Nonblocking Real-Time Garbage Collection

MARTIN SCHOEBERL and WOLFGANG PUFFITSCH
Vienna University of Technology

A real-time garbage collector has to fulfill two basic properties: ensure that programs with bounded allocation rates do not run out of memory and provide short blocking times. Even for incremental garbage collectors, two major sources of blocking exist, namely, root scanning and heap compaction. Finding root nodes of an object graph is an integral part of tracing garbage collectors and cannot be circumvented. Heap compaction is necessary to avoid probably unbounded heap fragmentation, which in turn would lead to unacceptably high memory consumption. In this article, we propose solutions to both issues.

Thread stacks are local to a thread, and root scanning, therefore, only needs to be atomic with respect to the thread whose stack is scanned. This fact can be utilized by either blocking only the thread whose stack is scanned, or by delegating the responsibility for root scanning to the application threads. The latter solution eliminates blocking due to root scanning completely. The impact of this solution on the execution time of a garbage collector is shown for two different variants of such a root scanning algorithm.

During heap compaction, objects are copied. Copying is usually performed atomically to avoid interference with application threads, which could render the state of an object inconsistent. Copying of large objects and especially large arrays introduces long blocking times that are unacceptable for real-time systems. In this article, an interruptible copy unit is presented that implements nonblocking object copy. The unit can be interrupted after a single word move.

We evaluate a real-time garbage collector that uses the proposed techniques on a Java processor. With this garbage collector, it is possible to run high-priority hard real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz system.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*

General Terms: Performance, Design

Additional Key Words and Phrases: Garbage collection, real-time, root scanning, nonblocking copying

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).

Authors' address: Martin Schoeberl and Wolfgang Puffitsch, Institute of Computer Engineering, Vienna University of Technology, Treitlstr. 3, A-1040 Vienna, Austria, email: {mschoeberl; wpuffitsch}@mail.tuwien.ac.at.

DOI 10.1145/1814539.1814545 http://doi.acm.org/10.1145/1814539.1814545

ACM Reference Format:
Schoeberl, M. and Puffitsch, W. 2010. Nonblocking real-time garbage collection. ACM Trans. Embedd. Comput. Syst. 10, 1, Article 6 (August 2010), 28 pages.
DOI = 10.1145/1814539.1814545 http://doi.acm.org/10.1145/1814539.1814545

1. INTRODUCTION

Garbage collection (GC) is a feature of 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) [Bollella et al. 2000], which targets soft real-time systems, defines an additional memory model, with immortal and scoped memory, to avoid GC.

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, runtime 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 GC in real-time systems.

In a system with a concurrent garbage collector, the GC thread and the mutators (i.e., the application threads, which mutate the object graph) 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. In this article, we propose two solutions that eliminate the blocking of these two tasks.

On the software side, we integrated the proposed solutions into a copying GC algorithm. The hardware portions of the presented approaches were implemented in the Java processor JOP [Schoeberl 2008], which runs at 100MHz. This platform was used to evaluate the usefulness of our concepts. It is possible to run a 10kHz high-priority task without a single deadline miss with ongoing GC. The maximum task frequency is limited by the scheduler and not by the garbage collector. It has to be noted that the proposed root scanning strategy and copy unit are not JOP specific. The copy unit can also be integrated in a standard RISC processor that executes compiled Java.

This article is based on prior work on nonblocking root scanning [Puffitsch and Schoeberl 2008] and nonblocking object copy [Schoeberl and Puffitsch 2008]. The evaluation section provides the results for the combination of both concepts. The article is organized as follows. In the remainder of this section, we discuss the issues to be solved in the areas of root scanning and object copy in a real-time garbage collector. Section 2 provides an overview of the related work in these fields. In Section 3, our solutions to make root scanning preemptible are presented. A hardware unit to allow nonblocking copying of objects is proposed in Section 4. Section 5 provides details of our implementation, which is then evaluated in Section 6. Section 7 concludes the article and provides an outlook on future work.

1.1 Root Scanning

Tracing garbage collectors traverse the object graph to identify the set of reachable objects. The starting point for this tracing is the *root set*, a set of objects that is known to be directly accessible. On the one hand, these are references in global (static in Java) variables, on the other hand these are references that are local to a thread. The latter comprise the references in a thread's runtime stack and thread-local CPU registers. The garbage collector must ensure that its view of the root set is consistent before it can proceed, otherwise objects could be erroneously reclaimed.

For stop-the-world garbage collectors, the consistency of the object graph is trivially ensured. Incremental garbage collectors however require the use of *barriers*, which enforce the consistency of marking and the root set [Baker 1978; Dijkstra et al. 1978; Steele 1975; Yuasa 1990]. While barriers are an efficient solution for the global root set, they are considered to be too inefficient to keep the local root sets consistent. Even frequent instructions like storing a reference to a local variable would have to be guarded by such a barrier, which would cause a considerable overhead and make it difficult if not impossible to compute tight bounds for the worst-case execution time (WCET). The usual solution to this problem is to scan the stacks of all threads in a single atomic step and stall the application threads while doing so.[1] The atomicity entails that the garbage collector may not be preempted while it scans a thread's stack, which in turn causes a considerable release jitter even for high-priority threads.

However, the atomicity is only necessary with respect to the thread whose stack is scanned, because a thread can only modify its own stack. If the garbage collector scans a thread's stack, the thread must not execute and atomicity has to be enforced. Other mutator threads are allowed to preempt the stack scanning thread. If a thread scans its own stack, it is not necessary to prohibit the preemption of the thread – when the thread continues to execute, the stack is still in the same state and the thread can proceed with the scanning without special action. Consequently, preemption latencies due to root scanning can be avoided. With such a strategy, it is also possible to minimize the overhead for root scanning. It can be scheduled in advance such that the local root set is small at the time of scanning.

In this article, we present two solutions for periodic and sporadic threads that make use of this approach, and evaluate their trade-offs. Furthermore, we show how the worst case time until all threads have scanned their stacks can be computed.

1.2 Object Copy

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 [Wilson and Johnstone 1993] is too high to be acceptable. A garbage collector that performs heap compaction as part of the collection task eludes this fragmentation issue.

[1]The former implies the latter on uniprocessors, but not on multi-processors.

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 trade-off 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 article, we propose a memory unit for nonblocking 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.

2. RELATED WORK

Real-time GC research dates back to the 1970s where collectors for LISP and ML have been developed. Therefore, a vast number of papers on real-time GC have been published. A good introduction to GC techniques can be found in Wilson's survey [Wilson 1994] and in Jones [1996].

2.1 Root Scanning

The idea of delegating local root scans to the mutator threads was proposed by Doligez and Leroy [1993] and Doligez and Gonthier [1994]. They point out that this allows for more efficient code and reduces the disruptiveness of GC. Mutator threads should check an appropriate flag from time to time and then scan their local root set. However, the authors remain vague on when the mutators should check this flag and do not investigate the effect of various choices. As they aim for efficiency rather than real-time properties, they do not consider a thread model with known periods and deadlines.

