Real-time Garbage Collection for a Multithreaded Java Microcontroller

S. Fuhrmann, M. Pfeffer, J. Kreuzinger, Th. Ungerer
Institute for Computer Design
and Fault Tolerance
University of Karlsruhe
D-76128 Karlsruhe, Germany
{s_fuhrm, pfeffer, kreuzinger, ungerer}@ira.uka.de

U. Brinkschulte
Institute for Process Control,
Automation and Robotics
University of Karlsruhe
D-76128 Karlsruhe, Germany
brinks@ira.uka.de

Abstract

Keywords: *garbage collection, microcontroller, Java microprocessor, real-time, multithreading*

We envision the upcoming of microcontrollers and systems-on-a-chip that are based on multithreaded processor kernels due to the fast context switching ability of hardware multithreading. Moreover we envision an extensive market for Java-based applications in embedded real-time systems.

This paper introduces two new garbage collection algorithms that are dedicated to real-time garbage collection on a multithreaded Java microcontroller. Our garbage collector runs in a thread slot in parallel to real-time applications. We show that our algorithms require only about 5% of the processor time for an effective garbage collection concerning our real-time benchmarks.

1. Introduction

Garbage collection is a research field that has been investigated since decades. The extensive use of Java in the last years yields a renewed interest in garbage collection. Java lacks explicit memory release methods and therefore a garbage collector is needed to reclaim heap memory that is no more in use.

The application of Java in embedded real-time systems produces requirements that are not fulfilled by most existing garbage collection algorithms. To preserve the predictability of the real-time application program execution, the requirements for real-time garbage collection are an incremental algorithm, predictability, feasibility, small synchronization needs, robustness and efficiency. Moreover embedded systems require to get by on small memory.

For the future we envision the upcoming of microcontrollers and systems-on-a-chip based on a multithreaded processor core due to the fast context switching ability of hardware multithreading. Motivated by our innovative multithreaded Komodo microcontroller [6], we go one step further in garbage collection and examine the use of garbage collectors in multithreaded hardware with real-time requirements. Current approaches investigate the use of multithreaded hardware to bridge latencies that arise by cache misses, pipeline hazards or long running instructions. Multithreaded processors are able to bridge these latencies efficiently, if there are enough parallel executable threads as workload and if the time necessary for switching of threads is very small. In consequence, recent announcements of high-performance processors by industry concern a 4-threaded Alpha processor of DEC/Compaq [9] and Sun's MAJC-5200 processor, which feature two 4-threaded processors on a single die [10]. Both processors are designed as high-performance processors and will not be suitable for low-cost embedded systems. However, the multithreaded MediaProcessor of MicroUnity [11] is specialized for multimedia appliances and the recent multithreaded Network Communication Processor of XStream Logic [15] is a dedicated network processor.

Our Komodo project [4] explores the suitability of multithreading techniques in embedded real-time systems. We propose multithreading as an event handling mechanism that allows efficient handling of simultaneous overlapping events with hard real-time requirements. We design a microcontroller with a multithreaded processor core. This core features the direct execution of Java bytecode, a zero-cycle context switch overhead and hardware support for scheduling and garbage collection [5]. Because of its application for embedded systems, the target architecture is kept on the level of a simple pipelined processor kernel, which is able to issue one instruction per cycle. To scale up for larger systems, we propose the parallel execution on several microcontrollers connected by a real-time middleware [7].

Based on the Komodo microcontroller, we build an

adapted Java Virtual Machine with two slightly different garbage collection algorithms. Both algorithms are incremental in a very fine-grained manner; one gets by without synchronized regions and the other features very short sojourns in its synchronized regions.

For better understanding of the requirements for our garbage collectors, section 2 explains the multithreaded Komodo microcontroller. Section 3 lists requirements for garbage collection in a multithreaded real-time environment and discusses potential algorithmic approaches. Our own approach is described in section 4, illustrated in section 5, and discussed concerning real-time issues in section 6. Section 7 summarizes related work. Performance evaluations are shown in section 8 and section 9 concludes the paper.

