Decoupled Root Scanning in Multi-Processor Systems

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

ABSTRACT

Garbage collection (GC) is a well known approach to simplify software development. Recently, it has started to gain acceptance also in the real-time community. Several hard real-time GC algorithms have been proposed for uniprocessors. However, the growing popularity of multi-processors for real-time systems entails that algorithms and techniques have to be developed that allow hard real-time GC on multi-processors as well.

We propose a novel root cache, which aggregates information of the processor-local root sets in multi-processor systems. It allows that the root scanning phase of the garbage collector is decoupled from the root scanning phase of working threads. Thread-local root scanning can be scheduled flexibly, without impeding the garbage collector. Additionally, the new cache lowers both the blocking time and the memory bandwidth consumption due to the root scanning phase of GC.

The proposed solution has been implemented for evaluation in a chip multi-processor system based on the Java Optimized Processor. We show how bounds on the garbage collector period can be extended to take into account the root cache. We also present experimental results, which highlight the advantages and limits of the proposed approach.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Memory management (garbage collection)*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Theory, Experimentation

Keywords

multi-processor, garbage collection, real-time

1. INTRODUCTION

GC reliefs the programmer from the task of manually managing memory and is part of a number of programming languages. However, simple garbage collectors cause pauses, which are hardly acceptable for interactive applications. This problem was an incentive for the development of early real-time garbage collectors, which focused on keeping interruptions small rather than achieving temporal predictability. Especially the use of Java in real-time systems has drawn new attention to this area, now also focusing on hard real-time systems, where temporal predictability is of utmost importance.

The authors of the Real Time Specification for Java (RTSJ) [4] were not convinced of the GC techniques available at the time – the specification efforts started in the late 1990s – and introduced a special programming model for memory management to Java, through the use of scoped memory. The programming model with scoped memory is however unusual to most programmers and forces the Java virtual machine (JVM) to check all assignments to references. Failing to adhere to the programming model will trigger run-time exceptions – arguably a different level of safety than most Java programmers would expect. Therefore, hard real-time GC – although not being strictly necessary – is still considered a valuable goal for designers of real-time JVMs.

While there are hard real-time GC algorithms for uniprocessors [20, 15, 2], the known algorithms for multi-processor GC rather focus on soft real-time systems [13, 6], if considering this topic at all.

In this paper we present a novel cache, which is used for caching the references each processor in a chip multi-processor (CMP) system accesses; a garbage collector can use the information in this cache to construct its root set without interrupting the working threads. Therefore, the cache allows the temporal decoupling of the garbage collector and the working threads in a CMP system. It also lowers the number of memory accesses for each processor, removing some pressure on the memory bandwidth, a typical bottleneck in CMP systems.

This paper is organized as follows: the rest of this section highlights the challenges of real-time GC on multi-processor systems. Section 2 describes the proposed cache and its implications; it is evaluated in Section 3. Related work is covered in Section 4, and Section 5 concludes the paper.

1.1 Garbage Collection Algorithms

Traditionally, there are three approaches to GC: reference counting, mark-sweep algorithms and copying algorithms [10].

Reference counting counts the number of references to each object and frees them, when this count drops to zero. While this looks appealing on the first sight, the approach suffers from problems with cyclic references. The solutions to circumvent these problems rely on a certain programming model, require a considerable amount of overhead, or need a mark-sweep phase to reclaim cyclic structures. As we are not convinced that these work-arounds are feasible for real-time systems, reference counting will not be discussed further in this paper.

Mark-sweep and copying algorithms differ in a number of details, but share the same concept: First, a set of *roots* is determined, which contains all objects that are directly accessible. Starting from this root set, the object graph is traversed to find all reachable objects. The objects not discovered in this second phase are obviously garbage and can be recycled. Mark-sweep garbage collectors suffer from heap fragmentation, but can be extended with a compaction phase to circumvent this problem; they are then usually referred to as mark-compact garbage collectors.

In order to lower the blocking time of GC, incremental algorithms have been developed, which allow interleaved execution of working threads and the garbage collector. The working threads have to execute *read barriers* [3] or *write barriers* [24] when accessing memory to ensure the correctness of GC.

