Hard Real-Time Garbage Collection for a Java Chip Multi-Processor

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

ABSTRACT

Garbage collection is a well known technique to increase program safety and developer productivity. Within the past few years, it has also become feasible for uniprocessor hard-real-time systems. However, garbage collection for multi-processors does not yet meet the requirements of hard real-time systems. In this paper, we present a hard real-time garbage collector for a Java chip multi-processor that provides non-disruptive and analyzable behavior.

For retrieving the references in local variables of threads, we propose a protocol that minimizes disruptions for high-priority tasks while still providing good bounds on the time until stack scanning finishes. Also, we developed a hardware unit that enables transparent, preemptible copying of objects, which eliminates the need to block tasks while copying objects.

Evaluation of the hardware shows that the copy unit introduces only little overhead and does not limit the critical path. Measurements resulted in release jitter for high-priority tasks of 224 µs or less on an embedded multi-processor with 8 cores clocked at 100 MHz. This indicates that with the proposed garbage collector, high scheduling quality and garbage collection do not contradict each other on chip multi-processors.

Categories and Subject Descriptors
D.3.4 [Language Constructs and Features]: Processors—Memory management (garbage collection); C.3 [Special-Purpose and Application-based Systems]: Real-time and embedded systems

General Terms
Algorithms, Measurement

Keywords
Garbage Collection, Real-Time, Java, Multi-Processor

1. INTRODUCTION

Real-time systems keep growing more and more complex. One means to fight this complexity and increase the productivity of developers is the usage of safe high-level languages such as Java. The initial revision of the real-time specification for Java (RTSJ) [5] introduced “scoped memory” so real-time tasks can avoid the uncertainties of garbage collection. However, in the past years, real-time garbage collection has become a mature technique, at least for uniprocessors. While there exist real-time garbage collectors for multi-processor systems, most of them at some point rely on mechanisms that are not suitable for hard real-time systems. In this paper, we present a garbage collector for a chip multi-processor that truly qualifies as hard real-time. All mechanisms provide predictable behavior with bounded execution times, from the garbage collector itself down to the arbitration of individual memory accesses.

There are two key contributions of this paper: First, we describe a protocol for the stack scanning phase of the garbage collector that minimizes disturbance of high-priority tasks while at the same time has low overhead and limits the time until stack scanning finishes. Second, we present a hardware unit that enables transparent, preemptible copying of objects during the compaction phase of the garbage collector. These features make garbage collection feasible for hard real-time systems that require high scheduling quality, especially with regard to release jitter.

This paper is organized as follows: The rest of this section briefly describes the key concepts of garbage collection and clarifies terminology. Section 2 discusses related work in the area of real-time garbage collection, focussing on scheduling of the garbage collector, the stack scanning phase and fragmentation handling. The runtime system the garbage collector is targeted at is described in Section 3, while the garbage collection algorithm itself is presented in Section 4. The implementation of that algorithm is evaluated in Section 5, with regard to the hardware overhead and its scheduling quality. Finally, Section 6 concludes the paper and provides an outlook on future work.

1.1 Garbage Collection Basics

Garbage collection automatically frees memory that is known not to be accessible to the application any more. Objects that are referenced by local or global variables always may be accessed by the application. These references form the roots of the object graph; the phase in which the garbage collector determines this root set is called root scanning. The part of this phase where local variables are scanned for references is called stack scanning. Starting from the roots, the garbage collector then traces, or marks, the object graph by following the references in the fields of objects. Afterwards, it frees the objects it has not visited during tracing in the sweep phase.

A garbage collector that implements these three phases is commonly called mark-sweep garbage collector. If it uses an additional phase to compact the heap and eliminate fragmentation, it is called mark-compact. In a copying garbage collector, tracing and heap
compaction is integrated. Such a garbage collector divides the heap into two semi spaces; during tracing, objects are copied from one semi space to the other. Upon the start of a garbage collection cycle, the notion of the two semi spaces is exchanged, or flipped.

There are several possible implementations for tracing the object graph. However, the tricolor abstraction [9] enables reasoning about tracing without being concerned with the details of the implementation. Objects that have been visited during tracing and do not need to be visited again are black. Objects are gray if the garbage collector is aware that they need to be visited. Unvisited objects are called white. At the end of tracing, the objects that are unreachable from the root set are white and can be reclaimed.

A garbage collector is concurrent if it can execute in parallel to the application. A parallel garbage collector can use more than one thread simultaneously. Note that a concurrent garbage collector is not necessarily parallel and vice versa. An incremental garbage collector breaks down the garbage collection work into smaller parts and can be interrupted between these steps. A stop-the-world garbage collector pauses all application threads to perform a whole garbage collection cycle; it is therefore neither concurrent nor incremental, but still potentially parallel. In the context of concurrent or incremental garbage collection, application threads are often also referred to as mutator threads, because they may modify the object graph during garbage collection.

Work-based garbage collectors perform garbage collection work when mutator threads allocate memory. The simplest implementation is to perform a full garbage collection cycle if an allocation request cannot be satisfied. Garbage collection can also be performed in a separate thread; this approach is called time-based, because time rather than the amount of memory allocations drives the execution of garbage collection work. These approaches can also be combined, by performing garbage collection work during allocations as well as in a separate thread that uses spare computing capacities. Another combination could be to trigger a garbage collection cycle of a time-based garbage collector only after a certain amount of memory has been allocated.

Please note that we use the term thread when referring to a thread of execution in general, and task when also taking into account a thread’s scheduling properties, such as its period and priority. For a specific instance of a task we use the term job.

2. RELATED WORK

There is a wealth of literature on garbage collection in general and real-time garbage collection in particular. In this section, we try to outline at least the most important principles and techniques in this area.

2.1 Scheduling

Most real-time garbage collectors use a time-based approach. One notable exception is Siebert’s garbage collector [29], which uses a work-based approach. However, more recent work on that garbage collector also integrates a time-based approach [31]. A combination of a work- and a time-based approach is also presented in [2].