2. The Komodo Microcontroller

The Komodo microcontroller [6] is a multithreaded Java microcontroller, which supports multiple threads with zero-cycle context switching overhead and several scheduling algorithms. Because of its application for embedded systems, the processor core of the Komodo microcontroller is kept at the hardware level of a simple scalar processor. The four stage pipelined processor core consists of an instruction-fetch unit, a decode unit, an operand fetch unit and an execution unit. Four stack register sets are provided on the processor chip.

The instruction decode unit contains an instruction window (IW) for each hardware thread. A priority manager decides from which IW the next instruction will be decoded. We define several scheduling algorithms to handle real-time requirements. In detail, we implemented the fixed priority preemptive (FPP), the earliest deadline first (EDF), the least laxity first (LLF), and the guaranteed percentage (GP) scheduling schemes [14]. The basic GP scheduling used in our evaluations in section 8 allows to assign a specific percentage of processor time to each thread (of together not more than 100%). The priority manager applies one of the implemented thread schedulers for IW selection. In case of GP scheduling, the percentage conditions are observed in a very fine-grained manner within 100 processor cycles. However, latencies may result from branches or memory accesses. To avoid pipeline stalls, instructions from threads of less priority can be fed into the pipeline. There is no overhead for such a context switch. No save/restore of registers or removal of instructions from the pipeline is needed, because each thread has it's own stack register set. In our current implementation, the Komodo microcontroller holds the contexts of up to four threads, which are directly mapped to hardware threads.

The Komodo microcontroller's garbage collector is executed in one of the hardware thread slots running in parallel to the real-time application threads. Our two proposed garbage collectors affect the hardware in two ways. First, each stack entry is enhanced by an *object reference bit*, which indicates that an entry is a reference to an object. This makes it easy to find all root pointers on the stack. Second, new instructions have been implemented to allow read access from one stack to the stack of another thread. Additionally, the trap routines for byte codes, which manipulate objects, are enriched with instructions for the garbage collector.

3. Real-time Multithreaded Garbage Collection

The Komodo microcontroller poses specific requirements on garbage collection algorithms due to its hardware multithreading and its real-time application field:

- *Incremental* — Real-time garbage collectors must be incremental, i.e. the garbage collection is done in small portions and the garbage collector is preemptive, because a real-time service routine cannot be deferred for an unlimited amount of time[18]. Most of the well-known garbage collectors are not *incremental*, but apply the *stop-and-copy-technique* [16] that stops all applications during a garbage collection run. Incremental collectors can be classified into two groups:
  - *Snapshot at beginning* — the stack is copied (short stop of the application thread) before the garbage collection starts and the collector accesses only the copy, whereas the application threads still access the original stack (needs write barriers only).
  - *Incremental update* — the application threads notify the garbage collection, whenever the stack is updated (needs read/write barriers).
- *Predictability* — Hard real-time systems require a predictable behavior such that an *a priori* worst-case performance analysis is possible. The garbage collector must maintain fixed upper bounds of run-time interference concerning the real-time application thread(s).
- *Feasibility* — The high reliability and robustness requirements of embedded software require an a priori analysis of run-time *feasibility*. The garbage collection should not disturb real-time scheduling.
- *Robustness* — The garbage collection strategy should help to realize safe and robust programs. The programmer should be freed from writing complex memory management software that may result in hard-to-find run-time errors.

- *Efficiency* — Processor time is a scarce resource, in particular in embedded systems. The garbage collection should utilize the available processor time most efficiently and limit its influence on the execution of the application threads.
- *Small synchronization needs* — A garbage collector should be *non-disturbing*, i.e., it should not prevent the application threads from their time-critical tasks by consuming too much processor time. The collector should perform its task not only in small portions, but also in time steps that are sufficiently far away from each other. Each pause of the application threads that is caused by the garbage collector should be limited.
- *Non copying* — Memory is a limited resource in embedded systems. Therefore *copying garbage collectors* are less applicable in embedded systems, because memory requirements are doubled. Also complex algorithms that would need much memory space are undesirable.

A well-known garbage collector scheme is the *tricolor marking* that was first introduced by Dijkstra [8]. Baker's *incremental copying technique* [1] is one of the best-known real-time techniques, even though it shows significant weakness (see [13], p. 136). Baker developed also a *non-copying collector* [2] that eases the weaknesses of his copying collector. The collector works with four double-chained lists, which represent the execution state of objects in the garbage collection:

new-set links newly allocated objects,

free-set manages freed objects,

from-set lists all objects marked before the garbage collection, i.e. the *white objects* of the tricolor scheme,

to-set concerns all objects being processed, i.e. linking the *black* objects (the already visited) and the *gray* objects (the objects under treatment).

