Universität Stuttgart

**INSTITUT FÜR
KOMMUNIKATIONSNETZE
UND RECHNERSYSTEME**
Prof. Dr.-Ing. Andreas Kirstädter

Copyright Notice

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47, D-70569 Stuttgart, Germany
Phone: ++49-711-685-68026, Fax: ++49-711-685-67983
Email: mail@ikr.uni-stuttgart.de, http://www.ikr.uni-stuttgart.de

An On-Chip Garbage Collection Coprocessor for Embedded Real-Time Systems

Matthias Meyer

Institute of Communication Networks and Computer Engineering
University of Stuttgart, Germany
meyer@ikr.uni-stuttgart.de

Abstract

Garbage collection considerably increases programmer productivity and software quality. However, it is difficult to implement garbage collection both efficiently and suitably for real-time systems. Today, garbage collection is exclusively realized in software and either fails to guarantee a small upper bound for pause times or suffers from considerable synchronization overhead.

In this paper, we present the design and implementation of an on-chip garbage collection coprocessor that closely cooperates with the main processor. The benefits of this configuration include low garbage collection overhead, low-cost synchronization of collector and application programs, and hard real-time capabilities.

We successfully realized the garbage collection coprocessor along with a pipelined RISC processor on a single FPGA. Performance measurements on the prototype show that the longest pauses caused by the garbage collector are less than 500 clock cycles and that the total runtime overhead is as little as a few percent if the application is provided with a reasonable amount of memory headroom.

1. Introduction

Software quality is the most fundamental requirement for security- and safety-critical embedded systems. While a faulting desktop application might be annoying and ruin hours worth of work, a single error in an embedded system can have disastrous consequences such as causing considerable damage or even harming people. Facing the constantly growing complexity of embedded systems, it becomes more and more important to support correct and robust software for these sensitive systems by appropriate design methods and architectures.

A widely accepted method to control software complexity and to increase software quality is automatic dynamic memory management, also referred to as garbage collection. Unfortunately, it is difficult to implement garbage collection for real-time systems. Because of high overhead and unpredictable pauses, most embedded system designers still consider garbage collection an unaffordable luxury. Consequently, they have to resort to manual memory management with the well-known problems: Freeing memory too early causes "dangling references", while freeing memory too late

or not at all gives rise to memory leaks. But worst of all, manual memory reclamation requires the programmer's global view of a software system and contradicts modularization.

Most garbage collection algorithms trace memory starting with a set of roots, usually consisting of processor registers and the program stack [4]. Objects that cannot be reached from the root set are safely reclaimed. While "stop-the-world" implementations suspend all application processing for the entire duration of a garbage collection cycle, incremental garbage collectors allow the application (in this context referred to as the mutator) to proceed while garbage collection is performed. In return, it becomes necessary to synchronize the application with the garbage collector to ensure the integrity of the heap. In software, this synchronization has to be realized by a compiler that inserts code sequences into the generated machine code. The most common type of synchronization sequences are read and write barriers that need to be inserted after each pointer load or before each pointer store, respectively. Regrettably, synchronization in software results in considerable code blow-up and runtime overhead [10].

Real-time applications require a small upper bound on any pause the garbage collector might cause. Garbage collection in software, however, faces an inevitable trade-off between the granularity of garbage collection (i.e. the maximum pause length) and the code-size and runtime overhead caused by synchronization. To prevent this overhead from becoming unjustifiably high, software collectors have to rely on indivisible operations such as processing an entire object or the complete root set [4]. Since the duration of these operations is potentially unlimited, these software collectors are not suited for hard real-time environments.

In this paper, we present a specialized garbage collection coprocessor for real-time embedded systems. It tightly cooperates with the main processor, thereby allowing for fine-grained synchronization with very little runtime overhead and without any code-size overhead. Most importantly, the system ensures by design that the duration of garbage collection pauses never exceeds a small constant in the order of a couple of hundred clock cycles.

This paper is organized as follows. Section 2 discusses representative related work. Section 3 gives an overview of our system and motivates the chosen configuration. Section 4 outlines the special architecture of the main processor and

our implementation thereof. Section 5 describes the garbage collector in detail, including the used algorithm, the coprocessor's architecture and the hardware implementation of all synchronization mechanisms. Section 6 presents experimental results from our prototype, and Section 7 discusses the contributions of this work. Finally, Section 8 provides a conclusion.

2. Related work

Hardware support for garbage collection has been introduced by language-directed architectures in the 1980s. Examples include processors specialized for LISP (e.g. Symbolics [6]) or Smalltalk (e.g. Mushroom [9]). All these architectures support read or write barriers in hardware and primarily focus on improving a system's throughput and interactive response rather than guaranteeing worst-case latencies.

