Non-blocking Object Copy for Real-Time Garbage Collection

Schoeberl, Martin; Puffitsch, Wolfgang

Published in:
Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)

Link to article, DOI:
10.1145/1434790.1434802

Publication date:
2008

Document Version
Early version, also known as pre-print

Non-blocking Object Copy for Real-Time Garbage Collection

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology, Austria
wpuffits@mail.tuwien.ac.at

ABSTRACT

A real-time garbage collector has to fulfill two conflicting properties: avoid heap fragmentation and provide short blocking time. The heap needs to be compacted to avoid probably unbounded fragmentation. During compaction all objects are copied; copying is usually performed atomically to avoid interference with mutator threads. Copying of large objects and especially large arrays introduces long blocking times that are unacceptable for real-time systems.

In this paper an interruptible copy unit is presented that implements non-blocking object copy. The unit intercepts object and array field access and redirects the access either to the source or destination part of the moving object. The unit can be interrupted after a single word move. The resulting maximum blocking time is the time for a memory word read and write. We have implemented the proposed non-blocking copy unit in the Java processor JOP and are able to run high priority real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz system.

1. INTRODUCTION

Garbage collection (GC) is a feature in modern object oriented languages, such as Java and C#, that increases programmer productivity and program safety. However, dynamic memory management is usually avoided in hard real-time systems. Even the real-time specification for Java (RTSJ) [3], which targets soft real-time systems, defines an additional memory model, with immortal and scoped memory, to avoid garbage collection.

However, the memory model introduced by the RTSJ is unusual to most programmers. It also requires that the Java virtual machine (JVM) checks all assignments to references. If a program does not adhere to the specified model, run-time exceptions are triggered. Arguably, this is a different level of safety than most Java programmers would expect. Therefore, much research activity is spent to enable garbage collection in real-time systems.

One of the open issues that need to be solved is heap compaction with short blocking times. Heap fragmentation is one of the main reasons to avoid dynamic memory management in hard real-time systems and safety critical systems. The worst case memory consumption within a fragmented heap is too high to be acceptable. A garbage collector that performs heap compaction as part of the collection task eludes this fragmentation issue.

In a system with a concurrent GC, the GC thread and the mutator threads have to synchronize their work. Several operations (e.g., barrier code, stack scanning, and object copy) need to be performed atomically. Stack scanning and object copy in atomic sections can introduce considerable blocking times. This paper deals with one of those atomic sections, the potentially long operation of an object copy in a compacting GC. In an accompanying paper [14] we propose a solution to avoid blocking at root scanning of the stack.

Heap compaction comes at a cost: objects need to be moved in the heap. This object copy consumes processor execution time, memory bandwidth, and needs to be performed atomically. We can accept the first two cost factors as a tradeoff for safer real-time programs. However, the blocking time introduced by the atomic copy operation can be in the range of milliseconds on actual systems. This value can be too high for many real-time applications.

In this paper we propose a memory unit for non-blocking object copy. The memory copy is performed independent of the activity in the CPU, similar to a direct memory access (DMA) unit. The copy unit executes at the priority of the GC thread. When a higher priority thread becomes ready, the copy unit is interrupted. The memory unit stores the state of the copy task. The object field and array access is also performed by this memory unit. When a field of an object under copy is accessed by the mutator, the memory unit redirects the access to the correct version of the object: to the original object when the field has not yet been copied or to the destination object when the field has already been copied.

The non-blocking copy unit is evaluated by an implementation in the context of the Java processor JOP [18]. The resulting maximum blocking time due to the object copy is 120 ns (12 clock cycles) on the 100 MHz system, orders of magnitudes lower than other atomic operations in the JVM. It is possible to run a 10 kHz high priority task without a single deadline miss with ongoing garbage collection. The maximum task frequency is limited by the scheduler and not by garbage collection. It has to be noted that the proposed copy unit is not JOP specific. The unit can also be integrated in a standard RISC processor that executes compiled Java.

The paper is organized as follows: in the next section related work on real-time collectors and approaches to minimize GC blocking

JTRES'08, September 24-26, 2008, Santa Clara, California, USA

time is presented. Section 3 presents the idea of non-blocking object and array copy. The implementation within the Java processor JOP is described in Section 4. The design is evaluated in Section 5 and the findings are discussed in Section 6. The paper is concluded in Section 7.