4. Garbage Collection for the Komodo Microcontroller

4.1 General Layout of Memory Management

The memory management system of the Komodo microcontroller is separated into two layers: The lower layer with the explicit memory management and the upper layer with the automatic memory management. The lower layer provides the explicit memory management methods (*alloc()*, *free()*), visible for the garbage collector, but invisible to the application. The garbage collection is part of the upper layer, which itself hides memory disposal from the application threads, but provides object allocation methods for the application.

The memory management chosen for Komodo allocates memory blocks in sizes of power of two. Allocation requests are being rounded up to the next power of two and then resolved by a table lookup. This solution achieves good performance, but induces fragmentation [19] yielding an average memory usage of about 75%, which may be unacceptable for embedded systems with very small memory.

The order of memory operations (allocation, freeing) is arbitrary. Allocation requests of memory blocks are satisfied in fixed portions (powers of two). Allocation information is stored implicitly by storing each block into a separate list for its size class. Using this strategy a *best-fit* algorithm is implemented.

In the worst-case the memory management has to look up *all* block classes to satisfy a request of the smallest block size. The time complexity of the tests in the free lists is $O(\log n)$ with a memory size of n (in our implementation a maximum of 15 tests for 1 MB memory size).

4.2. Garbage collection

In the following section, we propose two new algorithms, which are based upon Dijkstra's tricolor marking [8] and Baker's non-copying collector [2]. The two algorithms differ in how gray marked objects are being treated.

The algorithms mark themselves off the real-time algorithms presented in the prior section and stand in contrast to the trend of heuristic copying garbage collectors. The philosophy of the new collectors is *not* only a fast execution in the common case that may have a long worst-case execution time. The aim is a simple, homogenous, and therefore computable algorithm with a *predictable maximum execution time*.

For reasons of simplicity the first algorithm is named NADEL (needle) and the second KETTE (chain). The name NADEL (needle) stems from the search for gray objects that is similar to the search for the so-called *'needle in the haystack'*.

In the following unvisited and unused objects are colored white. Used objects are colored gray until all their descendants are visited, and then they are colored black.

NADEL and KETTE are similar in their proceeding. They adapt the tricolor marking to a multithreaded execution environment. They operate in the following phases:

1. Mark
   (a) In the beginning mark all objects **white**.
   (b) Mark all static objects. Since the garbage collection does not remove unused Java classes from

memory, all objects referenced by static fields of a class are being marked **gray**. This excludes the possibility of wrongly freed objects.

(c) Mark the objects directly reachable from the stack **gray**. All stacks of the application threads must be inspected for marking the objects that are directly reachable from the stacks as being alive.

(d) Mark *all* reachable objects on the heap **gray** (including their *descendants*). The prior marking operations only affected the objects *directly* reachable from the stacks or classes. To complete the marking phase, all objects indirectly reachable from this marked set must be also marked. Objects with all descendants marked gray are marked **black**. Because this phase is independent of the location of the stacks it has been split from the prior marking to reach a higher level of independency.

2. Sweep

(a) Free all objects colored **white**. Since the mark phases colored all reachable objects **black**, all unreachable objects are *not* marked and therefore still colored **white**.

NADEL handles gray objects as follows: NADEL holds all objects in a global list `GC_OBJS`. If an object's color changes, NADEL stores a byte value representing the color in the object's header. The object colors only change monotonically in a garbage collection cycle from white to gray to black. Therefore it is not necessary to synchronize reading and writing the color of objects.