Figure 1 shows how the execution of an incremental mark-sweep or copying garbage collector may look like in a multi-processor system. One CPU is responsible for GC, while three other CPUs execute working threads (we assume one thread per CPU for simplicity here). At point A, the garbage collector initiates a new GC cycle. This has to be atomic with regard to interactions of the working threads with the memory management, namely the creation of new objects and the execution of barriers. The garbage collector then (at $B1$) scans its own stack for root references and waits until all other threads have done so as well; static variables are scanned afterwards (at $B2$). While the scanning of thread stacks has to be atomic per thread, the scanning of static variables does not necessarily have to be so. At C, the garbage collector starts to trace the object graph and to copy the used objects or sweep the unused objects, depending on the algorithm. When the GC is finished at D, the GC thread goes idle until the start of the next GC cycle at A'.

When using an incremental garbage collector, the working threads can continue with their work throughout all phases of GC. They only need to be halted while their local root set is scanned.

A crucial point in Figure 1 is that GC may not proceed until all threads have scanned their stacks. One could interrupt all threads and force them to scan their stacks; obviously this will result in a low waiting time, but means that even highest-priority threads must be interruptible. One could also signal them that they should scan their stack and wait until they have done so. From a threads perspective, stack scanning ideally takes place at the end of its period, because its stack is usually low then and virtually no overhead has to be put into checking when to do a stack scan; such a strategy however will result in a long waiting time for the garbage collector. The implications of letting threads scan their local roots sets at the end of their periods are analyzed in [14].

1.2 Real-Time Garbage Collection

Real-time GC has to ensure three fundamental properties: bounded blocking times, acceptable utilization of working threads and bounded memory consumption. While the last requirement does not refer to timing in the first place, a thread that runs out of memory cannot provide its result on time – it will provide no result at all. Blocking times from GC have several reasons:

1. Retrieval of the root set
2. Copying of objects
3. Scanning of objects for graph traversal
4. Access to data structures which are shared between the working threads and the garbage collector

The access to shared data structures such as the mark stack must inevitably be synchronized to ensure these structures are not corrupted. However, the access to them appears only at certain well defined instructions (from the working threads' perspective) and the critical section is typically only a few instructions long. This blocking has to be taken into account for instructions that allocate objects (e. g., new in the JVM) and instructions that may execute a read or write barrier.

The issue of object scanning is a problem only for certain algorithms. Using an appropriate algorithm, the blocking time can be reduced to the access to shared data structures.

Copying of objects is an issue for copying and mark-compact garbage collectors, because copying must be atomic to ensure the consistency of the involved data. Traditional mark-sweep algorithms are not considered feasible for hard real-time systems (where memory consumption must be bounded) due to their problems with fragmentation. A mark-sweep garbage collector has been proposed in [21] to eliminate copying and external fragmentation at the expense of internal fragmentation. In [18], hardware support for non-blocking copying of objects is proposed. Extending this solution to multi-processors however remains future work.

The retrieval of the root set is not considered for the blocking time of a garbage collector in many papers. Incremental root scanning algorithms have been proposed to lower the blocking time for uniprocessor systems [5, 22, 25], but it is not clear if these can be extended to multiprocessors without introducing a considerable overhead to ensure the consistency of the root set. They are also more costly than traditional root scanning schemes; in [22], more than one million references have to be saved each second to "root arrays" and a runtime overhead of 11.8% is estimated. Furthermore, root scanning usually induces memory accesses proportional to the size of the stack, rather than to the overall memory consumption. A computationally expensive thread that rarely accesses memory can therefore cause a considerable amount of traffic, unnecessarily increasing the pressure on the memory bandwidth in a CMP system.

2. ROOT CACHE

The root set for a garbage collector consists of any references that are "always" accessible (any references in static variables in Java) and thread-local references. The latter comprise any references in local variables, the operand stack and CPU registers. In the JVM, all this data is part of the run-time stack. As a running thread constantly modifies this data, it is impossible to get a consistent view of it without cooperation; usually, execution is stopped until all relevant data is saved. Registers of a processor may not be accessible to other processors in a CMP system, so a processor has to scan them for references by itself. The view of the root set also has to be consistent such that no reachable object appears unreachable to the garbage collector; establishing this consistency makes it necessary to use some form of synchronization.

The idea behind the proposed cache is that in Java, a processor cannot "invent" a reference. As there is no pointer arithmetic in Java, references can only point to previously allocated objects. There are only two possibilities how a reference can enter a processor: either it is read from main memory or it is created by the memory management. It is evident that in the first case the memory access is observable from outside the processor. In the latter case, the interaction between a working thread and the memory management ensures that the reference cannot remain undetected.

