Replicating Real-Time Garbage Collector

Tomas Kalibera*

Purdue University, West Lafayette, IN 47907, USA; Charles University, Prague, 147 00, Czech Republic

SUMMARY

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, if a real-time collector has to relocate objects in the heap, it must do so 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 4% on a Linux/x86 platform.

Compared to an existing technique for parallel systems, our technique targets uni-processor green-threading embedded systems, allowing us to design simpler and more predictable mutator barriers. Thanks to cheap synchronization provided by green threading, our technique features atomic writes to object replicas, providing programmers with the natural behavior in face of unprotected access to shared variables.

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 [29, 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

†A version of this work was presented at the 7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), September 23–25, 2009, Madrid, Spain.

*Correspondence to: Tomas Kalibera, Department of Distributed and Dependable Systems, Charles University, Malostranske nam. 25, 118 00 Praha 1, Czech Republic

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 at the time of this writing still not part of the RTSJ.

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, which 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 speed-up is 4% as measured on a Linux/x86 system.

1.1. Terminology

We use the following terms related to garbage collection and Java. Most of these are common to the garbage collection literature:

Back-branch. Conditional or unconditional jump instruction in the Java byte-code with jump target at a numerically lower address than the jump instruction. Due to the Java compiler design, loops at Java source-code level correspond to back-branches at byte-code level.
Barrier. Piece of code inserted by the compiler at particular mutator operation related to the heap, i.e. reading or writing object fields or array elements, comparing objects, etc.

Black object. Object identified (or assumed) to be reachable, which has already been processed during object graph traversal.
Collector cycle. Unit of collection that includes all steps necessary to identify garbage and reclaim it for further allocation. All collector activity is formed by repeated collector cycles.
Collector pause. Time interval during which the mutator has to stop due to collector activity.
Collector phase. A particular part of collector cycle.
Concurrent collector. Collector that can run concurrently with mutator (on multiprocessor).
Copying collector. Collector that copies (relocates) objects as a part of its operation.
Dijkstra barrier. Write barrier for (over)writing pointer values. The object pointed to by the new pointer value is marked grey.
From-space. Region of the heap from where the collector copies objects.
Green threading. Java threads are implemented by the VM on top of a single operating system thread, giving the VM full control over preemption and atomicity. Can only use one CPU.
Grey object. Object identified (or assumed) to be reachable, which has not yet been processed during object graph traversal.
Incremental collector. A collector that does not need to stop the mutator for the whole GC cycle. On some collector designs, it means working in preferably short increments. On others, including ours, it means being quickly interruptible at almost any time, while making forward progress.
Mutator. The Java application. It mutates the heap, generating work for the collector.
Parallel collector. Collector that parallelizes some of its tasks (on multiprocessor).
Read barrier. Barrier inserted at points where the mutator reads from memory.
Stop-the-world collector. Collector that runs its cycles atomically, blocking the mutator.
To-space. Region of the heap which is a target for object copying performed by the collector.
Weak tri-color invariant. An invariant of the object graph as maintained by the collector. It 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.
White object. Object not yet processed during object graph traversal. Nothing is known or assumed about the object reachability.
Write barrier. Barrier inserted at points where the mutator writes to memory.
Yuasa barrier. A write barrier for overwriting pointer values. The object pointed to by the old pointer value is marked grey.

1.2. 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).

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 this copy. 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.

