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

A Novel Processor Architecture with Exact Tag-Free Pointers

ALONG WITH THE SUCCESS OF JAVA, GARBAGE COLLECTION ADVANCES TO EMBEDDED AND REAL-TIME SYSTEMS. OUR NOVEL PROCESSOR ARCHITECTURE LAYS THE FOUNDATION FOR EFFICIENT REAL-TIME GARBAGE COLLECTION IN HARDWARE AND GUARANTEES POINTER INTEGRITY AT THE MACHINE-CODE LEVEL.

Matthias Meyer
University of Stuttgart

Automatic dynamic memory management, also known as garbage collection, is indispensable in controlling software complexity since it increases programmer productivity and software quality. However, garbage collection is still considered an unaffordable luxury for application areas such as embedded, real-time, and safety-critical systems.

I propose a novel RISC processor architecture that introduces the robustness gained by garbage collection at the machine-code level. The architecture provides the basis for a hardware garbage collector that closely cooperates with the processor. The advantages of this configuration include low garbage collection overhead, low-cost synchronization of collector and application programs, high robustness, and hard real-time capabilities.

Garbage collection

To start with, this article summarizes the most commonly used garbage collection methods and points out problems of contemporary software-based solutions to motivate architectural support for garbage collection.

Basic algorithms

The basic algorithms for garbage collection are reference counting, copying, and mark-sweep collection.[1] Reference counting maintains a counter for each object, which holds the number of references pointing to the object. The garbage collector can reclaim objects with a counter of zero. Reference counting, however, fails to free cyclic garbage. In contrast, mark-sweep and copying algorithms, also called tracing algorithms, avoid the cost of updating reference counters and overcome the problem with cycles. These algorithms trace memory starting with a set of roots, usually consisting of processor registers and the program stack. Mark-sweep collectors first mark all reachable objects (mark phase) and then reclaim all unmarked objects (sweep phase). Copying collectors divide the heap into two areas called semispaces and use only one semispace at a time. During collection, they copy all reachable objects from one semispace to the other, thereby inherently compacting the heap. Although copying collectors need twice as much memory as their mark-sweep counterparts, they are particularly attractive

Published by the IEEE Computer Society

because the cost of garbage collection is proportional to the amount of reachable objects rather than the size of the entire heap.

Traditional implementations of tracing algorithms suspend application processing for the entire duration of a garbage collection cycle and can cause long and unpredictable pause times. For this reason, they are not applicable to interactive systems or real-time environments.

Incremental garbage collection

Incremental garbage collection techniques aim to improve the interactive response of garbage-collected systems. Such techniques allow application processing to continue during garbage collection. In this context, the application program is usually referred to as the *mutator*, because it changes the graph of objects while the collector is traversing the heap.

To illustrate, consider describing incremental garbage collection with Dijkstra's tricolor abstraction.[2] In this abstraction,

- *black* indicates that the collector has finished with an object for the current garbage collection cycle,
- *grey* indicates that the collector hasn't 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, the collector will reclaim white objects.

Problems can arise if the mutator writes a pointer to a white object into a black object. If the original pointer to the white object is destroyed and no further pointer to the white object exists, the collector will illegally discard the object at the end of the garbage collection cycle. Usually, either read or write barrier methods prevent the mutator from disrupting the garbage collector. Read barriers ensure that the mutator never sees a white object. Whenever the mutator accesses a pointer to a white object, the collector immediately visits the object. In contrast, write barriers ensure that black objects turn grey as soon as the mutator installs a pointer to a white object. To realize read or write barriers, the compiler must emit a few instructions before each pointer load or store, which incurs substantial runtime overhead and increases the code size considerably.[3]

Many incremental garbage collection algorithms are designated as real-time, because garbage collection pauses are, in the majority of cases, too short for a user to notice. Hard real-time systems, however, require correct responses to environmental changes within a short and specified time. Because software-based incremental algorithms usually must rely on some sort of atomic action, such as scanning the complete root set or processing an entire object, they are not applicable to hard real-time environments. Although there are software-based garbage collected systems that claim to satisfy hard real-time requirements, these implementations suffer considerable execution time overhead (due to expensive object accesses, write barriers, synchronization points, and root set copies) as well as memory space overhead (because of auxiliary data structures and internal fragmentation).[4]