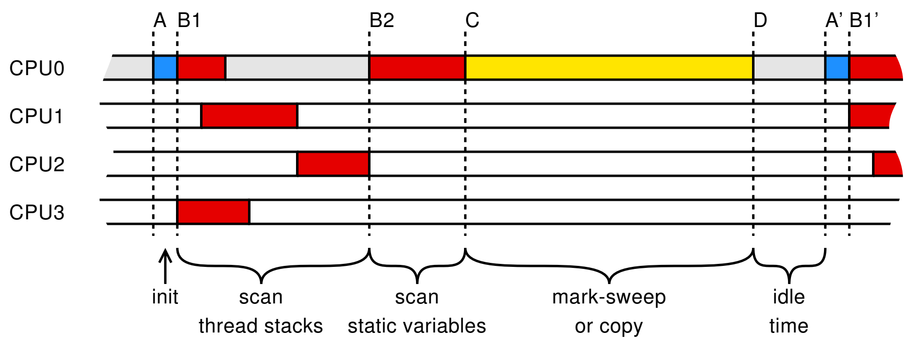

Figure 1: Multi-processor garbage collection

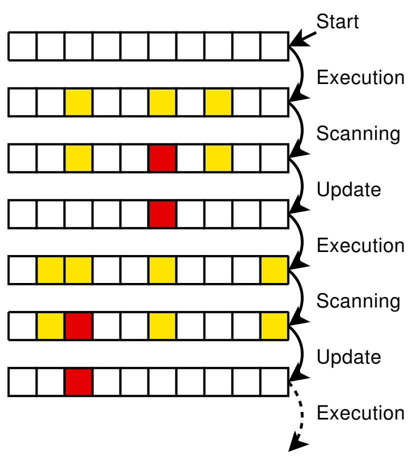

Figure 2: Algorithm for a single processor

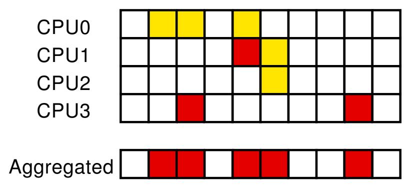

Figure 3: Snapshot of execution with four CPUs

This fact is utilized by snooping memory accesses of running processors and marking the detected references as live in a cache. In a second step, a processor scans the thread-local data, and marks the references it finds in a different color. After it has marked the cache entries, the references marked only during snooping can be marked as unused again, because they are known to be dead. The *cache narrowing* by the processor ensures that the amount of marked references in the cache is bounded.

In the following, we will use the colors *white*, *yellow* and *red* for marking. These colors were chosen to avoid confusion with Dijkstra's tricolor abstraction [8], which uses white, grey and black for marking. Gray scale reproductions of Figures 2 and 3 will show yellow as light gray and red as dark gray.

Figure 2 shows the algorithm for a single processor. Upon startup, all references are marked as unused (white). The processor then starts execution, and references it accesses are marked yellow. When the processor scans its stack, it marks the references it finds red, usually leaving some entries yellow. These entries are removed in the next step, where all yellow entries are reset to white. When the processor starts execution again, red entries become yellow again and, as in the previous execution phase, accessed references are marked yellow.

Note that a color always corresponds to the same bit pattern in the cache. Changing the interpretation of patterns would have resulted in a considerably larger hardware implementation than changing the values in the cache.

The algorithm ensures that a superset of all references which are contained in a root set is marked yellow or red. During execution, a reference is marked when it is accessed – as a reference has to be accessed to become live, all references which are live at some point of execution are marked yellow after that point. When the processor scans the thread-local data, it marks all live references red; after the cache narrowing, all yellow references are known to be dead and can be marked as unused again. When execution continues, red entries are reset to yellow, so they are not erroneously identified as live when the cache entries are narrowed the next time. Updating the references from red to yellow and from yellow to white is done by the hardware.

In a CMP system, the information of the caches for each processor has to be combined somehow. It also has to be ensured that the gathered information is consistent such that no reference in the root set can remain undetected. This is achieved with a memory in which the content of the per-processor caches is aggregated. In this memory, a reference is marked if it is marked yellow or red in any per-processor cache. The aggregate state computation is essentially a logical OR of the entries in the processor-local caches, which can be implemented efficiently in hardware.

