# Mark-Sweep or Copying? A "Best of Both Worlds" Algorithm and a Hardware-Supported Real-Time Implementation

Sylvain Stanchina Matthias Meyer

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47
70569 Stuttgart, Germany
{sylvain.stanchina, matthias.meyer}@ikr.uni-stuttgart.de

## Abstract

Copying collectors offer a number of advantages over their mark-sweep counterparts. First, they do not have to deal with mark stacks and potential mark stack overflows. Second, they do not suffer from unpredictable fragmentation overheads since they inherently compact the heap. Third, the tospace invariant maintained by many copying collectors allows for incremental compaction and provides the basis for efficient real-time implementations. Unfortunately, however, standard copying collectors depend on two semispaces and therefore need at least twice as much memory as required for the maximum amount of live data.

In this paper, we introduce a novel mark-compact algorithm that combines the elegance and simplicity of Baker's copying algorithm with the memory efficiency of mark-sweep algorithms. Furthermore, we present a hardware-supported implementation for real-time applications in the framework of an object-based RISC architecture.

Measurements of Java programs on an FPGA-based prototype show that our novel mark-compact algorithm outperforms a corresponding copying collector in every respect. It requires far less memory space for real-time behavior and, at the same time, reduces the overall runtime overhead under typical operating conditions.

***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, Mark-Compact Collection, Object-Based Processor Architecture, Hardware Support

## 1. Introduction

Tracing garbage collection algorithms are subdivided into two main families. Mark-sweep algorithms, as their name implies, operate in two passes. They first mark all live objects by recursively tracing the graph of objects. Then, they reclaim all unmarked objects. Standard copying algorithms, on the other hand, divide the heap into two equally-sized semispaces and, in a single pass, copy all live objects from one semispace to the other. In this way, they implicitly reclaim dead objects.

The huge advantage of mark sweep algorithms is that they do not depend on semispaces, but allow applications to always use the entire heap. Unfortunately, this advantage comes at the price of two serious problems. First, mark-sweep collectors need explicit storage for a tracing stack and must address the potential problem of stack overflows. In contrast, copying collectors do not have to worry about overflows because they dynamically embed a tracing queue into the heap. Second, and most importantly, mark-sweep collectors cause fragmentation, and with it unpredictable amounts of memory overhead. In contrast, copying collectors inherently compact the heap, which furthermore allows for extremely low-cost allocation.

The two problems of mark-sweep collectors are particularly severe for embedded systems. These systems have to achieve deterministic behavior despite of tightly restricted resources, both with respect to memory space and processing time. Regarding mark stacks, they are difficult to realize by means of limited memory, and robust implementations have to reserve one stack entry per object (e.g. [7, 24]). Regarding fragmentation, its actual extent is difficult to predict in advance. For deterministic memory behavior, mark-sweep collectors have to use an additional compaction pass, resulting in mark-sweep-compact or mark-compact collectors. However, standard compaction methods, such as threading [10] or table-based methods [8] cannot be made incremental since they render the heap unusable until compaction completes. Incremental methods, on the other hand, are extremely expensive. For this reason, known mark-sweep-compact algorithms for real-time systems either decompose objects into blocks of a fixed size [24], or they only process small portions of the heap at a time [11, 2, 22]. Unfortunately, the first approach simply trades external for internal fragmentation, while the second approach reduces part of the fragmentation only, and that at the cost of longer pauses and/or higher runtime overhead.

In this paper, we present a novel mark-compact garbage collection algorithm for real-time applications. Like a copying collector, it traverses the graph of objects without a marking stack, but unlike standard copying collectors, it does not depend on semispaces for compaction. In this way, the algorithm combines the elegance and simplicity of copying algorithms with the memory efficiency of mark-sweep algorithms.

To substantiate the benefits of our algorithm, we furthermore present a hardware-supported implementation for a system including an object-based RISC processor supported by a microprogrammable garbage collection coprocessor. Thanks to the concurrency gained by the coprocessor and efficient synchronization in hardware, pauses caused by garbage collection or synchroniza-



*ISMM'07,* October 21–22, 2007, Montréal, Québec, Canada.

tion are usually very short and always guaranteed to be less than a few hundred clock cycles. Therefore, garbage collection is almost invisible to application programs.

The rest of this paper is organized as follows. In Section 2, we analyze Baker's copying collector and show how its essential properties can be used to design a collector without semispaces. Section 3 introduces an object-based architecture for hardware-supported garbage collection and describes the implementation of our mark-compact algorithm for that architecture. In Section 4, we compare the performance of the algorithm with that of a Baker-style copying algorithm and present measurement results from our FPGA-based prototype. Section 5 discusses related algorithms and architectures. Finally, Section 6 provides a conclusion.

## 2. Algorithm

Baker's algorithm [3] is the best-known incremental copying algorithm. It is simple, elegant, and, with Steele's extensions, in principle suitable for fine-grained incremental compaction. For these reasons, it has been used as the basis for numerous real-time implementations [17, 14]. Being a standard copying collector, however, Baker's algorithm requires two semispaces and therefore at least twice the maximum amount of live memory needed by an application.

The novel algorithm we introduce in this paper is best described as a "best of both worlds" combination of a Baker-style copying collector and a mark-sweep collector: From Baker, it inherits incremental compaction. From mark-sweep, it inherits memory efficiency.

This section is organized as follows. First, we describe Baker's algorithm with Steele's extensions. Next, we analyze and extract its essential mechanisms and use them as the basis for a novel mark-compact algorithm. Finally, we identify a number of potential implementation issues.

In this section, we initially abstract from implementation details. We merely assume that objects are composed of a header and a body, and that the collector and the application (mutator) are able to unambiguously distinguish between pointers and non-pointer data.

### 2.1 Baker's Algorithm

Like most copying algorithms, Baker's algorithm divides the heap into two equally-sized semispaces referred to as *fromspace* and *tospace*. At the start of a collection cycle, the collector stops the application, swaps tospace and fromspace, and initializes two pointers *scan* and *free* to point to the bottom of tospace. Then, the collector evacuates all objects referenced by the root set from fromspace to tospace. To evacuate an object, the collector reserves an empty object frame at the position referred to by *free* and doubly links the empty object frame to the corresponding fromspace original (Figure 1a, assuming that A and B are referenced by the root set). As soon as the collector has finished with the roots, it resumes the mutator and enters the main collection loop.

At each iteration of the main collection loop, the collector fills the empty object frame referred to by *scan*. For this purpose, it follows the backlink in the frame header and successively copies the fromspace original to tospace. In doing so, each time the collector encounters a fromspace pointer, it checks whether the corresponding object has already been evacuated. If so, it reads the forwarding pointer. If not, the collector evacuates that object. In either case, the collector stores a tospace pointer into the tospace copy. When the collector has finished with an object, it deletes the backlink in the tospace copy. A cycle terminates when *scan* catches up with *free*. For illustration, the example in Figure 1b shows the moment the collector copies and scans object B.

