Exploiting the Efficiency of Generational Algorithms for Hardware-Supported Real-Time Garbage Collection

Sylvain Stanchina
sylvain.stanchina@ikr.uni-stuttgart.de

Matthias Meyer
matthias.meyer@ikr.uni-stuttgart.de

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47
70569 Stuttgart, Germany

ABSTRACT

Generational garbage collectors are more efficient than their non-generational counterparts. Unfortunately, however, generational algorithms require both write barriers and write barrier handlers and therefore degrade worst-case performance.

In this paper, we present novel hardware support for generational garbage collection. In contrast to previous work, we introduce a hardware write barrier that does not only detect inter-generational pointers, but also executes all related book-keeping operations entirely in hardware. For the first time, write barrier detection and handling occur completely in parallel to instruction execution, so that the runtime overhead of generational garbage collection is reduced to near zero.

For evaluation purposes, we extended a system with hardware-supported real-time garbage collection with our hardware support for generational garbage collection. Measurements of Java programs on an FPGA-based prototype show that the generational extensions reduce the total duration of garbage collection activities by a factor of 5 and the memory traffic caused by the collector by a factor of 4 on average.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – Memory Management (Garbage Collection)

General Terms

Design, Languages, Measurement, Performance, Experimentation

Keywords

Real-Time Garbage Collection, Generational Garbage Collection, Write barrier, Object-Based Processor Architecture

1. INTRODUCTION

In real-time systems, garbage collection must be performed concurrently with application execution. However, the concurrent execution of collector and application requires synchronization mechanisms that prevent the application from interfering with the collector's work and ensure that no objects are prematurely reclaimed [16]. Usually, these synchronization mechanisms, be it for read or write barriers, for incremental compaction, or for mutual exclusion, are implemented in software by compiler-inserted code sequences. Yet, this kind of synchronization shows three major drawbacks. First, the synchronization code inflates the program code. Second, it slows down application execution. Third and finally, the synchronization code introduces strong dependencies of the compiled application code from the particular garbage collection algorithm.

Generally, garbage collectors for real-time systems trade synchronization overhead for garbage collection granularity. If incremental algorithms are to bound garbage collection pauses to milliseconds, they will suffer from prohibitive synchronization costs. At the other extreme, generational garbage collectors do not reduce worst-case pause lengths, but get by with moderate synchronization.

Most problems of implementing garbage collection for real-time systems are inherent to the sequential nature of software. In contrast, hardware has the potential to operate in parallel. Motivated by this insight, Meyer proposed a novel RISC processor architecture that provides the basis for efficient garbage collection and synchronization in hardware [5]. In a first proof-of-concept design, he realized Baker's incremental copying algorithm with extensions for incremental compaction. Thanks to synchronization in hardware, his system is able to predictably bound any garbage collection pause to less than 500 clock cycles. In addition, and in contrast to software implementations, garbage collection needs no compiler support and implies a small amount of runtime overhead only [6].

Since the collector used in this system is based on a standard copying algorithm, it must copy all live objects in every collection cycle. To address this issue, generational algorithms concentrate their effort on the region of the heap where objects are most likely to die.

In this paper, we show that it is possible to combine the fine-graindness of Meyer's initial implementation with the efficiency of a generational algorithm. Furthermore, we show that the additional runtime overhead of generational garbage collection can be reduced to near zero by realizing both the write barrier and, for the first time, the corresponding book-keeping of inter-generational pointers by means of relatively simple hardware.

The rest of this paper is organized as follows: Section 2 introduces our initial system with hardware-supported real-time garbage collection. Next, Section 3 provides the basics of generational algorithms, and Section 4 describes our novel hardware support for generational garbage collection. Section 5 presents experimental results that we obtained from our prototype. Finally, Section 6 discusses related work, and Section 7 provides a conclusion.