Pointer finding

All garbage collection methods face the problem of *pointer finding* or *pointer identification*. If the garbage collector cannot unambiguously distinguish pointers from nonpointers, it can only employ conservative garbage collection methods and has to consider any bit pattern that looks like a pointer to be a pointer, to avoid freeing memory still in use. The conservative approach, however, has drawbacks. A value mistakenly identified as a pointer might refer to an object that recursively could contain pointers to other objects, thus keeping the collector from freeing unpredictable amounts of memory. Furthermore, a conservative collector is incapable of moving objects because this necessitates the update of pointers, and updating some data erroneously taken as a pointer can be disastrous. For this reason, conservatively collected heaps are subject to fragmentation unless the compiler employs costly measures such as using handles. So copying collectors or other compacting collectors, such as mark-sweep-compact, are unusable unless exact information about the location of each pointer is available.

To realize exact (nonconservative) garbage collection (also known as precise or accurate garbage collection), many implementations

spend a lot of effort in searching and exactly identifying pointers. In object-oriented languages, pointers in heap objects are often identifiable by type descriptors. These descriptors are usually available in each object because they are necessary for dynamic binding and runtime checks. However, locating pointers in the program stack and in processor registers is more difficult, particularly in the context of optimizing compilers. Although it is possible to maintain maps describing which stack locations or processor registers contain pointers, the cost of updating such maps at runtime is prohibitively high. For this reason, most software-based solutions rely on tables that describe all pointer locations in the stack and in registers. The compiler constructs a set of tables for each program point where a collection might occur (gc-points).[5]

Software-based methods for exact pointer identification, however, share disadvantages. The tables necessary to describe all pointer locations at all gc-points usually increase code size dramatically. Moreover, real-time environments must ensure that suspended threads reach the next gc-point within a bounded amount of time. Finally, these methods depend heavily on compiler support and are not generally applicable.

Related work

Known since 1966, capability-based architectures use capabilities instead of pointers to refer to memory.[6] Examples of capability-based computers are the Plessey System 250, the IBM System/38, and the Intel iAPX 432. *Capabilities* provide a safe mechanism to uniformly address both main memory and secondary storage. A capability does not contain the object's physical address. Instead, it contains a unique object identifier and the access rights for the object. The unique identifier locates a descriptor that specifies the physical location and the object's size. Capability-based architectures either tag capabilities or keep them separate from ordinary data to guarantee that users never compromise them. They provided for exact capability identification long before garbage collection became a major topic of interest.

Starting in the 1980s, various architectures evolved to support particular programming languages, most notably Lisp and Smalltalk. Example architectures include SPUR (Symbolic Processing Using RISCs),[7] Symbolics,[8] SOAR (Smalltalk On A RISC),[9] and Mushroom (Manchester University Software and Hardware Realization of an Object-Oriented Machine).[10] These architectures distinguish among types of data (including pointers) by augmenting every memory word with one or more tag bits and provide limited hardware support for garbage collection, such as hardware-assisted read or write barriers.

More recently, researchers have proposed active memory modules with hardware support for garbage collection, including Nilsen and Schmidt's garbage collected memory module,[11] Wise et al.'s reference count memory,[12] and Srisa-an et al.'s active memory processor.[13] Because these architectures rely on specialized memory, the hardware cost for the memory modules is relatively high and, above all, depends on the size of the garbage-collected memory.

For this work, three observations about systems with hardware support for exact pointer identification, garbage collection, or both are of particular interest. First, many architectures are designed for special programming languages. Second, only the expensive active memory modules used by Nilsen and Schmidt, and Srisa-an et al., target hard real-time applications, but they do not address the problem of processing the root set in a bounded amount of time. Third, apart from early capability-based computers, all architectures for exact garbage collection use tags to identify pointers, and most of them provide no hardware mechanism to guarantee pointer integrity.

Processor architecture

Exact hardware-supported garbage collection must be able to unambiguously distinguish pointers from nonpointers at the hardware level. If the architecture uses tag bits for this purpose, a 32-bit system needs main memory that is 33 bits wide. Moreover, a tagged architecture must expend considerable effort in initializing and maintaining these tags, and a garbage collector must examine every single word in order to find all pointers. Finally, the use of tag bits alone is insufficient to guarantee pointer integrity.