2. RELATED WORK

Real-time garbage collection research dates back to the 1970s where collectors for LISP and ML have been developed. Therefore, a vast number of papers on real-time garbage collection have been published. A good introduction to garbage collection techniques can be found in Wilson's survey [24] and in [7].

The simplest way to avoid blocking times due to object copying is simply to avoid moving objects at all. The real-time GC of the JamaicaVM does exactly this [22]. Objects and arrays are split into fix sized blocks and are never moved. This approach trades external fragmentation for internal fragmentation. However, the internal fragmentation can be bounded.

The Metronome GC splits arrays, similar to the JamaicaVM approach, into small chunks called Arraylets [1]. Metronome compacts the heap to avoid fragmentation and the Arraylets reduce blocking time on the copy of large arrays. Both approaches, the JamaicaVM GC and Metronome, have to pay the price of a more complex (and time consuming) array access.

Another approach to allow interruption of GC copy is to perform field writes to both copies of the object or array [6]. This approach slows down write access, but those are less common than read accesses. The main drawback of this solution is the additional pointer needed between the two copies of the object. Nettles and O'Toole propose a GC where the mutator is allowed to modify the original copy of the objects [9]. All writes are recorded in a mutation log and the GC has to apply the writes from this log after updating the pointer(s) to the new object copy.

Nilsen and Schmidt propose hardware support, the object-space manager (OSM), for real-time GC on a standard RISC processor [10]. The concurrent GC is based on [2], but the concurrency is of finer grain than the original Baker algorithm as it allows the mutator to continue during the object copy. The OSM redirects field access to the correct location for an object that is currently being copied.

The clever usage of atomic two-field compare-and-swap (CAS) operations for an incremental object copy is proposed in [13]. During the copy process, an object is expanded to an intermediate wide version and an uninitialized narrow version in tospace. The wide version is protected by CAS operations. However, this solution introduces some overheads to the mutator field access especially during the copy process. In the worst case, the mutator has to expand the object to the wide version on a field write.

Meyer presents a hardware implementation of Baker's read-barrier [2] in an object-based RISC processor [8]. The cost of the read-barrier is between 5 and 50 clock cycles. The resulting minimum mutator utilization (MMU) for a time quantum of 1 ms was measured to be 55%. For a real-time task with a period of 1 kHz the resulting overhead is about a factor of 2. We consider the 50 cycles, even if they are quite low, too expensive for a read-barrier and use the Brooks-style [4] indirection instead.

The solution proposed by Meyer for object-oriented systems also contains a garbage collection coprocessor in the same chip. Close interaction between the RISC pipeline and the GC coprocessor allow the redirection for field access in the correct semi-space with a concurrent object copy. The hardware cost of this feature is given as an additional word for the back-link in every pointer register and every attribute cache line. The only additional runtime cost is on an attribute cache miss. In that case, two instead of one memory accesses resolve the cache miss. It is not explicitly described in the paper when the GC coprocessor performs the object copy. We assume that the memory copy is performed in parallel with the execution of the RISC pipeline. In that case, the GC unit *steals* memory bandwidth from the application thread. Our copy unit, in contrast, respects thread priorities and has no influence on the worst-case execution time (WCET) of hard real-time threads.

The Java processor SHAP [26], with a pipeline and cache architecture based on the architecture of JOP, contains a memory management unit with a hardware GC. That unit redirects field and array access during a copy operation of the GC unit.

The three hardware-assisted GC proposals [10, 8, 26] do not address the influence of the copy hardware on the WCET of the mutator threads. It is known that background DMA complicates WCET analysis. In our proposal, we allow object copy only when the GC thread is running. Therefore, that task is simple to integrate into the schedulability analysis. Scheduling the GC thread at low priority and providing an interruptible (non-blocking) object copy result in 100% MMU for high priority real-time tasks.

Compared to a to a fix block size GC (e.g., the JamaicaVM approach) a compacting GC avoids the internal fragmentation. The tradeoff is the additional memory bandwidth that is consumed by the object copy and the handling of the object relocation. Field and array accesses need two and three memory accesses for an object layout with a handle indirection. In the JamaicaVM the cost of the field and array access varies and depends on the location of the field or the layout of the array. In the average case the cost of a field access is close to one and of an array access close to two memory accesses [21]. With the jvm98 benchmarks most arrays are allocated contiguous instead of a tree of fix sized blocks. However, in the worst case a tree needs to be traversed and 2+d memory accesses are needed for a tree of depth d. Constant access time is simpler to handle by WCET analysis than a location and organization dependent access time.