The best known hardware-supported garbage collector for real-time applications is the garbage-collected memory module proposed by Nilsen and Schmidt [7]. The module connects to a standard microprocessor and accommodates the actual memory devices, a private microprocessor, and a number of custom devices, including two elaborate CAM-like devices. With respect to real-time performance, the authors report worst-case latencies of typically 16,000 clock cycles at the beginning of a garbage collection pass. Unfortunately, the hardware costs for the memory module are prohibitive, particularly for most embedded applications. Furthermore, the module's data throughput is considerably inferior to that of standard memory, especially when compared with modern, burst-oriented memory devices. Lastly, a significant overhead is caused by communicating the location of pointers to the module, most notably regarding stack operations.

Siebert [8] proposes a garbage collector for hard real-time environments without any hardware support. To avoid pauses that depend on the size of objects and to fight fragmentation in a non-moving collector, he suggests building objects from constant-sized 32-byte blocks. As a consequence, a program has to follow linked lists or tree structures for every single object and array access. In addition to this overhead, frequent compiler-inserted code sequences such as write barriers and synchronization points further degrade application performance and inflate the program code. Finally, the constant block size introduces a considerable amount of internal fragmentation.

Bacon et al. [2] have presented the most recent software-only implementation of a real-time garbage collector. They achieve a low memory space overhead (factor 1.6–2.5) by using a mostly non-copying algorithm. Time-based scheduling guarantees a CPU utilization of at least 45% for the mutator while the garbage collector is active. The overall runtime overhead of the garbage collector amounts to approx. 30–40%. A highly optimized software read barrier accounts for an additional 4% of runtime overhead. To bound the duration of the collector's atomic actions, the compiler splits large arrays into so-called arraylets. Incremental stack processing, however, is an open issue. Apart from the stack problem, the duration of pauses is limited to 6 ms (3,000,000 CPU cycles at 500 MHz).

3. System overview

The key idea of our approach is to extend a processor by a small, low-cost coprocessor dedicated to and specialized for garbage collection, and to integrate them both onto a single device. Figure 1 shows the configuration of the two processors. Each is provided with separate ports to an on-chip memory controller. At its external interface, the device behaves like a standard uniprocessor and interfaces to standard memory devices such as SDRAM.
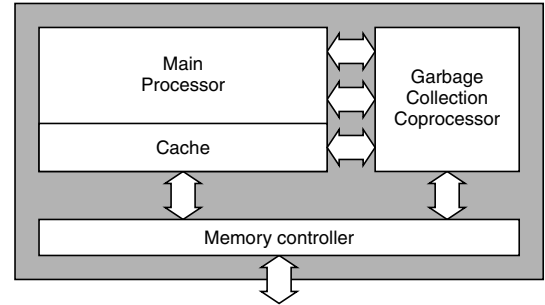

Figure 1. System overview

A major problem with software garbage collectors is their devastating effect on cache locality. During a single garbage collection cycle, they usually examine the entire heap, and, in doing so, repeatedly displace the entire contents of the cache. To resolve this issue, our coprocessor directly connects to the memory controller rather than accessing memory through the main processor's cache. In this way, the cache remains largely unaffected by the garbage collector's activities. To ensure cache coherency, the coprocessor inspects and, if necessary, flushes single cache lines by means of a dedicated cache port that resembles the snoop port of a standard cache. At the end of a garbage collection cycle, the coprocessor invalidates all cache lines that contain dead objects and thereby efficiently eliminates unnecessary memory traffic they would otherwise cause.

Because of the poor *temporal* locality of garbage collection algorithms, garbage collection itself will not profit from a cache, and consequently we realized the coprocessor without one. However, most garbage collection activities such as scanning and copying objects show a fair amount of *spacial* locality. To exploit this property, the coprocessor internally buffers a number of subsequent memory locations and, similar to the main processor's cache controller, takes advantage of efficient burst memory transfers.

Apart from cache coherency, the main processor and the garbage collection coprocessor tightly cooperate in many ways. Section 5 covers these synchronization mechanisms in detail. But first, the next section describes the main processor's architecture and the particular implementation we have realized as the basis for our system.

4. Main processor

4.1. Processor architecture

Hardware support for garbage collection requires the knowledge of pointers and objects at the architectural level. How-
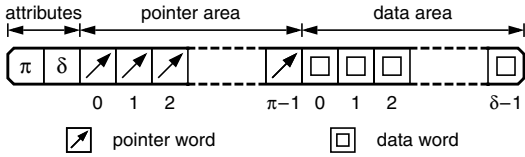
Figure 2. Object layout

Figure 3. Stack object

ever, standard processors are neither able to distinguish pointers from non-pointers, nor are they able to identify objects in memory. Therefore, the main processor in our system is realized according to a novel RISC architecture that was first presented in [5]. Instead of using tag bits for identifying pointers, this architecture strictly separates pointers from ordinary non-pointer data. At any time, it maintains the invariant that (1) every pointer can be exactly identified, and (2) every pointer value is either null or uniquely associated with an existing object *(system invariant)*.

To realize the separation of pointers from non-pointers in the register set, the architecture provides a data register set and a pointer register set. Data registers are used as general-purpose registers whereas pointer registers are exclusively for referring to objects in memory. It is not possible to copy values from data registers to pointer registers or vice versa.