As an alternative approach, this article

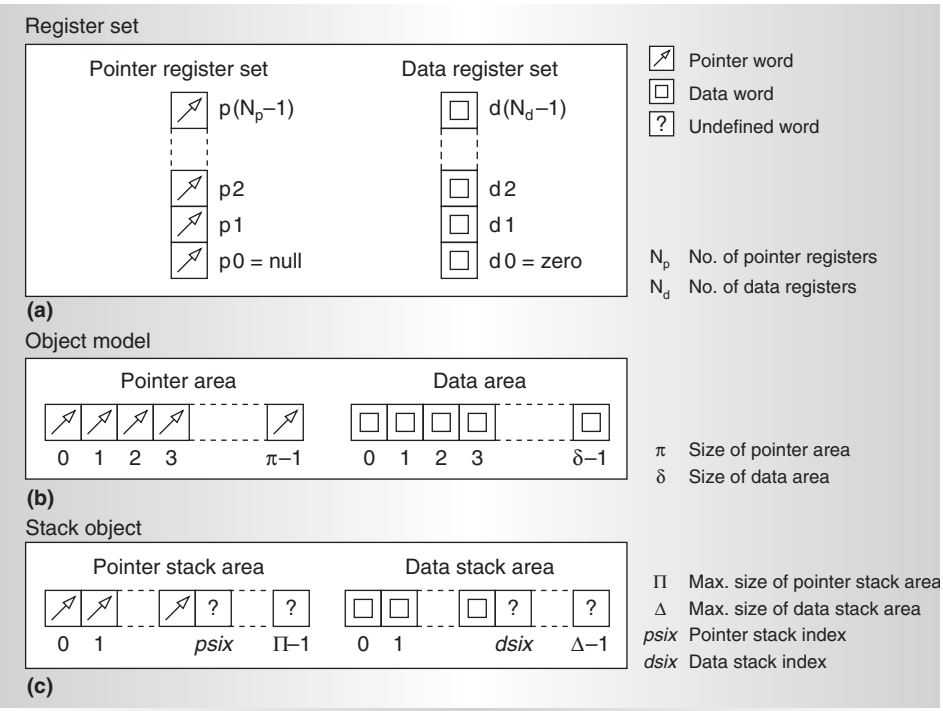

Figure 1. Processor architecture's programming model. The register set (a), object model (b), and stack object (c) strictly separate pointers from nonpointer data.

introduces a processor architecture that strictly separates pointers from nonpointer data, allowing for tag-free pointer identification. To ensure exact pointers, the architecture maintains the following system invariant:

- First clause (§1). The architecture identifies every memory word or register to be a pointer or not.
- Second clause (§2). Each pointer value is either null or uniquely associated with an existing object.

Programming model

Figure 1 shows the programming model of the processor architecture. It specifies separate register sets for data and pointers. Data registers serve as general-purpose registers whereas pointer registers are exclusively for referring to objects in memory. To comply with the system invariant, it must be impossible to write arbitrary values to pointer registers or to perform arithmetic operations on pointer registers.

The processor architecture's memory model is object-based. An object has a separate data area and a separate pointer area. The object's δ-attribute describes the size of the data area, and the object's π-attribute describes the size of the pointer area. The attributes are determined when the object is created and cannot be altered thereafter.

The stack object, like every object, consists of a data area and a pointer area and can thus be thought of as two separate stacks. Pointer register p1 holds the reference to the stack object. In each of the two areas, a stack index indicates the first free location and separates the respective area into the actual stack and a free area. The architecture reserves two data registers for the stack indices. Register d1 holds the pointer stack index (*psix*), and register d2 holds the data stack index (*dsix*).

However, if the architecture treats the stack like an ordinary object with constant size, the system cannot tell whether a pointer word belongs to the actual stack or to the free area. Since every word in the pointer area is considered a pointer, there might be many pointers within the free area pointing to objects that are no longer necessary. Though this constitutes no threat to the system invariant, a garbage collector will not be able to reclaim these objects because they are still accessible, causing unpredictable amounts of floating garbage.