Levanoni and Petrank [2001] coined the term "sliding view" for the independent scanning of local thread states in a reference counting garbage collector. This scheme was later extended to a mark-sweep garbage collector [Azatchi et al. 2003]. Again, these works do not consider the implications on the timing of a real-time system.

The approach presented in this article builds to some degree on an approach by Schoeberl and Vitek [2007]. They propose a thread model which does not support blocking for I/O and where threads cannot retain a local state across periods. They also propose that the garbage collector runs at the lowest priority, which entails that the stacks of all threads are empty when a GC cycle starts. Consequently, the garbage collector only needs to consider the global root set.

Yuasa introduces a *return barrier* in Yuasa [2002]. In a first step, the garbage collector scans the topmost stack frames of all threads atomically. Then, it continues to scan one frame at a time. When a thread returns to a frame that

has not yet been scanned, it scans it by itself. Return instructions consequently carry an overhead for the respective check. Furthermore, the proposed policy makes it difficult to compute tight WCET bounds, because it is difficult to predict when a scan by a thread is necessary. A further critical issue is that the topmost frames of *all* threads have to be scanned in a single atomic step. Therefore, the worst-case blocking time increases with the number of threads. An overhead of 2 to 10 percent due to the necessary checks for the return barrier is reported in Yuasa [2002]. Depending on the configuration, 10 to 50 μs were measured as worst-case blocking time on a 200-MHz Pentium Pro processor for two single-threaded benchmarks.

Cheng et al. [1998] propose a strategy for lowering the overhead and blocking of stack scanning. The mutator thread marks the activated stack frames and the garbage collector scans only those frames that have been activated since the last scan. However, this technique is only useful for the average case. In the worst case, it is still necessary to scan the whole stack atomically.

In the JamaicaVM's garbage collector, the mutator threads are responsible of keeping their root set up to date in "root arrays" [Siebert 2001]. The average overhead for keeping these root arrays up to date is estimated as 11.8%.

2.2 Object Copy

The JamaicaVM takes a simple approach to avoid blocking times due to object copying: it avoids moving objects at all [Siebert 2000]. 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 garbage collector splits arrays, similar to the JamaicaVM approach, into small chunks called Arraylets [Bacon et al. 2003b]. Metronome compacts the heap to avoid fragmentation and the Arraylets reduce blocking time on the copy of large arrays. Both approaches, the JamaicaVM garbage collector and Metronome, have to pay the price of a more complex (and time consuming) array access. The defragmentation algorithm in Metronome evacuates the objects from almost empty pages to nearly full pages [Bacon et al. 2003a]. This minimizes the amount of data to be moved, and the overall effort for defragmentation. However, it still requires to atomically move considerable amounts of data.

Another approach to allow interruption of GC copy is to perform field writes to both copies of the object or array [Huelsbergen and Larus 1993]. This approach slows down write access, but those are less common than read accesses. The writes to the two copies must be performed atomically to ensure the consistency of the data. An additional pointer is also needed between the two copies of the object. We consider the overhead for establishing the atomicity for the two writes too high for this solution to be practical. Nettles and O'Toole [1993] propose a garbage collector where the mutator is allowed to modify the original copy of the objects. All writes are recorded in a mutation log and the garbage collector has to apply the writes from this log after updating the pointer(s) to the new object copy.

The clever usage of atomic two-field compare-and-swap (CAS) operations for an incremental object copy is proposed by Pizlo et al. [2007]. 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. Pizlo et al. [2008] explored two more variants of using CAS for consistent object copying, which rely on a probabilistic understanding of time bounds. Furthermore, it is admitted for the original variant that "In a small probability worst-case race scenario, repeated writes to a field in the expanded object may cause the copier to be postponed indefinitely." As a hard real-time system has to guarantee time bounds also in worst-case scenarios, we do not consider these approaches to be suitable for such systems.

Nilsen and Schmidt [1992] propose hardware support, the object-space manager (OSM), for real-time garbage collector on a standard RISC processor. The concurrent garbage collector is based on Baker [1978], 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. Schmidt and Nilsen [1994] extend the OSM to a GC memory module where a local microprocessor performs the GC work. In the paper the performance of standard C++ dynamic memory management is compared against garbage collected C++. The authors conclude that C++ with the hardware supported garbage collection performs comparable with traditional C++.

One argument against hardware support for GC might be that standard processors will never include GC specific instructions. However, Azul Systems has included a read barrier in their RISC based chip-multiprocessor system [Click et al. 2005]. The read barrier looks like a standard load instruction, but tests the TLB if a page is a GC-protected page. GC-protected pages contain objects that are already moved. The read barrier instruction is executed after a reference load. If the reference points into a GC-protected page a user-mode trap handler corrects the stale reference to the forwarded reference.

Meyer [2006] presents a hardware implementation of Baker's read-barrier [Baker 1978] in an object-based RISC processor. 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 [Brooks 1984] indirection instead.

The solution proposed by Meyer for object-oriented systems also contains a GC 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 WCET of hard real-time threads.

The Java processor SHAP [Zabel et al. 2007], with a pipeline and cache architecture based on the architecture of JOP, contains a memory management unit with a hardware garbage collector. That unit redirects field and array access during a copy operation of the GC unit.

The three hardware-assisted GC proposals [Nilsen and Schmidt 1992; Meyer 2006; Zabel et al. 2007] 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 (nonblocking) object copy result in 100% utilization for high priority real-time tasks.

3. PREEMPTIBLE ROOT SCANNING

Due to the volatile nature of a thread's stack, the garbage collector and the mutator thread must cooperate for proper scanning. If a thread executes arbitrary code while its stack is scanned, the consistency of the retrieved data cannot be guaranteed. Therefore, a thread is usually suspended during a stack scan. In order to ensure the consistency of the root set, the stack is scanned atomically to avoid preemption of the garbage collector and inhibit the execution of the respective thread.

When the GC thread scans a stack it is not allowed to be preempted by that thread. The runtime stacks of any two threads are however disjoint – otherwise they could not execute independently of one another. Therefore, preemption by any other mutator thread is not an issue. When inhibiting the preemption of the garbage collector only for the thread whose stack is scanned, a thread will only suffer blocking time due to the scanning of its own stack. A high priority thread, which has probably a shallow call tree, will not suffer from the scanning of deeper stacks of more complex tasks. The protection of the scan phase can be achieved by integrating parts of the GC logic with the scheduler. During stack scanning only the corresponding mutator thread is blocked.

We generalize this idea by moving the stack scanning task to the mutator threads. Each thread scans its own stack at the end of its period. In that case mutual exclusion is trivially enforced: the thread performs either mutator work or stack scanning. The garbage collector initializes a GC period as usual. It then sets a flag to signal the threads that they shall scan their stacks at the end of their period. When all threads have acknowledged the stack scan, the garbage collector can scan the static variables and proceed with tracing the object graph. Why the static variables are scanned after the local variables is discussed in Section 3.1.3.

