to the heap (the value always comes from stack in Java), ensures that a black object can never point to a white object, and thus it again does not matter if such a white object would be reachable from "a grey object through a chain of white objects" (see pseudo-code in Figure 3). Even with Dijkstra barrier, the stack of a single thread has to be scanned atomically. This is however a very fast operation on Ovm, as pointers are stored in a separate pointer stack. [6].

Pointers being written have to be updated to point to new locations. Otherwise, dirty pointers could spread without control in the heap, making it impossible to have the pointers in the heap completely cleaned by the collector. With Brooks forwarding, pointer update is simply an indirection of the forwarding pointer. With replication, the barrier uses the *old bit* in object header which identifies the old location. This bit is, in addition to the barrier, only used by marking and sweeping code. The mutator only accesses it in this barrier, which is not as frequent as other barriers.

Pseudo-code for the barrier is shown in Figure 3. The code in the figure is indeed simplified for readability. The full version can be found in the source code, which is publicly available.

Optimizations

Barriers are implemented in Java, translated to C by Ovm's Java-to-C compiler, and are always inlined when compiled by GCC (by the GCC inliner). Compiling ahead of time and knowing that barriers will always be inlined allows us to specialize the barriers based on properties of pointers known by static analysis to the Java-to-C compiler, based on the context in which a particular barrier instance is used. The Java-to-C compiler automatically adds a bitmap of assertions (an integer constant) known to hold for any pointer passed to a barrier. The barrier code, written in Java, can then freely use branches for these assertions. The conditions in the branches will, at every barrier instance, be turned into bit-operations on constants, and thus the branches will always be eliminated by the GCC compiler.

Most usable assertions for the barrier code are `KNOWN_NONNULL` (pointer is known not to be null) and `KNOWN_NULL`, which allow to significantly optimize all of the mentioned barriers. Ovm also has a special memory area for static data named *image*, which is treated specially by the GC – objects are neither reclaimed nor moved in the image. With the support for assertions on pointers, it is often known by static analysis that a pointer is in the image, and thus cannot have multiple copies. Sometimes, the Java-to-C compiler also knows statically that an array is so small that it can only have a single arraylet, which simplifies the array access barrier. The use of assertions in the barrier code is shown in Figure 3. We omit the assertions related to the image, as they are rather specific to our VM.

To speed up compilation of large applications, we have

[1] The experiments we show later had been run before this optimization was implemented.

identified the most common combinations of assertion values for the pointer write barrier and erased (zeroed) the assertions known for the less frequent combinations. By a wrapper method call at Java level for each specialized combination of the assertions and special pragmas controlling inlining, we made the GCC do most of the work only once for each combination, as opposed to repeat it at every barrier call site. The combinations could be easily reconfigured by modifying the Java code of the GC.

This said, further compiler optimizations could be implemented to elide unnecessary forwarding in Brooks-style read and write barriers, especially if we further exploit the presence of the green-threads scheduler and the knowledge where yield points are inserted.

2.3 Collection Cycle

The algorithm and invariants on clean and dirty pointers are different for atomic and incremental object copy. We first describe the GC cycle with atomic object copy, both for Brooks forwarding and replication.

Atomic Object Copy

1. `waitUntilMemoryIsScarce` The GC thread sleeps. Pointers are black, objects are allocated black. Pointers in the heap and stacks (and local variables) can be both clean and dirty. Dirty pointers can exist because objects might have been moved during defragmentation at the end of the previous cycle. For every relocated object, there can be pointers to both its old and new location.

2. `scanStacks` The meaning of black and white is inverted, making all objects white. Allocation color is again made black. Stacks of threads are scanned and discovered pointers are marked grey (this also stores the pointers into a list of reachable not-yet-scanned pointers). The mark operation always marks the up-to-date location (cleans its copy of the pointer before dereferencing it). The pointers in the stacks however cannot be cleaned yet, as the heap is dirty.

3. `markAndCleanHeap` For each pointer from the list of grey (reachable not-yet-scanned) pointers, the target object is marked black, scanned for pointers, and removed from the list. The scanning involves cleaning each pointer in the heap representation of the object and marking it grey if it is still white. The list implements a FIFO queue, and thus this is a depth-first-search traversal. Note that breadth-first-search traversal would typically require a longer list.