Similarly, objects in memory are composed of a separate pointer area and a separate data area (Figure 2). The size of the pointer area is referred to as the object's π-*attribute* and the size of the data area as the object's δ-*attribute*. In this way, an object offers two parallel index spaces, each starting with zero. Load and store instructions for pointers implicitly target the pointer area while load and store instructions for non-pointer data implicitly target the data area. Pointer integrity is enforced by range checks. The processor stores the attributes inside an object header that is entirely invisible at the assembly language level. An *allocate* instruction creates new objects and initializes the object's pointer area with null pointers. To guarantee hard real-time response times, allocate instructions may be temporarily suspended by interrupts.

As mentioned before, incremental stack processing represents a serious issue for real-time garbage collection. To solve this problem, the processor provides a special *stack object* of dynamic size. Like ordinary objects, the stack object is incrementally processed by the garbage collector. A special pointer register holds the reference to the stack object,
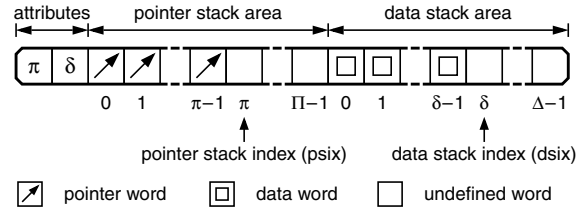
and two stack indices in special data registers describe the actual size of the two areas (Figure 3).

To realize more than one stack for multithreaded environments, the processor supports *static objects*. They are managed by the operating system apart from the heap and never moved by a collector. During a context switch, the processor writes the current stack indices into the attribute header of the corresponding static object. This way, the garbage collector exactly knows which pointers inside a currently inactive stack are actually valid.

In addition to ordinary and static objects, the architecture defines two further kinds of objects. *Constant objects* provide a safe and uniform mechanism to access constant data structures that are stored as part of the program code. *Uninitialized objects* are created when the initialization of objects during allocation is suspended due to interrupts. Uninitialized objects and static objects are restricted to supervisor mode.

4.2. Processor implementation

The main processor of our system is a 32-bit implementation of the architecture outlined above. Memory is byte-addressable in order to provide for byte and half-word accesses within the data area, and so the attributes π and δ actually describe the number of bytes in the respective area.

The processor is statically scheduled and issues up to three instructions in a clock cycle. Despite its features for pointer identification and object protection, the pipeline of the processor (Figure 4) is very similar to that of a standard RISC and can thus be implemented in an efficient way. Compared to a standard RISC, the processor pipeline shows the following three enhancements.

First, the register set within the decode stage is split into 16 data registers and 16 pointer registers. Each pointer register is supplemented by two attribute registers. Whenever a
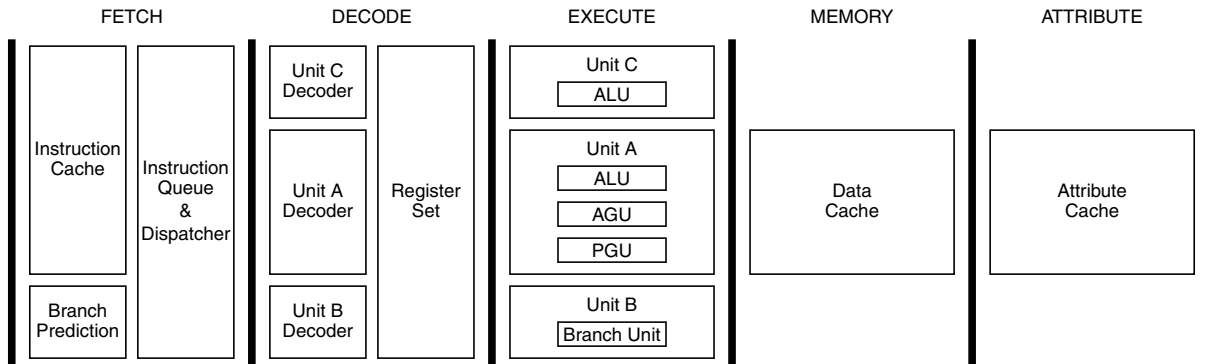

Figure 4. Processor pipeline

3

pointer register contains a non-null value, the corresponding attribute registers hold the attributes of the object the pointer register refers to.

Second, the execution stage contains different execution units for different types of target operands. While the two ALUs are responsible for instructions targeting data registers, the PGU (Pointer Generation Unit) takes care of instructions targeting pointer registers (e.g. *allocate*, *copy pointer*), and the AGU (Address Generation Unit) performs range checks and generates addresses for the cache in the subsequent memory stage. Since this cache is identical to the data cache in a standard processor, it is designated as "data cache" even though it contains pointers and non-pointer data. It is realized as an ordinary two-way set-associative copy-back cache with a cache line size of 8 words and a capacity of 8 KBytes.

Third and finally, the pipeline exhibits an additional attribute stage after the usual memory stage. Whenever a non-null pointer is loaded from memory, this stage loads the attributes of the corresponding object. It features an attribute cache in order to allow for attribute accesses without performance penalty in the common case. The attribute cache has 256 entries and, like the data cache, is realized as a two-way set-associative copy-back cache.