Time-based approaches can be divided into two categories: approaches that schedule small quantums of garbage collection work at top priority, and approaches that schedule the garbage collection task at some lower priority. The latter approach is sometimes referred to as “slack-based”.

The policy to schedule small quantums of garbage collection work at top priority can be found in the Metronome garbage collector [3, 1] and in a garbage collector by Kalibera [16].

Scheduling garbage collection work at a priority between high-priority real-time tasks and other tasks was proposed by Henriks son [13]. The Schism [21] and the Java RTS garbage collectors [6] use this approach. Additionally, these collectors also provide means to control the scheduling parameters to enable adaption of the scheduling policy to the application. We also follow this approach. Most importantly, it avoids the problem with scheduling at top priority that the size of garbage collection quantums limits the granularity for scheduling.

2.2 Stack Scanning

While scanning the thread stacks for root references, care has to be taken to avoid inconsistencies. A straightforward approach is to stop all threads and perform stack scanning atomically with regard to all threads. As this may result in unacceptably long pauses, more sophisticated mechanisms have been devised.

The approach in [1] performs stack scanning atomically with regard to individual threads. To avoid inconsistencies, writes to references in the heap have to be guarded by a double barrier. In a double barrier, both the old value of a reference field and the reference to be written are marked gray, unless they are gray or black already. In [24], it has been shown that this can also be achieved by marking the old value and allocating new objects gray.

Doligez, Leroy and Gonzalez [11, 10] introduced the idea of making mutator threads responsible for stack scanning. This idea was adapted for real-time systems in [24, 27]; it is proposed that tasks scan their stack at the end of a job’s execution, where it is usually shallow. In this paper, we also follow this approach, but refine it to achieve tighter bounds on the time needed to complete stack scanning.

A different approach is used in Siebert’s garbage collectors [29, 31]. When a thread reaches a “safe point”, i.e., a point in the execution where it may be preempted, it scans its stack for references and saves them to a root array. After the start of a garbage collection cycle, the garbage collector has to wait until all threads have reached such a safe point and can then use the root arrays to construct the root set. This approach has the drawbacks that it introduces overhead to update the root arrays and requires additional memory for them.

Other approaches to reduce the blocking time related to stack scanning, such as stacklets [7] or return barriers [33] are sometimes discussed within the scope of real-time garbage collection. Due to their unpredictability we do not consider these mechanisms suitable for hard real-time systems.

2.3 Fragmentation

A hard real-time garbage collector must not void the real-time properties of the system as a whole. Obviously, it must not keep tasks from meeting their deadlines. However, it must also be possible to reason about the maximum memory consumption of a system. As fragmentation would make such reasoning difficult if not impossible, hard real-time garbage collectors must take it into account.

A solution to cope with fragmentation is to entirely avoid external fragmentation. This can be achieved by organizing the heap in blocks of a fixed size [29]. This eliminates external fragmentation at the expense of a considerable amount of internal fragmentation. Also, arrays have to be organized as trees to achieve at least logarithmic access time.

The Java RTS garbage collector [6] overcomes fragmentation by allocating objects in small fragments if necessary. The organization of fragmented objects is somewhat similar to the fixed block organization. Accesses to fragmented objects have a considerably
higher overhead than accesses to non-fragmented objects. Therefore, average-case performance and worst-case behavior differ significantly.

Fragmentation can also be eliminated by defragmenting the heap, which requires the relocation of objects. During relocation, writes to the object that is about to be copied can potentially be lost or cause inconsistent data. In order to avoid consistency issues, objects are often copied atomically. Especially for large arrays, this introduces unacceptably high blocking times. In the Metronome garbage collector [1], arrays are therefore organized as arraylets, which are similar to arrays in a fixed-block layout, but at the granularity of memory pages. Fragmentation is however not completely eliminated and objects and smaller array chunks still need to be copied atomically. The approach of the Schism garbage collector [21] combines a fixed-block layout for objects with arraylets to cope with fragmentation and achieve constant access times for arrays.

Consistency during object relocation can also be achieved by writing both the old and the new location of an object. Apart from the obvious drawback of making writes more expensive, care must be taken that these writes are not reordered arbitrarily. The approach in the Sapphire garbage collector [15] exploits the fact that the Java memory model does not require sequential consistency of data. A garbage collector by Kalibera [16] is targeted at unprocessors with green threads and therefore can avoid preemption between the two writes, but is not applicable to more general systems.

The Stopless garbage collector [20] ensures consistency during copying by clever use of compare-and-swap (CAS) operations. Unfortunately, copying may not terminate in adverse situations, which makes the approach unsuitable for hard real-time systems.

### 2.4 Copying Hardware

Using hardware to assist transparent relocation of objects has been proposed by Nielsen and Schmidt [19] and is also used in the SHAP processor [34]. The basic principle, namely to use hardware to redirect memory accesses during copying to the appropriate location, is the same as in our approach. However, these approaches implement substantial parts or all of the garbage collector in hardware. In contrast, our approach aims at a small and simple hardware implementation, while leaving most of the garbage collection logic in software. This paper also focusses more on the integration of the copy unit with the arbitration logic in a chip multi-processor than previous descriptions.

### 3. **RUN-TIME SYSTEM**

It is pointless to strive for a predictable garbage collector on top of an unpredictable run-time system. Just as a chain is only as strong as its weakest link, a component that does not meet the requirements for hard real-time systems spoils the properties of the whole system.

The target platform for the garbage collector described in this paper is a multi-processor version of the Java Optimized Processor (JOP) [25]. It is targeted at embedded hard real-time systems and aims at simplifying worst-case execution time (WCET) analysis. This means that provable execution time bounds must be available from execution of an individual instruction up to the scheduler and the garbage collector. In the following, we describe the key components of the run-time system that enable time-predictability in our multi-processor platform.