By using such a scheme, it is not necessary to enforce the atomicity of a stack scan. Furthermore, the overhead for a stack scan is low; at the end of each period, the stack is typically small if not even empty. Such a scheme also simplifies exact stack scanning, because stack scanning takes place only at a few predefined instants. Instead of determining the stack layout for every point at which a thread might be preempted, it is only necessary to compute the layout for the end of a period. The required amount of data is reduced considerably as well, which lowers the memory overhead for exact stack scanning.

3.1 Consequences

In Puffitsch and Schoeberl [2008], we proved that delegating the scanning of the thread-local root sets to the mutator threads does not void the correctness of our garbage collector. It has to be assured that no reference can remain undetected by the garbage collector. Such a situation could happen, if a reference migrates from a not-yet-scanned local variable to a local variable that already has been scanned. We could prove that in such a situation, the proposed GC algorithm can compute the root set correctly. Threads can exchange data only through static variables and object fields. An appropriate write barrier can therefore make migrating references visible to the garbage collector. The proof revealed some other interesting issues, which we address in the following.

3.1.1 *Write Barrier*. Formal reasoning showed that a Yuasa-style snapshot-at-beginning barrier is sufficient to ensure the correctness of the GC, if new objects are allocated gray in terms of Dijkstra's tri-color abstraction [Dijkstra et al. 1978]. The idea behind this is that a snapshot-at-beginning barrier allows to approximate the history of the object graph if no object is black. On the one hand, overwritten references are marked gray, that is, they are visible to the garbage collector. On the other hand, the garbage collector follows the most recent state of the object graph during tracing. Therefore, the whole history of the object graph is visible to the garbage collector. If an object is black, it is not considered by the garbage collector for tracing, and its actual state would remain invisible to the garbage collector.

The usual solution to keep the view of the heap consistent for such garbage collectors is a *double barrier* [Auerbach et al. 2007]. It requires that the write barrier pushes both the old and the new value onto the mark stack during root scanning. With respect to predictability, a snapshot-at-beginning barrier is superior to a double barrier, because only zero or one references may be pushed onto the mark stack. For a double barrier, zero, one or two references may be pushed. Obviously, the latter has a higher variability in its execution time.

We are aware of the fact that allocating new objects gray is against "common knowledge", especially for a copying garbage collector. However, in the case of our GC algorithm (it is described in detail in Section 5.1), the impact of this can be kept considerably lower than for other garbage collectors. The notion of gray objects mainly refers to their status with respect to tracing the object graph. With our garbage collector, it is possible to allocate a new object in tospace and to also push it onto the mark-stack (one may think of these

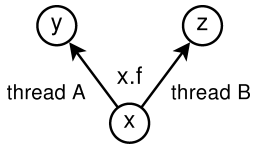


Fig. 1. Threads may have an inconsistent view of the object graph.

objects as being "anthracite"). The copying step is skipped for such objects, while tracing takes place as normal. This leaves us with a trade-off between a double barrier and some additional tracing effort for the garbage collector. The temporal variability of the allocation is slightly increased, because new objects are pushed onto the mark stack only during root scanning. Otherwise, we would not be able to ensure that tracing ever finishes. However, the increase of the temporal variability is small, compared to the overall costs of allocation.

It has to be noted that reading the old value and writing the new value in the write barrier has to be atomic, which is the case in our implementation. When guaranteeing this atomicity is too expensive, the double barrier is an alternative solution.[2]

3.1.2 *Memory Model.* Figure 1 shows a situation, where two threads, A and B, have an inconsistent view of the object graph. While for thread A the field x.f references object y, the same field references object z for thread B. Such a situation is acceptable in the Java memory model [Gosling et al. 2005], but poses problems for a garbage collector, because it would leave either object y or object z unvisited. Proper synchronization of course eliminates such coherence issues, but the authors consider correctness of synchronization to be an unreasonably strong precondition for GC. Flawed synchronization should not cause a failure of the garbage collector.

Inconsistent views of the object graph originate from the fact that threads are allowed to cache data locally. On uniprocessors, cache coherence is not an issue – all threads share the same cache – but thread-local registers may be used to store reference fields. As these registers are scanned for the computation of the local root set of a thread, it is ensured that references cached in registers are visited as well as references stored in the heap. It is therefore safe to assume that all threads have a consistent view of the object graph.

On multiprocessors, cache coherence must be ensured to allow consistent tracing of the object graph. It is beyond the scope of this article how the required degree of cache coherence can be achieved efficiently.

3.1.3 *Static Variables.* For our proof, we modeled static variables with an immutable root, which points to a virtual array that contains the static variables. This virtual array can then be handled like any other object and the scanning of static variables becomes part of the marking phase. As marking has to take place after root scanning, static variables have to be scanned after the local root sets.

[2]We thank Bertrand Delsart who pointed out this detail during the presentation at the JTRES 2008.

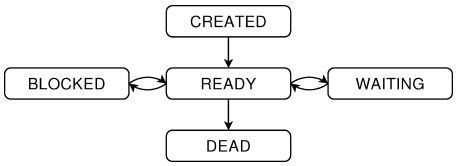


Fig. 2. Thread model.

There is also a more pragmatic reason for this – it is easy to construct an example where scanning static references before scanning local variables breaks the consistency of a garbage collector that uses a snapshot-at-beginning write barrier. Consider the case where during the scanning of static variables a reference is transferred from a local variable to a static variable that already has been scanned. The value of the local variable might be lost until it is scanned, and the new value of the static variable is not visible to the garbage collector. The variable already has been scanned, and the snapshot-at-beginning barrier retains the old value, but does not treat the new one. Therefore, the respective object may erroneously appear unreachable to the garbage collector. The consequence of this is that static variable have to be scanned after the local root sets have been scanned.

3.2 Execution Time Bounds

Functional correctness is not the only concern for real-time systems: the effects on the timing behavior of the GC thread also have to be analyzed. We found two solutions to apply the theoretical results: The first solution, which is described in Section 3.2.2, can be applied only to periodic tasks. The second solution can be applied to sporadic tasks as well; it is described in Section 3.2.3. The two solutions also provide a trade-off in terms of timing and memory overheads.

3.2.1 *Thread Model.* We assume that all threads are either periodic or have at least a known deadline. This is a reasonable assumption for real-time threads: it is impossible to decide whether a task delivers its result on time if no deadline or period is known.[3]

The thread model has five states: CREATED, READY, WAITING, BLOCKED and DEAD. Initially, a thread is in state CREATED. When a thread gets available for execution, it goes to the READY state. When it has finished execution for a period it becomes WAITING. At the start of the next period, it goes to the READY state again. If a thread terminates, it becomes DEAD. Threads are in state BLOCKED while they wait for locks or I/O operations. The time between the instant at which a thread becomes READY until it goes to state WAITING must be bounded – if it is not WAITING when its deadline arrives, it has missed the deadline. Figure 2 visualizes the possible state transitions of the thread model.