5. Garbage Collector

5.1. Garbage Collection Algorithm

Basically, the garbage collection coprocessor realizes a Baker-style copying collector [1] with several extensions. Baker's algorithm is extremely simple and elegant and therefore well suited for a hardware implementation. Since it is a copying algorithm, it furthermore compacts the heap and considerably eases allocation. Finally, copying collectors are usually more efficient than their mark-sweep counterparts.

5.1.1. Basic algorithm. For illustration purposes, object states during garbage collection are often described by Dijkstra's tricolor abstraction [3]. In this abstraction, *black* indicates that the collector has finished with an object for the current garbage collection cycle, *gray* indicates that the collector has not finished with the object or, for some reason, has to visit the object again, and *white* indicates that the object has not been visited by the collector. At the end of a garbage collection cycle, white objects are reclaimed.

Copying collectors like Baker's divide the heap into two areas called semispaces. During a garbage collection cycle, all objects that are reachable from a set of roots are copied from one semispace *(fromspace)* to the other semispace *(tospace)*, thereby inherently compacting the heap. At the beginning of a cycle, the collector flips the roles of fromspace and tospace and initializes two pointers called *scan* and *free* to point to the bottom of tospace. Next, the collector evacuates all objects referenced by the root set from fromspace to tospace (Figure 5a, assuming that objects A and B are referenced by the root set).

In order to bound the time needed to evacuate an object, the collector does not actually copy an object during evacuation. Instead, it merely reserves space in tospace and doubly links the empty object slot to the fromspace original. For this
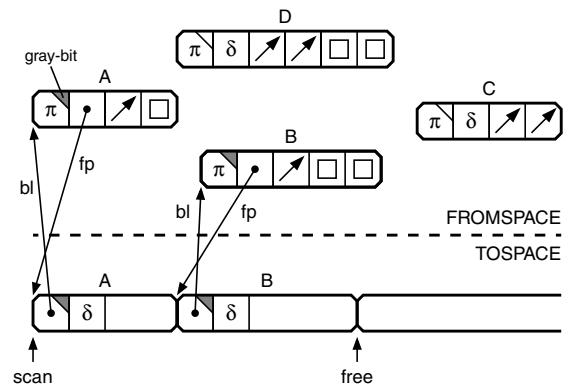

Figure 5a. Basic algorithm (1)

purpose, the collector first saves the object's δ-attribute to tospace and then overwrites it with a forwarding pointer (fp) to the tospace copy. In tospace, it initializes the field for the π-attribute with a backlink (bl) to the fromspace original (Figure 5a). In both the word holding the π-attribute in fromspace and the word holding the backlink in tospace, the collector sets a reserved bit called *gray-bit*. Finally, it updates *free* to indicate the next free location in tospace.

At each iteration of the main loop, the garbage collector follows the backlink in the attribute header currently pointed to by *scan* to find the corresponding source object in fromspace. Then, it scans the pointer area of the fromspace object. If a pointer refers to an object whose gray-bit has not yet been set, the collector evacuates the corresponding object. Otherwise, the object has already been evacuated and the collector reads the forwarding pointer. In either case, the collector writes the resulting tospace pointer to the tospace copy. Subsequently, it copies the data area to tospace. Finally, the collector "blackens" the tospace object by replacing the backlink with the π-attribute and by clearing the gray-bit and advances *scan* to point to the next object header (Figure 5b). The gray-bit of the fromspace original remains set. The algorithm terminates when *scan* catches up with *free*.

Baker's algorithm is incremental and allows the mutator to proceed during a garbage collection cycle. If, however, both the mutator and the collector are allowed to access the heap without restriction, problems may arise if the mutator writes a pointer to a white object into a black object. If the
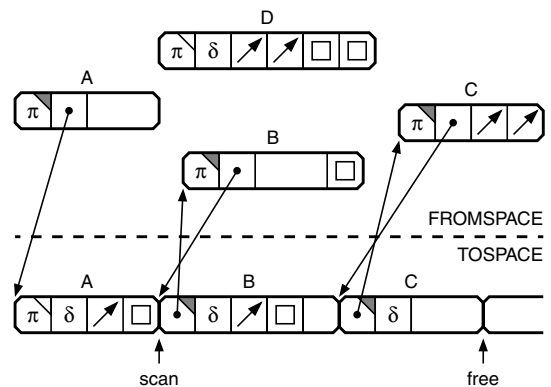

Figure 5b. Basic algorithm (2)

original pointer to the white object is destroyed and no further pointer to the white object exists, the object will be illegally discarded at the end of the garbage collection cycle. For this reason, Baker's algorithm erects a read barrier between the mutator and the heap to protect the garbage collector. This barrier examines every pointer load to ensure that the mutator never sees a white object. Whenever the mutator is about to access a pointer to an object in fromspace, the read barrier immediately evacuates the object, or, if the object has already been evacuated, reads the forwarding pointer.