One possible solution to this problem is explicitly storing a null in the appropriate stack location whenever a pointer is removed from the stack. However, this approach is not particularly attractive because of the related overhead, especially if many pointers are to be removed from the stack at the same time, such as during the deallocation of a stack frame at the end of a subroutine.

A better approach is treating the stack as an object of dynamic size by regarding *psix* as the π-attribute and *dsix* as the δ-attribute of the stack object. With respect to pointer identification and the system invariant, the architecture will only consider words at indices below *psix* to be pointers.

Instruction set

The processor architecture defines only pointer-related instructions, including load and store instructions. The design of other instructions, such as for arithmetic or program control, remains the choice of a particular implementation.

Allocate object. The processor architecture entirely abstracts dynamic memory management. A special *allocate object* instruction creates a new object and associates an object reference with it. The allocate instruction takes the values of the π- and the δ-attribute for the new object as arguments and returns the reference to the newly created object in a pointer register. Every pointer word in the pointer area of the created object is initialized with null before the allocate instruction completes.

There is no such thing as an instruction for the deletion of objects. The manual deletion of objects unavoidably leads to the dangling-reference problem, thus violating the system invariant. For this reason, the architecture relies on garbage collection at the processor level.

Load and store. The instruction set provides separate load and store instructions for data words and pointers. The *load data* and *store data* instructions exclusively move data words between an object's data area and data registers. Similarly, the *load pointer* and *store pointer* instructions move only pointers between an object's pointer area and pointer registers. To identify the memory word they should access, load and store instructions use a pointer register holding the reference to the object and a positive integer index. To compute this index, an implementation can provide indexing modes (the equivalent to addressing modes in conventional architectures) using constant indices, data registers, displacements, and scale factors.

Accessing memory words outside the data or pointer area of an object can cause unpredictable effects. Depending on the object layout, it might be possible to access pointers by means of out-of-range data load instructions or vice versa. For this reason, the architecture prescribes bounds checks. If a bounds check fails, the processor does not permit the load or store instruction to complete and raises an *index out of bounds exception*. For similar reasons, the processor prevents a memory access that uses a pointer register holding null and issues a *null pointer exception*.

Push pointer. To push a pointer value onto the pointer stack, a program has to store the pointer and increment *psix*. However, if the architecture were to permit a program to store the pointer before incrementing *psix*, it would offend the first clause (§1) of the system invariant between the two operations, as the pointer just stored is not yet identified as a pointer. If, on the other hand, a program increments *psix* before storing the pointer, it would offend the second clause (§2) between the two operations, since a word identified as a pointer does not yet contain a valid pointer.

The architecture solves this problem by providing a pointer store instruction (or, alternatively, an indexing mode for use with a pointer store instruction) that both stores a pointer at the first free pointer stack location and increments the pointer stack index in an indivisible fashion. Furthermore, this must be the only instruction to increment the pointer stack index. Any attempt to increment the pointer stack index by other instructions results in a *psix increment exception*.

Copy and compare pointers. In contrast to the multitude of register-to-register instructions for operations on data registers, the architecture defines a tightly restricted set of two instructions for pointer-related register-to-register operations. The *copy pointer* instruction

copies the contents of a pointer register to another pointer register, while the *compare pointers* instruction checks whether two pointer registers refer to the same object.

Query object attributes. The instruction set provides two instructions to query the attributes π and δ of an object. An attempt to query the attributes via a null pointer results in a *null pointer exception*.

Special objects

Apart from dynamically created objects, the architecture defines special types of objects that serve particular purposes.

Constant objects. These make it possible to access constant data in the same way as dynamically created objects. Constant objects exist before a program starts (for example, as part of the program code) and typically realize constant strings and compiler-generated structures, such as branch tables and type descriptors.

A pointer to a constant object is created by a *create constant pointer* instruction. Since this instruction includes the conversion of an address into a pointer, it substantially threatens the system invariant. For this reason, operations on constant pointers are only for read accesses, and the architecture confines accesses through constant pointers to the area for constant objects. As a consequence, constant objects can only contain constant pointers. Violating these rules at runtime will raise appropriate exceptions.

Static objects. In many systems, different operating modes require separate program stacks; for example, user or supervisor modes employ a separate stack. Moreover, multithreaded environments require separate program stacks for each thread of execution. The operating system usually arranges all these stacks apart from a garbage-collected heap.