#### 3.1 Memory Access

The time to access memory must be bounded. While this is usually simple to achieve on a uniprocessor, it is less so on a multi-processor. The arbitration unit must guarantee a worst-case latency. In many systems, bandwidth is more important than latency, as long as requests are served eventually. For WCET analysis however, it is more important to achieve a low latency in order to minimize overestimations.

JOP supports both a time-division multiple access (TDMA) and a round-robin arbiter. While the former provides better predictability, the latter improves average-case performance. A round-robin arbiter is especially useful if there is a lot of locking, because the bandwidth of blocked cores can be used by running cores. However, we have not yet found a way to integrate this advantage into WCET analysis.

#### 3.2 Caching

Even with very fast backing memory, memory access times in CMPs often make caching worthwhile. Although there is work towards time-predictable data caches [28, 14], our WCET analysis tool does not yet take data cache hits into account. Therefore, we assume in the following that data memory is not cached. Future versions will include caching of data memory; as we decided to implement a moving garbage collector, the caching solution must allow either cross-core cache invalidations or implement cache coherence [22]. JOP does however contain a method cache and a stack cache. As methods are constant and stacks are thread-local, these caches do not require cache coherence protocols. Accesses to the stack are known to be cache hits on JOP and are therefore trivial to analyze. Cache hits and misses on the method cache can only occur on invoking or returning from a method, and an appropriate analysis is part of the WCET analysis tool.

#### 3.3 Synchronization

Synchronization takes place on more than one level. Low-level synchronization mechanisms operate on the hardware level. While they are very useful for fine-grained synchronization, they are often difficult to use both correctly and efficiently. High-level synchronization mechanisms such as software locks are usually simpler to use; their implementation however requires some form of low-level synchronization.

##### 3.3.1 Low-Level Synchronization

Modern processors usually provide support for synchronization, e.g., CAS operations. Algorithms that are based on CAS or similar operations usually guarantee progress of some thread. In hard real-time systems, it is however usually more important to ensure progress of a specific thread.

On JOP, the means for low-level synchronization is a single global hardware lock. Due to the implementation in hardware, the overhead for acquiring/releasing the lock is reduced to a few cycles and comparable to the execution time of a CAS operation.

The hardware implementation also ensures that threads are served round-robin: in an n-way CMP, a core must wait at worst until the other cores have finished their critical section.

A single global lock severely limits parallelism. Therefore, it is only used to guard the critical sections that are required for the implementation of higher-level synchronization mechanisms. As it has little overhead, it can be used for fine-grained locking with very small critical sections without being overly expensive.

##### 3.3.2 Language-Level Locking

Language level synchronization uses per-object locks, as usual in Java. When a task tries to acquire the lock, its priority is raised to a ceiling priority, which is higher than the priorities of all other
application tasks and makes it effectively non-preemptible. If the lock is already held by another task, the acquiring task spins until it becomes the head of the FIFO queue and the lock is released by its current holder. The task then acquires the lock and removes itself from the queue. After having released all (nested) locks, the task returns to its normal priority.

The synchronization algorithm can be seen as special case of the MSRP protocol described in [12], with all locks being considered global. However, we do allow nesting of global locks, as we would otherwise have to preclude all nesting. The protocol therefore does not avoid deadlocks. Deadlock freedom has to be proven by the application designer.

The time between trying to acquire a lock and actually acquiring it is bounded, because a task has to wait until at most \( n - 1 \) other requests have been served. As we assume that all tasks have a bounded execution time, critical sections also must have a bounded execution time. The \( n - 1 \) requests therefore can be served within a bounded time, provided that deadlock freedom can be proven.

### 3.4 Scheduling

Real-time scheduling for CMPs is considerably more complex than for uniprocessors [8]. A scheduler on a CMP does not only have to decide when something is executed, but also where. On the one hand, this additional dimension makes formal reasoning more challenging. On the other hand, practical issues like the cost of thread migration further complicate things.

Our approach on this issue is straightforward: We do not allow thread migration. Every thread can be executed on exactly one processor core. Within a core, we use static priority scheduling. For the scope of this paper, we assume that a rate-monotonic priority assignment is used.

The strict partitioning of threads reduces the number of schedulable task sets compared to more flexible scheduling algorithms. However, it provides the benefit that schedulability analysis is reduced to uniprocessor schedulability analysis in many regards. One aspect that can unfortunately not be simplified is the analysis of blocking times, which must still consider that synchronization may take place across cores.

### 4. HARD REAL-TIME GC

The usage of the term “real-time” for garbage collectors dates back to the late 1970s [4]. However, this term has been used in very different ways since then. In this paper, we talk about hard real-time systems, i.e., systems where failure to meet a deadline may have catastrophic consequences. This means that a number of techniques for garbage collectors that are casually referred to as real-time collectors are not suitable in this context. The probably most prominent technique is generational garbage collection, as real-time collectors are not suitable in this context. The probability of these bounds must be provable, and violation of these bounds must be impossible, not merely very unlikely. As example consider an algorithm that loops until a compare-and-swap operation succeeds. For usage in a hard real-time system, it must be proven that the loop terminates after a finite number of steps. Showing that it usually does so in practice is not sufficient.

A hard real-time garbage collector must be predictable with regard to both time and memory. On the one hand, it must not void the system’s abilities to meet its hard real-time requirements. On the other hand, it must be possible to reason about the maximum memory consumption of a system. The execution time of the garbage collection algorithm must be bounded, and it must be provable that garbage will be eventually collected.

#### 4.1 General Algorithm

According to the classification presented in Section 1.1, the garbage collection algorithm presented in this paper is concurrent and incremental. This means that it executes while mutator threads are executing and that it may be preempted. However, apart from stack scanning, it does not qualify as parallel garbage collector.