2. INITIAL SYSTEM

Garbage collection, although recognized as indispensable for software quality, is frequently considered cumbersome because of its negative side-effects, including runtime overhead, memory overhead, and disruptive pauses. In an ideal garbage-collected system, objects would be allocated when required and simply vanish without any overhead as soon as they are no longer needed. Motivated by this vision, Meyer proposed and realized an object-based processor for real-time embedded systems [5]. This processor completely abstracts from memory management at the assembly language level and thereby allows the realization of garbage collection as well as all garbage-collection-related synchronization mechanisms completely in hardware. To exploit this potential, the processor is supplemented by a garbage collection coprocessor that reclaims memory behind the scenes and completely in parallel to application processing (Figure 1).

This section introduces the characteristics of both main processor and garbage collection coprocessor and shows how the synchronization mechanisms for hard real-time garbage collection are efficiently realized in hardware.

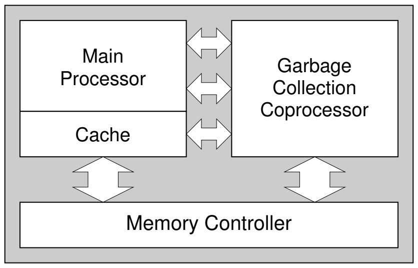

Figure 1: System overview

2.1 Main Processor

In order to perform exact garbage collection in hardware, objects and pointers must be known at the hardware level. For this reason, the main processor is based on an object-based architecture. Rather than using plain addresses, load and store instructions access memory by means of pointer/index pairs. Objects are created with a dedicated *allocate* instruction. However, there is no instruction to delete objects. The architecture relies on a hidden garbage collector that recycles memory in the background.

Pointers are identified by a strict separation of pointers and non-pointer data. The processor architecture ensures this separation by three mechanisms. First, each object is split into two dedicated areas, one for pointers, the other for non-pointers. Two object attributes π and δ describe the size of the pointer area and the data area, respectively. They are stored in an object header that is invisible at the assembly language level. Each area realizes a separate index space starting at zero (Figure 2). Second, the processor's register set is split into a pointer register set and a data register set. Third, separate instructions are provided for handling pointers and non-pointer data. Regarding load and store instructions, pointer load and pointer store instructions implicitly target the pointer area of an object, while load and store instructions for non-pointers implicitly target the data area.

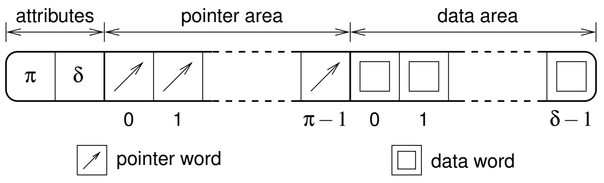

Figure 2: Object layout

In order to ensure the integrity of pointers, it is not possible to transfer the contents of a data register to a pointer register or vice-versa. Furthermore, range checking ensures that load and store instructions never violate the bounds of the respective area.

The implementation of the processor is based on a classical pipelined RISC design and extended to efficiently handle objects and their attributes (Figure 3). The main extensions are as follows: In the decode stage, the register set is split into 16 data registers and 16 pointer registers. In the execute stage, the processing units are split into an arithmetic logic unit (ALU) that performs standard data operations targeting data registers, a pointer generation unit (PGU) that performs operations targeting pointer registers, and an address generation unit (AGU) that processes pointers and indices to generate addresses for the data cache in the memory stage. For each memory access, the AGU requires the object attributes π and δ for address computation and range checking. Therefore, each pointer register is supplemented with attribute registers. After a pointer has been loaded from the data cache in the memory stage, the corresponding attributes are loaded from an attribute cache in a subsequent pipeline stage. This way, the loading of attributes is fully pipelined and, in the common case of cache hits, the load pointer instruction effectively completes in a single clock cycle.

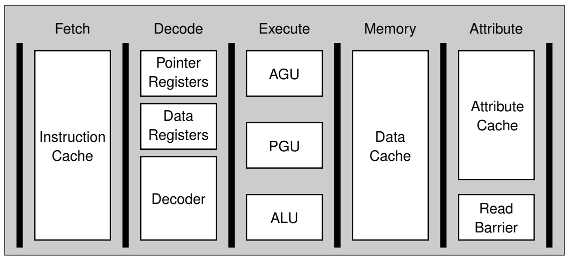

Figure 3: Main processor pipeline

2.2 Garbage Collection Coprocessor

In order to perform memory reclamation in parallel to application execution, the initial system takes advantage of a small on-chip garbage collection coprocessor that is tightly coupled to the main processor.