The states of objects during a collection cycle are usually described by Dijkstra's tricolor abstraction [6]. Accordingly, an object is *white* when it has not yet been seen by the collector. *Gray* indicates that the collector has started with an object, but has not yet finished with it. Finally, an object is *black* when the collector has finished with that object, i.e. when it has visited and grayed all its descendants. At the end of a collection cycle, all objects are either black or white. Black objects survive, while white objects are reclaimed as garbage.

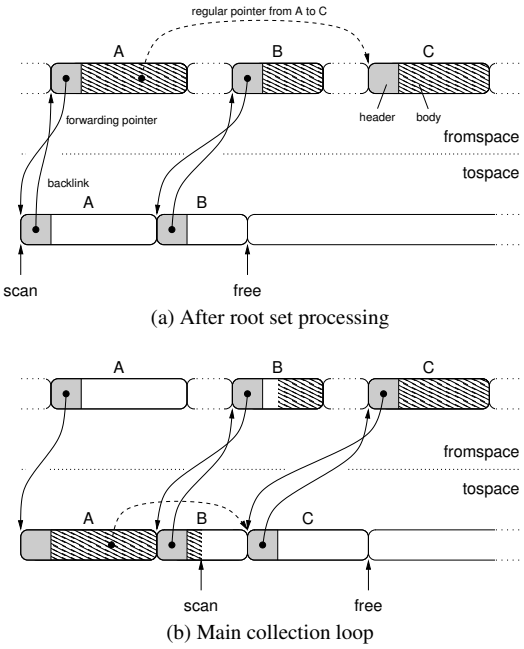



**Figure 1.** Baker's algorithm

Figure 2 illustrates the tricolor abstraction for Baker's algorithm. White objects exclusively lie in fromspace. Gray objects exist in both spaces at the same time. Black objects have been entirely copied and lie in tospace only. However, the fromspace original of black objects is still required to redirect any remaining fromspace pointers to tospace.

New objects are allocated in tospace, starting from the top. As a consequence, all objects allocated in the course of a collection cycle are initially black and can be ignored by the collector in that cycle.

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

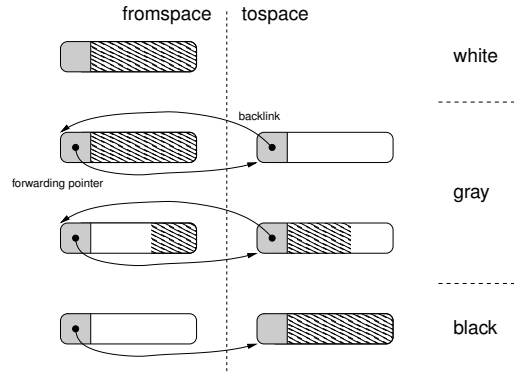



**Figure 2.** Object states (Baker's algorithm)

restriction, problems may 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 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. In this way, the read barrier realizes a so-called *tospace invariant*, i.e. the mutator will exclusively see tospace pointers.

### 2.2 Analyzing Baker's Algorithm

As mentioned in the introduction, the two main advantages of copying collectors over mark-sweep collectors are that they do not depend on a mark stack and that they compact the heap. In the following section, we are going to analyze how the corresponding mechanisms are realized by Baker's algorithm, i.e. how the algorithm gets by without a mark stack and how it uses the semispaces for compaction.

First, Baker's algorithm, like any tracing algorithm, requires some sort of a data structure to remember the set of objects that have been seen by the collector, but not yet scanned. Mark-sweep algorithms realize this set as a separate mark stack. In contrast, Baker's algorithm realizes this set in the form of a queue embedded into tospace. This tracing queue is delimited by *scan* and *free*. All objects, strictly speaking all empty object frames located in between these pointers, belong to the set of objects that remain to be processed by the collector. The queue grows when the collector or the read barrier advances *free*, i.e. while they evacuate objects from fromspace to tospace. Correspondingly, the queue shrinks when the collector advances *scan*, i.e. while the collector actually copies the objects and scans them for pointers. A collection cycle terminates when the queue is empty, i.e. when *scan* meets *free*.

Second, Baker's algorithm, like any mechanism for compaction, must relocate objects and update pointers. Basically, the pointer update requires a mechanism to map the original locations of all objects to their respective new locations. If the mutator is furthermore allowed to proceed during compaction and if it may access objects at their new locations before they have been completely copied, a second mechanism will be required to map the new locations back to the original locations.

For relocation, Baker's algorithm evacuates all live objects from fromspace and arranges them closely packed at the bottom of tospace. To map fromspace pointers to tospace pointers, each fromspace object is provided with a forwarding pointer that refers to the corresponding tospace copy. To furthermore allow the mutator to proceed during collection (and, with it, during compaction), the algorithm maintains the tospace invariant and provides each tospace copy with a backlink to the corresponding fromspace original. Thanks to the tospace invariant, the mutator only sees tospace addresses, independently of whether objects have already been copied or not. Thanks to the backlink, the mutator is able to find the original version of uncopied or incompletely copied objects.

Reconsidering the two mechanisms just analyzed, it is most interesting to note that neither of them depends on semispaces. First, each entry in Baker's tracing queue requires as much space as the entire object. Actually, however, a queue with simple pointers would be absolutely sufficient. For this reason, using a full semispace for the tracing queue is actually a waste of memory. Second, incremental compaction in itself does not depend on semispaces either. As long as there is some sort of a tospace invariant and some sort of method to make object references independent from object locations (such as providing a backlink), objects can actually be moved from anywhere to anywhere.

### 2.3 A New Mark-Compact Algorithm

#### 2.3.1 Overview

The analysis of Baker's algorithm has revealed that neither tracing nor incremental compaction requires the presence of entire objects in tospace. We exploit this discovery by dynamically dividing the heap into two spaces of different sizes. The first space corresponds to Baker's fromspace and contains entire objects. This space will be referred to as *object space*. The second space corresponds to Baker's tospace, but will contain references only. These references are designated as handles. They double as queue entries for scanning and as indirections for incremental compaction. Accordingly, the second space will be referred to as *handle space*.