3. NON-BLOCKING OBJECT COPY

Copying large arrays and objects in a compacting GC attributes to the largest blocking times. To avoid losing updates on the object during copying (write to fields that are already copied), it is usually performed atomically. To avoid those long blocking times in a real-time GC, we propose an interruptible copy unit. The copy unit has two important properties:

- It can be preempted at single word copy boundaries
- The copy process is executed at the GC thread priority

A real-time GC needs to be interruptible by higher priority threads. If the copy task is performed by the hardware, which works autonomously in its own hardware thread, that hardware also needs to be interrupted on a thread switch. Furthermore, the copy task

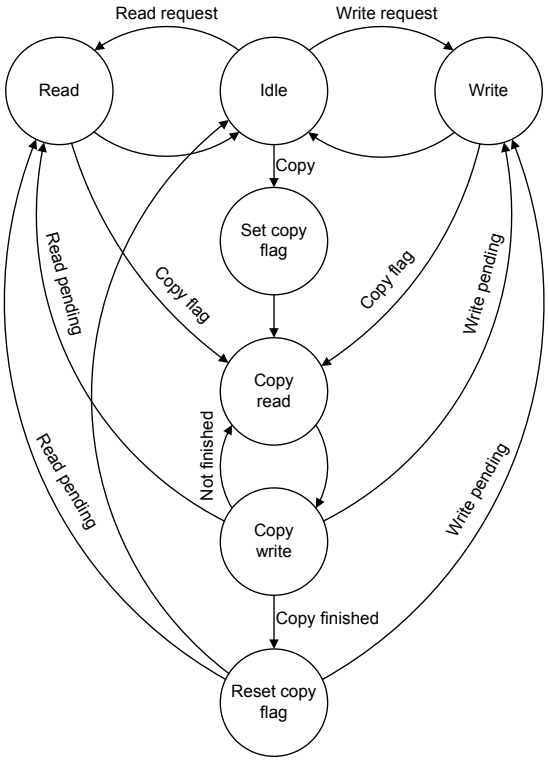

Figure 1: Memory controller state machine with background copy

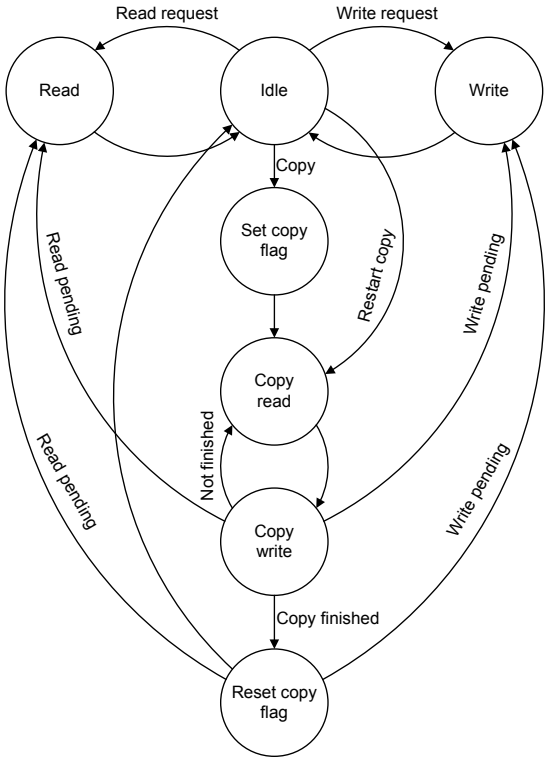

Figure 2: Memory controller state machine with interruptible copy

needs to be restarted at the correct time, i.e., when no thread with a priority higher than the GC thread priority is ready.

A simplified solution is to start the copy as a background DMA operation and let the GC thread wait for completion before continuing the GC work. However, this background activity, even when interruptible at word boundaries, changes the WCET of high priority threads. It *steals* memory cycles from those threads. Even when the copy unit starts at idle cycles it will still block incoming read or write requests from the real-time threads during the copy. Therefore, it will delay most of the load and store instructions.

Figure 1 shows a simplified state diagram of the memory controller that performs the background copy. From the *idle* state either a normal read, normal write or a start of the copy task is performed. The states of the copy task are: start with setting a flag that an object copy is pending, perform the copy (via states *copy read* and *copy write*), and end with the reset of the copy flag. After each write in the copy loop the CPU is checked for an outstanding read or write request. In that case the copy task stops and that request is fulfilled. A stopped copy is resumed from states *read* and *write* if the copy flag is set.