The decision not to parallelize the garbage collector was taken consciously. Garbage collection is a memory-bound task. On usual CMPs, parallelization also provides more bandwidth to the garbage collector and consequently provides a speed-up. On JOP in contrast, we can control the arbitration policy for the individual processor cores, and can provide the garbage collector with more bandwidth without the overhead for parallelization. Also, by avoiding parallel marking, the execution time of the marking phase does not depend on the structure of the object graph – for example, a simple linked list is naturally sequential and cannot be traced in parallel. For a discussion of limits of parallel marking, please refer to [30].

The fundamental algorithm behind our garbage collector is copying garbage collection, with extensions for making it incremental and concurrent. The initial design and the considerations behind it are best described in Chapter 7 of the JOP handbook [25]. Many of the design decisions of the initial algorithm can be found also in the algorithm described in this paper. However, the stack scanning phase has changed considerably to reduce blocking and ensure correctness on CMPs. Also, the copying unit described in Section 4.3, which is necessary to achieve concurrent copying on CMPs, has been developed since. The extensions described in this paper do not rely on a copying garbage collector and may be also applied to mark-compact garbage collectors.

Our garbage collector uses a handle-based object layout. In such a layout, meta data like the type of an object is located in a separate memory area instead of being located in a header to the actual object data. As references point to the handle, accesses to object fields have to follow an indirect route to the actual object data. Figure 1 shows a handle-based layout, with references on the stack pointing to the handles and indirection pointers in the handles pointing to the actual object data in the heap.

An advantage of a handle-based object layout is that only a single location, the indirection pointer, needs to be updated when an object is relocated. As all handles are the same size, the handle area itself is not subject to fragmentation and handles do not need to be relocated.1 In contrast, a layout with object headers, where references directly point to the object data, would require to patch the object graph and the thread stacks when moving objects. An alternative would be a fixed-block layout [29]; however, it is not yet clear whether such an approach would provide substantial benefits over our current approach. A fair comparison would at least have to include a study on how the results for space overheads of a copying collector [26] and a fixed-block layout [23] compare in practice and whether the logarithmic access time for arrays in a fixed-block layout can be compensated by avoiding copying. Contrary to popular belief, a copying collector does not require twice the amount of memory of a mark-compact collector. Rather, the overhead depends on the amount of live memory and the allocation rates of mutator threads, which makes a comparison more complex than it might appear at first sight.

1The fact that the part of the heap that is reserved for handles cannot be used by objects and vice versa may be regarded as a form of fragmentation, though with different implications.
The garbage collector described in this paper uses eight word handles. While this may seem rather large compared to other garbage collectors, the handle includes apart from the indirection pointer also the object type and size, data structures for garbage collection, and a pointer to a lock. The size of the handles could be reduced to six words, though with some impact on the performance. At the moment, we do not think that the benefits from reduced memory consumption grant the performance costs, but future developments might change this.

The correctness of incremental marking is ensured by a snapshot-at-beginning barrier [32]. In order to allow modification of the object graph during stack scanning, our garbage collector supports both double barriers [1] and anthracite allocation of new objects (i.e., new objects do not need to be copied, but are pushed onto the work list of the tracing algorithm and scanned for references). The proof of correctness for the latter option can be found in [24].

As pointed out in Section 2, the garbage collector presented in this paper is time-based; garbage collection takes place in a separate, periodic task. In line with traditional scheduling theory, the priority for the garbage collection task is preferably chosen using rate-monotonic priority assignment. The maximum period for the garbage collection task depends on the heap size, the allocation rate, and the amount of live memory. Details on how such a bound can be computed can be found in [26].

A garbage collector usually consumes considerably more execution time per release than other real-time tasks. In order to be able to meet its deadline, its period will typically also be larger than the periods of other tasks. In the following, we therefore assume that the garbage collector executes at lowest priority, as entailed by rate-monotonic priority assignment for the task with the longest period. However, the proposed solutions do not necessarily require this assumption, and tasks with longer periods or non-real-time tasks may execute at lower priorities than the garbage collector.

### 4.2 Stack Scanning

The garbage collector presented in this paper uses the approach from [24] that tasks scan their stack at the end of a job’s execution, where it is usually shallow. While scanning the stack at the end of execution reduces the effort, it introduces a delay for the garbage collector – garbage collection cannot proceed until stack scanning has completed. In general, this delay depends on the period or the response time of the tasks (depending on the implementation). Tasks with long response times can slow down garbage collection to the point where it cannot keep up with the allocations any more.

To overcome this limitation, we propose a strategy that does not block high-frequency tasks while still providing reasonable bounds for the time to complete stack scanning.

- High-frequency tasks scan their own stack. They can do so when their stack is shallow, which minimizes overhead. As they execute at a high frequency, the time until up-to-date roots are available to the garbage collector remains short.
- The stacks of low-frequency tasks are scanned by stack scanning events. These events have a priority that is higher than the priority of the tasks whose stacks it scans, but lower than the priority of high-frequency tasks. Therefore, stack scanning of these events can be accounted for with usual schedulability analysis.

There is usually one stack scanning event per core, which scans the stacks of all lower-priority tasks. On cores where all tasks scan their own stack, the stack scanning event may be omitted. In theory, it would be possible to have more than one stack scanning event. However, we believe that situations where this could provide benefits are rare in practice. Therefore, and for the sake of simplicity, we assume that there is at most one stack scanning event per core in the following.

The firing of stack scanning events is done via cross-core interrupts, which trigger the scheduler. As this interrupt is masked during the execution of high-priority tasks, we achieve low latency without disturbing the execution of high-priority tasks.