Our algorithm operates in three phases. The first phase is similar to Baker's algorithm. However, instead of evacuating objects by reserving empty object frames in tospace, the first phase of our algorithm merely creates object handles in handle space (Figure 3, gray, upper diagram). Analogously to Baker, objects in object space are doubly linked to the respective handles in handle space, i.e. our handles correspond to Baker's backlinks. At the end of this phase, all live objects are "marked" by a reference to a handle. Therefore, we refer to the first phase as the *mark phase*. The second phase is the *compaction phase*. In this phase, live objects are compacted within object space, and the handles are redirected accordingly (Figure 3, gray, lower diagram). Dead objects are implicitly overwritten. The third phase is designated *cleanup phase*. This phase removes all handles and returns the objects to the same state as observed before the mark phase.

The algorithm is entirely incremental and allows the application to proceed during all phases. In the mark phase, a read barrier ensures a *handle space invariant*, i.e. the mutator exclusively sees handles. Thanks to the handles and the invariant, the compaction phase is incremental as well. In the cleanup phase, another read barrier ensures an *object space invariant* so that the mutator will no longer see handles, but direct object pointers only.

#### 2.3.2 Heap Layout

For the best possible memory efficiency, our algorithm adjusts the size of the object space and the handle space dynamically. Since any live object requires a handle during compaction, and since all allocated objects potentially survive a collection cycle, we dynamically reserve space for one handle per allocated object.

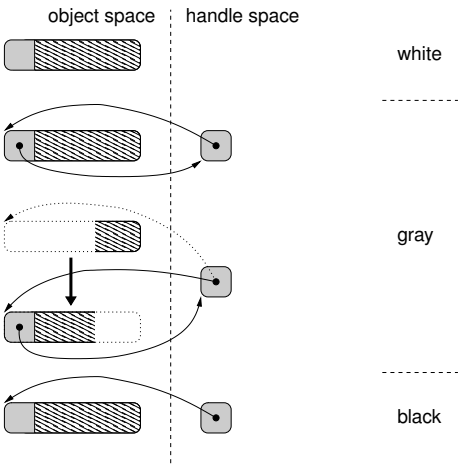



**Figure 3.** Object states (our algorithm)

The corresponding heap layout is shown in Figure 4. The object space is located at the bottom end of the heap. An allocation pointer *alc* always refers to the first free location at the end of the object space. The handle space is arranged at the top end of the heap and delimited by a pointer *alc_lim*. At each object allocation, *alc* is advanced by the size of the new object, and *alc_lim* is decreased by the size of an object handle. In doing so, the handle space is merely reserved and is initially completely empty. When *alc* and *alc_lim* meet, the heap is full.

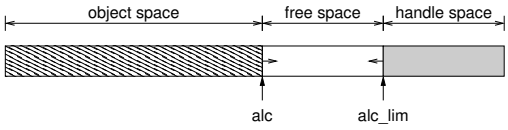



**Figure 4.** Heap layout

### 2.3.3 Mark Phase

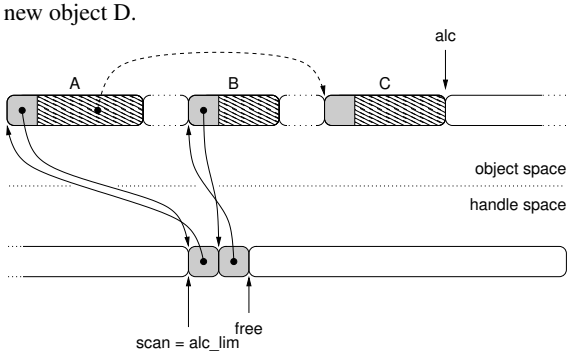
During the mark phase, the collector creates a handle for each reachable object and replaces all pointers by references to corresponding handles. As previously mentioned, this phase is virtually identical to Baker's algorithm. The only difference is that objects are not copied, but left at their place.

At the start of the mark phase, the collector stops the application and initializes two pointers *free* and *scan* to point to the bottom of handle space, i.e. to the current position referred by *alc_lim*. Then, it creates a handle for every object referenced by the root set and replaces each root set pointer with a reference to the corresponding handle. In doing so, all handles are successively stored in handle space at the position indicated by *free*, and *free* is incremented correspondingly. After root set processing, the collector activates the read barrier and resumes the application. Figure 5a shows a snapshot of the heap after root set processing. Like in Figure 1, the example assumes that the root set consists of objects A and B, and that object A contains a pointer to object C.

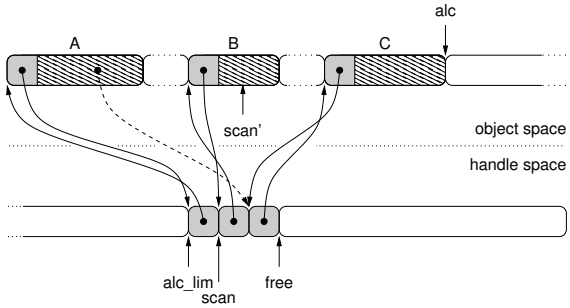
At each iteration of the marking loop, the collector follows the handle pointed to by *scan* in order to scan the corresponding object. In doing so, the collector advances a pointer *scan'* in object space to indicate the scanning progress. Whenever the collector encounters a pointer that refers to an object rather than a handle, it checks whether that object already contains a reference to a handle, i.e. whether it has been previously marked. If so, the collector reads that reference. Otherwise, it creates a handle and doubly links it to the object. In either case, the collector replaces the pointer at the position pointed by *scan'* by a reference to the corresponding handle. When the collector has finished with an object, it advances *scan* to point to the next handle (Figure 5b). The mark phase terminates when *scan* catches up with *free*.

Like Baker, we use a read barrier to protect the collector and to ensure that the mutator sees handles only. In case the mutator attempts to read a non-handle, i.e. a direct pointer to an object, the read barrier immediately creates a handle for this object, or, if a handle to this object already exists, reads the corresponding reference from the object.

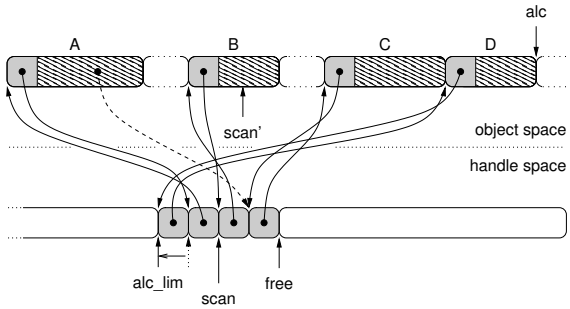
In contrast to Baker's algorithm, newly allocated objects must be specially treated in order to avoid that they are prematurely reclaimed. Since the compaction phase reclaims unmarked objects, i.e. objects without handles, objects allocated during the mark phase and the compaction phase must immediately be marked, i.e. objects and their handles must be created at the same time. As previously mentioned, each allocation reserves space for a handle by decreasing *alc_lim*. In the mark and compaction phases, this space is immediately used to store the handle created with the allocated object. In this way, handles for newly created objects are arranged at the left end of the handle space, and in particular not in between *scan* and *free*. Therefore, newly created objects are not part of the tracing queue and do not cause any needless work for the collector. Because of the handle space invariant, the mutator can never write non-handle references into objects, and so the mark phase can safely ignore objects created during the current garbage collection cycle. For illustration, Figure 5c shows the heap after the allocation of a new object D.