This coprocessor is micro-programmable and designed to efficiently trace and copy objects and to handle attributes. As garbage collection algorithms exhibit poor temporal locality, the collector does not use a cache, but is directly connected to the memory controller (Figure 1). To exploit the spatial locality of accesses, it provides burst buffers that take advantage of efficient burst memory transfers.

The proof-of-concept collector of the initial system targets hard real-time applications and is based on Baker's copying algorithm with Steele's extensions for fine-grained lazy copying [2].

2.3 Synchronization Mechanisms

Thanks to the tight coupling of processor and collector, all synchronization mechanisms for real-time garbage collection are efficiently realized in hardware.

The first synchronization mechanism ensures Baker's tospace invariant and is realized by a read barrier that inspects every pointer load. This read barrier is directly inserted into the attribute stage of the main processor pipeline (Figure 3). Both read barrier checking and read barrier fault handling are entirely realized in hardware [7].

The second mechanism ensures cache coherency. The collector inspects both the data and the attribute cache and flushes cache lines when necessary. Furthermore, a cache line locking mechanism guarantees exclusive access to objects.

Third, the collector can access the processor's pointer registers for root-set scanning. In addition, the collector is able to temporarily stop the processor to protect critical regions in the microcode.

Finally, incremental compaction is realized by dedicated circuitry in the AGU that dynamically determines whether the fromspace original or the tospace copy of an object is to be accessed.

By design, none of the synchronization mechanisms described above suspends application execution for more than a few hundreds clock cycles [6].

3. GENERATIONAL COLLECTION

Generational algorithms exploit the fact that most objects become garbage soon after they have been created [4]. For this purpose, they divide the heap into two or more generations and concentrate their effort on the youngest generation, i.e. the region of the heap where objects are most likely to die. Since the young generation is a small subset of the heap only, the time required for collecting the young generation (a minor collection) is much shorter than the time for collecting the entire heap (a major collection). This way, generational algorithms reduce the duration of the majority of collection cycles and significantly increase the efficiency of garbage collection.

In order to process the young generation independently of older generations, generational algorithms have to keep track of so-called inter-generational pointers, i.e. of pointers in old objects that refer to young objects. For this purpose, they use a write barrier to inspect every pointer store instruction whether it is about to write a pointer to a young object into an old object. If so, the collector has to consider that pointer as part of the root set for subsequent minor collections.

The realization of write barriers for generational collectors faces two challenges: First, the write barrier check itself must be implemented as efficiently as possible in order to keep the code-size and runtime overhead small. Second, the book-keeping procedure invoked on each write barrier fault must be efficient as well. In software, generational collectors have to trade runtime overhead for book-keeping granularity. Known implementations considerably differ in their granularity, ranging from Ungar's fine-grained remembered set [15] to Moon's page marking method [8] that merely records inter-generational pointers on a virtual memory page basis.

Apart from the write barrier, implementations must address the problem of duplicates in their book-keeping structures. Duplicates potentially blow up the collector's internal data structures and increase the time for root set processing of minor collections. Avoiding duplicates, however, is expensive since it usually involves search operations.

4. NOVEL HARDWARE SUPPORT

In this section, we extend the initial system with novel hardware support for generational garbage collection.

4.1 Design Considerations

As described in the previous section, synchronization for generational garbage collection depends on two basic mechanisms only: a write barrier that detects inter-generational pointers, and a book-keeping method for these pointers. Like all synchronization mechanisms in the initial system, these new mechanisms are efficiently realized in hardware.

For the book-keeping of inter-generational pointers, we use a remembered set, i.e. an array of references to old objects containing young pointers. A remembered set features a finer granularity than page or card marking techniques. Furthermore, a remembered set does not depend on virtual memory or structures like bitmaps or indirection tables.