A software garbage collector has to rely on the compiler to insert code sequences in order to realize this barrier. In contrast, the main processor in our system efficiently realizes the read barrier in hardware. Consequently, there is no need to insert any code sequences, and the compiler as well as the compiled code are entirely independent of the particular type or realization of a garbage collector.

5.1.2. Special objects. The algorithm described so far only works with ordinary objects created by successfully completed allocate instructions. During garbage collection, the collector scans these objects for pointers and moves them for compaction. To add support for constant, static, and uninitialized objects, we extended the algorithm as follows.

Constant objects are neither scanned for pointers nor moved and can be ignored by the collector. During the scan phase, the collector handles pointers to constant objects like null pointers and simply copies them to the tospace object.

Static objects are never moved, but they have to be scanned for pointers. One approach to handle static objects is to treat them as a part of the root set. In this case, the operating system has to inform the collector about the set of static objects. Although this approach is feasible, it unnecessarily widens the interface between operating system and garbage collector. For this reason, we propose a different solution.

During each garbage collection cycle, the collector builds a singly-linked list of handles for static objects. Whenever the collector encounters a pointer to a static object, it appends a handle for this static object to the list. To avoid duplicates without searching, it sets a *visited-bit* in the π-attribute of the static object. It then writes the handle to the position pointed to by *free*. When *scan* eventually reaches a handle, the collector enters a special loop that scans and updates the object's pointers without actually moving the object.

Uninitialized objects, finally, have to be moved, but must not be scanned. The processor tags uninitialized objects with an *uninitialized-bit* in the π-attribute. When the garbage collector evacuates an uninitialized object, an uninitialized-bit in the backlink entry is set as well, and when *scan* reaches such a backlink, it enters a special loop that ignores the values in the fromspace pointer area and writes null pointers to the tospace pointer area.

5.1.3. Scheduling garbage collection. Baker's original algorithm interleaves the mutator with the garbage collector on a single processor and does a bit of garbage collection during every allocation. This procedure is usually referred to as work-based scheduling. In contrast to Baker, we use a separate coprocessor for garbage collection, and the collector and the mutator are actually active at the same time. Thanks to the parallelism gained by the coprocessor, our garbage collector, similar to time-based scheduling, does not unevenly slow down the mutator during allocations.

Whenever the amount of available memory falls below an adjustable threshold, the coprocessor starts a new garbage collection cycle. Setting this threshold to zero results in non-concurrent and non-incremental "stop-the-world" behavior. Setting this threshold close to the semispace size will keep the garbage collector running continuously. For hard real-time performance, the garbage collector requires some memory headroom, and the threshold parameter has to be adjusted so that the collector finishes with a garbage collection cycle before the mutator *starves*, i.e. runs out of memory.

5.2. Garbage Collection Coprocessor

The structure of the garbage collection coprocessor is shown in Figure 6. Compared to a standard processor, it is rather simple and can be implemented at little cost. It is composed of a microcoded control unit, a register file, and a number of execution units. Two separate load/store units (LSUs) are responsible for accessing memory. While the burst LSU efficiently transfers the contents of objects between the collector and memory, the attribute LSU loads and stores individual double words containing attributes. Two arithmetic units execute register-to-register operations. The microcoded control unit operates all execution units in parallel.

The register file of the coprocessor contains 32 registers, including general purpose registers for microprogram variables and intermediate results, memory interface registers for interfacing the burst LSU and the attribute LSU, and some special registers for accessing the pointer registers and the stack indices of the main processor. In addition, the main processor's system registers for heap configuration are directly mapped to the coprocessor's register file.

The burst LSU contains two burst registers. Each burst register is eight words wide and covers the address range of a cache line in the main processor's data cache. To read from memory, the microprogram first loads the source burst register by means of an efficient burst transfer. Then, it subsequently processes the loaded words. To write to memory, the microprogram first assembles a burst in the target burst register. Then, an explicit microoperation triggers a burst transfer to store the contents of the target burst register to memory.

For each burst register, the unit maintains a microprogrammable lock that prevents the processor from accessing memory locations that are currently covered by the burst reg-
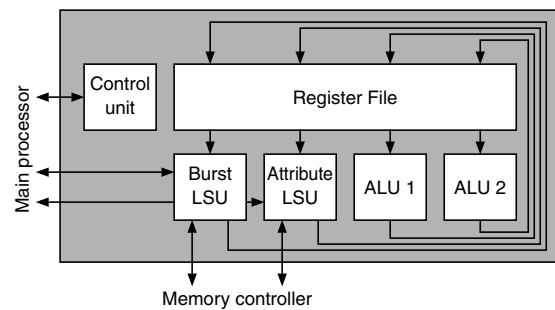

Figure 6. Garbage collection coprocessor

isters. Before the burst LSU locks a cache line, it inspects the main processor's data cache. In case the requested address range is contained in the cache, it inserts an "artificial" flush instruction into the main processor pipeline. This artificial flush writes any modified data back to memory and invalidates the corresponding cache line.

The attribute LSU unit loads and stores individual double words and maintains a lock that prevents the processor from accessing the attributes at the locked address. An explicit microoperation inspects the main processor's attribute cache, flushes the corresponding attribute cache line if necessary, and locks the corresponding attributes.