(a) After root set processing




(b) During scanning




(c) Object allocation

**Figure 5.** Mark phase

### 2.3.4 Compaction Phase

During the compaction phase, the collector reclaims unmarked objects and compacts surviving objects to the bottom of the heap. In doing so, it uses a so-called *sliding technique* that preserves the order of objects.

At the start of the compaction phase, the collector initializes two pointers *target* and *source* to refer to the bottom of object space (Figure 6a). During each iteration of the main loop, the collector checks the object pointed to by *source*. If this object is unmarked, i.e. if it does not contain a handle, the collector advances *source* to the next object (Figure 6b). Otherwise, the collector moves that object from its current position to the position referred to by *target*. To do so, the collector first copies the object header (i.e. the

reference to the object's handle) and updates the handle to refer to the new location. Then, it successively copies the contents of the object and advances *source* and *target* accordingly (Figure 6c and 6d). The compaction loop repeats until *source* catches up with the allocation pointer *alc* (Figure 6e).

During compaction, the free space between *target* and *source* becomes bigger and bigger. To avoid that newly created objects have to be moved soon after their creation, each allocation checks whether this free space is large enough to accommodate the new object. If so, the object is allocated at *target* (more precisely at the position that *target* will assume as soon as the compactor is done with the object it currently copies) instead of at *alc*. This way, less objects have to be moved, the compaction phase terminates earlier, and the overall efficiency of garbage collection is increased.

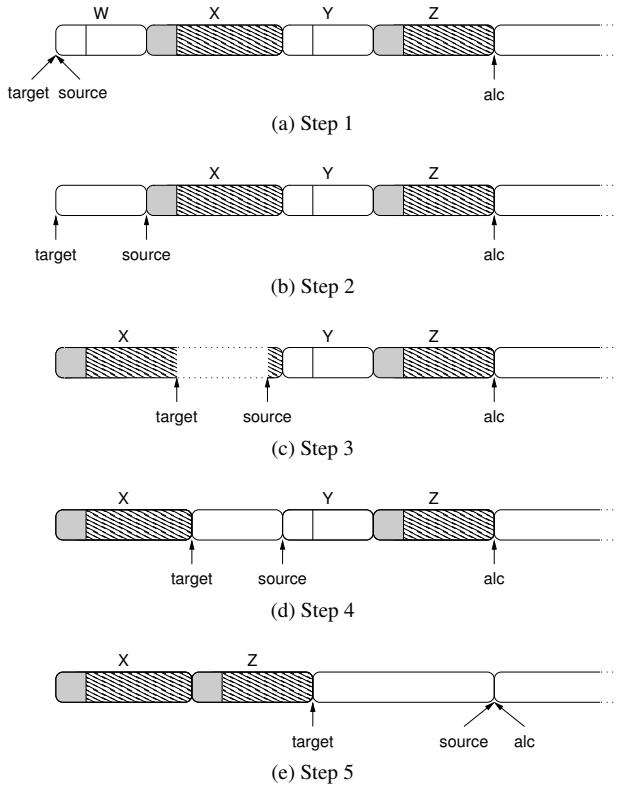



**Figure 6.** Compaction phase

#### 2.3.5 Cleanup Phase

During the cleanup phase, the collector removes all handles and replaces all references to handles by direct pointers to the corresponding objects. The presence of this phase yields three essential benefits. First, there are no handles and no indirections while the collector is idle. Second, it allows handles to double as tracing queue entries for marking and as indirections for compaction. Third, thanks to the removal of all handles at the end of a garbage collection cycle, there is no need to explicitly garbage collect the handle space itself.

At the beginning of the cleanup phase, the collector suspends the mutator for the duration of a second root set processing. This time, references to handles are replaced by direct object pointers, and the handle references in the corresponding objects are deleted. Next, the collector initializes *scan* to refer to the bottom of the handle space, i.e. to the current position referred to by *alc_lim*. The pointer *free* remains unchanged and still refers to the end of the used handle space. Then, the mutator is resumed.

An iteration of the main cleanup loop consists of the following steps. First, the collector follows the handle currently referred to by *scan*. Next, it deletes the handle reference in the corresponding object. Then, the collector scans the object like it did during the mark phase. This time, however, each time it encounters a handle reference, it replaces that reference by the value of the handle, thereby effectively removing the indirection caused by the handle. As soon as the collector has finished scanning an object, it advances *scan* to the next handle. The cleanup phase terminates when *scan* meets *free*.

Like during the mark phase, a read barrier maintains an invariant that allows the mutator to proceed during cleanup. This time, the read barrier ensures that the mutator never sees object handles. Whenever the mutator is about to read a reference to a handle, the read barrier immediately replaces that reference by the handle value, i.e. by a direct pointer to that object.

As always, allocations reserve handles for allocated objects during the cleanup phase, too. However objects are created without handles, like in the idle phase. In contrast, objects allocated during the mark and compacting phases are always created together with the corresponding handle.

After the cleanup phase, the handle space is effectively empty, and the exact upper bound for the number of live objects is now given by the difference of *free* and *alc_lim*. For this reason, the handle space can be shrunk to that size, and *alc_lim* can be increased accordingly.

### 2.4 Implementation Issues

Our algorithm, like Baker's algorithm, is both simple and elegant. At the same time, however, implementations threaten to become expensive.

First of all, our algorithm relies on a read barrier. However, read barriers are known to be more expensive than write barriers. On the one hand, load instructions are more frequent than store instructions. Accordingly, read barrier checking introduces more runtime overhead than write barrier checking. On the other hand, read barrier faults tend to cluster after root set scanning. This might reduce mutator progress to a degree that is intolerable for real-time systems.

Second, our algorithm depends on indirections during the mark and compaction phases. This indirection must be followed by every single load and store instruction, both for pointer and non-pointer accesses. Usually, the cost of such indirections is prohibitively high (e.g. [7]).

Third, fine-grained Steele-style compaction renders memory accesses even more expensive. For every single load and store instruction, the application code must check whether the collector is currently moving the corresponding object. If so, it must additionally determine whether the object is to be accessed at its original location or at its new location, and it must accordingly calculate the address of the requested field.