In Figure 3, an intermediate state is shown; while the individual processors are in different phases of execution, the aggregate state reflects all references which may be part of a local root set. The aggregate state corresponds to the union of all processor-local root sets and can be used by the garbage collector to construct the global root set at any time.

2.1 Scheduling of Local Root Scans

There is no need to synchronize the root scanning of individual processors to achieve a consistent view of their root sets. Different strategies can be used for different threads, without changing the garbage collector. The most appropriate strategy can therefore be chosen for each thread.

A simple strategy is to use a timer interrupt to trigger a cache narrowing from time to time. This however introduces some jitter, which may not be tolerable for some applications.

Another solution is to wait until a thread waits for its next period, as proposed in [19]. As the stack is almost empty then and no overhead has to be put into checking when a stack scan should be done, this is a very convenient strategy from a threads perspective. For traditional root scanning schemes, this introduces a considerable waiting time to the garbage collector (see Figure 1). The proposed cache however allows one to use such a strategy without introducing waiting times.

A third solution is to scan the thread-local data upon thread switches. The state of the old thread has to be saved and the state of the new thread has to be read from main memory, so the overhead for narrowing the cache in the course of doing so can be kept small for some processors. Again, such a strategy would have caused considerable waiting times in the garbage collector for traditional root scanning schemes, but does not do so for the proposed cache.

2.2 Garbage Collection Period

The temporal decoupling of local and global root scanning comes at the cost of longer life times of references. A reference which is not contained in a thread's stack may still be marked in the root cache. In this section, we will show how this influences the requirements for the GC period of a copying garbage collector. The formulas can easily be extended to a mark-compact garbage collector.

An upper bound for the memory consumption is provided by the worst case memory consumption of all processors before they scan their stacks plus the amount of memory they allocate during the worst case execution time (WCET) of two GC cycles. It is necessary to take into account the GC cycle twice, because a GC cycle which started just before stack scanning cannot free the respective memory. This bound is therefore dependent on the allocation rate of individual threads and the GC period T_{GC}.

In [17], it is shown that for a copying garbage collector the following inequation must hold for situations where data is generated by one thread ("producer") and consumed by a different thread ("consumer"):

$$T_{GC} \leq \frac{H_{CC} - 2\sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{1}$$

H_{CC} is the overall heap size, T_i is the period of a thread τ_i, a_i the amount of memory allocated during each period by τ_i and l_i the "life time factor". It takes into account that data produced by one thread is consumed by another thread. It is defined as[1]

$$l_i = \begin{cases} 1 & \text{for normal threads } \tau_i \\ \left\lceil \frac{2T_c}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases} \tag{2}$$

If cache narrowings take place at the end of a task's period, the lifetime of objects is not extended artificially beyond a period. Therefore, the assumptions of the equations above are fulfilled and these formulas can be applied to the proposed GC algorithm.

If cache narrowings are triggered by an interrupt, the definition of l_i has to be extended such that the time between the narrowings of the cache is taken into account. With $T_{u.i}$ as time between these cache narrowings w. r. t. a thread τ_i and C_u as WCET of such a narrowing, it can be redefined as

$$l_i' = \begin{cases} 1 + \left\lceil \frac{T_{u.i}+C_u}{T_i} \right\rceil & \text{for normal threads } \tau_i \\ \left\lceil \frac{2T_c+T_{u.c}+C_u}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases} \tag{3}$$

By replacing l_i in Equation 1 with l_i', an upper bound for the GC period for a copying garbage collector can be computed.

Due to the limited size of the caches, we do not only have to meet restrictions on the available heap memory, but also on the number of references. A copying garbage collector using the presented root scanning scheme also has to fulfill the following inequation:

$$T_{GC} \leq \frac{\widehat{H}_{CC} - 2\sum_{i=1}^{n} \widehat{a}_i l_i' - 2\sum_{i=1}^{n} \widehat{a}_i}{2\sum_{i=1}^{n} \frac{\widehat{a}_i}{T_i}} \tag{4}$$

where $\widehat{H}_{CC}$ means the maximum number of references and $\widehat{a}_i$ is the number of objects a thread τ_i allocates each period.