Figure 2 exemplifies the proposed stack scanning scheme. Tasks $\tau_1$ to $\tau_6$ are application tasks, $\tau_{s1}$ and $\tau_{s2}$ are stack scanning events, and $\tau_{gc}$ is the garbage collection task. Shaded areas indicate stack scanning, while white areas denote regular execution. The task set executes on two processor cores, as indicated by the line between $\tau_1$ and $\tau_2$. Tasks $\tau_1$ and $\tau_4$ have quite tight deadlines; they can afford to scan their own stacks, but any preemption or delay would cause them to miss their deadlines.

At time 0, the garbage collector starts a new cycle by flipping the notion of the semi spaces, and subsequently triggers the stack scanning events for both cores; the dashed arrow represents the firing of the event. Execution of $\tau_{s1}$ is delayed by the execution of $\tau_2$. Task $\tau_{s2}$ starts its execution immediately, just to be preempted by $\tau_4$ at time 5. Tasks $\tau_1$, $\tau_2$ and $\tau_4$ scan their own stacks and are not disturbed by stack scanning. At time 15, $\tau_1$, $\tau_2$ and $\tau_4$ have scanned their own stack, and the stacks of tasks $\tau_3$, $\tau_5$ and $\tau_6$ have...
been scanned by the stack scanning events. At this point, $\tau_{gc}$ can proceed and begin to traverse the object graph.

As usual, we use $T_i$ for the period of task $\sigma_i$ and $R_i$ for its response time. Furthermore, $\sigma$ is the set of tasks that scan their own stack, and $\rho$ is the set of stack scanning events. For a task $\sigma_i \in \sigma$, the time to scan its own stacks is part of the WCET and therefore included in $R_i$. For the events in $\rho$, the WCET includes the time to scan the stacks of all lower-priority tasks on the same core. An upper bound for the maximum time between triggering stack scanning and its completion, $t_{stackscan}$, is given by

$$t_{stackscan} = \max_{\sigma_i \in \sigma} (\max_{i \in \sigma} T_i + R_i, \max_{i \in \rho} R_i)$$

For the tasks in $\sigma$, we must take into account both their period and their response time. In the worst case, the garbage collector notifies a task $\sigma_i$ that it should scan its stack just after the task has decided not to do so. The garbage collector then has to wait for $T_i$ until the task is released again, and $R_i$ until the task has finally scanned its stack. As the events in $\rho$ are triggered by the garbage collector, it is sufficient to take their response time into account.

The garbage collector needs to wait until stack scanning has finished until it may proceed. Therefore, the choice which tasks scan their own stack and which have their stack scanned scanned by a stack scanning event influences its response time. This choice also influences the response time of the stack scanning events. Finding an optimal solution and integrating the above results with schedulability analysis however go beyond the scope of this paper and remain future work.

### 4.3 Copying

When relocating objects during garbage collection, care has to be taken to keep data consistent. A write to an object that is about to be copied could be missed if the write goes to the old location of the object, or might be overwritten if it goes to the new location of the object.

The location presented in this paper builds on a hardware unit to support transparent, preemptible copying of objects. A uniprocessor version of such a unit was presented in [27]. The unit presented in that paper is part of a core’s memory unit and redirects memory accesses to fields that already have been copied to the new location. However, as this unit is local to a core, it can do the translation only for a single core.

For a CMP copy unit, we face the following requirements: First, copying must be preemptible, transparent and must retain data consistency. Second, copying must not influence the arbitration of memory requests of other cores. Third, it should be possible to reuse existing arbiters. The first requirement basically states what we want to achieve. The second requirement entails that we cannot simply lock the arbiter in order to achieve consistency, because doing so would influence the timing of memory accesses for other cores. Consistency must be achieved by taking into account memory access from other cores between reading and writing the data to be copied rather than preventing the other cores from accessing memory. Also, the copy unit should either have its own slot for arbitration or use the slots of the core that issues the copying command rather than stealing bandwidth from other cores. The third requirement takes into account the fact that arbiters are non-trivial pieces of hardware and rewriting an arbiter is not a task to be taken lightly. However, the requirement slightly complicates copying, as even a TDMA arbiter reorders memory accesses and it might not be possible to tell copying related memory accesses from ordinary memory accesses after the arbitration.

Figure 3 shows a block diagram of the copy unit. While the connections from cores 1 to $n−1$ are forwarded directly to the arbiter, core 0 is special and the only core on which the garbage collector may execute. Normally, accesses from core 0 are also just forwarded to the arbiter. However, if core 0 triggers a copying step, the copy logic uses the arbitration slot of core 0 to read from the source and write to the destination location. After arbitration, the redirection logic changes the addresses of memory accesses if necessary. It is connected to the copy logic in order to communicate whether copying is in progress, the source, destination and offset of the current copying process. As a location that is about to be copied may be written to, the redirection unit must also detect such writes and update the internal buffer accordingly. While this is to some degree similar to cache coherence, it is fortunately simpler to retain consistency on a single word than on a whole cache.

Listing 1 shows how the copy unit is utilized by the garbage collector. As the ports of the copy unit are part of the local I/O space, accesses to them are not subject to arbitration. In the first lines, the source and destination for copying are stored in the copy unit, and the unit is activated. The actual copying is issued by writing the position to be copied to address Const.IO_CCCP_POS. After updating the redirection pointer, it is necessary to wait before turning off the copy unit again. Otherwise, other cores might use the old location for accesses without redirection, resulting in inconsistent data. The value of the redirection pointer is stored locally only in a few spots in the run-time system. Analysis of the respective code fragments showed that the time between reading and using an indication pointer is always bounded. Therefore, a simple delay loop is sufficient to ensure correctness.2

Listing 1: Copying code

```c
// copy object
Native.wr(addr, Const.IO_CCCP_SRC);
Native.wr(dest, Const.IO_CCCP_DST);
Native.wr(1, Const.IO_CCCP_AGT);

for (i = 0; i < size; i++) {
    // @WCA loop <= MAX_SEMI_SIZE outer
    Native.wr(i, Const.IO_CCCP_POS);
}

// update object pointer to the new location
Native.wr(dest, rel-OFF_PTR);
// wait until everybody uses the new location
for (i = 10; i > 0; --i); // @WCA loop = 10
// turn off address translation
Native.wr(0, Const.IO_CCCP_AGT);
```

2Of course, it must be guarded appropriately against being eliminated by optimizations.
therefore relates the loop bound to the bound of the outer space can be copied during the whole tracing phase. The annotation can cause random variations of the results.