## 3. Implementation

In fact, all the implementation issues mentioned at the end of the previous section are not caused by the complexity of our algorithm, but by the inability of software to efficiently synchronize applications with garbage collection. All the required synchronization mechanisms, be it for read barriers, for incremental compaction, or for mutual exclusion, have to be implemented in software by compiler-inserted code sequences and cause 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 between the compiled application code and a particular garbage collection algorithm.

In contrast to software, 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 [13]. In a first proof-of-concept design, he realized Baker's incremental copying algorithm with Steele-style extensions for incremental compaction. Thanks to a garbage collection coprocessor and synchronization in hardware, garbage collection is completely independent of any compiler support and causes little amounts of overhead only. Furthermore, the system is able to predictably bound any garbage collection pause to less than 500 clock cycles [14].

In this section, we show how to take advantage of Meyer's system to implement our novel algorithm in an efficient way. In the first part of this section, we introduce Meyer's initial system. In the second part, we present an implementation of our algorithm for this system and in particular describe the required extensions.

### 3.1 Initial System

An ideal computer system should provide the illusion of infinite memory. In such a system, objects would be allocated as required and forgotten about when they were no longer needed. Motivated by this vision, Meyer proposed a novel object-based architecture and realized a computer system with hardware-supported garbage collection for embedded real-time applications [13]. The architecture completely abstracts from memory management at the assembly language level and thereby allows garbage collection as well as all garbage-collection-related synchronization to be completely realized in hardware. The system consists of a RISC processor according to the object-based architecture (main processor) and a dedicated garbage collection coprocessor that operates completely in parallel to application processing (Figure 7).

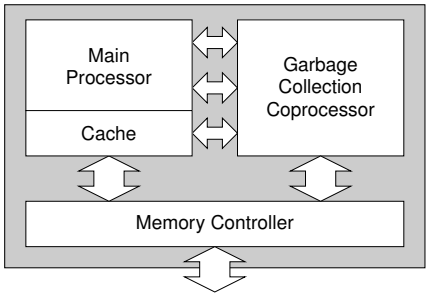



**Figure 7.** System overview

#### 3.1.1 Main Processor

For exact garbage collection in hardware, objects and pointers must be known at the assembly language level. For this reason, the main processor shows an object-based architecture. Rather than using plain addresses, load and store instructions use pointers with indices to access memory. The architecture provides a dedicated instruction to create objects. There is, however, no instruction to delete objects. Instead, the architecture relies on a hidden garbage collector to recycle memory behind the scenes.

The architecture exactly identifies pointers by strictly separating pointers from non-pointer data. It implements this separation by means of three mechanisms. First, objects are split into two dedicated areas, one for pointers, and one for non-pointers. Each of the two areas realizes a separate index space starting at zero. Two object attributes $\pi$ and $\delta$ describe the size of the pointer area and the data area, respectively. These attributes are stored in an object header that is invisible at the assembly language level (Figure 8). Second, the processor's register set is split into a pointer register set and a data register set. Third, separate instructions are provided for

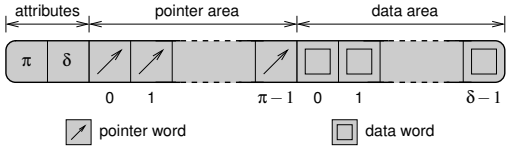



**Figure 8.** Object layout

pointers and non-pointer data. Regarding load and store instructions, for instance, pointer load and pointer store instructions implicitly target an object's pointer area, while load and store instructions for non-pointers implicitly target its data area, respectively.

In order to ensure the integrity of pointers, the content of a data register cannot be transferred to a pointer register or vice versa. Furthermore, range checking ensures that load and store instructions never violate the bounds of the corresponding pointer or data area, respectively.

The implementation of the processor is based on a pipelined RISC design that is extended to efficiently handle objects and attributes (Figure 9). Compared to a classical RISC, the processor pipeline shows the following three enhancements: First, the register set (decode stage) is split into 16 data registers and 16 pointer registers. Each pointer register is supplemented by two attribute registers. Whenever a pointer register contains a non-null value, the corresponding attribute registers hold the attributes of the object the pointer register refers to. Second, the execute stage contains different execution units for different types of target operands. While the ALU (Arithmetic Logic Unit) performs standard operations targeting data registers, the PGU (Pointer Generation Unit) takes care of operations targeting pointer registers, and the AGU (Address Generation Unit) performs range checks and generates addresses for the cache in the subsequent memory stage. 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.

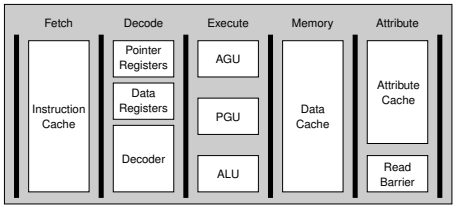



**Figure 9.** Main processor pipeline

#### 3.1.2 Garbage Collection Coprocessor

The garbage collection coprocessor is a low-cost microprogrammable device that is integrated with the main processor on the same chip (Figure 7). It realizes the complete garbage collection algorithm in the form of a single microprogram.

Because of the poor temporal locality of garbage collection, the coprocessor does not profit from a general-purpose data cache and is consequently designed without one. Typical garbage collection tasks such as scanning and copying, however, show a fair amount of spatial locality. To exploit this property, the coprocessor features burst registers that take advantage of efficient burst modes offered by modern memory devices.

The proof-of-concept collector of the initial system targets hard real-time applications. It is based on Baker's copying algorithm with Steele's extensions for fine-grained lazy copying. Whenever the collector evacuates an object from fromspace to tospace, it does

not actually copy the object, but merely reserves an empty object slot in tospace (Section 2.1). To do so, it sets a gray-bit in the original π-attribute, saves the δ-attribute to tospace, and overwrites the δ-attribute with a forwarding pointer (Figure 10). In tospace, it initializes the field for the π-attribute with a backlink to the fromspace original. In this way, the garbage collector distributes the attributes between fromspace and tospace and manages to doubly link the tospace copy to the fromspace original without causing any additional storage overhead.

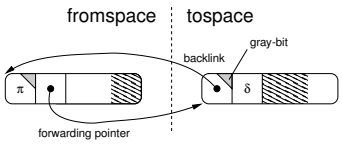



**Figure 10.** Linking of object attributes

#### 3.1.3 Synchronization Mechanisms

The tight coupling of the main processor and the garbage collection coprocessor allows for particularly efficient synchronization in hardware, including cache coherency, cache line locking, mutual exclusion, and incremental stack processing [14]. In the context of this paper, two synchronization mechanisms are of particular interest.