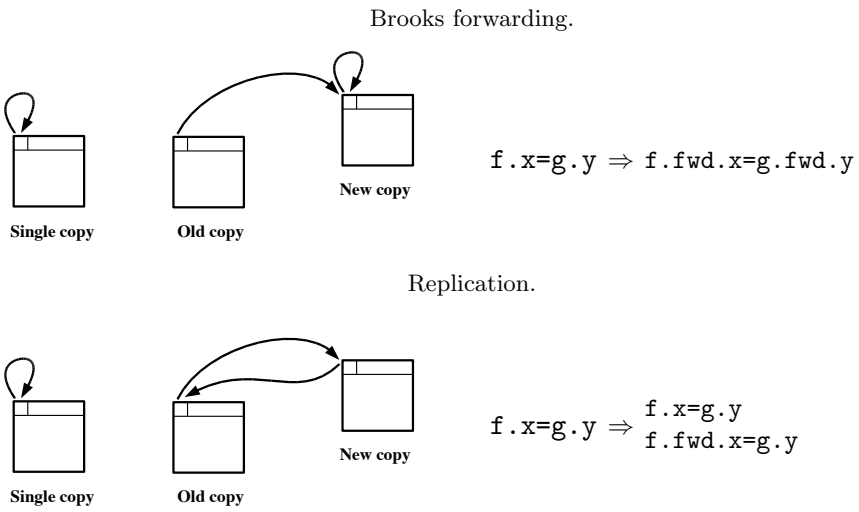


Figure 1. Forwarding pointers with pseudo-code for memory reads and writes.

1.3. 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 this copy (Figure 1).

The mutator has to perform every write twice. First using the pointer it has, and second using the forwarding pointer. The write operation via a non-null pointer is unconditional: the two writes are always executed, even if to the same location. Note that memory caches typically present in current processors allow to execute two successive writes of the same value to the same location very quickly.

1.4. 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 the new or the old copy, and then either a pointer dereference or a hash-table look-up. 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 the Java Memory Model (JMM). Sapphire is targeting 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 uni-processors 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 have the same overhead during all phases of a GC cycle (including when GC is not running). Our collector is snapshot-at-the beginning with a Yuasa barrier, which allows us to scan every thread's stack only once during marking. Sapphire does not have a Yuasa barrier. To ensure that all reachable objects have been marked it thus has to keep re-scanning thread stacks until it makes a pass through all the stacks without discovering any white object.

A replication scheme proposed for the 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 for 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 incorporate as the generational collector already had one. The log was processed atomically, which worked well, because ML programs had many immutable objects and 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 the heap frequently [13]. The solution is also different from ours in implementation details related to the language, i.e. ML does not have stack and ML programs tend to have object size distributions different from Java programs (they in general use smaller objects) [†].

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 than ours. 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

[†]More information on object sizes typical for SML/NJ and Java programs can be found in [7, 28].

writing twice, and both the read and write code include 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 is collected by 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 without specialized hardware. In hardware solutions [24, 31], a specialized hardware memory controller both relocates the objects and directs memory access operations 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, while 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, the worst-case analysis must assume that the memory is fragmented, and thus arrays be always 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 the Metronome RTGC [5], regular objects are allocated in single 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 are also unlikely to be present in Java applications. Objects smaller than 16K are relocated, being 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 support defragmentation.

Aonix's PERC Java Virtual Machine [2, 22] has a copying as well as mark-sweep collector, which take turns in collecting individual memory regions. The VM thus supports object relocation.

2. Minuteman RTGC Framework

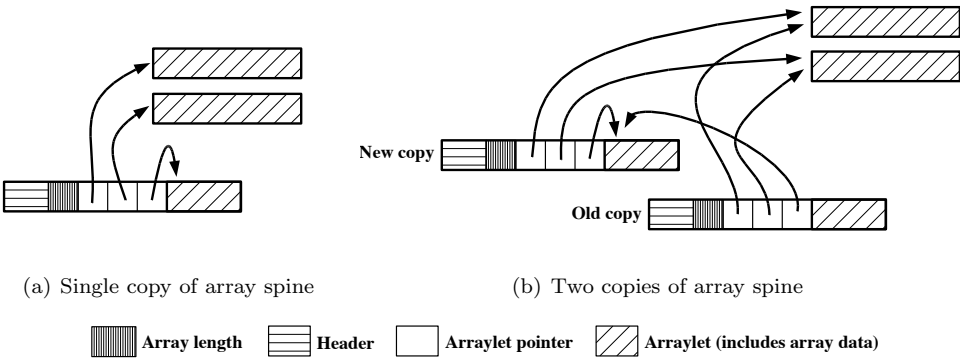
We use the Minuteman RTGC framework in the Ovm Java Virtual Machine, which is an open-source RTSJ implementation from Purdue University [23]. Ovm is an ahead-of-time compiling VM, written mostly in Java. At build time, it compiles all Java bytecode of the application, the VM itself, and all needed Java libraries into C source, which is then compiled by a C compiler (GCC). All reflective calls an application is allowed to 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, because barriers do not have back-branches, and thus barriers are atomic. An earlier study reported that the latency (time between occurrence of an event and preemption of the current Java thread) is below 6 μs [3]. If needed, preemption points could also be inserted into method prologues and/or epilogues such as in the Jikes Research VM, with only minor changes to Ovm.