The simulation tasks S1, S2, and S3 are not hard real-time tasks, but are needed to simulate the environment and provide a self-contained benchmark. They are grouped on cores 1 and 2 to isolate the exception of the simulation tasks S1, S2 and S3, tasks with a higher priority than the stack scanning events. The tasks are sorted vertically to respect their priorities. For example, A3 has a higher priority than the stack scanning event SE on that core. Tasks A1 to A7 and F1 to F4 are considered real-time tasks and have to meet their deadlines. With the exception of the simulation tasks S1, S2 and S3, tasks with a period of 50 ms or less scan their own stacks and therefore have a higher priority than the stack scanning events. The cut-off period of 50 ms was chosen in order to have the stack scanning events at their priorities. For example, A3 has a higher priority than the stack scanning event SE on that core. Tasks A1 to A7 and F1 to F4 are considered real-time tasks and have to meet their deadlines. With the exception of the simulation tasks S1, S2 and S3, tasks with a period of 50 ms or less scan their own stacks and therefore have a higher priority than the stack scanning events. The cut-off period of 50 ms was chosen in order to have the stack scanning events at different relative priorities (top or medium priority) on the individual cores. On cores 3 and 4, there are no low-priority tasks, so the stack scanning events could be omitted. Relative priorities within a core are the same as in the original benchmark, except for A2 and A6, where the relative priorities are inverted so A2 has a higher priority than the stack scanning event. The priorities are mostly rate-monotonic, with the exception being the simulation tasks S2 and S3.

5. Evaluation

5.1 Hardware

For the evaluation, we use the CMP version of JOP with 8 cores, which runs at 100 Mhz on an Altera DE2-70 board. This board contains 2 MB synchronous SRAM; both reads and writes take 3 cycles.

In order to assess the overhead for the copy unit, we synthesized a plain variant of the processor and a variant of the processor that includes the copy unit. In order to be useful, the size of the copy unit must not slow down the critical path, which would require to run the processor at a lower frequency. Also, it must be reasonably small so the associated cost remains low.

The original version of the TDMA arbiter allows zero-cycle arbitration, i.e., cores that issue a memory access are granted access immediately if they happen to hit their slot. In a configuration with 8 cores, 3-cycle memory accesses and uniform 3-cycle slots, this results in a worst-case memory access time of 26 cycles. Two cycles have to be taken into account, because accesses can only be issued in the first cycle of the slot – otherwise the access would occupy parts of the next core’s slot. It then takes 21 cycles until the slots of the other cores have passed, and further 3 cycles for the actual memory access to be completed.

Unfortunately, zero-cycle arbitration results in a relatively long path from the core to the memory interface. Inserting more logic into that path would have lowered the maximum frequency considerably. Therefore, we decided to make the arbiter fully registered. This results in a slightly higher worst-case memory latency of 27 cycles instead of 26 cycles – one cycle being added by the registers – but does not reduce the available bandwidth and allows the processor to run at the maximum frequency the core provides.

Table 1 shows the results of the hardware synthesis for three variants of JOP. The line labeled “zero-cycle” shows the results for the original version of JOP. It is notable that the maximum frequency is only 93.51 MHz; the critical path is between the cores and the memory interface. Breaking the critical path by replacing the zero-cycle arbiter with a registered arbiter increases the maximum frequency to 99.86 MHz. The reduced pressure on the paths between the cores and the memory interface leads to different optimizations being applied by the synthesis tool. Therefore, this version requires more logic cells (LCs) to be implemented, but fewer registers.

Optimization effects also lead to the paradoxical effect that adding the copy unit to JOP increases the maximum frequency to 100.03 MHz. As the critical paths are in a part of the processor that did not change between the two versions, we assume that the difference is caused by the heuristic nature of fitting, place and route algorithms, where even minimal changes to the hardware design can cause random variations of the results.

5.2 System Evaluation

As benchmark we use a Java port of the Papabench [18] benchmark, jPapabench [17]. The benchmark implements an autopilot for an unmanned aerial vehicle and was ported to Java in order to provide a realistic benchmark for real-time Java implementations. Table 2 shows the task set of jPapabench, including the garbage collection task. The periods for the tasks are the same as in the original benchmark. Table 3 shows how the assignment of tasks to individual cores, including the garbage collection task and the stack scanning events. The tasks are sorted vertically to respect their priorities. For example, A3 has a higher priority than the stack scanning event SE on that core. Tasks A1 to A7 and F1 to F4 are considered real-time tasks and have to meet their deadlines. With the exception of the simulation tasks S1, S2 and S3, tasks with a period of 50 ms or less scan their own stacks and therefore have a higher priority than the stack scanning events. The cut-off period of 50 ms was chosen in order to have the stack scanning events at different relative priorities (top or medium priority) on the individual cores. On cores 3 and 4, there are no low-priority tasks, so the stack scanning events could be omitted. Relative priorities within a core are the same as in the original benchmark, except for A2 and A6, where the relative priorities are inverted so A2 has a higher priority than the stack scanning event. The priorities are mostly rate-monotonic, with the exception being the simulation tasks S2 and S3.