First, a hardware read barrier realizes read barrier checking as well as read barrier fault handling entirely in hardware. The corresponding circuitry is located in the processor's attribute stage (Figure 9), causes no runtime overhead for read barrier checking, and handles read barrier faults within a few clock cycles [15].

The second mechanism allows for concurrent compaction. To realize Steele's extensions to Baker's algorithm, the mutator must always decide whether a requested field inside a particular object is already available in tospace. If not, the mutator has to recalculate the field's address, using the backlink entry found in the yet incomplete tospace copy.

To implement this elaborate procedure in an efficient way, the processor treats the backlink (i.e. an object's location in fromspace) as a third object attribute and adds a corresponding entry to every pointer register and to every attribute cache line. Whenever the processor is about to access a field inside a gray object, the AGU calculates two addresses in parallel: a tospace address based on the actual pointer, and a fromspace address based on the backlink. In case the tospace address is greater or equal than the garbage collector's *scan* pointer (Figure 1b), the field has not yet been copied and the AGU simply replaces the tospace address with the fromspace address. The procedure just described is implemented by relatively simple combinatorial logic that does not extend the critical path in the execute stage. Consequently, Steele-style address generation in hardware is as fast as standard address generation.

The only runtime cost associated with gray objects occurs if a pointer to a gray object is to be loaded and, at the same time, causes an attribute cache miss. In this case, since the π-attribute of a gray object resides in fromspace and its δ-attribute in tospace, the attribute cache requires two separate memory accesses to resolve the cache miss (Figure 10). Since the cache implicitly loads the backlink during the first memory access, maintaining the backlink attribute in the cache and the registers does not cause any additional runtime overhead.

### 3.2 Extensions for our Algorithm

Basically, our new algorithm relies on exactly the same synchronization mechanisms as Baker's original algorithm, in particular read barriers to maintain invariants and indirections in conjunction with corresponding address generation for incremental compaction. For this reason, our extensions are actually minor and mainly caused by the different heap layout used by our algorithm.

#### 3.2.1 Main Processor

The modified heap layout affects object allocation and address generation. Accordingly, the processor's PGU and AGU require a couple of small extensions.

The PGU is extended in three ways. First, our algorithm reverses the direction of allocation. Correspondingly, we generalized the PGU and made the allocation direction configurable. Second, the object allocation circuitry in the PGU can now decrease the value of *alc_lim* in order to reserve space for object handles. Third, the PGU supports a second allocation pointer *alc'* to alternatively allocate in the free space between *target* and *source* (Section 2.3.4).

The AGU is extended in two ways. First, despite the tospace invariant, the addresses generated by the original AGU will refer to fromspace if the corresponding object or the corresponding part of an object has not yet been copied. In contrast, and despite the handle space invariant, our extended AGU will exclusively generate addresses in object space. Second, in the original design, a simple address comparison is sufficient to determine whether an object needs to be accessed at its original or at its new location. In contrast, to support single-space compaction, our AGU additionally needs to know whether the accessed object is currently moved by the collector or not.

#### 3.2.2 Coprocessor

Since the hardware garbage collector is realized as a microprogrammable device, our new algorithm is implemented as easily as by extending the initial microprogram. Additionally, we added a few new microinstructions in order to improve the efficiency of two novel synchronization features, namely for locking the alternative allocation pointer *alc'* and for updating handles inside the main processor (see Section 3.2.3 below).

#### 3.2.3 Synchronization Mechanisms

Regarding synchronization of garbage collection and application processing, we extended the read barrier, added a locking mechanism for the alternative allocation pointer, and implemented a circuitry to effectively update handle values inside the main processor.

Regarding the read barrier, the circuitry for read barrier checking is exactly the same. While the boundaries of the corresponding space change once per collection cycle with Baker's algorithm, they change twice per cycle with our new algorithm. In contrast, the read barrier fault handling circuitry needs a minor extension. Since the read barrier creates handles instead of empty object frames, *free* must now be advanced by the size of a handle instead of the size of the corresponding object.

The second extension allows both processors to access the alternative allocation pointer *alc'* in a mutually exclusive way. While the main processor modifies this pointer whenever it allocates an object in between *target* and *source*, the garbage collection coprocessor modifies this pointer as soon as it starts copying an object for compaction. To avoid potential conflicts, the coprocessor is able to lock *alc'*. In the case of a conflict, the main processor will wait until the lock is cleared.

The third and final modification concerns the backlinks stored along with π and δ in the form of a third object attribute with every pointer register. In our new algorithm, these backlinks act as handle values. While backlinks, once created, do not change for the duration of the remaining collection cycle, handle values must be updated as soon as the collector starts moving the corresponding object. To allow the collector to implement the handle update process in a particularly efficient way, each pointer register is supplemented with a small comparison circuitry that updates the cor-

responding handle register as soon as the location of the referred object changes. These circuits operate within a single clock cycle and completely in parallel to instruction execution.

### 3.3 Summary

All the extensions described in this section are entirely configurable, i.e. the extended system supports our new algorithm as well as Baker's original algorithm. As a benefit, the desired algorithm can easily be selected at runtime by exchanging the microcode in the coprocessor and by modifying some configuration bits in the corresponding system registers.

By design, our new algorithm preserves the upper bound on synchronization pause times introduced by the initial system. Furthermore, our system, like the initial system, implements read barrier fault handling completely in hardware, thereby limiting the runtime penalty of read barrier faults to a few clock cycles only. As a resulting benefit and despite fault clustering, all applications we examined show minimum mutator utilizations of more than 50% within arbitrary time intervals of 1 ms [15].

## 4. Experimental Results

### 4.1 Measurement Platform

To demonstrate the feasibility and efficiency of our algorithm, we extended a hardware prototype of the initial system. This prototype consists of the main processor and the garbage collection coprocessor, both of which are jointly realized on an Altera Stratix II device (EP2S60 [1]), of standard DDR-SDRAM modules for main memory, and of various peripheral devices, including an Ethernet interface for file access via NFS. The entire prototype is synchronously operated at 25 MHz.

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 disposes of a microcode memory of 256 words with 96 bit each. The coprocessor uses approximately 20% of the chip area. The extensions for our algorithm occupy less than 3%.

For clock-cycle accurate measurements, we integrated a monitoring framework into the main FPGA that allows to trace up to 32 internal processor signals. By means of a dedicated, on-board Gigabit Ethernet interface, the measurement data is transmitted at a constant rate of 800 MBit/s to a measurement PC, written to multiple hard disks in parallel, and analyzed offline.

On the software side, we have developed a static Java compiler that translates standard Java bytecode to the processor's native machine code. Moreover, we realized a subset of the Java class libraries supporting text-based applications in order to facilitate the execution of representative programs.