For time predictability we need a complete stop of the copy task on a software thread switch (from the GC thread to an application thread). Two solutions are possible: (a) integrate the control of the copy task into the scheduler, or (b) let the copy unit itself detect a thread switch.

For the first solution the stopping of the copy unit is integrated into the scheduler. On a non-GC thread dispatch the scheduler has to explicitly stop the copy task. However, this approach needs integration of GC related work into the scheduler, which is not possible in all JVMs.

The second approach is to interrupt the copy task by a normal memory operation (read or write). Interruption can be detected by the memory unit by a pending read or write request. During the object copy the GC thread performs a busy wait on the status of the copy. Therefore, the GC thread does not access main memory at this time. If the memory unit recognizes a read or write request it comes from an application thread that interrupted the GC thread. That request is the signal to stop copying. The state machine for this behavior is depicted in Figure 2. As in the former state machine the copy loop can be interrupted by a pending read or write request. The difference is that there is no automatic transition from the *read* and *write* state back to the copy loop. The copy task needs to be explicitly restarted from the processor as indicated by the transition from *idle* to *copy read*.

The remaining question is how to restart the copy task? Similar to the stopping of the copy unit, two solutions are possible: (a) the scheduler restarts the copy task, or (b) the GC thread performs the restart.

The scheduler integration works as follows: When the GC thread is about to be rescheduled, the scheduler has to resume the copy operation as well. This approach is only possible when the scheduler has knowledge about the thread types (mutator or GC thread). Scheduling and dispatching are atomic operations with respect to software threads (only higher priority interrupt handlers are allowed to interrupt the scheduler). Performing the copy unit restart within

```
startCopy(src, dst, size);
while (!copyFinished()) {
    if (copyInterrupted()) {
        restartCopy();
    }
}
synchronized (GC.mutex) {
    updateHandle(handle, dst);
}
```

Figure 3: Busy waiting copy loop in the GC thread with copy restart

this atomic operation slightly increases the blocking time introduced by the scheduler.

The proposed solution lets the GC thread resume the copy task when getting rescheduled. To perform this function, the GC thread needs to know that it was preempted – an information that is usually not available for a thread. However, the copy unit preserves this information and the state *interrupted* can be queried by the GC from the copy unit in the copy loop.

Figure 3 shows the copy code in the GC. The GC thread kicks off the copy task with startCopy() and performs a busy wait till the copy task is finished – copyFinished() returns true. Within the loop the state of the copy state machine is checked with copyInterrupted() and the copy task is restarted if necessary. It has to be noted that this busy waiting loop does not consume any memory bandwidth. The code is executed from the instruction cache, stack operations are performed in the stack cache, and all state queries go via an on-chip bus directly to the memory controller. The memory controller can perform the copy at maximum speed during the GC busy wait. At the end of the copying process, the reference to the object in the handle is updated atomically.

A further simplification of the copy unit is possible when the GC thread triggers only single word copies in a tight loop. The copy process is automatically preempted when the GC thread gets preempted. No restart is necessary due to the incremental copy trigger and the polling for the finished copy task can be omitted. The disadvantage of this simplification is the slower copy of the object.

4. IMPLEMENTATION

We implemented the proposed non-blocking copy unit in the Java processor JOP [18]. JOP was designed from scratch as a real-time processor [16] to simplify the low-level part of WCET analysis. The main benefit of a Java processor for real-time Java is the possibility to perform WCET analysis at bytecode level [19].

In the following section the GC algorithm that is part of the JOP runtime environment is briefly described. It has to be noted that the proposed copy unit is independent of the processor platform and also independent from the GC algorithm.