2.3 Design Considerations

It would be resource intensive and inefficient if the references had to be stored in the cache along with the colors. Therefore, it is necessary that there is a simple mapping between references and positions in the cache. This is the case if the object layout is regular, e. g., if a dedicated area for object handles is used (like in JOP [16]) or if fixed size blocks for objects are used (such as in the JamaicaVM [21]). By making use of this mapping, two bits are needed in the cache for each reference to store the colors.

It also would be very resource intensive to update the coloring of the references (red $\rightarrow$ yellow, yellow $\rightarrow$ white) and the aggregate state in a single cycle. This task hence has to be spread across several cycles. Moreover, these updates should not delay the processors' execution to keep the jitter as low as possible. The solution is time-multiplexed access to the caches. In the first cycle, the processor may access the values; in the second cycle, the update "demon" updates the coloring and computes the aggregate state. This entails that access to the caches has to be at least two times faster than access to main memory in order not to affect performance. The "demon" runs continuously and does not need to be triggered in any way. The time until a color change in a processor local cache is reflected in the aggregate state is therefore dependent on the number of entries in the caches, the number of entries updated in one step of the "demon" and the time for one step.

Splitting updates into several cycles is not problematic with regard to references which move from one processor to the other, because the state for a reference is always updated within a single cycle. What *could* pose problems is the fact that the aggregate state may not reflect a recently created object. This may however be circumvented by either allocating new objects in an area which is not garbage collected in this GC cycle or by coloring objects upon allocation. After the start of a GC cycle, the garbage collector has to wait until the update "demon" has computed the aggregate state for all references, i. e., until all objects created before the start of the GC cycle are reflected in the aggregate state. Otherwise, objects that were created just before the start of the GC cycle could be garbage collected even though they are live.

A different issue with the split update is that processors have to wait after scanning their stack, until the update has unmarked all yellow entries. If it does not wait long enough, the update will only be partial, and stale objects may be considered live. It is however safe to use this data, as no live root object is marked as unused. If it can be guaranteed that the time between successful updates is bounded, the amount of stale objects is bounded as well.

[1]This is reformulation of the original definition

$$l_i = \begin{cases} 1 & \text{for normal threads } \tau_i \\ 2\left\lceil \frac{T_c}{T_i} \right\rceil & \text{for producer } \tau_i \text{ and consumer } \tau_c \end{cases}$$

which provides tighter bounds but is still safe.

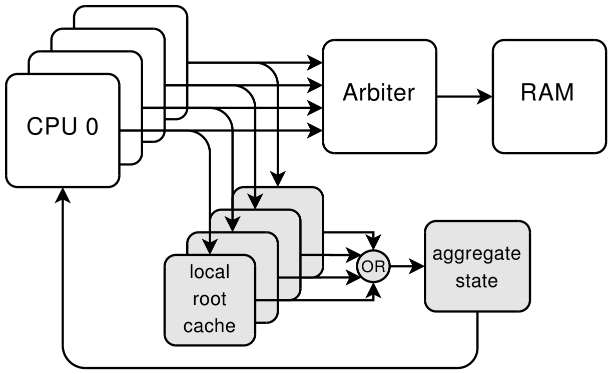

Figure 4: Block diagram

3. EVALUATION

We implemented the root caches in a CMP system based on Java Optimized Processor (JOP) [16, 11, 12]. While the principal algorithm is not tied to any specific processor, the results presented in this paper were obtained with this implementation and will be different in some places for other processors.

In our implementation, a single CPU takes care of garbage collection. While it would be possible to distribute the work load to more than one processor, we do not expect any gains from doing so, due to the synchronization overhead and the fact that access to shared memory rather than computation power appears to be the main limiting factor for GC in a CMP system.

Apart from the CPU dedicated to GC, all other CPUs are free to execute real-time threads. Figure 4 shows a block diagram of the individual components. The components we added to the standard configuration are shaded. While each CPU updates its state privately, one CPU can access the aggregate state to perform garbage collection. Accessing the per-processor caches and the main memory can be done in parallel.

We used a CMP based on JOP with 8 cores for evaluation. The CMP comprises 8 JOP cores, an arbiter for access to the shared memory, and synchronization and I/O modules. The board we used features an Altera Cyclone-II field programmable gate array (FPGA) and 512 KB SRAM with a 16-bit interface. The processor is clocked at 100 MHz; 32-bit accesses to the SRAM take 6 cycles. Each core includes 1 KB of on-chip memory for the method cache (a special instruction cache) and 1 KB for the stack cache. The root cache for each processor is also 1 KB in size, allowing for 4096 objects to be under the control of the garbage collector.