To permit manually managed memory areas outside the heap, this architecture uses *static objects*. The operating system creates static objects by means of a privileged instruction and lays them out in a dedicated memory area.

Uninitialized objects. To satisfy the system invariant, each pointer in a newly created object must be initialized before the corresponding allocate instruction can complete. Therefore, there is no small upper bound on the execution time for the allocate instruction. Thus, this sort of initialization is not acceptable for hard real-time applications.

To provide for interruptible allocate instructions, the architecture introduces *uninitialized objects* (strictly speaking, *incompletely initialized objects*). The processor creates uninitialized objects only if it suspends allocate instructions because of interrupts. To observe the system invariant, instructions are not allowed to dereference pointers to uninitialized objects.

Summary. The processor architecture supports four types of objects: dynamic (ordinary), constant, static, and uninitialized. Dynamic and uninitialized objects reside in the heap, static objects in a static area, and constant objects in the area for program code and constant data. Static and uninitialized objects are available only in supervisor mode.

It is possible to define the four object types by the way a compacting (moving) garbage collector treats them. *Ordinary dynamic objects* require collectors to scan them for pointers and move them for compaction. *Static objects* require scanning for pointers, too, but collectors must not move them. In contrast, collectors must move *uninitialized objects* during compaction, but must not scan them because these objects can contain invalid pointers. Finally, *constant objects* must not be moved or scanned. It is remarkable that, although motivated by rather practical reasons, these four object types completely fill the space spanned by the scan and move properties.

Processor implementation

My colleagues and I have implemented a 32-bit RISC processor conforming to the proposed architecture to demonstrate that this architecture can yield an efficient implementation. This processor will operate without a hardware garbage collector as long as the operating system provides a software garbage collector.

Object layout

An object in memory consists of a two-word attribute header for π and δ followed by

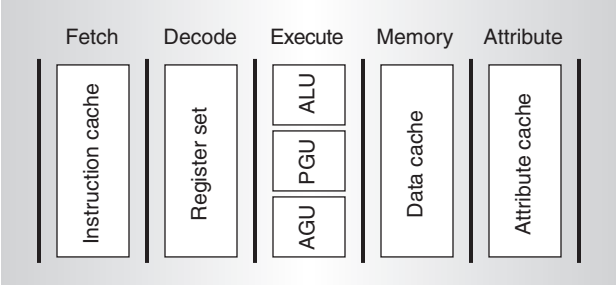

Figure 2. Basic processor pipeline structure. An additional pipeline stage, the attribute stage, provides efficient attribute access.

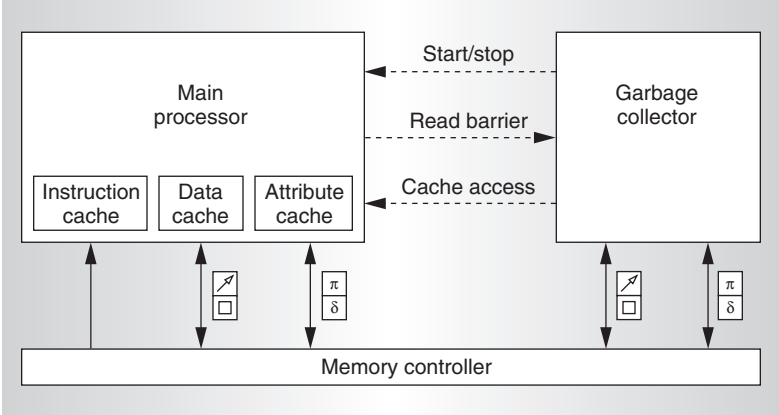

Figure 3. System overview. The garbage collector is a separate, microcoded coprocessor that tightly cooperates with the main processor.

the pointer and the data areas. The attributes describe the number of bytes in these areas. For efficiency, objects are always double-word aligned. The processor encodes a pointer as the address of the object that the pointer refers to. Because of byte addressing and alignment constraints, some bits in words holding a pointer or π are available for identifying the four different object types and for tagging purposes during garbage collection.

Processor pipeline

Figure 2 shows the basis for our implementation: a straightforward pipelined RISC design that extends to efficiently handle objects and attributes.