KETTE also uses a color value in the object header, but additionally uses four lists: `GC_WLIST`, `GC_GLIST`, `GC_BLIST`, `GC_OBJS`. Grey *and* white objects are in the double linked list `GC_WLIST`. Grey objects are additionally in the single linked list `GC_GLIST` (fast access to gray objects). After all children of an object have been examined by the garbage collection, the object is colored black and transferred from the double linked list `GC_WLIST` to the double linked (black) list `GC_BLIST`. New objects are stored in a separate double linked list `GC_OBJS` by the `new`-traps and are marked black at the beginning of their lifetime. These two lists for black objects avoid collisions between the garbage collection and object creation when coloring objects black (therefore less synchronization overhead). `GC_BLIST` and `GC_OBJS` are being merged when swapping the colors black and white and transferred to `GC_WLIST`. To maintain list coherency, all list operations on shared lists must be *synchronized* with all other application threads.

The algorithms have the following features in common:

- Non-moving mark-and-sweep-collectors,
- Non-overlapping mark and sweep phases,
- Exact pointer handling (stack & heap),
- Incremental update,
- Garbage collection in a *separate thread*,
- Runnable in a separate thread slot,
- Permanent execution of the application threads possible (no long pauses).

For further discussions of the real-time issues of our algorithms see section 6.

5. Example

Figure 1 depicts a snapshot of the NADEL garbage collector. On top there is the runtime stack that forms the major set of root pointers. Each circle represents an object on the heap. The solid arrows depict pointers to objects on the heap. The colors of the circles stand for the garbage collection state of the respective object. White objects have not been processed yet. The gray objects have been visited, but not their descendants. The black objects have been processed completely including their descendants.

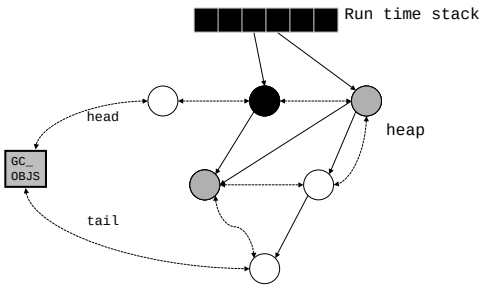

Figure 1. Snapshot of a NADEL garbage collection run

All objects are kept in a single double linked list `GC_OBJS`, which is outlined by dashed arrows. For searching a gray object, a traversal of the linked list `GC_OBJS` is necessary. In the worst case a search in a linked list has a time complexity in $O(n)$. Because each object has to be found once, the resulting time complexity for the garbage collector is $O(n^2)$.

Figure 2 shows the same example with the KETTE garbage collection. Comparing to NADEL there are more lists for the object management. The `GC_WLIST` list contains the white and gray objects. The gray objects are additionally held by the `GC_GLIST` list. Black objects reside in the `GC_BLIST` list. For searching gray objects only one list operation in the `GC_GLIST` list is necessary, this results in a complexity of $O(n)$ for the garbage collection.

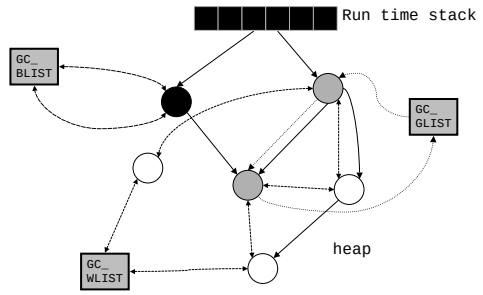

Figure 2. Snapshot of a KETTE garbage collection run

6. Discussion of Real-time Issues

Since our garbage collector runs incremental using its own hardware thread slot, the interference with the application real-time threads is widely reduced. But in fact this interference is not null.

There are two basic issues, where the garbage collector affects the application real-time threads: first, the additional barrier code that is being executed on the context of the application thread, second, the synchronization between the KETTE garbage collection thread and the application threads.

Additional barrier code is needed for both algorithms, NADEL and KETTE. The implementation of the read/write barrier of NADEL requires two instructions, if the processed reference is the *null pointer*, nine instructions if the object has already been marked, and 15 instructions if the object has to be marked by barrier code.

The read/write barrier of KETTE also requires two instructions for the *null pointer*, 11 instructions for a marked object, and 28 instructions if the object must be marked.

The shortest executed instruction with additional barrier overhead is the `astore` bytecode. The execution of our `astore` trap needs about 40 cycles to succeed. The unpredictable execution time factor of the application thread is therefore in the worst-case $\frac{15+40}{2+40} - 1 \approx 30\%$ for NADEL and $\frac{28+40}{2+40} - 1 \approx 60\%$ for KETTE (the fixed slice of execution is 40 cycles and the dynamic slice ranges from 2 to 15 resp. 28 cycles). Since `astore` is the shortest instruction with barrier overhead, the calculated values can be considered to be absolute worst-case for an application thread in a hard real-time environment.