4. `cleanStacks` As all pointers in the heap are clean, objects on the stack can now be cleaned. This is done incrementally similarly to stack scanning.

5. `sweep` All pointers in reachable objects and on the stacks are clean. All white objects are unreachable, and thus are reclaimed. With Brooks forwarding, the old locations of live relocated objects are reclaimed naturally as they are white (unreachable). With replication, the forwarding pointer of the new copy has to be updated when the old copy is being reclaimed, so that the mutator stops writing to the old copy. After sweep, all objects in the heap are black. All pointers are clean.

6. `defragment` If the free memory is below a given threshold, defragmentation is started, relocating objects from less occupied pages to more occupied pages. After the next sweep, the evacuated pages will become free, unless they were in the meantime used by the mutator. Forwarding pointers are updated atomically with copying.

Incremental Object Copy

The invariants on clean an dirty pointers differ from atomic object copy as follows

1. `waitUntilMemoryIsScarce` There may be relocated objects in the system, but all pointers always point to the old locations. Clean pointers are thus only those that point to objects with a single copy. Pointer updating in pointer write barrier is not active.

2. `scanStacks` Pointer updating is activated. From now on, clean pointers to objects with two copies can spread both to the heap and to stacks. It is important that newly allocated objects will from now on only contain clean pointers.

3. `markAndCleanHeap` Pointer updating is still active. At the end, all reachable objects in the heap are clean.

4. `cleanStacks` Pointer updating is still active. At the end, all reachable pointers in the heap and on the stacks are clean. The pointer updating is thus deactivated.

5. `sweep` Pointer updating is still inactive.

6. `defragment` It is crucial that pointer updating is still inactive, and thus the mutator does not have pointers to new locations of relocated objects. The size of the increments by which objects are copied is controlled by a build-time constant (number of bytes that can be touched atomically). Note that external arraylets are not copied (there is no need to do so as arraylet size is the same as page size). Array spines with an internal arraylet have to be copied atomically, because as we have shown they always use Brooks forwarding.

3. EVALUATION

The goal of the evaluation is to empirically compare performance of Brooks forwarding pointers and replication in Ovm with Minuteman RTGC framework.

3.1 Benchmarks

We run 8 benchmarks from the DaCapo [7] suite (antlr, bloat, fop, hsqldb, luindex, lusearch, pmd, and xalan), the pseudo-jbb modification of the SPEC JBB 2005 [27], and 7 benchmarks from the SPEC JVM 98 [26] suite (compress, db, jack, javac, jess, mpegaudio, and mtrt). These are all application benchmarks and most of them even use libraries and code base that is/was being deployed and tested in real applications. The benchmarks are unfortunately all non-realtime, but we use them as we do not have access to any sufficiently complex real-time application benchmarks. We use periodic time-based RTGC scheduling, so that the benchmarks can be run unmodified even though they do not have slack.

```
// with assignment "f.x = y", where f.x and y are pointers to objects
//   oldPtr is f.x, newPtr is y
//   aNewPtr are statically known assertions about y
//   storeToAddr is address at which f.x is stored
void pointerWriteBarrier( Address oldPtr, Address newPtr, int aNewPtr, Address storeToAddr ) {
    /* Yuasa */
    if (oldPtr != null) {
        oldPtrUpd = updateNonNullPtr(oldPtr);
        markFromBarrier(oldPtrUpd);
    }
    if ( KNOWN_NONNULL(aNewPtr) || ( !KNOWN_NULL(aNewPtr) && newPtr!=null )) {
        /* Dijkstra */
        newPtrUpd = updateNonNullPtr(newPtr);
        markFromBarrier(newPtrUpd);
        /* The write itself, with pointer updating */
        if (REPLICATION && INCREMENTAL_OBJECT_COPY && !updatingPointersOn) {
            storeToAddr.store(newPtr);
            forwardNonNullPtr(storeToAddr).store(newPtr);
        } else {
            storeToAddr.store(newPtrUpd);
            if (REPLICATION) { forwardNonNullPtr(storeToAddr).store(newPtrUpd); }
        }
    } else {
        /* newPtr is null */
        storeToAddr.store(null);
        if (REPLICATION) { forwardNonNullPtr(storeToAddr).store(null); }
    }
}

// with comparison "A==B", where A and B are pointers to objects
// ptrA is A, aPtrA are statically known assertions about A
// returns true iff "A==B"
boolean pointerComparisonBarrier( Address ptrA, Address ptrB, int aPtrA, int aPtrB ) {
    if (KNOWN_NULL(ptrA) && KNOWN_NULL(ptrB)) {
        return true; /* null == null */
    }
    if ((KNOWN_NONNULL(ptrA) && KNOWN_NULL(ptrB)) || ((KNOWN_NULL(ptrA) && KNOWN_NONNULL(ptrB))) {
        return false; /* null != !null */
    }
    if (KNOWN_NULL(ptrA) || (!KNOWN_NONNULL(ptrA) && ptrA==null)) { /* A is null */
        return (ptrB==null);
    }
    if (KNOWN_NULL(ptrB) || (!KNOWN_NONNULL(ptrB) && ptrB==null)) { /* B is null */
        return (ptrA==null);
    }
    if (ptrA==ptrB) { return true; }
    ptrAFwd = forwardNonNullPtr(ptrA);
    if (ptrAFwd==ptrB) { return true; }
    if (REPLICATION) { return false; } else {
        ptrBFwd = forwardNonNullPtr(ptrB);
        return (ptrAFwd==ptrBFwd);
    }
}

Address updateNonNullPtr( Address ptr ) {
    if (REPLICATION) {
        if (ptr.isOldBitSet()) { return forwardNonNullPtr(ptr); }
        else { return ptr; }
    } else { return forwardNonNullPtr(ptr); } /* Brooks forwarding */
}
```