The register set includes 16 data and 16 pointer registers. In the execute stage, the arithmetic logic unit (ALU) processes instructions targeting data registers, and the pointer generation unit processes instructions (such as allocate) that target pointer registers. The address generation unit (AGU) generates addresses for accessing the cache in the memory stage. Because this cache serves the same purpose as a data cache in conventional architectures, the figure shows it as a "data cache" even though it contains ordinary data and pointers.

Before an instruction can use a pointer to access an object, it is necessary to have the object's attributes for range checking and for address generation. For this reason, attribute registers supplement every pointer register. Whenever a pointer register contains a non-null value, the corresponding attribute registers hold the attributes of the object to which the pointer register refers. In this way, the effort for dereferencing a pointer register is as low as that of address generation in conventional architectures. Range checking is not a performance penalty because it occurs in parallel with address generation.

Whenever an instruction loads a pointer register from memory, it must also load the associated attribute registers. This happens within an additional pipeline stage after the usual memory stage. This *attribute stage* uses an attribute cache to allow for attribute accesses without a performance penalty for the common case.

All caches are two-way set-associative copy back. The cache line size of the instruction and data caches is eight words. In contrast, an attribute cache line is two words wide and holds only the attributes of a single object.

Garbage collector implementation

Figure 3 shows how we realize a hardware garbage collector as a separate, microcoded coprocessor. The memory controller provides separate ports for both the main processor and the garbage collector. Because of this configuration, the garbage collector's inherently nonlocal behavior does not disrupt cache locality.

Garbage collection algorithm

The implemented garbage collection algorithm derives from the incremental copying algorithm that Baker introduced.[14] Figure 4 shows that whenever the garbage collector evacuates an object from fromspace to tospace, it only reserves space in tospace instead of actually copying the object. In doing so, the collector tags the π-attribute of the object and saves the δ-attribute to the

tospace copy and then overwrites it with a forwarding pointer. In tospace, the field for the π-attribute is initialized with a backlink to the fromspace original. When the collector traverses the yet empty object in tospace, it scans each pointer in the corresponding fromspace object by either evacuating the referenced object or, if the object has already evacuated, by reading the forwarding pointer, and writes the resulting tospace pointer to the tospace object. Subsequently, the collector copies the data area and blackens the tospace object by replacing the tagged backlink with the untagged π-attribute.

A garbage collection cycle starts whenever the amount of available memory falls below an adjustable threshold. Real-time behavior requires some memory headroom to ensure that the collection cycle terminates before the mutator "starves" (runs out of memory).

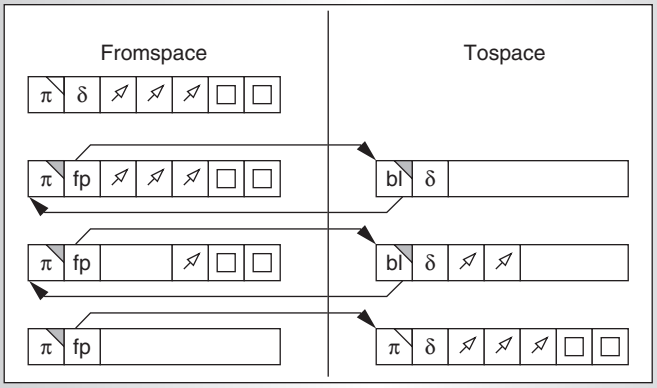

Figure 4. Object states during garbage collection. The attributes of grey objects are split between fromspace and tospace. A forwarding pointer (fp) and a backlink (bl) doubly link the tospace copy to the fromspace original.

Synchronization

Figure 3 shows that the synchronization of the processor and the garbage collector occurs on different levels. The garbage collector ensures cache coherence by checking the address tags and valid bits in the data and attribute caches, and by flushing cache lines if necessary. A cache line locking mechanism enforces exclusive access to objects. On the garbage collection level, a hardware read barrier triggers a garbage collector interrupt whenever the processor tries to access a pointer to a white object. Finally, the garbage collector can stop the processor pipeline to scan the root set. Because the architecture treats stacks as static objects, the garbage collector must only scan the pointer registers atomically and processes stacks in an incremental way. For any synchronization mechanism, our implementation guarantees by design that synchronization pauses never exceed a small constant on the order of a couple of hundred clock cycles. As an additional benefit, the synchronization mechanisms do not require any compiler support.