The waiting time through synchronization of list accesses occurs only in the KETTE algorithm. Application threads can be blocked when executing barrier code. While the duration of such an event is limited, the number of these events cannot be predicted. This may lead to priority inversion. With the use of methods against priority inversion the critical intervals can be bounded in the case of a FPP or EDF scheduler. In the case of GP methods against priority inversion still have to be examined. A possible solution would be to add the percentage of a blocked thread to the blocking thread resulting in some sort of percentage inheritance.

7. Related Work

To our knowledge no investigations have been done yet to implement real-time garbage collection on multithreaded processors. Comparing our new algorithms with classic garbage collection algorithms, the most similarity exists with Baker's non-copying collector. Objects are managed in lists and the algorithms are therefore appreciable in time cost. In contrast to Baker's non-copying collector there's no strong separation of objects between the lists, i.e. in the new algorithms, the presence of an object in *two lists at the same time* is allowed for the reason of efficiency (not allowed in Baker's scheme).

In contrast to Baker's *copying collector*, nothing is done against fragmentation by both of our collectors. This design decision makes it possible to estimate the runtime of the garbage collection. The collectors don't consume processor time for copying objects and don't need the double amount of memory like a copying collector does.

The garbage collection of Huelsbergen & Winterbottom ([12], non-real-time) and of Wallace & Runciman ([17], worst-case time complexity $O(n^2)$) serve as a motivation for emphasis on synchronization, management cost (memory words per object) and worst-case performance. These points separate our new main algorithm into the two algorithms NADEL and KETTE.

A completely different approach to real-time garbage collection is done by the *Real Time Specification for Java* [3]. Memory is divided into three different areas: scoped memory, physical memory and immortal memory. Scoped memory contains objects with a limited lifetime. Physical memory is a memory area for objects with special important characteristics as e.g. a memory with fast access. Immortal memory stores objects that are never garbage collected. In contrast to that, our approach uses a faster and less complex memory structure, keeping memory management invisible to the application threads. Our garbage collection algorithms are dedicated to a multithreaded processor, in contrast to the Real Time Specification for Java. If the Real Time Specification would be applied to a multithreaded processor, our algorithms may be integrated into the specification.

8. Performance Evaluation and Results

The performance tests presented in this section were done with a software simulator for the Komodo microcontroller. The simulated Java Virtual Machine was in a

stage of development not yet incorporating speed-up techniques like *quick opcodes* and *resolved methods* known from SUN's picoJava processor. With such techniques the application threads are faster in producing objects and therefore the garbage collection needs more processor time.

8.1. Average and Worst-Case Performance Benchmark

We developed specific benchmarks to investigate the average and the worst-case performance of NADEL and KETTE. A linear list of objects is collected as *average case* in the marking direction, and against the marking direction representing the *worst case*. NADEL needs a long time for the *worst-case* test, because the complexity of the garbage collection time is $O(n^2)$, where n is the number of objects. KETTE shows no differences between average and *worst case* test. Each algorithm has to execute some list operations for carrying on. In the worst-case NADEL has to search through the whole heap, whereas KETTE only performs one list operation.

8.2. Caffeine Embedded Benchmarks

As second benchmark, we used the *Caffeine Embedded Benchmarks*, which are Java benchmarks especially designed for microcontrollers. They need only a minimum number of API classes and measure the basic performance of the microcontroller.

Like all Java applications, the Caffeine Embedded Benchmarks feature a specific behavior of memory allocation. It is of particular interest to study the gain of free memory, when the garbage collection, which is running in parallel to the benchmark, is getting more calculation time.

For the tests, the garbage collection is executed in a never-ending loop. For an exact assignment of the processing time, we used the *guaranteed percentage* scheduling (see section 2) with *accurate rates*. That means, no thread takes advantage of the latencies of other threads and always gets its fixed time slice.