4.1 The GC Algorithm

The collector for JOP is a concurrent copy collector [15, 20] based on [2, 5]. Baker's expensive read-barrier is avoided by using a write-barrier and performing the object copy in the collector thread. Therefore, the collector is concurrent and resembles the collectors presented by Steele [23] and Dijkstra et al. [5]. The collector and

```
private static void putfield_ref(int ref, int value,
                                 int index) {

    synchronized (GC.mutex) {

        // snapshot-at-beginning barrier
        int oldVal = Native.getField(ref, index);
        // Is it white?
        if (oldVal!=0 &&
            Native.rdMem(oldVal+GC.OFF_SPACE)!=GC.toSpace) {
            // mark gray
            GC.push(oldVal);
        }
        // assign value
        Native.putField(ref, value, index);
    }
}
```

Figure 4: Snapshot-at-beginning write-barrier in JOP's JVM

the mutator are synchronized by two barriers. A Brooks-style [4] forwarding directs the access to the object either into tospace or fromspace. The forwarding pointer is kept in a separate handle area as proposed in [11]. The separate handle area reduces the space overheads as only one pointer is needed for both object copies. Furthermore, the indirection pointer does not need to be copied. The handle also contains other object related data, such as type information, and the mark list. The objects in the heap only contain the fields and no object header.

The second synchronization barrier is a *snapshot-at-beginning* write-barrier [25]. A snapshot-at-beginning write-barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array. The *to be overwritten* field is shaded gray as shown in Figure 4. An object is shaded gray by pushing the reference of the object onto the mark stack.[1] Further scanning and copying into tospace – coloring it black – is left to the GC thread. One field in the handle area is used to implement the mark stack as a simple linked list.

This write-barrier and atomic stack scanning allow using expensive write-barriers only for reference field access (putfield, putstatic, and aastore in Java bytecode). Local variables and the operand stack need no barrier protection.

Note that field and array access is implemented in hardware on JOP. Only write accesses to reference fields need to be protected by the write-barrier, which is implemented in software. During class linking all write operations to reference fields (putfield and putstatic when accessing reference fields) are replaced by a JVM internal bytecodes (e.g., putfield_ref) to execute the write-barrier code as shown in Figure 4.

The methods of class Native are JVM internal methods needed to implement part of the JVM in Java. The methods are replaced by regular or JVM internal bytecodes during class linking. Methods getField(ref, index) and putField(ref, value, index) map to the JVM bytecodes getfield and putfield. The method rdMem() is an example of an internal JVM bytecode and performs a memory read. The null pointer check for putfield_ref is implicitly performed by the hardware implementation of getfield that is executed by Native.getField(). The hardware implementation of getfield triggers an

[1] Although the GC is a copying collector a mark stack is needed to perform the object copy in the GC thread and not by the mutator.

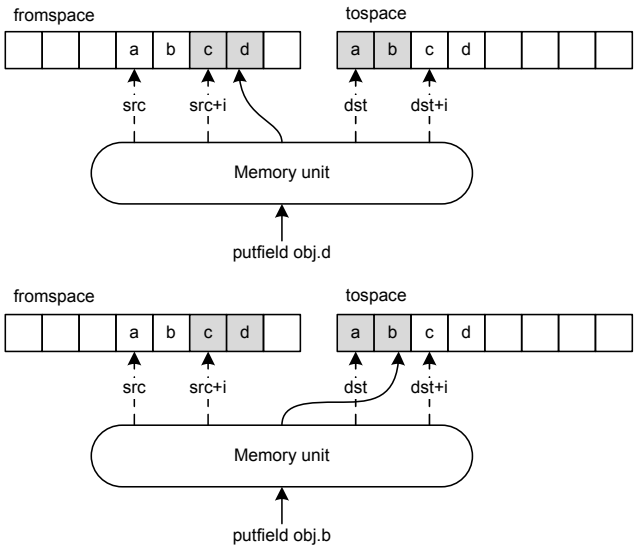

Figure 5: Redirection of a putfield operation by the memory unit

exception interrupt when the reference is null. The implementation of the write-barrier shows how a bytecode is substituted by a special version (pufield_ref), but uses in the software implementation the hardware implementation of that bytecode (Naitve.putfield()).

In principle this write-barrier could also be implemented in microcode to avoid the expensive invoke of a Java method. However, the interaction with the GC, which is written in Java, is simplified by the Java implementation. As a future optimization we intend to inline the write-barrier code.

The collector runs in its own thread and the priority is assigned according to the deadline, which equals the period of the GC cycle. As the GC period is usually longer than the mutator task deadlines, the GC runs at the lowest priority. When a high priority task becomes ready, the GC thread will be preempted. Atomic operations of the GC are protected simply by turning the timer interrupt off.[2] Those atomic sections lead to release jitter of the real-time tasks and shall be minimized. It has to be noted that the GC protection with interrupt disabling is not an option for multiprocessor systems.