For the calculation of the execution time bounds, we assume that threads scan their stack when they become WAITING. For periodic tasks in the RTSJ

[3]For threads without a known deadline, the garbage collector can fall back to blocking the thread and scanning the stack itself.

[Bollella et al. 2000], this can be done implicitly when waitForNextPeriod() is invoked. There is no need to change the application code. If no such method needs to be called by tasks, the scanning can be integrated into the scheduler. In the current version of the RTSJ, sporadic threads do not invoke such a method; their stack is however empty when they do not execute, which in turn makes root scanning trivial. The overhead for stack scanning of course has to be taken into account for calculating the WCET of tasks.

We assume that the GC thread runs at the lowest priority in the system. On the one hand, a garbage collector usually has a long period (and deadline), compared to other real-time tasks. It follows from scheduling theory that it should have a low priority [Liu and Layland 1973].

3.2.2 *Solution for Periodic Tasks.* For periodic threads, the time between two releases is known and the time between two successive calls of waitForNextPeriod() is bounded. For this solution, the individual tasks push the references of the local root set onto the mark stack of the garbage collector if an appropriate flag is set. The garbage collector must wait until all tasks have acknowledged the scan before it can proceed. In the worst case, a task has become WAITING very early in its period when the garbage collector starts execution and becomes WAITING very late in its next period.

Let R_i be the worst-case response time of a thread τ_i, Q_i its best case response time and T_i its period. The response time of a task is the time between the instant at which a thread becomes READY until it goes to the WAITING state again. $C_{stackscan}$ is the worst-case time until all threads have scanned their local root set and the garbage collector may proceed. $C_{stackscan}$ can be computed as follows:

$$C_{stackscan} = \max_{i \geq 0}(T_i - Q_i + R_i) \tag{1}$$

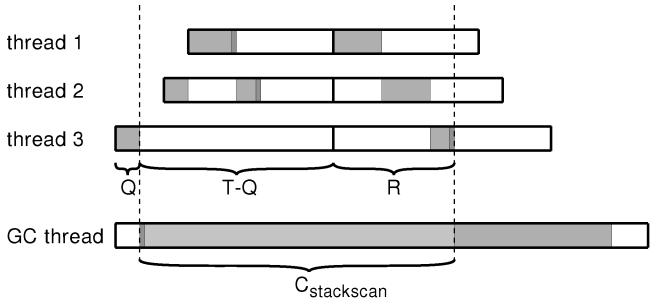
Figure 3(a) visualizes the formula above. It shows that the worst case between two completions of a thread is $T-Q+R$. Consequently, this is the longest time the garbage collector must wait for this thread.

To avoid the computation of the best and worst-case response times – especially the former is typically unknown –, this can be simplified to

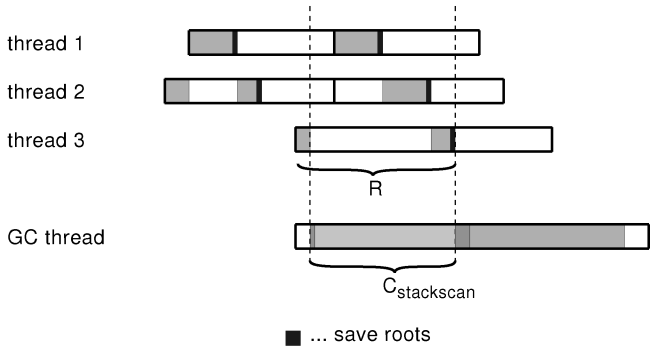
$$C_{stackscan} = 2T_{max} \tag{2}$$

3.2.3 *Generalized Solution.* The considerations for periodic tasks cannot be applied to sporadic tasks in the general case. For sporadic tasks, the minimum interarrival time is known, but usually not the maximum inter-arrival time. Therefore, the worst-case time until the garbage collector may proceed is potentially unbounded. A similar issue occurs with threads that have a very long period; for such threads, $C_{stackscan}$ for the simple solution may become prohibitively large.

The stack of a thread is only modified if the respective thread executes. Therefore, the garbage collector can reuse data from previous scans and only needs to wait for threads which may have executed since their last scan. These are – apart from the initialization and destruction of threads – the threads which are not in state WAITING.



(a) Solution for periodic tasks



(b) Generalized solution

Fig. 3. Visualization of the WCETs for root scanning.

We adapt the root scanning scheme such that threads save their stack on every call of waitForNextPeriod() to a *root array*. For WAITING threads, the content of the root array from their last scan is used by the garbage collector; for all other threads, the garbage collector waits until they have updated their root array. With this scheme, it is sufficient to take into account the worst-case response time for the execution time of stack scanning.

$$C_{stackscan} = R_{max} \tag{3}$$

This time can be further improved: the garbage collector can only execute if no other thread is READY. If threads scan their stacks when becoming BLOCKED, the garbage collector can therefore never encounter threads that have executed since their last stack scan. Trivially, this is also the case, if a system does not support blocking operations at all. As the information in the root arrays is always consistent when the garbage collector executes, it is never necessary to wait for any thread to scan its stack and

$$C_{stackscan} = 0 \tag{4}$$

Enforcing scanning upon blocking requires more effort than enforcing it upon waiting. It entails that the implementation of wait() needs to be changed

accordingly. Depending on the organization of a JVM, this may or may not be possible.

3.2.4 *Discussion.* As pointed out by Robertz and Henriksson [2003] and Schoeberl [2006a], the allocation rate and the size of the heap determine the maximum GC period. The proposed solutions introduce a waiting time for the garbage collector and therefore may make it necessary to increase its period. Such an increase may lead to a situation where it cannot be guaranteed anymore that the garbage collector can cope with the allocation rate of a system.

$C_{stackscan}$ has to be added to the response time of the GC thread; the impact of this delay depends on the thread periods. If the response time of the garbage collector is far greater than the periods of the mutator threads, the relative impact is small. If there is some slack between the maximum and the actual GC period, the effect can probably be hidden.

An advantage of the generalized solution is that $C_{stackscan}$ is considerably smaller than for the simple solution. The downside of the generalized solution is however that a dedicated memory area is needed to save the roots of the individual threads. It is not possible anymore to let the mutator thread push its root set onto the mark stack. To avoid blocking in this scheme, it is necessary to use two memory areas for each thread to allow for double buffering. If the maximum number of roots is unknown, each of these areas occupies as much memory as the stack.

The overhead for completing the root scanning is larger on the garbage collectors side for the generalized solution. This is due to the fact that the garbage collector itself has to push the references onto the mark stack. On the threads' side, the overhead is slightly smaller, because the content of the stack only has to be transferred to the root array, without performing any computations. However, the increased overhead for scanning is most likely far smaller than the time that is spent on waiting for the other threads. It is mandatory to use such a scheme for sporadic tasks; applying it to periodic tasks as well allows to trade time to wait for a root scan with additional memory consumption.