### 4.2 Measurement Results

In a first experiment, we measured the smallest possible heap size for various applications[1] and for both garbage collection algorithms (Table 1). In case of the copying algorithm, the measured heap size exactly corresponds to twice the maximum amount of live data. In case of our algorithm, the measured heap size exactly corresponds to the maximum amount of live data plus the memory required for the corresponding handle space. The results show that the minimum memory overhead caused by garbage collection is always 100% of the maximum amount of live data in case of the copying algorithm, while it is only 1% to 27% in case of our new algorithm. Correspondingly, our mark-compact algorithm reduces the minimum memory overhead of garbage collection by a factor of 3 at minimum and by a factor 6 on average for all the applications we have examined. This gain is a direct consequence of replacing Baker's tospace by our handle space and depends on the average size of objects used by an application.

[1] Our system does currently not support threads, reflection and dynamic class loading. This restricts the range of benchmarks we are able to run.

| program | minimum heap size (% of max. live data) | |
|---|---|---|
| | copying collector | m.c. collector |
| cup | 17420 K (200%) | 11062 K (127%) |
| db | 20692 K (200%) | 12622 K (122%) |
| javac | 11212 K (200%) | 6671 K (119%) |
| javacc | 3870 K (200%) | 2361 K (122%) |
| jflex | 4096 K (200%) | 2168 K (106%) |
| jlisp | 266 K (200%) | 144 K (109%) |
| search | 10376 K (200%) | 5240 K (101%) |

**Table 1.** Minimum heap size required by both collectors

Next, we measured the runtime overhead of garbage collection for various relative heap sizes. For this purpose, we first provided each application with virtually infinite memory, deactivated the garbage collector, and measured the corresponding execution times $t_{\text{off}}$. Then, we limited the heap to various relative sizes, i.e. sizes relative to the maximum amount of live data, activated the garbage collector, and measured the corresponding execution times in order to determine the garbage collection runtime overhead. The results are given in Table 2. Values printed in bold indicate real-time behavior without mutator starvation.

For all applications we examined, our new algorithm achieves real-time behavior with less memory than the copying algorithm. In particular, our algorithm shows real-time behavior with relative heap sizes as little as 200% for some applications, whereas the copying algorithm hardly allows to execute applications within this size.

Under relevant operating conditions (i.e. with appropriate memory headroom), the runtime overhead caused by our new collector is as little as few percent only. Furthermore, this overhead is smaller than the overhead caused by the copying collector in the vast majority of cases. In the remaining cases, the difference is negligible.

By means of a third experiment, we aimed at evaluating the overall efficiency of the two collectors. For this purpose, we run each application with a heap size large enough to allow real-time behavior for both algorithms and measured the mean duration of a garbage collection cycle (Figure 11) as well as the fraction of total application execution time with garbage collection activities (Figure 12).

| program | GC | relative heap size (%) | | | |
|---|---|---|---|---|---|
| | | 200% | 300% | 400% | 500% |
| cup | cop. | 697.7% | 15.8% | 2.6% | **0.3%** |
| $t_{\text{off}}$=49.4s | m.c. | 38.8% | 10.9% | **1.5%** | **1.0%** |
| db | cop. | 58.9% | **3.1%** | **3.1%** | **2.2%** |
| $t_{\text{off}}$=239.0s | m.c. | 6.6% | **1.5%** | **0.8%** | **0.4%** |
| javacc | cop. | 271.8% | 22.1% | 5.1% | 0.3% |
| $t_{\text{off}}$=18.9s | m.c. | 37.4% | 13.0% | 4.8% | **1.2%** |
| javac | cop. | 115.7% | **0.9%** | **0.5%** | **0.2%** |
| $t_{\text{off}}$=31.4s | m.c. | **4.7%** | **2.0%** | **0.7%** | **1.0%** |
| jflex | cop. | 7.1% | 0.2% | **0.1%** | **0.0%** |
| $t_{\text{off}}$=25.2s | m.c. | 2.1% | **0.2%** | **0.0%** | **0.0%** |
| jlisp | cop. | 0.9% | **0.2%** | **0.0%** | **0.0%** |
| $t_{\text{off}}$=59.3s | m.c. | **0.6%** | **0.2%** | **0.1%** | **0.0%** |
| search | cop. | 61.1% | **0.1%** | **0.0%** | **0.0%** |
| $t_{\text{off}}$=47.8s | m.c. | **0.0%** | **0.0%** | **0.0%** | **0.0%** |

**Table 2.** Runtime overhead versus relative heap size

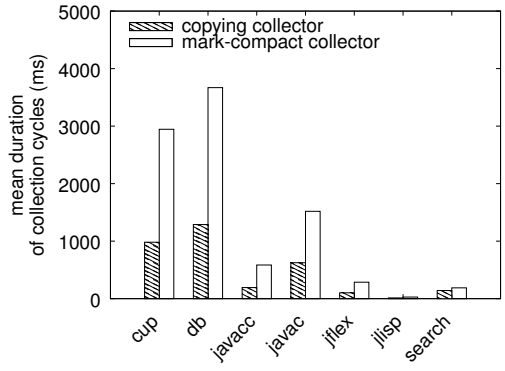



**Figure 11.** Mean duration of collection cycles

Figure 11 shows that a collection cycle of our collector lasts 1.5 to 3 times as long as that of the copying collector. This difference is not surprising since our algorithm requires three passes whereas the copying collector requires only one. On the other hand, the results in Figure 12 reveal that our algorithm reduces the relative duration of garbage collection activities by a factor of 1.1 to 4.5 (1.7 on average). In conclusion, and despite longer collection cycles, our collector, provided with the same amount of memory as the copying collector, increases the overall efficiency of garbage collection, i.e. it collects more garbage per time unit.

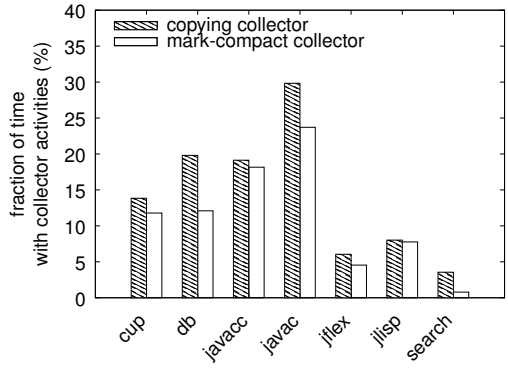



**Figure 12.** Collector activity (% of application execution time)

## 5. Related Work

This section summarizes related work in two distinct areas. First, it gives an overview of real-time compacting collectors without semispaces. Second, it presents related architectures for hardware-supported garbage collection.