Figure 3: Pseudo-code for pointer write and pointer comparison barriers.

3.2 Configurations

We evaluate the following configurations, identified by features present (A stands for arraylets, B for Brooks forwarding, R for replication, I for incremental copy):

·A·	Arraylets, no defragmentation
B··	Brooks forwarding, atomic copy
BA·	**Brooks forwarding, arraylets, atomic copy**
R··	Replication, atomic copy
R·I	Replication, incremental copy
RA·	**Replication, arraylets, atomic copy**
RAI	Replication, arraylets, incremental copy

The two highlighted configurations, RA· and BA·, are the two one would really use with our collector, and thus allow the most realistic performance comparison of replication and Brooks. These configurations are most natural with our collector, because objects larger than 2K are never defragmented. Thus, while non-array objects larger than 2K don't exist in real applications, we need arraylets to avoid fragmentation due to large arrays. Since only objects smaller than 2K are copied, there is no need for incremental object copy. The other configurations allow to quantify overheads of replication and incremental object copy, which might apply when our replication is incorporated into different types of collectors. They also show overheads of the collector.

3.3 Results

The percentage overheads of replication of Brooks forwarding are shown in Table 3. With arraylets, the replication is on average by 3% faster than Brooks forwarding. Without arraylets, it is faster by 4.3%. The maximum speedup is 8.6% with mtrt benchmark from SPEC JVM 98. The maximum and only slowdown is with pmd benchmark from DaCapo (2.6%).

Confidence intervals for mean execution time with different configurations and benchmarks are shown in Table 2. With the exception of the bloat benchmark from the DaCapo suite and pseudo-jbb, the results are very stable. The table columns are ordered such that pairs of configurations interesting for comparison are in adjacent columns.

3.4 Ratios of reads and writes

The ratios of read and write operations in the Dacapo and SPEC JVM 98 benchmarks are shown in Table 4. On geometric average, there were 5.2 times more reads than writes. However, the overheads of individual barriers for reads and writes depend on the context in which the barriers are inserted, as the context determines the efficiency of optimizations performed by the Java-to-C and GCC compilers. Also, the impact of replication on the total execution time also depends on the actual rate of the read/write operations in the benchmark.

3.5 Comparability to Products

To verify that the performance of our VM is within a reasonable margin of state-of-the-art production systems, we run some of the benchmarks also with IBM WebSphere Real-Time. We use both the hard real-time version (WRT) and the soft real-time version (SRT) of the product. Both SRT and WRT have a Metronome RTGC with a scheduler that we configured with the same parameters as Ovm (10 ms window, 500 μs quantum, 1 ms maximum pause time).