The simulation tasks S1, S2, and S3 are not hard real-time tasks, but are needed to simulate the environment and provide a self-contained benchmark. They are grouped on cores 1 and 2 to isolate

<table>
<thead>
<tr>
<th>Configuration</th>
<th>LCs</th>
<th>Registers</th>
<th>( f_{\text{max}} ) (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero-cycle</td>
<td>45 077</td>
<td>14 079</td>
<td>93.51</td>
</tr>
<tr>
<td>registered</td>
<td>45 308</td>
<td>13 677</td>
<td>99.86</td>
</tr>
<tr>
<td>copy unit</td>
<td>45 525</td>
<td>14 275</td>
<td>100.03</td>
</tr>
</tbody>
</table>

Table 1: Synthesis results

<table>
<thead>
<tr>
<th>#</th>
<th>Task</th>
<th>Period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>RadioControl</td>
<td>25</td>
</tr>
<tr>
<td>A2</td>
<td>Stabilization</td>
<td>50</td>
</tr>
<tr>
<td>A3</td>
<td>LinkFBWSend</td>
<td>50</td>
</tr>
<tr>
<td>A4</td>
<td>Reporting</td>
<td>100</td>
</tr>
<tr>
<td>A5</td>
<td>Navigation</td>
<td>250</td>
</tr>
<tr>
<td>A6</td>
<td>AltitudeControl</td>
<td>250</td>
</tr>
<tr>
<td>A7</td>
<td>ClimbControl</td>
<td>250</td>
</tr>
<tr>
<td>F1</td>
<td>TestPPM</td>
<td>25</td>
</tr>
<tr>
<td>F2</td>
<td>SendDataToAutopilot</td>
<td>25</td>
</tr>
<tr>
<td>F3</td>
<td>CheckFailSafe</td>
<td>50</td>
</tr>
<tr>
<td>F4</td>
<td>CheckMega128Values</td>
<td>50</td>
</tr>
<tr>
<td>S1</td>
<td>SimulatorFlightModel</td>
<td>25</td>
</tr>
<tr>
<td>S2</td>
<td>SimulatorGPS</td>
<td>250</td>
</tr>
<tr>
<td>S3</td>
<td>SimulatorIR</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 2: jPapabench tasks

<table>
<thead>
<tr>
<th>Priority</th>
<th>Core</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td></td>
<td></td>
<td>F1</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A3</td>
<td>F3</td>
<td>F4</td>
<td>A1</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SE</td>
<td>SE</td>
<td>SE</td>
<td>SE</td>
<td>SE</td>
<td>SE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
<td>A4</td>
<td>S1</td>
<td>S2</td>
<td></td>
<td>A7</td>
<td>A6</td>
<td>A5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GC</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Task partitioning

The size of the copy unit is not exactly the difference between the “registered” and the “copy unit” configurations. The actual size of the copy unit is 347 LCs, of which 178 are registers. Compared to the overall size, we consider this overhead negligible.
their execution from the scheduling of the hard real-time tasks. Furthermore, tasks from the fly-by-wire module F1 to F4 are grouped on core 3 and 4, while the autopilot tasks A1 to A7 are grouped on cores 5, 6 and 7. As jPapabench is not a complete port of the original Papabench, the tasks A3 and A4 are empty. To evaluate the impact of the garbage collector on tasks that run on the same core, we use these tasks without compromising the correctness of the overall system and assigned them to core 0.

### 5.2.1 Analysis

In order to get a better understanding of the behavior we can expect for the benchmark, we analyzed the WCETs of the individual tasks. The reported times do not include blocking, but otherwise reflect the worst case with regard to timing, even in the implementation of complex bytecodes such as checkcast. The analysis results are shown in Table 4.

For most tasks, the execution times seem to be reasonable. For A5 and the simulation tasks however, the WCETs seem unreasonably high. Further investigation revealed that these tasks use floating-point operations. These are implemented in software on JOP, which is costly even when considering average-case performance. As the analysis must assume the worst-case behavior for every single operation, the WCET contains considerable overestimation.

When comparing the WCETs and the task periods, it becomes clear that the tasks that use floating-point operations potentially violate their deadlines. For the scope of the evaluation, we left the tasks and the periods as they were. In a real system of course, appropriate measures would have to be taken to avoid the risk of deadline violations.

Apart from the execution time, we also analyzed the worst-case heap allocations (WCHAs). We analyzed the number of objects that are allocated per release and their raw size separately. In total, the tasks allocate 504 objects consuming 9.75 KB per second. When taking into account not only the actual object data, but also the eight words of the meta data in the handles, the byte per second figure rises to 25.50 KB. While these figures are probably not impressive, please remember that we aim to evaluate predictability rather than performance. Of course, we would have liked to increase the task frequencies to put higher stress onto the garbage collector. However, the tasks that use floating-point operations already require a higher budget than we can provide. Therefore, we have refrained from doing so.

We also performed a preliminary WCET analysis on the garbage collector. The analysis reported a WCET of around 25 seconds. However, we are aware of several sources of overestimations in the analysis. For example, we can formulate constraints that the number of handles limits the number of objects and the number of arrays to consider for tracing, but not that the sum of objects and arrays cannot be larger than the number of handles. Also, it is assumed that all handles must be swept, although an object that has been visited during tracing must not have its handle swept. The tightness of the WCET bound could also be improved by taking into account the number of reference fields in objects instead of assuming in the analysis that every field might be a reference. Eliminating these sources of overestimation and potentially other, less obvious ones, requires considerable changes in our WCET analysis tool and remains future work.

Interestingly, copying is cheap compared to other parts of the garbage collector, being responsible for 117 ms of the overall WCET. Although this is not a definite proof, it indicates that avoiding copying is unlikely to provide a significant performance gain.

### 5.2.2 Measurements

For our measurements, we used the BraunschweigFlightplan, which is part of jPapabench and takes about 200 seconds to complete. In order to maximize the interference from the garbage collection task, we chose a period of 199 ms, which is roughly its average observed response time and is not evenly divided by the periods of other tasks. We measured both with all release offsets set to 0 ms and with release offsets such that higher-priority tasks are released 1 ms after lower-priority tasks.