Figure 3 compares the worst case scenarios of the solution for periodic tasks and the generalized solution. For the generalized solution, the threads save their stack in a root array at the end of each period. The garbage collector only has to wait for threads that have executed since their last scan. In Figure 3(b), thread 3 is BLOCKED when the garbage collector starts execution, but does not scan its stack. Therefore, the garbage collector has to wait for this thread. In the worst case, this waiting time equals the maximum response time.

It is possible to mix root scanning strategies to find an optimal solution. For high frequency threads, jitter is usually very important, and the waiting time of the solution for periodic threads may be negligible. For medium frequency threads, the generalized solution with an impact in the order of one period may be a better trade-off. Low-frequency threads are probably less sensitive to jitter and root scanning by the garbage collector may not hurt them. However, it is not possible to propose a generic solution to this problem without knowledge about the properties of the whole system.

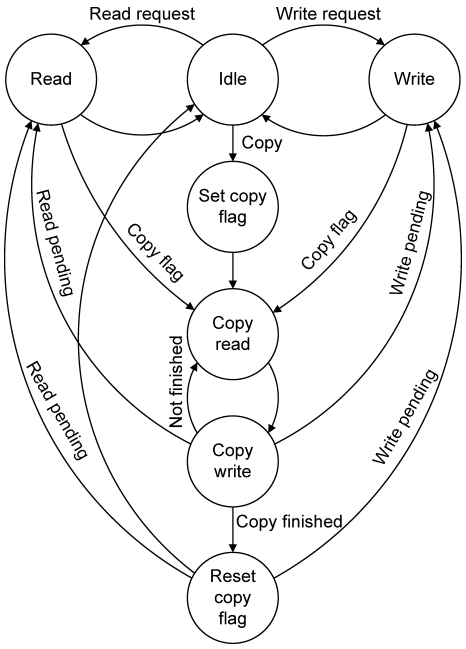


Fig. 4. Memory controller state machine with background copy.

4. NONBLOCKING OBJECT COPY

Copying large arrays and objects in a compacting garbage collector 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 garbage collector, 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 garbage collector 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, the hardware also needs to be interrupted on a thread switch. Furthermore, the copy task needs to be resumed at the correct time, that is, 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. The copy unit starts at idle cycles, but 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.

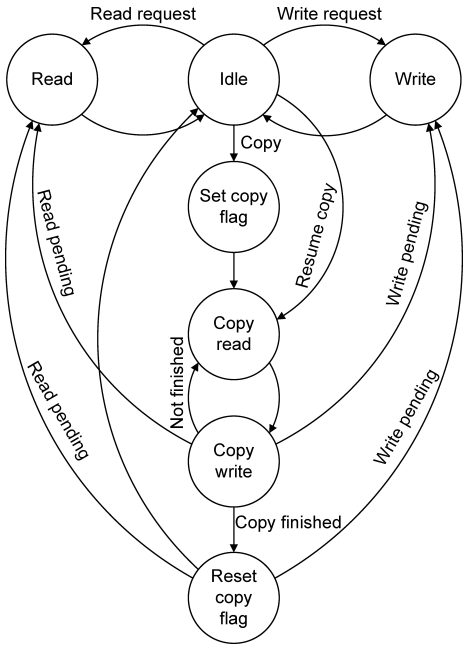


Fig. 5. Memory controller state machine with interruptible copy.

Figure 4 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 5. As in the former state machine the copy loop can be

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

Listing 1. Busy waiting copy loop in the GC thread with a copy resume.

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 resumed from the processor, as indicated by the transition from *idle* to *copy read*.

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

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 thread from the copy unit in the copy loop.

Listing 1 shows the copy code in the garbage collector. 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 resumed if necessary. On a resume, the copy unit just continues to copy the object; it is not a restart of the copy task, as in Gruian and Salcic [2005], that can result in starvation of the GC copy. 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 garbage collector busy wait. At the end of the copying process, the reference to the object in the handle is updated atomically. The copy unit still redirects the access to the correct location to avoid any race condition. The redirection is updated at the next object copy (with startCopy()).

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 resume 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.

5. IMPLEMENTATION

We implemented the proposed nonblocking copy unit in the Java processor JOP [Schoeberl 2008]. JOP was designed from scratch as a real-time processor [Schoeberl 2006b] 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 [Schoeberl and Pedersen 2006].

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.

The described GC algorithm is intended for hard real-time systems where allocation rate and object lifetime can be analyzed. As a fallback, when the analysis was wrong, an allocation will be blocked till the GC has freed enough memory. The real-time GCs, which are part of practically all available RTSJ implementations, are usually optimized for soft or mixed real-time applications. These GCs support applications that are not analyzable by (self-)tuning of GC parameters.

5.1 The GC Algorithm

The collector for JOP is a concurrent copy collector [Schoeberl 2006a; Schoeberl and Vitek 2007] based on the garbage collectors of Baker [1978] and Dijkstra et al. [1978]. 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 [1975] and Dijkstra et al. [1978]. The collector and the mutator are synchronized by a read and a write barrier. A Brooks-style [Brooks 1984] forwarding directs the access to the object either into tospace or fromspace. Indirection through the forwarding pointer is implemented in hardware and is therefore an atomic operation. On a standard uniprocessor preemption points are common practice for short critical section to synchronized mutator and GC threads. The forwarding pointer is kept in a separate handle area, as proposed by North and Reppy [1987]. The separate handle area reduces the space overheads, because 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. It has to be noted that the size of the handle area needs to be chosen according to the application characteristics.

The second synchronization barrier is a *snapshot-at-beginning* write barrier as proposed by Yuasa [1990]. 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 Listing 2. An object is shaded gray by pushing the reference of the object onto the mark stack.[4] Further scanning and copying into tospace – coloring it

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

```
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);
    }
}
```

Listing 2. Snapshot-at-beginning write barrier in JOP's JVM.

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 appropriate 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 JVM internal bytecodes (e.g., putfield_ref) to execute the write barrier code as shown in Listing 2.

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 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 Java method the hardware implementation of that bytecode (Native.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 garbage collector, 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 garbage collector runs at the lowest priority. When a high priority task becomes ready, the GC thread will

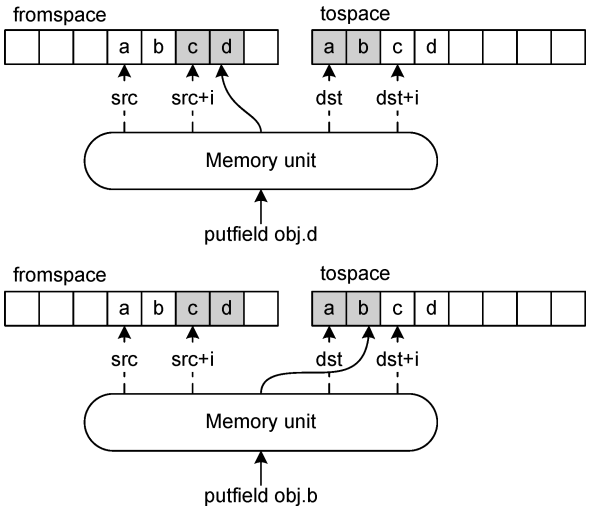


Fig. 6. Redirection of a putfield operation by the memory unit.

be preempted. Atomic operations of the garbage collector are protected simply by turning the timer interrupt off.[5] 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.