	RA·/ BA·	R··/ B··	RAI/ BA·	R·I/ B··
Antlr	-2.3	-2.7	2.9	-4.9
Bloat	-6.1	-11.8	-5.2	-5.5
Fop	-2.4	-3.5	-2.1	-3.4
Hsqldb	-3.6	-4.4	-4.2	-3.7
Luindex	-3.0	-4.0	-0.4	-1.3
Lusearch	-1.8	-5.6	0.1	-5.2
Pmd	2.6	-4.5	-0.5	-6.1
Xalan	-3.7	-3.8	-3.9	-3.0
Geo-mean	-2.6	-5.1	-1.7	-4.1
Pseudojbb	-2.9	-4.5	3.2	-8.3
Compress	-4.4	13.8	-4.5	-7.3
Db	-2.5	-3.8	-2.8	-3.6
Jack	-1.1	-5.9	-1.3	-8.0
Javac	-1.7	-6.4	-1.8	-5.2
Jess	-3.5	-6.5	-0.1	-5.7
Mpegaudio	-3.0	-6.4	-3.2	-7.6
Mtrt	-8.6	-7.0	-7.8	-8.2
Geo-mean	-3.6	-3.4	-3.1	-6.5
Geo-mean	**-3.0**	**-4.3**	**-2.0**	**-5.4**

Table 3: Percentage overhead of replication over Brooks forwarding. In the natural configuration for our collector (first column), the speedup is 3%.

	FR/FW	AR/AW	R/W
Antlr	8.4	2.3	6.5
Bloat	4.5	6.3	4.6
Fop	6.4	5.4	6.2
Hsqldb	6.6	7.0	6.7
Luindex	5.3	1.7	3.8
Lusearch	4.7	2.0	4.0
Pmd	3.2	2.2	3.0
Xalan	6.8	3.7	5.8
Compress	5.0	2.2	4.0
Db	15.4	7.6	11.7
Jack	4.5	3.0	4.1
Javac	4.4	3.4	4.2
Jess	7.5	5.8	7.1
Mpegaudio	6.0	6.1	6.0
Mtrt	4.6	4.5	4.6
Geo-mean	5.8	3.8	**5.2**

Table 4: Ratio of reads and writes (F=field, A=array, R=read, W=write).

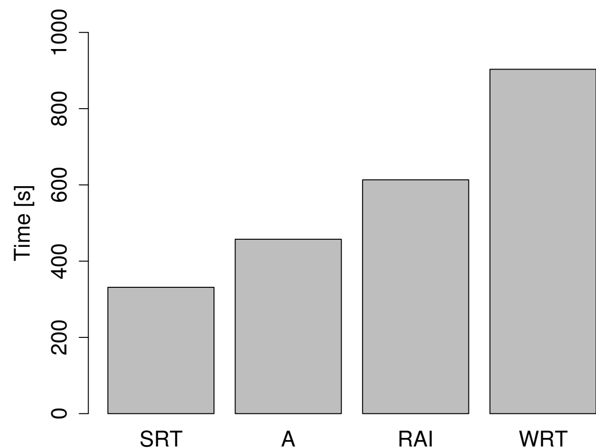


Figure 4: Pseudo-jbb with 2x400,000 transactions, mean execution time.