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 like 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. 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 mostly non-copying, mark-sweep, snapshot-at-the-beginning Yuasa style [30] with weak tri-color invariant. Further we describe our collector focusing on aspects important for 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 the same 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.



(a) Single copy of array spine (b) Two copies of array spine

Figure 2. Array representation with arraylets.

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 an inline arraylet. 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 generate simpler access code. Also, the scheme allows even larger arrays to be contiguous b by placing all external arraylets right after the spine, which is important for internal VM structures, in particular for large arrays out of the heap. These contiguous large arrays do not move.

As the arraylets already introduce one level of indirection to every array access, there is no need for any special barrier to support defragmentation of arrays. External arraylets don't move. The only array data that may move is stored in internal arraylets. However, there is at most one internal arraylet per array and only small objects move, thus the amount of data moved is still bounded by 2K. When moving an array spine, we 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. Together with VM runtime modifications (Section 2.4), they allow incremental collection, object relocation, and split representation of arrays using arraylets.

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. In the following description, marking an object grey is always conditional: the object becomes grey only if it was still white. Marking an object black is unconditional.

Read Barrier. With Brooks forwarding pointers, every read from an object has to follow the forwarding pointer of the object. 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 with a constant divisor that is a power of two (the arraylet size). In particular, branches are not needed. If the value being read is a pointer, it is not forwarded in our collector (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. With replication, every write is executed twice, first using the current pointer location, and then using the forwarded location. 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 from the mutator the fact that objects can have multiple copies. 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, 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

```
// 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);
    }
}
```

Figure 3. Pseudo-code for pointer comparison barrier.

skip the second pointer forwarding. 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, otherwise dirty pointers could spread without control in the heap. To support correct identification of the reachable objects, sometimes a pointer that is being overwritten has to be marked grey (Yuasa barrier [30]). 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 operations are always performed even when not needed, targeting predictable overhead.