5.2 Root Scanning

The implementation of the new root scanning strategies is straight forward. The logic for stack scanning is inserted into the implementation of waitForNextPeriod(). The garbage collector is modified such that it waits for the application threads to scan their stacks instead of doing it itself. A third change is the gray allocation of new objects during the root scanning phase, which is not necessary for atomic stack scanning. In total, less than 100 lines of Java code are specific to the root scanning strategies proposed in Section 3.2.

5.3 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 overhead of the read-barrier (the handle indirection) and speeds up null pointer and bounds checks [Schoeberl 2007]. 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 6 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,

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

fields c and d are in the original object in fromspace. The upper figure shows the access 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.

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 resuming it after preemption, the copy task is continually triggered for individual words in the garbage collector 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 our 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 to read the word and 3 cycles to write the word) 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,[6] 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 [Schoeberl 2007]. Therefore, the pointer of the access just needs to be compared with the 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.

[6]Interrupts are only accepted at bytecode boundaries.

Table I. Thread Properties of the Test Programs

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	6
τ_p	2 ms	2 ms	5
τ_c	10 ms	10 ms	4
τ_s	15 ms	15 ms	3
τ_{log}	25 ms	25 ms	2
τ_{gc}	50 ms	50 ms	1

Table II. Release Jitter with Blocking Copy, Task Set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$, Varying Array Sizes

	Jitter (μs)			
Array Size	base	single	scan	save
256 B	536	136	82	91
512 B	533	130	82	91
1 KB	537	135	84	90
2 KB	535	142	128	134
4 KB	531	244	241	237
8 KB	537	447	445	455
16 KB	856	857	856	866
32 KB	1677	1671	1677	1685
64 KB	3313	3316	3311	3323

The hardware resource consumption of the copy unit is moderate. The additional registers, adders, and multiplexors in the memory unit consume 322 additional logic cells (LC). This is about 10% of the complete processor. However, it doubled the size of the memory unit from 301LC to 623LC. The memory unit is now almost as large as the execution unit (679LC).

6. EVALUATION

For the evaluation we used following hardware setup: JOP implemented in an FPGA and configured for 100MHz.[7] JOP is configured with 4KB instruction cache and 1KB stack cache. The main memory consists of 1MB static RAM with 15ns access time, resulting in a single word read access in two clock cycles and a single word write access in three clock cycles.

For jitter measurements, we used 6 different tasks, which are similar to the tasks presented in Puffitsch and Schoeberl [2008] and Schoeberl and Puffitsch [2008]. We chose to unify these two slightly different experiments, so we can present consistent figures throughout all aspects of our evaluation. Rate monotonic priority ordering is used to determine the tasks' priorities. The task properties are described in the following and subsumed in Table I. The figures presented in Tables II, III, IV, and V were obtained by measuring the maximum release jitter of the highest priority thread during a run of 15 minutes. For the measurements, we slightly modified the periods of the threads. We used prime numbers (e.g., 2003μs instead of 2000μs) to avoid a regular phasing of the threads, which could have led to too optimistic results.

[7]The actual synthesis results with medium effort on optimization for the low-cost Altera Cyclone-I FPGA is 97MHz.

Table III. Release Jitter with Copy Unit, Task Set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$, Varying Array Sizes

	Jitter (μs)			
Array Size	base	single	scan	save
256 B	532	132	73	86
512 B	532	125	73	75
1 KB	528	126	73	75
2 KB	527	125	73	74
4 KB	527	131	73	86
8 KB	526	131	73	86
16 KB	526	126	75	86
32 KB	526	124	72	86
64 KB	525	122	71	74

Table IV. Release Jitter with Blocking Copy, Varying Task Sets, Array Size 4 KB

	Threads						Jitter (μs)			
Task Set	τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	no GC	base	single	scan	save
A	✓			✓	✓	76	517	199	78	86
B	✓	✓	✓		✓	70	531	244	241	237
C	✓	✓	✓	✓	✓	84	683	242	230	247

Table V. Release Jitter with Copy Unit, Varying Task Sets, Array Size 4 KB

	Threads						Jitter (μs)			
Task Set	τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	no GC	base	single	scan	save
A	✓			✓	✓	70	531	196	80	74
B	✓	✓	✓		✓	69	527	131	73	86
C	✓	✓	✓	✓	✓	75	683	198	82	92

The most important thread with respect to the measurements is the high-frequency task τ_{hf} with a frequency of 10kHz. It computes its own release jitter and does nothing else. This task has the highest priority of all tasks and all jitter figures in this section refer to the release jitter of this thread.

Two more threads, τ_p and τ_c, act as producer/consumer pair exchanging arrays. τ_p produces one array every two milliseconds and τ_c consumes the available arrays every 10ms. A simple list is used to pass the objects from τ_p to τ_c. These threads have the second- and third-highest priorities in the system. $\tau_{p'}$ is a variant of τ_p, which uses a preallocated pool of objects instead of dynamic allocation. It is used to evaluate the behavior of the system if no GC takes place or if GC could not cope with the allocation rates.

τ_s is a thread that occupies the stack such that it is not empty when waitForNextPeriod() is invoked. Consequently, a strategy as proposed by Schoeberl and Vitek [2007] cannot be used if this thread is part of the task set, because the strategy assumes that the threads' stacks are empty at the time of root scanning. It is also the thread with the deepest stack. The period of τ_s is 15ms.

To record the measurements, we used a logging thread τ_{log} with a period of 25ms. It prints results every 40^{th} iteration, that is, once per second. The artificially short period is necessary to keep the waiting time for the *scan* strategy sufficiently low. The GC thread, τ_{gc}, has a period of 50 ms and is

consequently the lowest-priority thread in the system. The GC period was chosen shorter than necessary to force the GC thread to run practically as a background thread. This setting maximizes the interference between the GC thread and the mutator threads.

The careful reader will note that the release jitter in Tables II, III, IV, and V exceeds the period of τ_{hf} for some measurements. In the scheduler we used for this experiment, we chose not to adjust the following release times in these cases. On the one hand, this may lead to queuing up of releases, heavily overloading the system. On the other hand, this allows the threads to "catch up" in the following releases and avoids the release times to drift off. At least for our experiment, the latter behavior is more useful, because a single deadline miss then does not affect the measurement for all following releases.