To prevent duplicates in the remembered set, we add a supplementary one-bit tag (the so-called *r-bit*) to each object. This bit is stored together with the attributes in each object's header and will be set as soon as the corresponding object is included in the remembered set. Since both the main processor and the collector have access to object attributes, they can easily determine whether an object is already in the remembered set or not.

The rest of this section describes the implementation of the remembered set, the write barrier, and extensions to the coprocessor.

4.2 Implementation of the Remembered Set

The remembered set is stored in main memory, apart from the heap, as a single array of pointers.

Both the write barrier handler and the garbage collection coprocessor access the remembered set in a sequential way. To take advantage of the resulting spatial locality, processor and collector, instead of directly accessing the remembered set, use small on-chip FIFO buffers (Figure 4). These buffers connect to the memory controller and access memory by means of efficient burst transfers. A simple control circuitry monitors the level of both buffers and initiates burst transfers when necessary. Thanks to these buffers, all operations in conjunction with the remembered set can be performed in a single clock cycle, provided that the read buffer is not full, or the write buffer is not empty, respectively. Because accesses from the write barrier handler and the coprocessor are rarely clustered, these two conditions are almost always met.

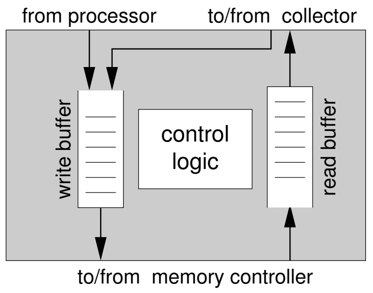

Figure 4: Remembered set buffers

4.3 The Write Barrier

The write barrier detects inter-generational pointers generated by the processor and inserts them into the remembered set. Accordingly, the implementation of the write barrier can be split into two parts: detection and handling.

4.3.1 Detection

The following conditions must hold to get a write barrier fault:

1. the instruction is a pointer store instruction;
2. the written pointer refers to a young object;
3. the stored-into object belongs to the old generation;
4. the stored-into object isn't already in the remembered set.

When processing a pointer store instruction, all information required to check for these four conditions is available in the processor's memory stage. Thanks to this, the write barrier check can be realized by simple logic and a few comparators, and can occur completely in parallel to the pointer store instruction without introducing any delay (Figure 5).

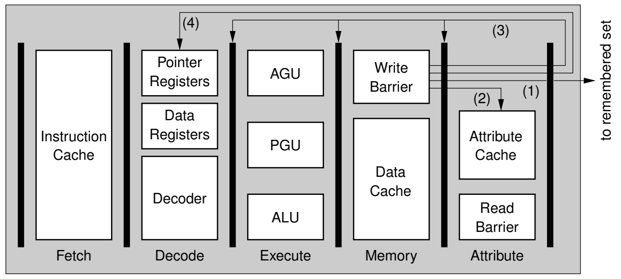

Figure 5: Integration of the write barrier in the main processor pipeline

4.3.2 Handling

Write barrier handling involves two steps: First, the pointer to the old object must be written to the remembered set. Second, the *r-bit* in this old object must be set.

Pointers are added to the remembered set in parallel to instruction completion, i.e. when faulting pointer store instructions leave the attribute stage. (Figure 5, arrow 1).

Setting the *r-bit* of an object requires its attributes to be updated. The attributes of single object, however, may be stored in several locations within the pipeline.

The first of these locations is the attribute cache. As pointer store instructions do not access the attribute cache, the write barrier handling logic is free to set the *r-bit* in the cache without causing a structural conflict (Figure 5, arrow 2). In the common case of an attribute cache hit, this access introduces no additional delay.

Second, object attributes are stored in several registers within the processor pipeline, i.e. with each pointer in the pointer register set and with each pointer in the pipeline registers between the decode stage and the attribute stage. Since any of these registers can hold a pointer to the stored-into object, all of them must potentially be updated. This takes place while the instruction is committed in the write-back stage (Figure 5, arrows 3 and 4). The update is performed by a comparator and some simple logic associated with each register.