In figures 3 and 4 the graphs of several simulation runs of NADEL and KETTE are depicted with different rates of processing time given in percentages of the GP scheduling scheme. A falling of the memory level is provoked by the benchmark, a rise by the garbage collector. Due to the permanently running garbage collector the marking phases can be recognized as a falling or constant memory level and the sweeping phases as a mainly rising or constant memory level.

The figures 3 and 4 show that the memory needed for the execution of the Caffeine Benchmarks is getting smaller at a high rate of garbage collection. The lowest graph in the figures represents the Caffeine Benchmark without garbage collection.

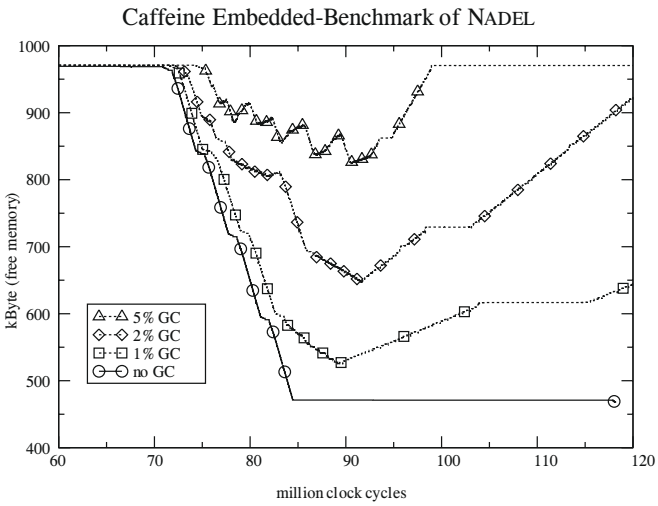

Figure 3. Free memory of NADEL with a fixed rate of processing time

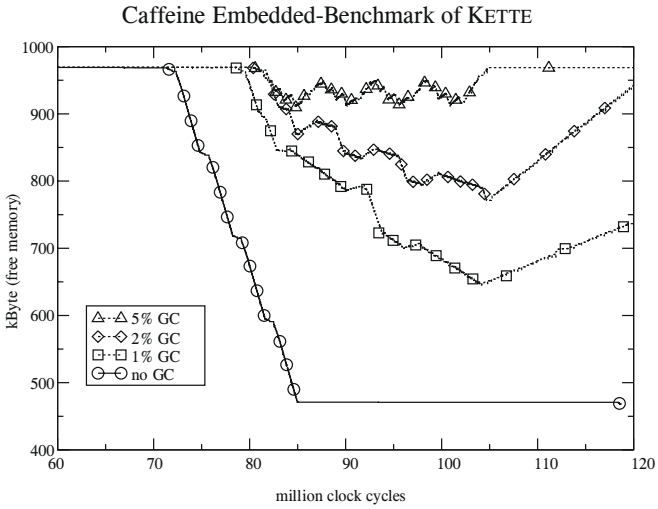

Figure 4. Free memory of KETTE with a fixed rate of processing time

The most interesting effect of the graphs can be recognized at the positions of the latest three slopes in the 5%-curves in figures 3 and 4, e.g. the maximum of the latest slope in KETTE occurs later than in NADEL. KETTE interferes with the Caffeine Embedded Benchmark in its critical sections. Due to the exact scheduling KETTE only gets its small amount of processing time, even though nothing else is being computed. Thus the critical sections of KETTE are enlarged compared to a different scheduling strategy. In spite of their high amount of processing time the Caffeine Embedded Benchmarks wait for the garbage collection, which has a lower priority. This effect resembles the *priority inversion*, even though there are only two active

threads. The graph for KETTE looses significance due to this effect. The inheritance of percentages proposed in section 6 may be a solution for this problem.

8.3. Producer Consumer Benchmark

As last benchmark we simulated a *producer consumer benchmark*, which is a typical application of embedded systems. The motivation for this benchmark is an autonomous guided vehicle equipped with the Komodo microcontroller as a control unit [4]. The vehicle orients itself by a line on the floor, which is being recognized by a line camera in a 64-bit vector (producer thread). This vector has to be processed for the calculation of the driving direction (consumer thread).

The benchmark is based on these specifications. The microcontroller executes two threads in parallel: the producer and the consumer. The producer creates a byte vector with dimension 64, fills it with random values and stores it in a shared container object. The consumer waits for new data from the camera in the container object, picks them up when available and does a few operations on the data.