We evaluated four different root scanning strategies. The strategy labeled "base" in Tables II, III, IV, and V scans the stacks of all threads in one atomic step. The "single" strategy scans one stack at a time atomically. The "scan" and "save" strategies implement the solution for periodic tasks and the generalized solution as described in Section 3.2. For the "scan" strategy, tasks push their local root set onto the mark stack at the end of their period. For the "save" strategy, tasks save their stack into root arrays, and the garbage collector pushes the references onto the mark stack.

Tables II and III show results for a fixed task set and various arrays sizes. The size of the arrays that τ_p and τ_c exchange is shown in the first column. The task set for arrays of up to 4 KB is $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$. For larger arrays, it was necessary to use the task set $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_{log}, \tau_{gc}\}$, i.e., preallocated arrays are used. The garbage collector could not keep up with the allocation rates with dynamic allocation. Still, it tries to garbage collect the preallocated arrays and therefore has an effect on the system behavior. While Table II presents the numbers for a system that copies objects atomically, the results in Table III were obtained on a system with a copy unit.

The "base" strategy yields a release jitter of around 535μs, for arrays of up to 8KB. The jitter due to the root scanning is large enough to hide the jitter that is caused by the atomic copying of the arrays up to this size. The *single* strategy lowers the release jitter to around 135μs for small arrays. In Table II, the jitter increases linearly with the array size, starting at 2KB, to up to 3316μs. The copy unit removes this effect, such that the *single* strategy can achieve a release jitter of around 130μs for all array sizes. The results for the *scan* and *save* strategies are similar: both lower the jitter to around 85μs for small arrays. Again, the jitter for larger arrays increases without the copy unit. Table III shows that with the copy unit low jitter can be achieved also for large arrays. Tables II and III show that the copy unit and the new root scanning strategies allow to achieve low release jitter for high frequency threads.

One question to be answered as well is in how far GC affects the release jitter when comparing it to a system without GC. In Tables IV and V, various task sets are compared. The first column displays the labels for the task sets, the following six columns indicate which tasks are part of a specific task set. For the measurements in the column labeled "no GC", τ_p is replaced with $\tau_{p'}$, which uses a pool of preallocated objects instead of dynamic allocation, but

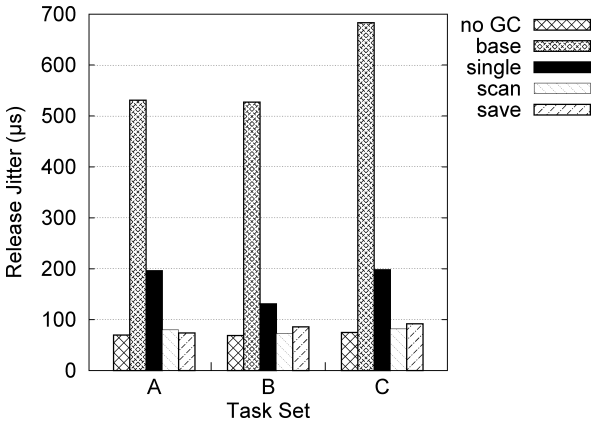


Fig. 7. Release jitter with copy unit, varying task sets, array size 4 KB.

otherwise is equivalent to τ_p. For these measurements, τ_{gc} is not part of the task set, while it is part of the task set for all other measurements. Therefore, the figures in the "no GC" column indicate how a system without GC would behave. The size of the arrays that are passed between τ_p and τ_c is 4 KB for all measurements in Tables IV and V. Figure 7 shows a graphic representation of the numbers in Table V; the evaluated task sets and strategies are the same.

As in Tables II and III, the *base* strategy introduces a considerable amount of jitter. Up to 683μs of release jitter could be observed for task set C, which is almost one order of magnitude larger than the jitter without GC. Scanning the stacks of individual threads atomically results in a lower jitter, but the atomic array copying results in up to 244μs in Table IV. This jitter is removed in Table V, resulting in up to 198μs of jitter for the "single" strategy. Comparing the task sets with and without τ_s confirms that the effect depends on the size of the largest thread stack.

When comparing the system without dynamic memory allocation to the "scan" and "save" strategies, the jitter is increased by less than 20μs. Up to 75μs of jitter can be observed without GC, and must therefore be attributed to scheduling and synchronization in the application logic. Future work will have to concentrate on this area to allow the scheduling of real-time threads with periods of less than 100μs.

The new strategies "scan" and "save" slightly degrade the scheduling quality. Up to 82μs of jitter could be observed for the "scan" strategy, up to 92μs for the "save" strategy. As we made only minimal changes to the scheduler, and no significant atomic sections were introduced, we had expected only a smaller deviation in these results. However, further research will be necessary to find the source of the jitter increase.

6.1 Discussion

The results presented in the previous section demonstrate that traditional root scanning techniques are inadequate when considering high frequency real-time threads. Scanning all thread stacks in a single atomic step easily increases the

release jitter by one order of magnitude, scanning one thread stack at a time atomically almost triples the jitter. Of course, the effects depend on the actual application, but they tend to become worse with larger applications.

The copying of large arrays is also a key factor in achieving a high scheduling quality. The impact of atomic copying grows linearly with the array sizes; without appropriate measures to reduce this effect, it can easily introduce an unacceptable amount of jitter.

One important result derived from the Tables IV and V is that it is necessary to use both a nonblocking root scanning strategy and a nonblocking copy mechanism to achieve low jitter. Improved root scanning is futile if the copying of large arrays introduces considerable jitter. Lowering the preemption latency of array copying to the granularity of single words is rendered useless if whole thread stacks are scanned in one atomic step.

The results presented in Figure 7 demonstrate that the the GC techniques proposed in this article can lower the jitter to almost the same level as in a system without GC. For the evaluated system, scheduling is the largest source of jitter. For high frequency tasks, it is therefore necessary to improve the scheduler; the garbage collector is not the limiting factor anymore.

Of course, GC does not come for free. It introduces memory and performance overheads and may therefore make it necessary to use more expensive hardware for a given system. On the other hand, dynamic memory management increases programmer productivity and program safety. The low intrusiveness of the proposed GC mechanisms allows deciding on this trade-off without sacrificing scheduling quality.

7. CONCLUSION AND OUTLOOK

We investigated the root scanning phase of GC and could show three important properties. First, atomicity for stack scanning is only necessary with respect to the thread whose stack is scanned. Second, atomicity is not required at all if mutator threads scan their own stack. And third, a snapshot-at-beginning write barrier is sufficient to allow complete decoupling of local stack scans.

Furthermore, we provided two approaches for how these theoretical properties can be utilized and showed the implications on the execution time of a garbage collector. The first approach can only be applied to periodic tasks and delays the garbage collector by up to two times the longest task period. The second approach is more general and has a smaller impact on the execution time of the garbage collector, but has a higher memory overhead.

In this article, we also proposed and evaluated a hardware extension to eliminate the blocking time due to atomic copying of large arrays. A copy unit performs the object and array copy and redirects field and array access to the correct 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.