Concurrent compaction

To support concurrent object copying, a backlink entry supplements every pointer register and every attribute cache line. If the processor loads the attributes of a grey object from memory, it also loads the corresponding backlink. Whenever an instruction is about to access a grey object, the AGU determines whether the tospace pointer or the backlink is to be used for address generation, and the instruction accesses either the fromspace original or the tospace copy. This way it is not necessary to lock an entire object while the garbage collector is copying it.

Experimental results

We have developed the processor and the hardware garbage collector in VHDL, putting them together on a single, advanced, programmable-logic device, an Altera APEX 20K1000C. The garbage collector occupies approximately 20 percent of the chip area. The prototype operates at 25 MHz and features 8-Kbyte instruction and data caches, and a 2-Kbyte attribute cache. We have also built an experimental computer system based on the processor, using standard SDRAM modules for main memory. To facilitate the execution of representative programs, we implemented a static Java bytecode compiler and a subset of the Java class libraries supporting text-based applications.

Figure 5 shows the garbage collection overhead that we measured on the prototype by running several real-world applications with different semispace sizes. To provide a basis for comparison, all test cases execute with a nonincremental software garbage collector and with the concurrent hardware garbage collector. Apart from the type of garbage col-

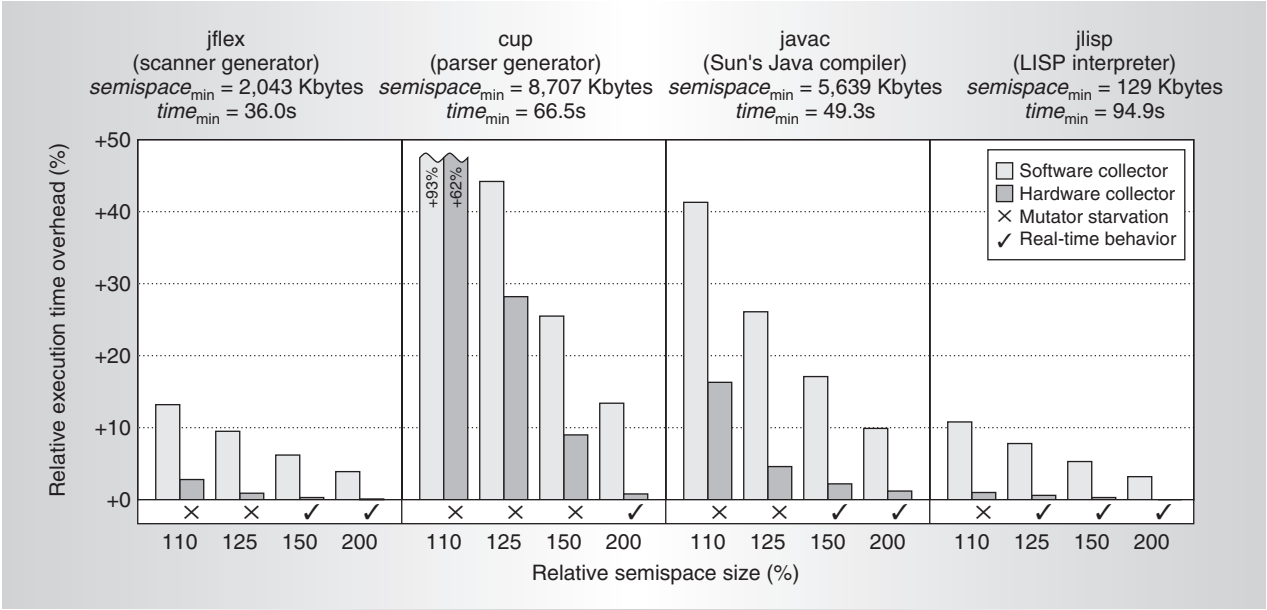

Figure 5. Relative execution time of software and hardware garbage collectors for several applications and semispace sizes. A relative semispace size of 100 percent denotes the maximum amount of live memory that an application uses ($semispace_{min}$). This figures give the relative execution time with respect to the execution time with a virtually infinite semispace size ($time_{min}$).

lector, the environment is exactly the same in both cases. For the hardware collector, Figure 5 shows a symbol indicating whether pauses occurred due to mutator starvation ("×") or whether enough memory headroom was available to ensure hard real-time behavior ("✓").