The Yuasa and Dijkstra barriers ensure that no reachable object is accidentally 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 4). 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 4). With the Dijkstra barrier, the mutator can run in between of scanning stacks of different threads. In Ovm, scanning a single thread stack is atomic. The operation is very fast, because pointers are stored in a separate pointer stack. [6].

```
// 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); }
    }
}

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

Figure 4. Pseudo-code for pointer write barrier.

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 that update pointers. One exception to the rule that barriers are activated at all times for the sake of predictability is needed: with incremental object copy, the pointer updating on writes has to be disabled when objects are being relocated. This is because otherwise the mutator could obtain a pointer to the new copy of the object being copied. It could then attempt to read using that pointer, possibly reading from the part of the new object copy that does not yet contain valid data.

Unlike the relatively heavy-weight Dijkstra and Yuasa barrier, pointer updates are cheap and keeping them activated at all times (except the mentioned case) makes sense not only for predictability, but also for performance. Pseudo-code for the barrier is shown in Figure 4. The code in the figure is simplified for readability.

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, depending 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 relocated in the image. With the support for assertions on pointers, it is often known by static analysis that an object pointed by 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 Figures 4 and 3. We omit the assertions related to the image, because they are rather specific to our VM.

To speed up compilation of large applications, we have 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 LIFO (stack), 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 and 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 amount of 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 and 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.

2.4. VM Runtime

In addition to the barriers, the runtime code of the VM also requires certain modifications, such as array copy functions, locking, interface to native code, I/O, and reflective access to data structures in the heap.

Array copy. The standard Java API provides methods for copying blocks of elements within a single array or from one array to another (`System.arraycopy`). At the level of the virtual machine that supports object relocation, this breaks down to several different functions: for pointer/non-pointer array, for inter/intra array copy, and in the case of intra array copy also for the two directions of the copy. In order to be fast, the array copy operation needs to be optimized for each configuration: it cannot be implemented as a sequence of array access operations at Java level, neither with nor without an RTGC. Arraylets, replication or Brooks forwarding pointers thus all require special handling.

The array copy operation has to be incremental, as to avoid starving out the garbage collector or other threads. In VMs with native scheduling, this would be handled by the thread scheduler in the operating system and proper synchronization. In Ovm, the copy operation has to be incremental by design: the pauses can be controlled by setting the maximum amount of bytes that can be copied atomically.

The copy operation also has to ensure that the heap is consistent at any point when the collector can interrupt. In Ovm, the code of the operation uses explicit preemption points; the automated insertion of preemption points at back-branches is locally disabled. Not only that this approach allows to execute barriers in "batches" for all atomically copied data, it also allows to elide some of them. The Dijkstra barrier is never needed, because pointers are always being copied from the heap (not stack) and indeed from a reachable object. In an intra array copy, the Yuasa barrier is not needed when overwriting a pointer value that will be stored elsewhere in the array; this however only holds for the direction of the copy that is the same as the direction of incremental scanning of the array by the collector during the marking phase. If the direction of scanning was the opposite, the pointers being copied might be hidden from