The GC algorithm used by JOP [19] is an incremental garbage collector, with a snapshot-at-beginning write barrier [24]. It is based on the copying collector by Baker [3], but uses a forwarding pointer placed in *object handles* to avoid the costly read barrier.

Memory arbitration is done by a time-sliced arbiter, which makes it possible to reason about the worst case latency of memory accesses. We chose a slot size of 6 cycles, which equals the time for one 32-bit memory access. As copying of objects must be atomic, the worst case memory access latency is however not only dependent on the slot size. Rather, the worst case memory latency for a CPU depends on the length of the atomic operations of all other CPUs. For the CPU that does GC, the longest atomic operation is the copying of the largest object in the system. For all other CPUs, it is the access to arrays, which includes three back-to-back memory accesses. Hardware support to eliminate the blocking due to copying is proposed in [18]. Future work will have to extend this solution to multi-processors.

3.1 Resource Consumption

Two bits are needed for each reference to realize the three possible colors. The number of objects depends on the application and is constrained by the overall heap size. JOP uses a dedicated area for object headers, a forwarding pointer and other GC information; each of these object handles is 8 words in size. Therefore, even if only objects that do not carry any data are created, no more than $\frac{heapSize}{4*8}$ objects can exist. This number will typically be lower, because objects consume memory as well, decreasing the maximum possible number of objects. Note that not all of the main memory is heap memory, because it contains also the bytecodes and class descriptions. Even very small Java programs occupy several dozen KB of main memory with such data.

With 1 MB of heap memory, 32768 objects can exist at most; furthermore, the payload of objects has to be taken into account, and – for a copying collector – that only one semi-space can contain live objects. An average object size of 33 Bytes is reported in [9], so a reasonable estimate for the number of handles would be $\frac{heapSize}{4*(8+8.25*2)}$, yielding 10700 handles for 1 MB of heap memory.

A safe limit can however only be determined by proper memory analysis. Such an analysis is necessary in any case to confirm that the heap is large enough to satisfy all allocation requests. Determining the maximum number of objects is even simpler than this analysis – using an object size of 1 for all objects, the "memory consumption" equals the number of objects.

As a concrete example, consider the system we used for evaluation, with eight CPUs and 512 KB main memory. Allowing 4096 handles, this yields a memory consumption of $\frac{2*4096}{1024*8}=1$ KB for each per-CPU cache. The aggregate state cache is 0.5 KB in size. For the whole system, 8.5 KB of on-chip memory are required for the root caches, which is an overhead of 1.66% when comparing it to the size of the main memory. In the JOP-based CMP system, other on-chip memories (method cache, stack cache and microcode ROM) occupy 4.625 KB per processor. The overall memory consumption is 45.5 KB, of which the root caches occupy 18.7%.

The system in consideration is balanced with regard to the number of handles and the available heap memory for average object sizes of 8 to 9 words for medium sized applications (about 100 KB of non-heap data). These figures are consistent with the average object sizes reported in [9].

For the configuration presented above, 32 entries can be updated in one cycle without inferring additional memory blocks (one cache consists of two 4 Kbit memory blocks, which together allow 64 bit access in one cycle). An update consequently takes 256 cycles, or 2.56 μs at 100 MHz. This time has to be taken into account for the blocking time of a processor for root scanning.

The whole CMP system occupies 26546 logic cells (LCs). Each processor core consumes about 2600 LCs; arbitration, synchronization and I/O modules consume about 3150 LCs. The update logic occupies 2588 LCs, which is 9.7% of the total LC count. This number could have been lowered by updating fewer entries in one cycle at the expense of a longer update period.