	RAI	RA·	BA·	·A·	B··	R··	R·I
Antlr	30.72± 0.99	29.15± 0.05	29.85± 0.03	22.23± 0.06	29.47± 0.1	28.67± 0.05	28.04± 0.09
Bloat	472.36±30.17	467.96±29.46	498.39±27.68	298.92±19.87	508.12±28.55	448.13±28.03	480.09±27.78
Fop	6.21± 0.01	6.19± 0.01	6.34± 0.01	5.12± 0.01	6.22± 0.01	6.0 ± 0.01	6.01± 0.01
Hsqldb	21.02± 0.04	21.17± 0.05	21.95± 0.04	16.43± 0.03	20.69± 0.06	19.77± 0.04	19.93± 0.03
Luindex	31.26± 0.02	30.47± 0.04	31.4 ± 0.08	24.59± 0.02	29.44± 0.03	28.27± 0.1	29.05± 0.03
Lusearch	92.12± 0.03	90.41± 0.17	92.04± 0.08	69.24± 0.09	109.94± 0.84	103.81± 0.8	104.22± 0.73
Pmd	77.52± 0.2	79.92± 0.29	77.91± 0.11	64.27± 0.26	77.36± 0.36	73.86± 0.26	72.67± 0.23
Xalan	718.89± 2.57	719.79± 1.95	747.76± 2.19	650.72± 2.78	734.19± 2.02	706.38± 1.12	712.39± 1.37
Pseudojbb	613.3 ± 2.06	577.35± 1.56	594.36± 1.61	457.5 ±16.57	615.11± 1.48	587.38± 3.18	564.2 ±20.17
Compress	8.09± 0.02	8.1 ± 0.02	8.47± 0.02	7.74± 0.02	7.55± 0.01	8.59± 0.01	7.0 ± 0.02
Db	13.0 ± 0.01	13.05± 0.01	13.38± 0.02	11.75± 0.03	12.94± 0.04	12.45± 0.03	12.48± 0.03
Jack	11.71± 0.01	11.74± 0.01	11.87± 0.0	9.03± 0.01	12.34± 0.01	11.61± 0.01	11.35± 0.01
Javac	10.11± 0.03	10.12± 0.05	10.3 ± 0.05	7.4 ± 0.02	10.23± 0.03	9.58± 0.04	9.7 ± 0.04
Jess	10.62± 0.05	10.26± 0.01	10.63± 0.01	7.68± 0.01	10.45± 0.01	9.77± 0.01	9.85± 0.02
Mpegaudio	9.03± 0.02	9.05± 0.03	9.33± 0.02	7.93± 0.01	7.93± 0.01	7.42± 0.02	7.33± 0.02
Mtrt	3.33± 0.01	3.3 ± 0.01	3.61± 0.01	3.03± 0.03	3.41± 0.0	3.17± 0.0	3.13± 0.0

Table 2: Benchmark execution times in seconds, averages with 95% confidence intervals.

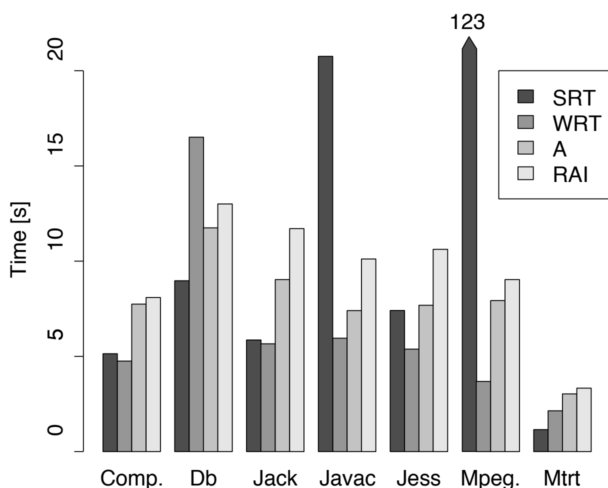


Figure 5: SPEC JVM 98, mean execution time.

Unfortunately, we were not able to run the Dacapo benchmarks with the amount of physical memory available on our system (Ovm could run, because it needed less memory than SRT and WRT).

The results for the pseudo-jbb benchmark are shown in Figure 4. The benchmark run 2 warehouses and 400,000 transactions, which is the combination that would have been reported by the original SPEC JBB 2005 benchmark on our system. As we checked on several other configurations, different numbers of warehouses and transactions lead to very different performance results. We used the same GNU Classpath implementations of Java collection classes with Ovm, SRT, and WRT, to focus the comparison more on the VM and the GC itself rather than the class libraries. With this setting, the ·A· configuration is faster than WRT, but slower than SRT. Note that the ·A· configuration is by its RTGC features closest to SRT and WRT, as SRT and WRT use arraylets, but not defragmentation. The RAI configuration is, however, still faster than WRT in this benchmark.