the collector, resulting in potential reclamation of a live object. For arraylets, the array copy operation then has to identify largest contiguous blocks of data that can be copied atomically (each of these blocks is a contained within a single arraylet).

Locks. Ovm uses fast-locks: locking an unlocked monitor only involves a bitwise operation on a header field of the object being locked. Only when the lock operation should block the calling thread, a monitor representation with a queue of waiting threads is used. This monitor is created lazily when the first thread is to block on it. The monitor is just a Java object on the garbage collected heap, which can be copied or replicated. The fast locking operations thus need to handle this, depending on the forwarding style used.

To speed up the collection and elide some of the RTGC barriers from the locking code, the GC treats the life-cycle of monitors specially, exploiting their semantics in the virtual machine. A monitor is created for a particular object (is owned by that object) and shall be reclaimed when that object dies. Most of the objects do not have a monitor in Java applications (monitors of most objects are never used or at least never contended), and thus it would be prohibitive to scan every Java object for a monitor during every collector cycle. Instead, monitors are stored in a special linked-list managed by the collector. This list keeps the monitors alive and links them to their owners (not keeping the owners alive). The list is scanned by the garbage collector after marking, removing entries of monitors whose owners have been identified unreachable. Without any further modification, the monitors will then be re-claimed during the next collector cycle. Monitors are still referenced from their owner objects, but this reference neither needs Dijkstra nor Yuasa barrier, nor does it need to be scanned by the collector.

Native code and I/O. The object relocation, replication, and arraylets have to be hidden from the native code, mostly operating system libraries in Ovm. Since Ovm uses green threading, it has to turn all blocking operations (like I/O) into non-blocking ones to prevent blocking the whole VM. While this code is relatively complex and its necessity often regarded as a weakness of the green threading approach, it makes handling of relocation and replication relatively simple. The VM knows that the GC cannot interrupt the native code; with Brooks forwarding, it suffices to pass forwarded pointers to the native code; with replication, synchronization of replicas may be needed after the call. With arraylets, arrays passed to native code are either allocated as contiguous, or operations are split into multiple calls on consecutive segments of the array (I/O calls).

3. Evaluation

The goal of the evaluation is to empirically compare performance of Brooks forwarding pointers and replication in Ovm with Minuteman RTGC framework, to verify that the implementation is reasonably bug-free, and to provide a sanity-check performance comparison against related production systems.

Benchmarks. We run 8 benchmarks from the DaCapo 2006 [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 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 that would also exercise the memory management in a non-trivial manner. We use periodic time-based RTGC scheduling (like in Metronome GC), so that the benchmarks can run unmodified even though they do not have slack. This also allows us to compare to production VMs that use this type of scheduling.

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, N for none of the previous):

`N`	No defragmentation, no arraylets
`A`	Arraylets, no defragmentation
`B`	Brooks forwarding, atomic copy
`BA`	**Brooks forwarding, arraylets, atomic copy**
`R`	Replication, atomic copy
`RI`	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 use with our collector in practice, and thus they allow the most realistic performance comparison of replication and Brooks. These configurations are the most natural with our collector, because objects larger than 2K are never defragmented by the collector. While non-array objects larger than 2K do not exist in real applications, we need arraylets to avoid fragmentation due to large arrays. Because only objects smaller than 2K are copied, there is no need for incremental object copy: the copy operation has very short latency. The `RI` and `RAI` configurations allow us to quantify overheads of replication and incremental object copy, which might apply when our replication was incorporated into a different collector. The `N` and `A` configurations then allow us to quantify the overheads of arraylets, defragmentation, and a combination of both.