All units inside the garbage collection coprocessor are controlled by 80-bit microcode words that are stored inside a 256 x 80 bit on-chip microcode memory. Consequently, the coprocessor does not require any memory bandwidth for instruction fetching. The control unit itself supports conditional branch operations and one-level subprogram calls, is able to entirely stop the main processor's pipeline, and allows the read barrier to interrupt the microprogram at synchronization points that are marked by explicit microoperations.

5.3. Synchronization

Synchronization between garbage collector and main processor is supported by relatively simple hardware extensions to the processor pipeline. These extensions are the key for the efficient implementation of all the synchronization mechanisms that are required for concurrent garbage collection without unbounded pauses.

Exclusive access to objects is assured by the source, target, and attribute locks described in the last subsection. They are realized by means of simple comparators in the memory and the attribute stages. When the processor tries to access a locked memory location, the corresponding instruction is aborted by the processor's standard exception handling mechanism. The only difference is that instead of invoking an exception handler, the processor restarts the aborted instruction as soon as the garbage collector signals that the corresponding lock has been released.

Similarly, the hardware read barrier is implemented by the use of two comparators in the attribute stage that check whether a fromspace pointer has been loaded. In this case, the respective instruction is suspended and the garbage collector's read barrier interrupt is triggered.

Before the garbage collector reads or writes a memory location, it has to make sure that the information is not contained in one of the processor's caches. This is facilitated without any runtime overhead by extending the data and the attribute caches with a snoop port for the address tags and the valid bits. In case the garbage collector detects that the required data is contained in the corresponding cache, it halts the main processor's instruction fetch stage for a single clock cycle and inserts an artificial flush instruction into the pipeline. If necessary, this instruction writes back any dirty data and invalidates the respective cache line.

To allow for concurrent object copying without the need to lock entire objects, the main processor has to decide whether the fromspace original or the tospace copy of an object is to be accessed. For this purpose, a backlink entry is added to every pointer register and to every attribute cache line. When the attributes of a gray object are loaded from memory, the corresponding backlink is loaded as well. If the processor is about to access a field inside a gray object, the AGU checks whether the field's address based on the tospace pointer lies beyond the current target address of the garbage collector. If so, the processor uses the fromspace address based on the backlink instead of the tospace address.

The garbage collector is able to stop the main processor by halting the instruction fetch stage and by inserting NOPs into the pipeline. In this state, the collector can read and update the processor's pointer registers for root set scanning. When a pointer register is updated by the collector, the attribute registers remain unchanged, and the previous content of the pointer register is copied to the backlink register.

The microprogram takes special care with all synchronization mechanisms under microprogram control. First, any loop that may iterate for more than a few times contains explicit synchronization points in order to bound the latency of the read barrier interrupt. Second, the collector explicitly stops the main processor for at most 500 clock cycles during root set scanning. Third and last, any lock established by the microprogram is changed or released in a strictly bounded amount of time.

6. Experimental results

To demonstrate the feasibility and practicability of our approach and to evaluate system performance, we have built up a working prototype of the proposed system. We modeled both the main processor and the garbage collection coprocessor at the register transfer level in VHDL and synthesized them for a single advanced programmable logic device (Altera APEX 20K1000C). The garbage collector uses less than 20% of the chip area. Furthermore, we have assembled an experimental computer system based on the garbage-collected processor. Standard SDRAM modules are used for main memory. Peripherals implemented for the system include standard serial and parallel interfaces, PS/2 interfaces for keyboard and mouse, and a 100 MBit Ethernet interface. Processor, SDRAM and the peripherals are synchronously operated at 25 MHz.

On the software side, we have developed a static Java compiler that translates standard Java bytecode to the processor's native machine code. The compiler includes a code scheduler that rearranges instructions in order to take advantage of the processor's parallel execution units and to hide instruction latencies as far as possible. Moreover, we realized a subset of the Java class libraries supporting text-based applications in order to facilitate the execution of representative programs. As part of the class library, we implemented an NFS client to provide for access to a file system.

For performance evaluation, we have run a number of real programs on the prototype. Because of some constraints of our current software environment (static compilation, text-based applications only, so far no thread support in the runtime system), our present collection of benchmark programs does not include all programs typically found in benchmark suites.

Table 1. Garbage collection runtime overhead

application	sss_{min}	t_{min}	relative semispace size		
			150%	200%	250%
compress	7,317 K	46.2 s	**+0.1%**	**+0.0%**	**+0.0%**
database	10,343 K	252.8 s	*+5.4%*	**+2.3%**	**+2.1%**
javac	5,642 K	34.5 s	**+4.5%**	**+3.0%**	**+1.5%**
jlisp	129 K	61.0 s	**+1,3%**	**+0.9%**	**+0.7%**
jflex	2,041 K	25.6 s	*+0.9%*	**+0.2%**	**+0.1%**
cup	8,707 K	52.6 s	*+15.9%*	**+2.0%**	**+1.7%**
javacc	1,931 K	20.4 s	*+23.2%*	*+6.8%*	*+1.5%*

To measure the impact of the garbage collector, we determined the runtime overhead in dependence of the semispace size. To do so, we first provide each application with virtually infinite memory and turn off the garbage collector in order to find the minimum execution time t_{min}. Next, we turn on the garbage collector and reduce the semispace size step by step in order to find the smallest possible semispace size sss_{min}. Please note that this size is of theoretical interest only since an application will never be run close to this minimum. Then, we run each application with more realistic semispace sizes that are 150%, 200% and 250% of the required minimum. The results are shown in Table 1. Values printed in bold indicate real-time behavior (no mutator starvation).