The results for SPEC JVM 98 benchmarks are shown in Figure 5. When running WRT and SRT, we had to enable the synchronous non-incremental GC on out-of-memory condition to be able to run the SPEC JVM 98 benchmarks on our system with the same heap size we used with Ovm. We used the original IBM class libraries. Assuming that the poor results of SRT with mpegaudio and javac were due to a bug or misconfiguration, let's for every benchmark compare Ovm to the faster of SRT and WRT. At worst, Ovm was 2.6x slower, it was against SRT with mtrt. On geometric average, it was 1.7x slower. We believe this is still an acceptable margin with which relative performance measurements within Ovm can be regarded as credible.

3.6 Conclusion

We present a defragmenting real-time garbage collector for Java, which uses replication instead of Brooks forwarding pointers. Compared to Brooks forwarding, replication is faster. In our implementation within Ovm and the Minuteman RTGC framework, we have observed 3% performance improvement on average. We measured with DaCapo, pseudo-jbb, and SPEC JVM 98 benchmarks. Our version of replication is a natural choice for a uniprocessor VM with green threading model, which is in turn a natural choice for embedded systems.

Our work complements Sapphire [16, 14], a replicating garbage collector for highly parallel systems. In Sapphire, the support for parallelism comes at a high price of the memory being consistent only at accesses to volatile fields and releases/acquisitions of locks. This limitation is argued to be in-line with Java Memory Model (JMM), but it indeed at least complicates development and use of legacy applications. Our collector allows replication on uni-processors (with green threading) without this limitation. Moreover, the barriers involved are much simpler and incur more repeatable overhead.

Acknowledgments

The code of the RTGC is based on an earlier non-defragmenting non-arraylet version written by Filip Pizlo. The author would like to thank Bertrand Delsart and Jan Vitek for their valuable comments and suggestions that have helped to improve this paper.

4. REFERENCES

[1] AICAS. The Jamaica virtual machine. http://www.aicas.com.

[2] AONIX. The PERC virtual machine. http://www.aonix.com.

[3] Austin Armbruster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1), 2007.

[4] Joshua Auerbach, David F. Bacon, Florian Bömers, and Perry Cheng. Real-time music synthesis in Java using the Metronome garbage collector. In *Proceedings of the International Computer Music Conference (ICMC)*, 2007.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003.

[6] Jason Baker, Antonio Cunei, Tomas Kalibera, Filip Pizlo, and Jan Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurrency and Computation: Practice and Experience*, 2009.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2006.

[8] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real-time specification for Java. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 2003.

[9] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP)*, 1984.

[10] Eric J. Bruno and Greg Bollella. *Real-Time Java Programming with Java RTS*. Prentice Hall, 2009.

[11] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1993.

[12] Sven Gestegard Robertz, Roger Henriksson, Klas Nilsson, Anders Blomdell, and Ivan Tarasov. Using real-time Java for industrial robot control. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES)*, 2007.

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

[14] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande (JGI)*, 2001.

[15] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. *SIGPLAN Not.*, 28(7):73–82, 1993.

[16] Lorenz Huelsbergen and James R. Larus. Sapphire: copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003.

[17] IBM. WebSphere Real Time. http://www.ibm.com/software/webservers/realtime.

[18] IBM. DDG1000 Next generation navy destroyers. http://www.ibm.com/press/us/en/pressrelease/21033.wss, 2007.

[19] Nicolas Juillerat, Stefan Müller Arisona, and Simon Schubiger-Banz. Real-time, low latency audio processing in Java. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007.

[20] Tomas Kalibera, Marek Prochazka, Filip Pizlo, Martin Decky, Jan Vitek, and Marco Zulianello. Real-time Java in space: Potential benefits and open challenges. In *Proceedings of Data Systems in Aerospace (DASIA)*, 2009.

[21] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices, 1993.

[22] Kelvin Nilsen. Differentiating features of the PERC virtual machine. http://www.aonix.com/pdf/PERCWhitePaper_e.pdf, 2009.

[23] Purdue. The Ovm virtual machine. http://www.ovmj.org.

[24] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES)*, 2008.

[25] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES)*, 2007.

[26] SPEC. SPECjvm98 benchmarks, 1998.

[27] SPEC. SPECjbb2000 benchmarks, 2005. http://www.spec.org/jbb2005.

[28] Sun Microsystems. Sun java real-time system. http://java.sun.com/javase/technologies/realtime, 2008.

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

[30] Martin Zabel, Thomas B. Preuber, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded java microarchitecture. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, 2007.