Results.

The speed-ups of replication versus Brooks forwarding are shown in Table II. With arraylets, the replication is on average by 4% faster than Brooks forwarding. Without arraylets, it is faster by 5%. The maximum speed-up is 9% with compress benchmark from SPEC JVM 98, closely followed by 8% speed-up with mtrt benchmark from SPEC JVM 98. The measured average speed-up is also 9% with bloat benchmark from DaCapo, but this value is not completely reliable due to a relatively high variation in bloat results (Table I). Without arraylets and excluding bloat, the maximum speed-up is 12% speed-up with mtrt benchmark. With arraylets, there was no measured slow-down, though one of the benchmarks (javac from SPEC JVM 98) did not show any speed-up, either. Without arraylets, 1% slow-down was measured with compress benchmark and no performance change with lusearch benchmark from DaCapo.

Confidence intervals for mean execution time with different configurations and benchmarks are shown in Table I. With the exception of bloat benchmark from DaCapo, the results are very stable. Even with bloat, the half-width of the confidence interval is (only) within 5% of the mean value. The table columns are ordered such that pairs of configurations interesting for comparison are in adjacent columns.

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

	RAI	RA	BA	A	N	B	R	RI
Antlr	16.03± 0.06	15.92± 0.02	16.81± 0.02	15.66± 0.06	15.25± 0.01	15.9 ±0.04	15.47± 0.03	15.42±0.02
Bloat	258.36±12.14	241.78±12.72	264.67±12.83	234.94±11.77	223.18±10.95	269.91±9.9	235.5 ±11.93	261.94±7.34
Fop	3.28± 0.01	3.13± 0.01	3.25± 0.0	2.99± 0.01	2.9 ± 0.0	3.05±0.01	2.99± 0.01	2.96±0.0
Hsqldb	11.72± 0.03	11.77± 0.02	11.9 ± 0.02	11.24± 0.03	10.7 ± 0.02	11.26±0.01	10.99± 0.02	11.03±0.03
Luindex	15.0 ± 0.05	15.04± 0.03	15.63± 0.01	14.56± 0.01	12.69± 0.02	14.3 ±0.01	13.79± 0.02	13.83±0.03
Lusearch	52.51± 0.38	52.17± 0.24	52.51± 0.04	50.65± 0.04	60.99± 0.25	62.39±0.4	62.38± 0.5	61.47±0.45
Pmd	46.26± 0.52	44.75± 0.17	46.96± 0.08	43.47± 0.15	40.93± 0.09	43.81±0.03	42.86± 0.18	42.7 ±0.08
Xalan	369.85± 1.96	360.14± 1.28	373.86± 0.48	359.11± 1.18	359.58± 1.09	367.69±1.38	356.27± 1.62	358.44±1.98
Pseudojbb	328.7 ± 2.75	327.81± 1.6	338.29± 1.71	323.62± 1.87	294.58± 1.67	316.93±2.23	306.65± 1.39	307.83±0.29
Compress	5.84± 0.01	5.82± 0.0	6.39± 0.01	5.49± 0.01	3.87± 0.0	5.02±0.01	5.05± 0.01	5.03±0.0
Db	6.19± 0.01	6.24± 0.02	6.43± 0.01	6.09± 0.01	5.33± 0.02	5.9 ±0.01	5.46± 0.01	5.46±0.01
Jack	5.26± 0.01	5.29± 0.01	5.38± 0.01	5.18± 0.01	4.74± 0.01	4.96±0.01	4.84± 0.02	5.01±0.02
Javac	5.26± 0.01	5.35± 0.01	5.36± 0.01	5.11± 0.01	4.78± 0.01	5.17±0.01	4.99± 0.01	5.08±0.01
Jess	5.26± 0.0	5.09± 0.03	5.12± 0.01	4.97± 0.0	4.53± 0.01	4.84±0.01	4.56± 0.02	4.5 ±0.0
Mpegaudio	5.38± 0.01	5.37± 0.01	5.56± 0.01	5.18± 0.0	4.32± 0.01	5.38±0.01	4.83± 0.0	4.86±0.01
Mtrt	1.8 ± 0.0	1.81± 0.0	1.97± 0.0	1.87± 0.0	1.65± 0.0	1.87±0.0	1.65± 0.0	1.66±0.0

In an earlier version of this work presented at JTRES workshop, we reported similar relative performance improvements. The new results were obtained with a recent compiler (GCC 4.4) and recent operating system kernel (Linux 2.6.32) and run on a recent hardware (Intel Xeon CPU at 2.27 GHz with 8M L3 Cache). The new code also includes several bug-fixes and some adaptations to run with a newer version of the C library.

Both the current results and the earlier ones show a larger speed-up of replication over Brooks when arraylets are disabled (compare RA/BA with R/B and RAI/BA with RI/B). The overhead of arraylets is relatively large, 8% on average, which is in-line with the difference in the average speed-ups. Still, two of the benchmarks report high overheads of arraylets, while showing a higher improvement of replication with arraylets than without: compress and mtrt from SPEC JVM 98. With lusearch, the arraylets actually cause a speed-up (17%), and the relative speed-up of replication over Brooks is higher with arraylets than without.

Ratios of reads and writes. The ratios of read and write operations performed by the DaCapo and SPEC JVM 98 benchmarks are shown in Table III. On geometric average, there were 5.2 times more reads than writes. Still, 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. In addition to that, the impact of replication on the total execution time also depends on the actual rate of the read/write operations in the benchmark.