The write barrier circuitry performs both write barrier detection and handling in parallel with instruction execution. In the common case of cache hits and available space in the remembered set write buffer, the write barrier involves no performance penalty whatsoever.

4.4 Extensions to the Coprocessor

In order to enable generational collection, only a couple of supplementary micro-operations need to be added to the coprocessor. These micro-operations provide access to the remembered set and allow the handling of the *r-bit*. They neither affect the features nor the performance of the collector and preserve the bound on maximum synchronization pause times of the initial system.

The extended algorithm realizes two generations. Apart from the write barrier and the remembered set, the algorithm is implemented in microcode without further hardware support. The size of the two generations is dynamic and adapts to the behavior of the running program. Objects are promoted to the old generation during major collection cycles only. In doing so, all objects are promoted, except those that have been allocated since the last collection cycle. Major collections are scheduled when the young generation fills beyond a certain configurable threshold, or when the percentage of objects surviving a minor collection exceeds another configurable threshold.

5. EXPERIMENTAL RESULTS

To demonstrate the feasibility of our approach, we extended a prototypical implementation of the initial system by our hardware support for generational garbage collection.

The main processor as well as the garbage collection coprocessor are described in VHDL and synthesized for an Altera Stratix-II-FPGA (EP2S60, [1]).

The main processor is realized as a 3-way multiple-issue explicitly parallel RISC, with 8K instruction cache, 8K data cache and 2K attribute cache. The garbage collection coprocessor can execute the initial algorithm and the extended generational algorithm. In either case, the algorithm gets by with 256 microcode words with 96 bit each. The coprocessor including the microcode memory uses approximately 20% of the chip area. Our extensions for generation garbage collection occupy less than 2%.

To realize a platform for measurements, we supplemented the main processor and the coprocessor with standard DDR-SDRAM and peripheral devices such as an Ethernet interface and a terminal interface. The entire prototype is synchronously operated at 25 MHz.

An integrated measurement framework allows to monitor up to 32 internal processor signals in every clock cycle. With an on-board gigabit ethernet interface, the measurement data is transmitted to a measurement computer, written to several hard disks in parallel, and analyzed offline.

For evaluation, we ran several statically compiled Java applications with both collectors. For each application, we chose the smallest heap size required for real-time behavior (i.e. without mutator starvation).

We first measured the fraction of time with garbage collector activities relative to application execution time (Figure 6). The corresponding results reflect the efficiency of garbage collection. On average over all benchmarks, collector activity is reduced by a factor of 5. Of our benchmarks, *javac* represents the worst case for a generational collector: It compiles Java classes in a sequential manner and independently of each other. In cases like this, the generational hypothesis does not apply, and almost all live objects must be collected at every collection cycle. Even in this case, however, the generational collector does not perform worse than the initial collector thanks to the efficient implementation of the generational extensions. In contrast to *javac*, *jlisp* (a Lisp interpreter written in Java) is ideal for a generational copying collector. It shows a very high allocation rate, and most allocated objects die relatively

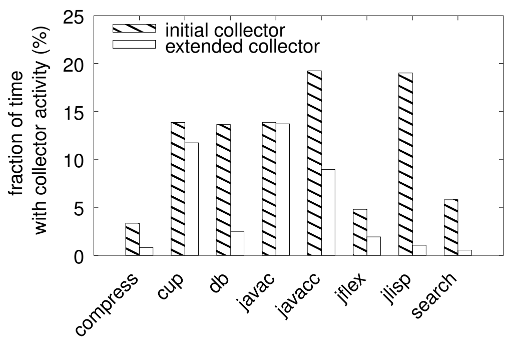

Figure 6: Collector activity (% of application execution time)

young. In this case, the activity of the collector is reduced by a factor of 12.