4.2 The Memory Controller

The memory controller in JOP already implements the field and array access in hardware. The hardware implementation of those functions reduces the overheads of the read-barrier (the handle indirection) and speeds up null pointer and bounds checks [17]. This memory controller is extended with a copy function and the redirection of field and array accesses to the correct part of the object.

Figure 5 shows an example of the write access to an object that is under copy from address src to address dst. The index i points to the next word that will be moved. The object contains four fields (a, b, c, and d). Gray memory cells show the current locations of the fields. Fields a and b are already in tospace, fields c and d are in the original object in fromspace. The upper figure shows the access

[2]If interrupt handlers are allowed to change the object graph those interrupts also need to be disabled.

```
memCopy:
    distance <= dst-src;
    tmp <= read(src+i);
    write(dst+i, tmp);

getfield:
    if (ref==0) trigger exception;
    addr <= read(ref);
    if (addr==src and index<i)
        value <= read(addr+index+distance);
    else
        value <= read(addr+index);
```

Figure 6: The algorithm in the memory unit for a copy step and bytecode getfield

to field d that goes to the original object. The lower figure shows the redirection of the access to field b into the tospace copy of the object.

Figure 6 shoes the hardware implementation of the copy function and bytecode getfield in pseudo code. Note that all operations without data dependencies are executed in parallel in the hardware, e.g., the calculation of distance and the start of the memory access with read().

We have implemented the simplified version of the copy unit with the simple interaction with the GC thread. Instead of kicking off the whole copy task once and restarting it after preemption, the copy task is continually triggered for individual words in the GC loop. The following code fragment shows that loop.

```
for (i=0; i<size; i++) {
    Native.memCopy(dst, src, i);
}
```

The method memCopy() is mapped to a JVM internal bytecode and triggers the hardware to perform a single word copy from src to dst at offset i. Note that this loop is not protected by a synchronized block and can be preempted when a high priority thread becomes ready. The copy task is preempted implicitly as well. When the GC thread is running again, it just continues to copy the object.

The advantage of that implementation is a simple state machine in the memory unit and less hardware resource consumption. The disadvantage is the slower copying of the object. A hardware implementation of the copy operation could perform a single word copy in 5 cycles (two cycles for the word read and 3 cycles for the word write) on the actual platform. Copy of a single word with the simplified solution takes 27 cycles: 12 cycles are spent in the JVM internal bytecode and 15 cycles are loop overhead and pushing the arguments for memCopy() onto the operand stack. The maximum blocking time of the copy operation is the execution of the internal bytecode,[3] therefore, 12 clock cycles.

One important feature of the memory controller is the redirection of field and array access to the correct copy of the object. Field and array access are already part of the memory unit [17]. Therefore, the pointer of the access just needs to be compared with the

[3]Interrupts are only accepted at bytecode boundaries.

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	5
τ_p	2 ms	2 ms	4
τ_c	10 ms	10 ms	3
τ_{log}	1000 ms	40 ms	2
τ_{gc}	50 ms	50 ms	1

Table 1: Task set for the evaluation

pointer of the object currently copied and the index with the copy pointer. If the index is higher than the copy pointer the access is performed normal – the pointer in the handle indirection points to the old copy until the whole copy is performed. The handle is updated afterwards atomically by the GC thread. If the access goes to a field or array element that is already copied, the access is redirected. To speedup the redirection the memory unit precalculates the distance between the old copy and the new copy of the object at the start of the copy operation. This offset is simply added at the effective address calculation when a redirection is necessary.

The redirection is performed in the same cycle as the effective address calculation. Therefore, field and array access takes the same time as in the original implementation. The calculation of the offset and the redirection is carefully designed to avoid introduction of a slow critical path in the memory unit that would reduce the maximum operation frequency of the processor.

The hardware resource consumption of the copy unit is moderate. The additional registers, adders, and multiplexors in the memory unit consume 310 additional logic cells (LC). This is about 10% of the complete processor. However, it doubled the size of the memory unit from 296 LC to 605 LC. The memory unit is now almost as large as the execution unit (669 LC).

5. EVALUATION

For the evaluation of the copy unit we have setup a similar experiment as in [20]. The task set contains a 10 kHz high-priority thread τ_{hf}. We measure the release jitter of this thread to reason about the blocking time introduced due to the GC thread. Two threads (τ_p, τ_c) act as a producer/consumer pair exchanging arrays. For the experiment the array size is varied to measure the blocking time due to the array copy. The logging thread τ_{log} prints out the maximum release jitter of the high-priority task. Thread τ_{gc} with the lowest priority performs the GC work.