Comparability to products. We provide a performance comparison of our VM to two production systems as a sanity check that our base VM is not too slow to render the measured relative speed-ups unrealistic, as well as that it is not too fast suggesting that it would be missing an important feature. If the VM lacked an important performance optimization, say if the non-defragmenting arraylet implementation of `System.arraycopy` was too slow and the new implementation with replication was well optimized, the resulting speed-up would not be inherent to replication. If, on the other hand, the VM missed an important feature (say arraylets), the relative speed-ups measured will seem higher, because replication would account

Table II. Ratio of execution times for replication over Brooks.

	RA/ BA	R/ B	RAI/ BA	RI/ B
Antlr	0.95	0.97	0.95	0.97
Bloat	0.91	0.87	0.98	0.97
Fop	0.96	0.98	1.01	0.97
Hsqldb	0.99	0.98	0.98	0.98
Luindex	0.96	0.96	0.96	0.97
Lusearch	0.99	1.0	1.0	0.99
Pmd	0.95	0.98	0.99	0.97
Xalan	0.96	0.97	0.99	0.97
Geo-mean	0.96	0.96	0.98	0.97
Pseudojbb	0.97	0.97	0.97	0.97
Compress	0.91	1.01	0.91	1.0
Db	0.97	0.93	0.96	0.93
Jack	0.98	0.98	0.98	1.01
Javac	1.0	0.97	0.98	0.98
Jess	0.99	0.94	1.03	0.93
Mpegaudio	0.97	0.9	0.97	0.9
Mtrt	0.92	0.88	0.91	0.89
Geo-mean	0.96	0.94	0.96	0.95
Geo-mean	**0.96**	**0.95**	**0.97**	**0**.96

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

	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**

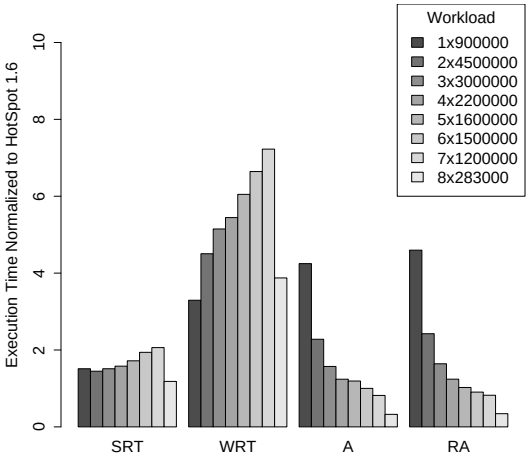


Figure 5. Pseudo-jbb with different workloads (warehouses,transactions). Execution times are normalized to (non-realtime) HotSpot server 1.6.

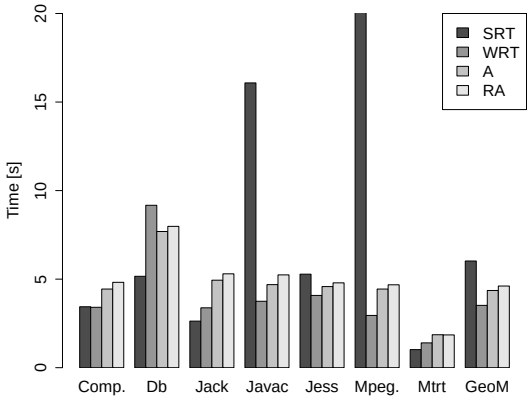


Figure 6. SPEC JVM 98, mean execution time.

for more of the overall performance (Amdahl's law). While it may discover some problems, this sanity check indeed cannot prove the absence of this kind of problems.

We compare the performance results of Ovm to that of 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. We configured the RTGC scheduling in the very same way as Ovm (10 ms window, 500 μs quantum, 1 ms maximum pause time). We use the pseudo-jbb benchmark in multiple configurations and the SPEC JVM 98 benchmarks.

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). We have run these experiments on Intel Core(2) CPU at 2.83 GHz with 6M L2 Cache, Linux 2.6.31. We used GCC 4.4 to build Ovm.

Figure 5 shows execution times of multiple configurations of the pseudo-jbb benchmark, normalized to (non-realtime) HotSpot server 1.6. The heap size for the non-realtime HotSpot VM was set such that the GC would start at the same memory usage level as the RTGC in the real-time VMs. The configurations use an increasing number of warehouses (1–8) and a particular number of transactions that the SPEC JBB 2005 benchmark would run on our machine. The number of warehouses is also the number of active threads that frequently acquire (uncontended) locks. While the number of warehouses has a critical impact on the results, the number of transactions is not that important – it only controls the duration of the experiment. The benchmark was run on a single core, because Ovm is a uni-processor machine. As the number of warehouses increases, Ovm is closing on the products and eventually it becomes faster. For eight warehouses, it is even faster than HotSpot. This can be easily explained by a very fast lock implementation in Ovm which is possible because of green threading. The production VMs use native scheduling and their lock implementations may not be optimized for uni-processors. On our machine, the SPEC JBB 2005 benchmark would use results from 2, 3, and 4 warehouses to calculate the final score. If we thus calculate a geometric mean overhead from these three configurations, we get 1.5 for SRT, 5 for WRT, 1.6 for `A` (which is the closest Ovm configuration to WRT), and 1.7 for `RA`. Ovm thus on average performs slightly worse than SRT and better than WRT. For the trivial case of a single warehouse, Ovm is slower than both products.

For the pseudo-jbb experiment, we used the same GNU Classpath implementation of Java collection classes with Ovm, SRT, WRT, and HotSpot, to focus the comparison more on the VM and the GC itself rather than the class libraries. This is supported by our experiment in which with WRT and SRT, the overhead of choosing GNU Classpath was as much as 100%. Ovm uses a historic version of GNU Classpath with backported bug-fixes, this comparison is thus not necessarily applicable to a recent version of GNU Classpath.