Next, we analyzed the efficiency of our circuitry for duplicate prevention. For this purpose, we measured (Table 1):

- the number of executed pointer store instructions;
- the number of write barrier faults without duplicate prevention;
- the number of write barrier faults with duplicate prevention.

program	number of		
	executed pointer store instructions	write barrier faults	
		without duplicate prevention	with duplicate prevention
compress	5,541,902	4	2
cup	35,835,933	3	1
db	178,385,747	57,289	3
javac	29,173,589	261,880	23,441
javacc	14,576,972	1,519	732
jflex	5,661,668	24,886	6
jlisp	25,878,020	363,108	74
search	4,802,776	10	1

Table 1: Number of executed pointer store instructions and write barrier faults

First of all, it is interesting to note that only a small fraction of pointer store instructions actually generates inter-generational-pointers, i.e. stores young pointers into old objects.

Second, duplicate prevention considerably reduces the number of write barrier faults (factor 2 to 19000). Thanks to the *r-bit* and the corresponding hardware logic, this reduction is accomplished without any runtime overhead. As a benefit, the maximum size of the remembered set is surprisingly small (Table 2).

program	maximum size (word)	average size (word)
compress	2	1.1
cup	4	2.4
db	3	2.1
javac	19664	7803.5
javacc	644	75.8
jflex	13	6.2
jlisp	13	6.1
search	0	0

Table 2: Size of the remembered set at the beginning of minor collection cycles

Finally, we inspected the memory traffic at the memory controller and measured, for each application, the memory traffic overhead caused by the collector relative to memory traffic of the corresponding application without garbage collection, i.e. with virtually infinite memory (Figure 7). Considering the average of all benchmarks, the initial collector generates overheads from 18% to 94%, while the extended collector reduces them to 1% to 28%. On average, our generational extensions reduce the overhead by a factor of 4. Again, *javac* represents the worst-case, and *jlisp* the best-case (from 43% down to 1%).

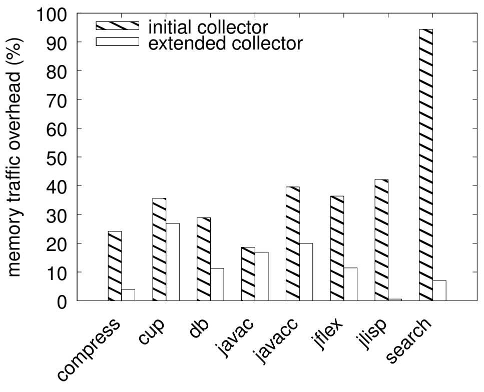

Figure 7: Memory traffic overhead of garbage collection

6. RELATED WORK

The system presented in this paper combines and leverages incremental garbage collection, generational garbage collection, and hardware support for both. In this section, we compare collectors introduced by previous work with respect to hardware support for generational garbage collection and real-time capabilities.

Hardware support for garbage collection appeared in the 1980s in many language-directed architectures. The best known systems are SOAR (Smalltalk On A RISC, [12]), SPUR (Symbolics Processing Using RISC, [14]), and the Symbolics 3600 [8].

SOAR and SPUR have been developed at the University of Berkeley and aim at improving the performance of Smalltalk (SOAR) and Lisp (SPUR). Both projects offer hardware support for generational garbage collection and rely on an object-based RISC architecture that uses tagged memory to distinguish pointers from non-pointer data. They segregate objects into several generations that are collected separately by a non-incremental copying algorithm in software [15]. A hardware-supported write barrier checks for the creation of inter-generational pointers and triggers an exception in case of a fault. The write barrier fault handling is realized by a software routine that maintains a remembered set.

The Symbolics 3600 is a special-purpose Lisp machine aimed at providing high performance for artificial intelligence applications. It relies on an object-based CISC architecture with tagged memory and offers hardware support for generational garbage collection. Like SOAR and SPUR, the Symbolics segregates objects into several generations, but collects them using an incremental garbage collection algorithm instead. Read barrier checks as well as write barrier checks are realized in microcode. In case of barrier faults, exceptions are triggered and handled in software. In order to keep track of inter-generational pointers, Symbolics uses a page marking technique. In doing so, Symbolics stores the address of each virtual memory page containing inter-generational pointers into a dedicated table. This table is physically implemented as two separate tables: the first is provided for swapped-in pages, realized in a special-purpose dedicated memory, and maintained by means of hardware support in order to minimize runtime costs. The second is provided for swapped-out pages, stored in non-pageable memory, and entirely maintained in software.