3.2 Execution Time

In the presented scheme, every processor only has to access the cache memory during stack scanning, which is not shared with other processors. The processors do not have to wait for their memory slot and stack scanning therefore has the same WCET regardless of the number of processors (i. e., it is as fast as on a single processor system). The presented scheme however infers a delay for waiting until the update is finished; the length of the delay depends on the configuration, but a value of 2.56 μs (see Section 3.1) is reasonable. This overhead is an order of magnitude smaller than the

Table 1: Measured execution times

	BCET	WCET
Traditional Root Scan	23 μs	87 μs
Cache Narrowing	13 μs	37 μs

WCET for stack scanning with time-multiplexed access to the main memory. A system consisting of 8 CPUs running at 100 MHz with 256 stack entries and a memory with 6 cycles access time needs at least 122.88 μs until all processors have written the contents of their stacks to main memory. Manual analysis of the bytecodes for cache narrowing resulted in a WCET of around 39 μs on JOP. Note that the figures presented in Section 3.3 are lower, because 64 words of the stack cache are not used for the run-time stack.

3.3 Experimental Results

For our measurements, we used the already mentioned CMP based on JOP, with 8 cores and 512 KB main memory. While one core was responsible for GC, the other seven cores executed the working threads.

Table 1 compares measurements for the best case execution time (BCET) and WCET for traditional root scanning and the proposed solution. These were obtained by triggering a stack scan at a period of 1 ms, while a task that occupied the stack was executed at various periods. The traditional root scanning was emulated by writing the stack contents to a dummy memory location. While the execution time of traditional root scanning depends on the access to main memory, the proposed scheme accesses only local memory and does not need to rival with other processors for this resource. The WCET for this task is therefore less than half for the proposed solution, compared to traditional stack scanning. The difference might be even bigger for arbitration schemes where some processors are preferred over others with regard to memory requests. While the preferred processors may have a lower WCET for cache narrowings, the memory access latencies (and hence the WCET) for other processors are inevitably increased.

Tables 2, 3 and 4 show our experiments with different cache narrowing policies. In the experiments for Tables 2 and 3, the cache narrowing is triggered periodically, with periods of 1 and 10 ms, respectively. For Table 4, the root cache is narrowed at the end of each period of a working thread.

Throughout all tests, the garbage collector is executed in a loop with a gap of 1 ms between iterations to allow for logging to be done. The working threads allocate various data structures, to test different aspects of the garbage collector. They also check the integrity of the allocated data to provide evidence for the correctness of the garbage collector.

The minimum periods were determined in two ways: on the one hand we measured the WCET, on the other hand we lowered the task periods every 100 seconds, until the system ran out of memory. The lowest period at which the system operated correctly, without missing any deadlines, is shown in the tables. The allocation rate shown is the rate reported by the test for this minimum period. The jitter displayed is the maximum release jitter we measured throughout the full run of a test.

ObjectTest is a "best case" example: The working threads allocate one simple object in each period. The objects do not need to be scanned and the tracing of the object graph is effectively a non-issue. ObjectTest is the only test which reaches the empirically determined WCET. For ListTest, the working threads allocate a linked list with 100 elements each period. For TreeTest, a tree structure

Table 2: Measurements with narrowing period of 1 ms

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
ObjectTest	0.25	26977	90
ListTest	27.5	25416	52
TreeTest	27.5	23954	51
ArrayTest	1.2	5654	206

Table 3: Measurements with narrowing period of 10 ms

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
ObjectTest	0.25	26428	79
ListTest	37.5	18665	40
TreeTest	30.0	21970	49
ArrayTest	1.7	3870	259

Table 4: Measurements with narrowing at end of task period

Test	Min. Period (ms)	Allocations (Objects/s)	Jitter (μs)
ObjectTest	0.25	27325	26
ListTest	30.0	23365	26
TreeTest	30.0	21952	26
ArrayTest	1.1	6229	62

comprising 94 elements is allocated. Obviously, the object graph for these structures is more complex than for ObjectTest. ArrayTest allocates integer arrays with 256 elements, resulting in a memory consumption of 1 KB for each array. While the object graph for this test is simple, it challenges the garbage collector through the size of the objects which must be copied.

ObjectTest, ListTest and TreeTest achieve similar allocation rates. ObjectTest is however bounded by its WCET, while ListTest and TreeTest run out of memory at higher execution frequencies. Further tests showed that the garbage collector can keep up with an allocation rate of 34884 objects per second and a minimal period of 160 μs for ObjectTest if the WCET is ignored. ArrayTest has a low allocation rate in terms of objects. This is not surprising, taking into account that it allocates large objects and thus the limiting factor is its allocation rate in terms of bytes rather than in terms of objects.