The results for SPEC JVM 98 benchmarks are shown in Figure 6. 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. This setting slightly favored the IBM products. As SPEC JVM 98 are complex application benchmarks that use wide functionality of class libraries and are closed-source, we had to use the original IBM class libraries. Any performance overhead inherent to GNU Classpath would thus be added to the performance overhead of Ovm. The average overhead of Ovm (`A`) over WRT is 24%. Ovm is by 28% faster than SRT. The poor performance of SRT measured by the javac and mpegaudio benchmarks could indeed be caused by a bug or misconfiguration, which we cannot fix. We can still get an upper bound for the Ovm overhead by comparing Ovm always to the better of SRT and WRT, which gives the overhead of 46%.

4. Concurrency Considerations

The proposed replication technique has been implemented and shown to be beneficial in a green threading uni-processor setting, but it is applicable also to other concurrency settings. The technique for instance should work with the synchronization scheme of the Sapphire collector, as proposed (though not implemented) in [16], allowing parallel execution of multiple threads, native scheduling, and non-parallel collection. This synchronization scheme adds synchronization primitives to read and write barriers and to the collector. The primitives have three alternative designs: compare-and-swap that synchronizes on objects, fetch-and-add that synchronizes on objects, and fetch-and-add that synchronizes on fields. Only volatile fields are synchronized for performance reasons. This is argued not to violate JMM, but it indeed causes an unnatural behavior of unprotected accesses to non-volatile fields, making programming more difficult. When applied to our replication scheme, this unnatural behavior would still be present, but would occur with much smaller probability, because our write barriers write to both replicas instantly. Although theoretically possible, adding the synchronization to accesses to all fields would likely have unacceptable overhead on multiprocessors.

Our replication technique should work in any system that would allow (cheap) atomicity of the consecutive writes to potentially two different replicas in the write barrier. An example of such a system would be a uni-processor with native scheduling and fast interrupt disabling/enabling. While in such a system replication might still provide better performance than Brooks forwarding, the main reason to consider it would be more likely the incremental object copy. The actual tradeoffs and real benefits can however only be found via a thoroughly optimized implementation of particular systems with different concurrency settings, which we do not provide in this work.

5. 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 average performance improvement of 4% on a Linux/x86 platform. We measured with DaCapo, pseudojbb, and SPEC JVM 98 benchmarks. Our version of replication is a natural choice for a uni-processor VM with green threading model. A uni-processor VM might then be a natural choice for embedded systems, as uni-processors still dominate the embedded systems market. While a green threading model has unique advantages for implementation of uni-processor VMs, our model of replication could also perform well in systems with other cheap means for atomicity of barriers, not necessarily the green threading model.

Our work complements Sapphire [16, 14], a replicating garbage collector for 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 the 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 in our collector 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. This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350, and by the Ministry of Education of the Czech Republic grant MSM0021620838.

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. Darko Stefanovic and J. Eliot B. Moss. Characterization of object behaviour in standard ml of new jersey. *SIGPLAN Lisp Pointers*, VII(3):43–54, 1994.
29. Sun Microsystems. Sun java real-time system. http://java.sun.com/javase/technologies/realtime, 2008.
30. Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.
31. Martin Zabel, Thomas B. Preußer, 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.