The actual realization of the benchmark resembles the approach for the Caffeine Benchmark. Both garbage collectors NADEL and KETTE were run with an *exact* percentage of processing time with *guaranteed percentage* scheduling. The producer and the consumer threads share the total processing time minus the fraction of processing time for the garbage collection.

The results of the benchmark are depicted in figures 5 and 6. The graphs show the free memory in relation to the execution time of the benchmark.

Both graphs show that a rate of 1% is not sufficient for garbage collection. The graphs of the 5% garbage collection show sufficient garbage collection ability. The garbage collection keeps up with the object allocation. The 10% garbage collection rate performs smoother and frees more memory than a 5% rate, but the gain is just in the range of about 1 kByte. With this information we suppose a high degree of saturation of the garbage collection.

All the measurements refer to a data interval of about 2 ms at a clock rate of 50 MHz. A higher data rate requires a higher rate of garbage collection. A new object is created on an average of every 50.000 clock cycles.

An upper limit of the garbage collection rate of KETTE can be calculated. The relation of the garbage collection in percentage of the total processor time, denoted a, to application time $(1 - a)$ equals the relation of the garbage collection time t_2 to the application time t_1. This results in the equation $\frac{a}{1-a} = \frac{t_2}{t_1}$, resolving to a leads to $a = \frac{t_2}{t_1+t_2}$. For t_2 we inserted the garbage collection time for the deletion of 20 objects, which takes 17 362 clock cycles (measured with the worst-case performance benchmarks). This is an upper bound estimation, because in reality only five objects are garbage collected. t_1 is the time interval in the left column of table 1. As you can see in the table, for a data interval of 100.000 clock cycles the calculated percentage is 15%. This rate is slightly too large, because the 10% graph scarcely shows any improvement. The garbage collection portion of NADEL is $a = \frac{n \cdot t_2^2}{t_1 + n \cdot t_2^2}$. It depends on the heap size n and is therefore not applicable for all heap sizes.

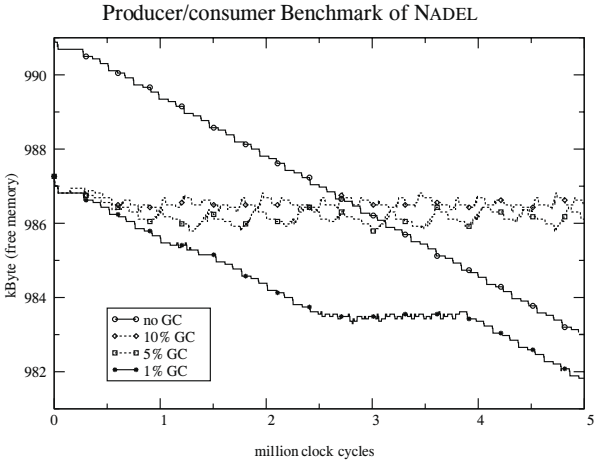

Figure 5. Free memory of NADEL with a fixed rate of processing time

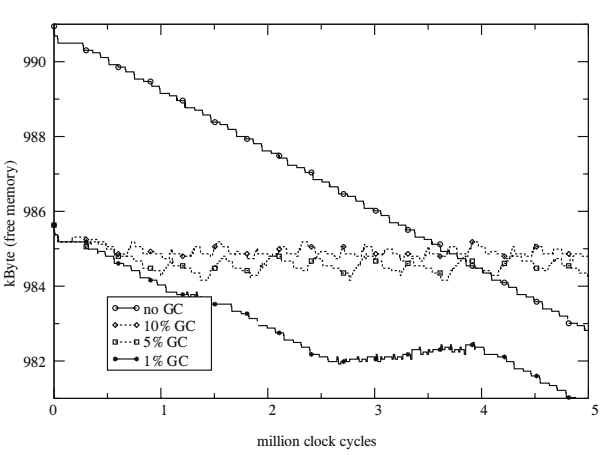

Figure 6. Free memory of KETTE with a fixed rate of processing time

9. Conclusion

Both of the implemented algorithms have strengths and weaknesses. KETTE has a better worst-case time complexity and should therefore be preferred to NADEL. A slow garbage collection means a high risk of lack of available memory. The small number of modified instructions (16 bytecodes) and the relatively small average slow-down (5%

Table 1. Calculated processing rate for garbage collection for given data rates

Data interval (in thousands of clock cycles)	Processing rate of garbage collection
50	26%
100	15%
150	10%
200	8%
250	6%
500	3%

resp. 7%) induced by the traps is a benefit of both algorithms.