The release jitter for ListTest and TreeTest is lower than for ObjectTest. The sources for the release jitter are however the same. Consequently, we have to assume that the measurement is too optimistic for ListTest and TreeTest. The release jitter ArrayTest is considerably worse, compared to the other tests. This is due to the atomic copying of objects and arrays, which causes a release jitter proportional to the size of the largest object or array.

Table 3 shows the effect of a longer narrowing period: the achievable allocation rate is lowered, while on average, the jitter remains the same. The lower allocation rate can be explained by the extended life time of objects in the root cache. As more objects are live at the same time, the system runs out of free memory earlier. The effect is not the same for all tests: while the minimum period for ListTest and ArrayTest is considerably increased, there is only a small increase for TreeTest. ObjectTest again reaches its WCET. An advantage of the longer narrowing period is however that the relative overhead for narrowing the cache is decreased.

Narrowing the root cache at the end of the task period yields similar minimal periods as narrowing the cache at a period of 1 ms. While for ListTest and TreeTest, the minimal period is only slightly higher, it is slightly lower for ArrayTest. For ObjectTest the WCET is reached again. The advantage of this policy is however, that the release jitter is lowered considerably. This is due to the fact that the thread stacks are almost empty at the end of a period. The measurement for ArrayTest seems to be too optimistic; we assume that this is caused by advantageous thread phasings. While the jitter may seem to be negligible when comparing it to the task periods of ListTest and TreeTest, it is an issue for a task with sub-millisecond periods such as ObjectTest. The relative overhead for narrowing the cache increases for smaller periods with this strategy.

The figures in Tables 2, 3 and 4 indicate that there is a trade-off between the needed task period/allocation rate and the allowable jitter and overhead. This trade-off is influenced by the allocation behavior of a task, which therefore must be considered to find an optimal solution.

4. RELATED WORK

While there are numerous papers on GC, only few take blocking times due to root scanning into consideration [22, 25, 2]. The work we are aware of in this area covers only uniprocessors. A more recent paper [1] identifies problems with the consistency of the root set in multi-processor systems, and suggests a special write barrier to overcome these. It still requires that the stack of a single thread is saved to main memory atomically.

The presented algorithm shares some concepts with reference counting techniques like the Deutsch-Bobrow algorithm [7] and one-bit reference counts as proposed in [23]. These algorithms use an approximation of the reference count, which is made precise on certain occasions. While the Deutsch-Bobrow however caches which references are probably dead in a "zero count table", the algorithm presented in this paper caches which references are probably live. One-bit reference counts are also used to remember the live references, but a full mark/sweep run is needed to overcome the problems of reference counting. A fundamental difference between reference counting approaches and the proposed root cache is however that the latter only "counts" the roots of the object graph, while the former apply reference counting to all objects.

5. CONCLUSION AND OUTLOOK

We presented a novel root cache, which allows that thread-local root scanning can be decoupled from the root scanning phase of a garbage collector in a CMP system. This enables flexible scheduling of thread-local stack scans without impairing the garbage collector. The proposed cache also lowers the WCET of stack scans compared to traditional root scanning techniques. Thread-local root scanning does not occupy memory bandwidth, which is a typical bottleneck in CMP systems.

The evaluation of the root cache demonstrated that the jitter caused by GC is reasonably low to allow even sub-millisecond task periods if only objects and small arrays are used. Further research is however necessary to avoid the atomic copying of larger arrays, which causes considerable jitter.

Acknowledgement

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

6. REFERENCES

[1] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 249–258, New York, NY, USA, 2007. ACM.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[4] Greg Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[5] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[6] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In Jan Vitek, editor, *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, Chicago, IL, June 2005. ACM Press.

[7] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[8] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[9] M. Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

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

[11] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

[12] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the IEEE Third Symposium on Industrial Embedded Systems (SIES 2008)*, Montpellier, France, June 2008.

[13] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. STOPLESS: A real-time garbage collector for multiprocessors. In Mooly Sagiv, editor, *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[14] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th international workshop on Java technologies for*

real-time and embedded systems (JTRES 2008), Santa Clara, California, USA, September 2008. ACM Press.

[15] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[16] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

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

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

[19] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Fifth International Workshop on Java Technologies for Real-Time Systems (JTRES)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[20] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[21] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.

[22] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, April 2001.

[23] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–9, 1977.

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

[25] Taichi Yuasa. Return barrier. In *Proceedings of the International Lisp Conference 2002*, 2002.