All applications that we have examined make extensive use of heap allocated memory. In the case of jflex and jlisp, most objects die relatively young, the best case for a copying collector. In contrast, applications like cup that require large amounts of live memory for long periods of time constitute the worst case. The results show that the applications we have examined can achieve real-time behavior by using semispace sizes that are 25 to 100 percent larger than the required minimum.

Under relevant operating conditions, the hardware collector's runtime overhead is small. In any case, the real-time hardware collector is significantly more efficient than the nonincremental stop-the-world software collector.

Our novel RISC processor architecture hides the details of dynamic memory management at the processor level. It ensures exact pointers without tags and provides a basis for hardware-supported real-time garbage collection. We present an efficient implementation of a processor conforming to the proposed architecture and a copying hardware garbage collector. Performance measurements on the prototype show that, with a reasonable amount of available memory headroom, the real-time hardware garbage collector will have almost no noticeable effect on application programs.

In the future, we plan to explore the potential of more elaborate garbage collection algorithms such as generational garbage collection to further reduce the cost of garbage collection in terms of memory and time overhead. MICRO

References

1. R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.
2. E.W. Dijkstra et al., "On-the-fly Garbage Collection: An Exercise in Cooperation," *Comm. ACM*, vol. 21, no. 11, Nov. 1978, pp. 966-975.
3. B. Zorn, *Barrier Methods for Garbage Collection*, tech. report CU-CS-494-90, Univ. of Colorado, 1990.
4. F. Siebert, "Hard Real-Time Garbage Collection in the Jamaica Virtual Machine,"

Proc. Sixth Int'l Conf. Real-Time Computing Systems and Applications, IEEE CS Press, 1999, pp. 96-102.
5. O. Agesen, D. Detlefs, and E. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines," *ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 98), ACM Press, 1998, pp. 269-279.
6. H.M. Levy, C*apability-Based Computer Systems*, Digital Press, 1984.
7. M. Hill et al., "Design decisions in SPUR," *IEEE Computer*, vol. 19, no. 11, Nov. 1986, pp. 8-22.
8. D.A. Moon, "Garbage Collection in a Large LISP System," *Proc. Conf. LISP and Functional Programming*, ACM Press, 1984, pp. 235-246.
9. D. Ungar et al., "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Annual Symp. Computer Architecture* (ISCA 84), IEEE CS Press, 1984, pp. 188-197.
10. M. Wolczko and I. Williams, "Multi-Level Garbage Collection in a High-Performance Persistent Object System," *Proc. 5th Int'l Workshop Persistent Object Systems*, Springer Verlag, 1992, pp. 396-418.
11. K.D. Nilsen and W.J. Schmidt, "A High-Performance Hardware-Assisted Real-Time Garbage Collection System," *J. Programming Languages*, vol. 2, no. 1, Jan. 1994, pp. 1-40.
12. D.S. Wise et al., "Research Demonstration of a Hardware Reference-Counting Heap," *LISP and Symbolic Computation*, vol. 10, no. 2, July 1997, pp. 159-181.
13. W. Srisa-an, C.D. Lo, and J.M. Chang, "Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices," *IEEE Trans. Mobile Computing*, vol. 2, no. 2, Apr.-June 2003, pp. 89-101.
14. H.G. Baker, "List Processing in Real Time on a Serial Computer," *Comm. ACM*, vol. 21, no. 4, Apr. 1978, pp. 280-294.

Matthias Meyer is a research group leader at the Institute of Communication Networks and Computer Engineering at the University of Stuttgart, Germany. His research interests include digital systems design, computer architecture, compiler construction, garbage collection, and Java. Meyer has a Diploma degree from the University of Stuttgart in Electrical Engineering.

Direct questions and comments about this article to Matthias Meyer, University of Stuttgart, IKR, Pfaffenwaldring 47, D-70569 Stuttgart; meyer@ikr.uni-stuttgart.de.