The results show that mutator starvation can be avoided by relative semispace sizes of 150% to 200% in most cases. Furthermore, the results demonstrate that the runtime overhead caused by the garbage collector is typically as little as a few percent or less under relevant operating conditions.

The different behavior of the applications we examined is caused by the fact that the cost of copying garbage collection depends on both the amount of live memory and the allocation rate. Traditional applications like *compress* require large amounts of live memory, but they allocate very little. In contrast, *jlisp* (LISP interpreter) and *jflex* (scanner generator)

show very high allocation rates, but get by with little amounts of live memory since most objects die relatively young. The parser generators *cup* and *javacc* constitute the worst case. They build up huge and long-lived data structures that are traversed while new objects are allocated at a very high rate.

In the second experiment, we measured the pause times caused by all synchronization mechanisms (read barrier, explicit processor stops, locks). We have chosen *jflex* as a best-case, *javac* as an average-case, and *javacc* as a worst-case application. The results are illustrated in Figure 7.

Analyzing the results, the number of pauses caused by the read barrier by far dominates the number of pauses stemming from locks and explicit processor stops, and the frequency of read barrier pauses decreases with their duration. The longest pauses are caused by explicit stops. Regarding their distribution, three discrete areas can be identified: Pauses below 50 clock cycles are caused by the stop that protects the comparison of static pointers with the processor's stack pointer before stack processing, pauses around 270 clock cycles come from cleaning up the processor's caches, and pauses above 300 clock cycles originate from root set scanning. Pauses caused by locks can be effectively ignored.

Most importantly, the measurements confirm that the duration of any pause does not exceed 500 clock cycles. Even with largely different semispace sizes and threshold values, neither the maximum duration nor the shape of the distributions change significantly. Only the absolute frequencies vary in orders of magnitude.

7. Discussion

The system we presented in this paper solves the problems of incremental garbage collection stated in the introduction. Synchronization is efficiently realized in hardware and neither increases code size nor depends on any compiler support. Fine-grained incremental object scanning and incremental compaction is realized by the cache line locking
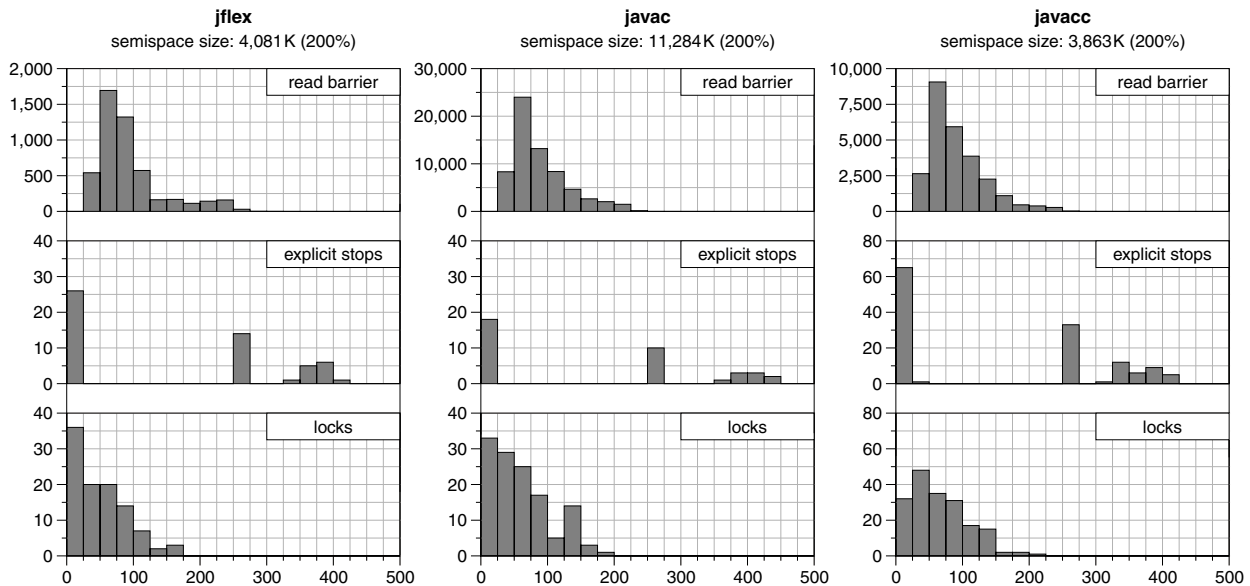

Figure 7. Frequency distribution of synchronization pauses (number vs. duration in clock cycles)

mechanism, the backlink entries, and the assistance of the AGU. This way, there is no need to ever lock an entire object, and no atomic action depends on the size of objects. Finally, the problem of root set scanning is solved by restricting the root set to the pointer registers and by treating stacks as static objects that are incrementally processed. All these mechanisms guarantee by design that the duration of all pauses caused by garbage collection never exceeds a small constant.