The maximum observed release jitter (MORT) and the maximum observed release jitter (MOJ) over ten benchmark runs are shown in Table 5, both for a version with atomic copying and a version with the copy unit. The MORT is the time between the (theoretical) release of a job and its observed completion. It includes release jitter, blocking times, and scheduling overhead. Therefore, it cannot be related directly to the worst-case execution times of the tasks that are executed on the respective core. The MOJ is the difference of the earliest and the latest start of the execution of a job relative to its release time. The results for S1 are not shown, be-

<table>
<thead>
<tr>
<th>#</th>
<th>WCET (µs)</th>
<th>WCHA objects</th>
<th>WCHA bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>3502</td>
<td>3</td>
<td>68</td>
</tr>
<tr>
<td>A2</td>
<td>4697</td>
<td>3</td>
<td>68</td>
</tr>
<tr>
<td>A3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A5</td>
<td>155511</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A6</td>
<td>636</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A7</td>
<td>2636</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F1</td>
<td>299</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F2</td>
<td>2580</td>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>F3</td>
<td>112</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F4</td>
<td>2161</td>
<td>3</td>
<td>68</td>
</tr>
<tr>
<td>S1</td>
<td>579361</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>S2</td>
<td>397641</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>S3</td>
<td>397641</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4: Analysis results

<table>
<thead>
<tr>
<th>#</th>
<th>atomic copy MORT (µs)</th>
<th>MOJ (µs)</th>
<th>copy unit MORT (µs)</th>
<th>MOJ (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>1826</td>
<td>870</td>
<td>533</td>
<td>65</td>
</tr>
<tr>
<td>A2</td>
<td>3904</td>
<td>921</td>
<td>2622</td>
<td>9</td>
</tr>
<tr>
<td>A3</td>
<td>989</td>
<td>982</td>
<td>145</td>
<td>139</td>
</tr>
<tr>
<td>A4</td>
<td>3536</td>
<td>3529</td>
<td>2174</td>
<td>2168</td>
</tr>
<tr>
<td>A5</td>
<td>22835</td>
<td>381</td>
<td>24793</td>
<td>12</td>
</tr>
<tr>
<td>A6</td>
<td>3935</td>
<td>3639</td>
<td>3502</td>
<td>3123</td>
</tr>
<tr>
<td>A7</td>
<td>3449</td>
<td>1832</td>
<td>2461</td>
<td>964</td>
</tr>
<tr>
<td>F1</td>
<td>1188</td>
<td>1171</td>
<td>103</td>
<td>86</td>
</tr>
<tr>
<td>F2</td>
<td>1261</td>
<td>1239</td>
<td>246</td>
<td>224</td>
</tr>
<tr>
<td>F3</td>
<td>1605</td>
<td>1588</td>
<td>760</td>
<td>743</td>
</tr>
<tr>
<td>F4</td>
<td>4225</td>
<td>1863</td>
<td>2407</td>
<td>764</td>
</tr>
<tr>
<td>S1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>S2</td>
<td>39900</td>
<td>414</td>
<td>38524</td>
<td>76</td>
</tr>
<tr>
<td>S3</td>
<td>44511</td>
<td>39616</td>
<td>43012</td>
<td>37979</td>
</tr>
</tbody>
</table>

Table 5: Measurement results
cause that task almost always violated its deadline, which rendered the measurement results useless.

In order to assess the MOJ we can expect to see from the benchmark, we measured the MOJ for a set of dummy tasks that continuously increments a shared counter within a synchronized block and an empty high-priority task. For the high-priority task, we observed a MOJ of 199 µs. While this may seem to be a rather large amount of jitter, it must be noted that a high-priority task cannot execute while a low-priority task on the same core waits for a lock or is within a synchronized block. In the worst case, the low-priority task has to wait until tasks on the N − 1 other cores have completed their critical section. As critical sections in the benchmark do more useful work than just incrementing a counter, we can expect a somewhat higher jitter than 200 µs.

With atomic copying, we observe a jitter of several hundred µs even for the highest priority tasks. This indicates that copying has a considerable effect on the scheduling quality. With the copy unit in contrast, we see release jitter of well under 100 µs for tasks A1, A2, A5, F1, and S2. Scheduling on JOP includes the replacement of the stack cache contents upon tasks switches; considering that 1 µs is sufficient only to load 4 words of memory, we think that the observed jitter is highly satisfactory. For task F2, we observe jitter of 224 µs even with the copy unit. This can be explained by the fact F4 cannot be preempted while holding the garbage collector lock during memory allocations and therefore delays the release of F2. Compared with the 200 µs observed for dummy tasks, the MOJ for F2 seems to be reasonable. The MOJ for A3 is also satisfying, which indicates that it is feasible to execute high-priority tasks on the same core as the garbage collector.

The maximum observed time for stack scanning was 44.4 ms, which is inline with the theoretical bound presented in Section 4.2. According to the task periods and the measured response times, up to around 60 ms would have been acceptable.

6. CONCLUSION

In this paper, we presented a garbage collector that is suitable for hard real-time systems on chip multi-processors. The proposed stack scanning protocol requires little overhead, provides reasonable timing bounds, and minimizes interference between garbage collection and high priority tasks. We also presented a hardware copy unit that enables transparent and preemptible copying on a chip multi-processor. In measurements, the release jitter of high-priority tasks was reduced significantly. The results indicate that high scheduling quality on chip multi-processors is achievable even in the presence of garbage collection.

Future work will focus on the analysis of the garbage collector; WCET analysis currently suffers from severe overestimations. These overestimations need to be reduced in order to compute bounds that are not only safe but also tight.

Availability

The platform described in this paper is open source and available via git clone git@www.soc.tuwien.ac.at:jop.git. The version used for the evaluation is located in the branch rtpc and tagged [test1]. For detailed instructions how to reproduce the experiments, please refer to file README.JTRES2011.

7. REFERENCES