An evaluation of the proposed solutions confirmed the theoretical results. Jitter of high priority threads, which can be attributed to GC, could be reduced considerably. The impact of the new root scanning strategies on the jitter due to scheduling and synchronization however still needs to be analyzed.

Future work will investigate if a tighter coupling of scheduling and root scanning is profitable. Merging the root arrays of the generalized solution with the memory areas for the thread contexts could lower the memory consumption without impairing the performance.

Exact stack scanning has not been handled in this paper. The proposed solutions lower the overhead for exact scanning, but tools to make use of this need to be developed. Furthermore, for hard real-time systems the execution time of the GC task needs to be bounded. We consider WCET analysis of the GC as future work.

The current implementation of our concepts is based on a uniprocessor. We plan to implement them also in the chip-multiprocessor (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 [Pitter 2008], 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. With regard to our root scanning approach, we are confident that the theoretical basis is applicable to CMP systems. Actual implementations may however offer new obstacles as well as new opportunities, especially in the area of cache consistency.

ACKNOWLEDGMENTS

We thank the reviewers for the detailed comments, which have helped to clarify the description of the presented ideas.

REFERENCES

AUERBACH, J., BACON, D. F., BLAINEY, B., CHENG, P., DAWSON, M., FULTON, M., GROVE, D., HART, D., AND STOODLEY, M. 2007. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*. ACM, New York, 249–258.

AZATCHI, H., LEVANONI, Y., PAZ, H., AND PETRANK, E. 2003. An on-the-fly mark and sweep garbage collector based on sliding view. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'03)*. ACM, New York.

BACON, D. F., CHENG, P., AND RAJAN, V. 2003a. Controlling fragmentation and space consumption in the Metronome, A real-time garbage collector for Java. In *Proceedings of the ACM SIGPLAN Conference on Languages, Computers, and Tools for Embedded Systems (LCTES'03)*.

BACON, D. F., CHENG, P., AND RAJAN, V. 2003b. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages*. New York.

BAKER, H. G. 1978. List processing in real-time on a serial computer. *Comm. ACM 21,* 4, 280–94.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, Reading, MA.

BROOKS, R. A. 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the ACM Symposium on Lisp and Functional Programming*, ACM, New York, 256–262.

CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proceedings of the SIGPLAN'8 Conference on Programming Languages Design and Implementation*. ACM, New York.

CLICK, C., TENE, G., AND WOLF, M. 2005. The pauseless GC algorithm. In *Proceedings of the 1st International Conference on Virtual Execution Environments, (VEE'05)*, ACM, New York, 46–56.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM 21,* 11, 965–975.

DOLIGEZ, D. AND GONTHIER, G. 1994. Portable, unobtrusive garbage collection for multiprocessor systems. In the *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

DOLIGEZ, D. AND LEROY, X. 1993. A concurrent generational garbage collector for a multi-threaded implementation of ML. In the *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 113–123.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java Language Specification, 3rd Ed.* The Java Series. Addison-Wesley, Reading MA.

GRUIAN, F. AND SALCIC, Z. 2005. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, (ACSAC'03)*. Springer-Verlag, Berlin, 281–294.

HUELSBERGEN, L. AND LARUS, J. R. 1993. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceeding of the 4th Annual ACM Symposium on Principles and Practice of Parallel Programming*. ACM, New York, 73–82.

JONES, R. E. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester. (With a chapter on Distributed Garbage Collection by R. Lins).

LEVANONI, Y. AND PETRANK, E. 2001. An on-the-fly reference counting garbage collector for Java. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Application (OOPSLA'01)*, ACM, New York.

LIU, C. L. AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20,* 1, 46–61.

MEYER, M. 2006. A true hardware read barrier. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'06)*, ACM, New York, 3–16.

NETTLES, S. M. AND O'TOOLE, J. W. 1993. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, New York.

NILSEN, K. D. AND SCHMIDT, W. J. 1992. Cost-effective object-space management for hardware-assisted real-time garbage collection. *Lett. Prog. Lang. Syst. 1,* 4, 338–354.

NORTH, S. C. AND REPPY, J. H. 1987. Concurrent garbage collection on stock hardware. In *Conference Record of the Conference on Functional Programming and Computer Architecture*. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 113–133.

PITTER, C. 2008. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. ACM Press, New York, 115–122.

PIZLO, F., FRAMPTON, D., PETRANK, E., AND STEENSGARD, B. 2007. STOPLESS: A real-time garbage collector for multiprocessors. In *Proceedings of the 5th International Symposium on Memory Management (ISMM'07)*, ACM, New York, 159–172.

PIZLO, F., PETRANK, E., AND STEENSGAARD, B. 2008. A study of concurrent real-time garbage collectors. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*. ACM, New York, 33–44.

PUFFITSCH, W. AND SCHOEBERL, M. 2008. 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'08)*.

ROBERTZ, S. G. AND HENRIKSSON, R. 2003. Time-triggered garbage collection—robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SILPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, New York.

SCHMIDT, W. J. AND NILSEN, K. D. 1994. Performance of a hardware-assisted real-time garbage collector. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. ACM, New York, 76–85.

SCHOEBERL, M. 2006a. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE Computer Society Press, Los Alamitos, CA, 424–432.

SCHOEBERL, M. 2006b. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE06)*, 800–805.

SCHOEBERL, M. 2007. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07)*. ACM, New York, 57–62.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *J. Syst. Arch. 54/1–2*, 265–286.

SCHOEBERL, M. AND PEDERSEN, R. 2006. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES06)*. ACM, New York, 202–211.

SCHOEBERL, M. AND PUFFITSCH, W. 2008. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES08)*. ACM, New York.

SCHOEBERL, M. AND VITEK, J. 2007. Garbage collection for safety critical Java. In *Proceedings of the 5th International WOrkshop on Java Technologies for Real-Time Systems (JTRES)*. ACM, New York, 85–93.

SIEBERT, F. 2000. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the Symposium on Compilers, Architectures and Synthesis for Embedded Systems (CASES'00)*.

SIEBERT, F. 2001. Constant-time root scanning for deterministic garbage collection. In *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*.

STEELE, G. L. 1975. Multiprocessing compactifying garbage collection. *Comm. ACM 18,* 9, 495–508.

WILSON, P. R. 1994. Uniprocessor garbage collection techniques. Tech. rep., University of Texas.

WILSON, P. R. AND JOHNSTONE, M. S. 1993. Truly real-time non-copying garbage collection. In *Proceedings of the OOPSLA/ECOOP'93 Workshop on Garbage Collection in Object-Oriented Systems*. ACM, New York.

YUASA, T. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw. 11,* 3, 181–198.

YUASA, T. 2002. Return barrier. In *Proceedings of the International Lisp Conference*.

ZABEL, M., PREUSSER, T. B., REICHEL, P., AND SPALLEK, R. G. 2007. 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'07)*. 59–62.

Received November 2008; accepted March 2009