During the 1990s, object-based architectures have been superseded by off-the-shelf processors, mainly for economical reasons. Following this trend, researchers proposed a number of active memory modules for standard processors. The best known of these modules is the so-called garbage-collected memory module (GCMM) [13] designed for hard real-time applications written in C or C++. Apart from the actual memory devices, the GCMM contains a standard microprocessor and a number of custom circuits, including an arbiter and two elaborate CAM-like devices. Unfortunately, however, the hardware costs of the GCMM are extremely high and in particular prohibitive for embedded systems. Furthermore, this module must separately process every single word and cannot use burst memory transfers. As a result, the module's data throughput is considerably inferior to that of modern memory devices like SDRAM. Finally, the loose coupling between main processor and memory module renders synchronization between application and collector particularly expensive. Simulations show that the synchronization overhead amounts to up to 100% [9].

Recently, together with the success of Java, a number of architectures emerged with hardware support for native Java bytecode execution. Surprisingly, however, they offer little to no support for garbage collection. Only Komodo [11] targets real-time applications and uses a combination of Dijkstra's algorithm and Baker's treadmill [3]. It realizes a write barrier in microcode, while the collection algorithm is entirely executed by a dedicated software thread. The main drawback of Komodo's approach is its high runtime overhead: the write barrier accounts for up to 60% of overhead, while other synchronization mechanisms produce an additional 10-15% [10].

In summary, among all architectures presented in this section, only the GCMM provides hardware support for real-time garbage collection, but suffers from prohibitive hardware costs and high runtime overhead. In contrast, all architectures with generational garbage collection (SOAR, SPUR, Symbolics) focus on interactive systems and garbage collection efficiency, but do not address real-time garbage collection at all. They realize write barrier checks in hardware, but handle write barrier faults in software.

7. CONCLUSIONS

In this paper, we introduce a system that combines hard real-time garbage collection with the efficiency of generational techniques. Thanks to write barrier detection and fault handling in hardware, the benefits of generational garbage collection, for the first time, come without any negative side-effects.

We built a prototype and ran various Java programs. Experimental results show the benefits of our approach: with reasonable hardware effort, the amount of garbage collector activities as well as the memory traffic generated by the collector are significantly reduced without degrading worst-case performance. Because of the resulting reduction in power consumption, the proposed extensions are of particular interest for power-aware mobile devices.

8. REFERENCES

[1] Altera. STRATIX II device family data sheet, 2006.

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

[3] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *ACM Sigplan Notices*, 27(3), 1992.

[4] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[5] M. Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.

[6] M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hong Kong, Aug. 2005.

[7] M. Meyer. A true hardware read barrier. In *ISMM'06 Proceedings of the Fifth International Symposium on Memory Management*, Ottawa, June 2006.

[8] D. A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, Boston, MA, June 1985.

[9] K. D. Nilsen and W. J. Schmidt. A high-performance hardware-assisted real time garbage collection system. *Journal of Programming Languages*, 2(1), 1994.

[10] M. Pfeffer. *Ein echtzeitfähiges Java-System für einen mehrfädigen Java-Mikrocontroller*. PhD thesis, University of Augsburg, Germany, Feb. 2004.

[11] M. Pfeffer, T. Ungerer, S. Fuhrmann, J. Kreuzinger, and U. Brinkschulte. Real-time garbage collection for a multithreaded Java microcontroller. *Real-Time Systems*, 26(1):89–106, 2004.

[12] A. D. Samples, D. M. Ungar, and P. Hilfinger. SOAR: Smalltalk without bytecodes. In *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 21(11), pages 107–118, Oct. 1986.

[13] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–85, Oct. 1994.

[14] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. In *ISCA '84: Proceedings of the 11th International Symposium on Computer Architecture*, pages 188–197, New York, NY, USA, June 1984.

[15] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 1984.

[16] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan. 1994.