The algorithms allow predictions about the application slow-down for homogenous instruction and data mixes. For KETTE with its linear time complexity dependent on the heap size, it's even possible to make adequate assumptions about the runtime of the garbage collection. NADEL with its quadratic time complexity does not allow estimations in such a simple manner.

The computation time slice that the *garbage collection* needs to stay head-on-head with the *object creation* is calculable, because of the linear order and is for the current implementation a maximum of 30%. This is a worst-case assumption for an infinite garbage creation. Usual applications will get by with much smaller slices. The measurements in the prior sections showed that 5-10% are sufficient.

NADEL is an attractive solution, because of its small barrier code, the absence of synchronization with the marker thread and the simple (and therefore less error-prone) implementation. However, NADEL's worst-case time complexity in $O(n^2)$ is a big drawback, reducing NADEL's usability in real-time systems.

KETTE's worst-case time complexity is in $O(n)$ and therefore a more attractive solution. The list operations in KETTE need synchronizations and are not immune against effects like priority inversion or long waiting for the processor mutex. KETTE needs four additional bytes in the object header for the system-wide gray list of objects.

The performance tests have shown the importance of efforts against priority inversion. For read and write barriers synchronization is invisible to the application and therefore more dangerous than explicit synchronization. The performance tests have also shown that the multithreaded processor technique of the Komodo microcontroller enables new scheduling techniques for garbage collection.

References

[1] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[2] H. G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[3] G. Bollella et al. *Real Time Specification for Java*. The Real Time for Java Experts Group, Dec. 1999. Draft Version.

[4] U. Brinkschulte, C. Krakowski, J. Kreuzinger, R. Marston, and T. Ungerer. The Komodo project: Thread-based event handling supported by a multithreaded Java microcontroller. *25th EUROMICRO Conference, Milano*, 2:122–128, September 1999.

[5] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer. Interrupt service threads - a new approach to handle multiple hard real-time events on a multithreaded microcontroller. *RTSS WIP sessions, Phoenix*, pages 11–15, December 1999.

[6] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer. A multithreaded Java microcontroller for thread-oriented real-time event-handling. *PACT, Newport Beach*, pages 34–39, October 1999.

[7] U. Brinkschulte, C. Krakowski, J. Riemschneider, J. Kreuzinger, M. Pfeffer, and T. Ungerer. A microkernel architecture for a highly scalable real-time middleware. *RTAS 2000 WIP sessions, Washington*, June 2000.

[8] E. W. Dijkstra et al. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[9] J. Emer. Simultaneous multithreading: Multiplying Alpha's performance. *Microprocessor Forum, San Jose, Ca*, October 1999.

[10] L. Gwennap. MAJC Gives VLIW a New Twist. *Microprocessor Report*, 13(12):12–15, September 1999.

[11] C. Hansen. MicroUnity's mediaprocessor architecture. *IEEE Micro*, pages 34–41, August 1996.

[12] L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. *International Symposium on Memory Management, Vancouver, Canada*, October 1998.

[13] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, 1997.

[14] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski. Real-time scheduling on multithreaded processors. *The 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000), Cheju Island, South Korea*, pages 155–159, December 2000.

[15] M. Nemirovsky. XStream Logic's network communication processor. *Microprocessor Forum 2000, San Jose, CA*, October 2000.

[16] B. Venners. Under the hood: Java's garbage-collected heap. *JavaWorld*, August 1996.

[17] M. Wallace and C. Runciman. An incremental garbage collector for embedded real-time systems. *Chalmers Winter Meeting, Tanum Strand, Sweden*, pages 273–288, 1993.

[18] P. R. Wilson. Uniprocessor garbage collection techniques. *International Workshop on Memory Management, Saint-Malo, France*, September 1992.

[19] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *International Workshop on Memory Management*, 1995.