Table 1 shows the period, deadline, and priorities of the task set. To avoid a constant phasing of the release times slightly different values (prime numbers) of the periods are used and each experiment run for at least 10 minutes.

For reference we performed three base line tests. The first test consisted of only two threads: the high-priority thread τ_{hf} and the logging thread τ_{log}. The maximum observed jitter was 41 μs. This jitter is introduced by the scheduler.[4] For the second test the GC thread τ_{gc} was added and a maximum release jitter of 56 μs was measured. In the last base line test the producer and consumer threads exchanging small objects. In that case two synchronized blocks, to add an object to the list and to remove it, are part of the benchmark. The maximum observed jitter was 68 μs.

[4]The scheduling decision and the thread dispatch consume about 20 μs.

	Release jitter	
Array size	original	copy unit
256 B	73 μs	68 μs
512 B	73 μs	67 μs
1 KB	72 μs	66 μs
2 KB	124 μs	66 μs
4 KB	226 μs	68 μs
8 KB	–	67 μs

Table 2: Release jitter measured on a 100 MHz processor for the high priority thread with different array sizes

	Release jitter	
Array size	original	copy unit
8 KB	415 μs	58 μs
16 KB	804 μs	57 μs
32 KB	1572 μs	57 μs
64 KB	3137 μs	58 μs
128 KB	6252 μs	58 μs
256 KB	12464 μs	58 μs

Table 3: Release jitter measured on a 100 MHz processor with statically allocated arrays of different sizes

For the copy unit test the producer and consumer exchange integer arrays. The producer allocates one array each period and appends it to the list. The consumer removes all array elements from the list. Depending on the phasing of the threads 4 to 6 elements are removed. We increased the array size until the GC thread was unable to keep up with the collection.

The maximum observed release jitter of τ_{hf} for various array sizes is reported in Table 2. The second column gives the jitter values for the GC that performs the object/array copy atomically. The third column shows the jitter values with the proposed copy unit. Performing the array copy atomically introduces considerable jitter for the high frequency thread. The interruptible array copy performed by the proposed copy unit introduces no additional jitter.

With an 8 KB array the JVM run out of memory as the GC thread could not keep up with the allocation of the producer thread. For arrays larger than 1 KB the blocking time of the atomic copy is longer than the period of the high-frequency thread. Therefore, some iterations of the periodic thread are either delayed for more than the period or lost.[5]

To check the observed blocking time in the original version we analyze the copy time. In the original version of JOP the array copy is performed in microcode; one iteration of the microcode loop (where one 32-bit word is copied) needs 18 cycles. Therefore, copying 1 KB takes 46 μs at 100 MHz ($1024/4 \times 18/100$). This value is in line with the measurement.

To measure blocking times for larger arrays we performed a second experiment. At program start, as many arrays of a given size

[5]Wether an iteration is completely lost or delayed depends on the scheduler implementation on a deadline miss. If the next release time is adjusted, the default implementation on JOP to avoid queuing up pending releases, some iterations can get lost. When the next release times are not adjusted, as configured for this experiment, the periodic thread can try to 'catch up' in the following releases.

as fit into one semi-space are allocated and only threads τ_{hf}, τ_{gc}, and τ_{log} are started. The period of τ_{gc} is set to 1 ms (a deadline that will be missed) so the GC thread runs effectively as a background thread. To provoke more GC runs we moved the memory initialization for new objects from the GC thread to the allocation operation (bytecode newarray). Table 3 shows the observed results.

For an atomic array copy we observe the expected linear increase of the blocking time with respect to the array size. The blocking time is about 50 μs/KB and verifies the results from the first experiment. The array copy with the copy unit results in no additional latency due to the GC thread. The release jitter times are lower in this experiment than in the former one as fewer threads are running (no producer and consumer threads).

6. DISCUSSION

From the evaluation it can be seen that an interruptible copy unit in the memory controller can eliminate blocking time due to compaction during GC. Without such a unit arrays larger than 1 KB introduce blocking and therefore release jitter for high priority real-time threads. Objects are usually way smaller and providing hardware support to copy those objects is therefore not necessary. For small arrays and objects, the blocking due to thread dispatching dominates. The jitter introduced by the scheduler is between 60 μs and 70 μs for the 100 MHz system. If we want to run higher frequency threads (e.g., at 100 kHz) we need to enhance thread scheduling and dispatch. The GC is not the limiting factor anymore.