While bounding synchronization pauses is a required precondition for hard real-time behavior, it is not necessarily sufficient. If these pauses are heavily clustered, the mutator's progress may be affected in a way that impedes responses to environmental changes within a specified amount of time. In this respect, Baker's algorithm usually shows a high read barrier fault rate at the beginning of a garbage collection cycle. In our system, a faulting read barrier merely reserves space for an object in the worst case, which typically takes between 50–100 clock cycles only. The actual work of processing the corresponding object is incrementally performed by the collector. Nevertheless, we are currently trying to reduce the cost of faulting barriers to further address this issue.

Mutator starvation constitutes another threat to real-time capabilities. In our system, we directly determine adequate semispace sizes and threshold values by measurement. As an alternative approach, some researchers [2, 8] measure (or assume) the maximum amount of live data and derive parameters for the configuration of their collectors therefrom. In any case, the real-time guarantees depend on values that are empirically determined, and consequently, no system known so far is able to actually prove its hard real-time capabilities in the strict sense.

Concerning the required memory overhead for real-time behavior, we typically need an overall factor of 3–5 which is very common for real-time garbage collectors. As with every copying collector, performance can be increased by simply providing more memory. Currently, we are trying to reduce the memory space overhead by investigating some more elaborate garbage collection algorithms.

All benefits with respect to hardware-supported garbage collection and low-cost synchronization are substantially enabled by the processor architecture that forms the basis of our system. Compared to a standard RISC, the implementation of this architecture entails some additional costs that have to be taken into account. First, there is a two-word overhead per object for the object's attributes. This cost, however, is attenuated by the fact that the length of arrays may be easily determined without explicit length fields and that no additional fields are required for garbage collection. Second, the additional attribute stage comes at some cost and results in a two-cycle latency of the load pointer instruction if the loaded pointer is subsequently dereferenced. Yet the scheduler in our compiler rearranges instructions in order to minimize related pipeline stalls as far as possible.

In addition to the obvious advantages for the implementation of a garbage collector, the architecture of the main processor significantly increases the robustness at the machine code level as it provides object-based memory protection. This feature is of particular interest for embedded systems that usually cannot rely on virtual memory. In contrast to vir-

tual memory, object-based memory protection considerably eases information sharing between different processes or between applications and the operating system since pointers are globally valid. As a further benefit of object protection, runtime checks required by many languages (e.g. array bounds checks in Java) are implicitly performed without any code-size or runtime overhead.

8. Conclusions

In this paper, we have presented the design and implementation of a processor with an on-chip hardware garbage collector. The processor guarantees pointer integrity in hardware and introduces the robustness stemming from garbage collection at the machine-code level. Synchronization of processor and garbage collector is efficiently realized in hardware. For almost all applications that we have examined, real-time behavior can be achieved by semispace sizes that are a factor of 1.5 to 2.5 larger than the required minimum. Furthermore, the overall runtime overhead of garbage collection is as little as a few percent or less in typical configurations. To our best knowledge, the prototype that we have realized is the first garbage-collected system that guarantees an upper bound on the duration of all garbage collection related pauses in the order of 500 clock cycles. Because of its potential for real-time and safety-critical applications, the proposed architecture is of particular interest for embedded systems.

References

[1] Baker, H. G.: List processing in real time on a serial computer, *Comm. ACM,* vol. 21(4), Apr. 1978, pp. 280–294.

[2] Bacon, D. F.; Cheng, P.; Rajan, V. T.: A real-time garbage collector with low overhead and consistent utilization, *13th ACM Symposium on Principles of Programming Languages,* Jan. 2003, pp. 285–298.

[3] Dijkstra, E. W., et al.: On-the-fly garbage collection: an exercise in cooperation, *Comm. ACM,* vol. 21(11), Nov. 1978, pp. 966–975.

[4] Jones, R., Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management,* Wiley, 1996.

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

[6] Moon, D. A.: Garbage collection in a large LISP system, *ACM Symp. on LISP and Functional Programming,* Aug. 1984, pp. 235–246.

[7] Schmidt, W. J., Nilsen, K. D.: Performance of a hardware-assisted real-time garbage collector, *6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct. 1994, pp. 76–85.

[8] Siebert, F.: *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages,* Dissertation, University of Karlsruhe, Germany, 2002.

[9] Williams I., Wolczko, M.: An object-based memory architecture, *4th Int. Workshop on Persistent Object Systems,* Sept. 1990, pp. 114–130.

[10] Zorn, B.: *Barrier Methods for Garbage Collection,* Tech. Report CU-CS-494-90, University of Colorado, Nov. 1990.