### 5.1 Compacting Collectors without Semispaces

In the past decade, researchers aimed at developing real-time compacting garbage collectors that use less memory than semispace-based copying collectors. The corresponding algorithms mainly differ in the method they use for incremental compaction.

A number of algorithms known in that context actually elude the compaction problem. Instead of moving objects to close the holes left by dead objects, they compose objects of constant-sized blocks that are always left in their place. For example, Siebert realized a real-time Java system based on this principle [24]. Approaches like this, however, do not actually solve the fragmentation problem. Instead, they merely trade external for internal fragmentation. Apart from that, they also cause significant runtime overhead since applications must follow linked lists or tree structures for every single object and array access.

Other algorithms only compact a small region of the heap at a time in order to reduce the pauses caused by compaction. This approach has first been proposed in [11] and was later extended for parallel compaction [4] or implemented by taking advantage of virtual memory page protection for concurrent pointer update operations [19]. Unfortunately, the basic idea has never been implemented, and the other two proposals must, every now and then, compact the entire heap in a stop-the-world fashion.

$MC^2$[22] is an incremental generational copying algorithm based on Mark-Copy [21]. It allows heap compaction without semispaces. To achieve this, $MC^2$ splits the old generation into a fixed number of equally-sized windows. Each major collection cycle consists of an incremental mark phase and several stop-the-world copy steps. At each copy step, the live content of one or more source windows is copied into a single free target window. Pointer updates are performed with window-specific remembered sets built during the mark phase. $MC^2$ is able to limit the size of the remembered sets by combining them with card-marking techniques. Experimental results show maximum pause times in the order of 40 ms. However, because the algorithm guarantees an upper bound on the space overhead, it cannot predictably limit the duration of pauses caused by the collector.

Bacon et al. [2] propose a mostly non-moving real-time collector. To bound the duration of garbage collection pauses, they split large objects into so-called arraylets. For incremental compaction, they rely on Brooks-style indirections [5]. Experimental results show memory overheads of 1.2 to 2.5 and runtime overheads in the order of 40%. On a 500 MHz processor, they managed to limit garbage collection pause times to a maximum of 6 ms.

The algorithm that most closely resembles the work presented in this paper has been proposed by Larose and Feeley [7, 12]. Like our algorithm, their algorithm realizes fine-grained Steele-style compaction by means of conditional address generation and object handles. In contrast to our work, however, they have to realize all the synchronization mechanisms by compiler-inserted code sequences. As a second difference, they use a fixed-size handle space that must be configured to 30% of the heap for worst-case scenarios. Third, objects are always accessed through handles, i.e. handles and objects are always allocated and reclaimed at the same time, respectively. Finally, their handles are exclusively used as indirections, i.e. they cannot be used as tracing queue entries. Instead their algorithm requires a supplementary field in each object which further increases the effective memory overhead. Unfortunately, however, experimental results show prohibitive runtime overheads, ranging from factors of 1.8 up to 8 (3 on average).

### 5.2 Hardware-Supported Garbage Collection

Hardware support for garbage collection has been introduced by language-directed architectures in the 1980s. Examples include processors specialized for LISP (e. g. Symbolics [16]) or Smalltalk (e. g. Mushroom [26]). 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 [23]. 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 [18].

Recently, triggered by the success of Java, various architectures with native Java bytecode execution have been proposed. Surprisingly, they offer little to no support for garbage collection. The PicoJava specification [25], for example, offers some basic support for hardware write barriers and for indirections via handles, but has never been realized in actual hardware. Komodo, another approach, focuses on hardware-supported multithreading [20] and realizes garbage collection by means of a dedicated software thread, supported by a microcoded write barrier. Unfortunately, however, Komodo's garbage collector suffers from enormous amounts of memory and runtime overhead and neglects the problem of incremental root set processing and incremental compaction.

## 6. Conclusions

In this paper, we have introduced a novel garbage collection algorithm and presented an efficient hardware-supported implementation. The algorithm combines the incremental compaction feature of copying collectors with the memory efficiency of mark-sweep collectors.

Thanks to the abstraction provided by an object-based architecture, we have been able to implement our algorithm for a garbage collection coprocessor that operates completely in parallel to applications executed on a main processor. Furthermore, all the synchronization mechanisms required by our algorithm, while potentially expensive on stock hardware, could be realized by relatively simple hardware and without any software support. These mechanisms include read barrier checking and read barrier fault handling, fine-grained incremental compaction, and indirections via handles.

Measurements of Java programs on an FPGA-based prototype show that our new collector outperforms a corresponding copying collector in every respect. It causes less runtime overhead, achieves real-time behavior with smaller heap sizes, and is generally more efficient than the copying collector, i.e. it collects the same amount of garbage in less time.

## References


[1] Altera. *STRATIX II Device Family Data Sheet*, Apr. 2006.

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

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

[4] O. Ben-Yitzhak, I. Goft, E. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In D. Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.

[5] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, Aug. 1984. ACM Press.

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

[7] D. Dubé, M. Feeley, and M. Serrano. Un GC temps réel semi-compactant. *Journées Francophones des Langages Applicatifs*, pages 165–181, Jan. 1996.

[8] B. K. Haddon and W. M. Waite. A compaction procedure for variable length storage elements. *Computer Journal*, 10:162–165, Aug. 1967.

[9] R. Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Oct. 1998. ACM Press.

[10] H. B. M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):25–30, July 1979.

[11] B. Lang and F. Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *ACM SIGPLAN Notices*, pages 253–263. ACM Press, 1987.

[12] M. Larose and M. Feeley. A compacting incremental collector and its performance in a production quality compiler. In Jones [9], pages 1–9.

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

[14] 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.

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

[16] 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.

[17] K. D. Nilsen. Cost-effective hardware-assisted real-time garbage collection. In *Workshop on Language, Compiler, and Tool Support for Real-Time Systems, PLDI94*, June 1994.

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

[19] Y. Ossia, O. Ben-Yitzhak, and M. Segal. Mostly concurrent compaction for mark-sweep GC. In A. Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, Oct. 2004. ACM Press.

[20] 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.

[21] N. Sachindran and E. Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, Nov. 2003. ACM Press.

[22] N. Sachindran, J. E. B. Moss, and E. D. Berger. $MC^2$: High-performance garbage collection for memory-constrained environments. In *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, Oct. 2004. ACM Press.

[23] 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.

[24] F. Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [9], pages 130–137.

[25] Sun Microsystems. *picoJava-II Programmer's Reference Manual*, Mar. 1999.

[26] I. W. Williams and M. I. Wolczko. An object-based memory architecture. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice (Proceedings of the Fourth International Workshop on Persistent Object Systems)*, pages 114–130, Martha's Vineyard, MA, Sept. 1990. Morgan Kaufman.