There is some room for improvement in the implementation. The simple implementation of triggering just a single word copy transaction results in a slow copy. A more advanced memory controller could speedup the copy task by a factor of 5. However, copy speed was not the main focus of the implementation. We aimed at fine grain granularity of atomic operations. Furthermore, there is some room to reduce the resource consumption in the memory controller.

We have implemented the copy unit in the context of a Java processor – a natural choice. It will be an interesting experiment to enhance a RISC processor that executes compiled Java with such a copy unit. The RISC processor needs to be extended by customized instructions for array access that cooperate with the copy unit. Enhancing a RISC pipeline with array access is a good idea anyway. Array bounds check and null pointer check without a MMU is expensive for compiled Java, even though some checks can be eliminated by the compiler.

For a chip multiprocessor (CMP) version of JOP [12] the memory controller must be split. The redirection of object and array access has to be performed for all cores. Therefore, this functionality has to be placed after the memory arbiter. The copy task itself will be a simple DMA master connected to the arbiter in the same way as CPU cores. For time sliced arbitration of the memory bus the copy task does not need to be interrupted on a thread switch. The arbiter itself performs the interruption and guarantees the bandwidth for the individual cores. The time sliced arbiter reserves bandwidth for the copy unit. This bandwidth consumption by the copy unit will be integrated into the WCET analysis as any other thread/processor.

On a CMP system the whole memory controller, including the logic for the field and array access, can be placed between the arbiter and the memory interface. Therefore, the pressure to reduce the resource consumption of that unit diminishes. However, in order to support local, program managed memory, some functionality of the memory controller core is still needed processor locally. The best balance between local and shard functionality of the memory controller is the topic of future research.

7. CONCLUSION

Atomic copy of large arrays by a compacting garbage collector introduces considerable blocking times for real-time threads. In this paper we proposed and evaluated a hardware extension to eliminate that blocking time. A copy unit performs the object and array copy and redirects field and array access to the correction version of the object or array. An important feature of the proposed copy unit is scheduling the copy task at GC priority. Therefore, a high priority real-time thread can interrupt the copy task at single word copy boundaries. As the copy task is completely interrupted (no background activity) it does not influence the WCET of real-time threads.

With the proposed solution the main source of blocking due to a real-time garbage collector is practically removed. The second source of blocking is due to atomic root scanning of the threads' runtime stack. This issue is addressed in an accompanying paper [14] by scheduling of the root scanning task. Both solutions together allow using a real-time GC concurrently to high frequency real-time tasks. In the 100 MHz JOP system, which is used for the evaluation of the approach, it is feasible to run a thread at 10 kHz concurrently to GC without losing a single deadline.

The current implementation of the copy unit is for uniprocessors. We plan to implement the copy unit in the CMP version of JOP. The copy unit needs to redirect access from all processors during the copy. Therefore, part of the functionality has to be placed after the memory arbiter. In a CMP setting with a time sliced arbiter the bandwidth is reserved for the copy task – the copy unit will act just like another CPU. In that case the copy task does not need to be interrupted as proposed for the uniprocessor version.

8. ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

9. REFERENCES

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[2] H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In P. C. G. L. Steele, Jr., editor, *LISP and Functional Programming. Conference Record of the 1984 ACM Symposium, Austin, Texas, August 6-8, 1984*, number ISBN 0-89791-142-3, New York, 1984. ACM.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[6] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7), pages 73–82, San Diego, CA, May 1993.

[7] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[8] M. Meyer. A true hardware read barrier. In E. Petrank and J. E. B. Moss, editors, *Proceedings of the 5th International Symposium on Memory Management (ISMM 2006)*, pages 3–16. ACM, June 2006.

[9] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 217–226, New York, NY, USA, 1993. ACM Press.

[10] K. D. Nilsen and W. J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–354, Dec. 1992.

[11] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[12] C. Pitter and M. Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[13] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[14] W. Puffitsch and M. Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[15] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[16] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

[17] M. Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 57–62, Vienna, Austria, September 2007. ACM Press.

[18] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[19] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[20] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[21] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems (CASES 2000)*, pages 9–17, New York, NY, USA, 2000. ACM.

[22] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

[23] G. L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[24] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan. 1994. Expanded version of the IWMM92 paper.

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

[26] M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Aug. 2007.