A Novel Processor Architecture With Exact Tag-Free Pointers

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

Abstract

Garbage collection has become a major cornerstone of almost any modern object-oriented programming language, as it considerably increases programmer productivity and software quality. Along with the success of Java, the benefits of garbage collection advance to the area of embedded and real-time systems, thus posing new challenges to processor architects.
This paper presents a novel RISC processor architecture that introduces the robustness stemming from garbage collection at the machine-code level. This is accomplished by an object-based architecture that maintains the invariant that (1) every pointer can be exactly identified, and (2) every pointer value is either null or uniquely associated with an existing object. The basic idea for ensuring this is to strictly separate pointers from ordinary data, thereby allowing for tag-free pointer identification.
The architecture provides the basis for garbage collection in hardware that closely cooperates with the processor. The advantages of this configuration include low garbage collection overhead, low-cost synchronization of collector and application program, high robustness and hard real-time capabilities.
A pipelined RISC processor conforming to the proposed architecture as well as a micro-coded hardware garbage collector have been realized on a single programmable logic device. A native Java bytecode compiler developed for the architecture facilitates the execution of large existing programs. Performance measurements on the prototype show that, despite the required synchronization, the real-time hardware garbage collector is significantly more efficient than a non-incremental "stop-the-world" software garbage collector.

1 Introduction

Today, automatic dynamic memory management, also known as garbage collection, is widely accepted as an indispensable method to control software complexity. For this reason, most object-oriented languages such as Smalltalk, Eiffel and Java are supported by garbage collection. There are, however, application areas such as embedded, real-time, and safety-critical systems for which the benefits of garbage collection are still considered an unaffordable luxury.

In this section, a summary of the most commonly used garbage collection methods is given. It concentrates on the problems of contemporary software-based solutions in order to motivate the architectural support for garbage collection that is presented in this paper.

Workshop on Application Specific Processors, December 2nd, 2003, San Diego

Basic Algorithms

The basic algorithms for garbage collection are reference counting, copying and mark-sweep collection [9]. Reference counting methods maintain a counter for each object that holds the number of references pointing to the object. Objects with a counter of zero can be reclaimed. Reference counting, however, fails to free cyclic garbage. To avoid the cost of updating reference counters and to overcome the problem with cycles, mark-sweep and copying algorithms, also referred to as tracing algorithms, are applied. They trace memory starting with a set of roots, usually consisting of processor registers and the program stack. While 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. Only one semispace is used at any given time. During collection, all reachable objects are copied 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 since the cost of garbage collection is proportional to the amount of reachable objects rather than the size of the entire heap.

Traditional implementations of these algorithms suspend application processing for the entire duration of a garbage collection cycle. They usually cause long and unpredictable pause times and are not applicable to interactive systems or real-time environments.

Incremental Methods

In order to improve the interactive response of garbage-collected systems, incremental garbage collection techniques are applied. They allow application processing to continue while garbage collection is performed. In this context, the application program is usually referred to as the "mutator" since it changes the heap behind the collector's back. For illustration purposes, incremental garbage collection is often described by Dijkstra's tricolor abstraction [4]: *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, white objects are reclaimed. 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.

Usually, either read or write barrier methods are employed to prevent the mutator from disrupting the garbage collector. *Read barriers* ensure that the mutator never sees a white object. Whenever a pointer to a white object is accessed, the object is immediately visited by the collector. In contrast, *write barriers* make sure that black objects turn grey as soon as a pointer to a white object is installed. In order to realize read or write barriers, the compiler must emit a few instructions before each pointer load or before each pointer store, respectively.

Many incremental garbage collection algorithms are described as real-time because garbage collection pauses are, in the majority of cases, too short to be noticed by a user. Hard real-time systems, however, require correct responses to environmental changes within a specified amount of time. As software-based incremental algorithms usually have to 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 (e.g. due to expensive object access, write barriers, synchronization points, root set copies) as well as memory space overhead (e.g. due to auxiliary data structures, fragmentation) [15].

Pointer Finding

All garbage collection methods face the problem of "pointer finding" or "pointer identification". If pointers cannot be unambiguously distinguished from non-pointers, only conservative garbage collection may be employed. This means that any bit pattern that looks like a pointer has to be considered a pointer in order to avoid freeing memory that is still in use. The conservative approach, however, suffers from a number of drawbacks. A value mistakenly identified as a pointer may 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 may not move objects as this necessitates the update of pointers, and updating some data erroneously taken as a pointer can cause disastrous effects. For this reason, conservatively collected heaps are subject to fragmentation as long as no costly measures are taken (e.g. the use of handles), and copying collectors or other compacting collectors such as mark-sweep-compact cannot be used unless exact information about the location of each pointer is available.

In order to realize exact (i.e. non-conservative) garbage collection (also known as precise or accurate garbage collection), many implementations spend a good deal of effort in searching and exactly identifying pointers. In the case of many object-oriented languages, pointers in heap objects can be identified by means of type descriptors that are usually available in each object as they are needed for dynamic binding and runtime type 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 generally considered prohibitively high. For this reason, most software-based solutions rely on the compiler to emit tables describing all pointer locations in the stack and in registers. A set of tables is constructed for each program point where a collection might occur (gc-points). Examples for this approach can be found in [5] for a Modula-3 compiler and in [1] for a Java Virtual Machine.

Software-based methods for exact pointer identification, however, share a number of disadvantages. First of all, the tables needed to describe all pointer locations at all gc-points usually result in considerable code size blow-up unless the tables are compressed [1, 5]. Furthermore, multi-threaded and 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.

The rest of the paper is organized as follows. Section 2 gives an overview of related architectures. Section 3 defines a novel processor architecture, and Section 4 outlines an efficient implementation thereof. In Section 5, the implementation is enhanced by a hardware garbage collector. Finally, Section 6 presents experimental results, and Section 7 provides a conclusion.

2 Related Work

This section highlights representative examples of architectures that provide support for exact pointer identification, garbage collection, or both.

Known since 1966, capability-based architectures use capabilities instead of ordinary pointers to refer to memory. Examples for capability-based computers include the Plessey System 250, the IBM System/38, the SWARD machine and the Intel iAPX 432 [10, 12]. Capabilities are unique object identifiers that provide a single mechanism to address both main memory and secondary storage. A capability does not contain the physical address of an object. Instead, it is used to locate a descriptor that specifies the physical location and the size of the object. Capabilities are either tagged or kept separate from ordinary data in order to guarantee that they are never modified by users. In this way, capability-based systems provided for "exact capability identification" long before garbage collection became a major topic of interest.

The Intel iAPX 432 [10] is an interesting example for a capability-based processor. Capabilities are associated with objects in a two-step mapping process. An object consists of two parts: a data part for scalars and an access part for capabilities. For each object, there is a unique entry in an object table that describes the location, the size, and the state of the object. Due to its architectural complexity, however, the iAPX 432 suffered from very poor performance [3] and has never been a commercial success.

During the eighties, various architectures in support of particular programming languages have evolved. Examples include the SOAR (Smalltalk On A RISC) [16], SPUR (Symbolic Processing Using RISCs) [8], and Symbolics [11] architectures. They distinguish different types of data (including pointers) by augmenting every memory word with a tag field and provide hardware support for read or write barriers [19].

More recently, Williams and Wolczko [17] suggest an object-based memory architecture for Smalltalk. Similar to capability-based systems, object identifiers do not directly refer to the

object's address in memory but are used to index a multi-level object table. Object identifiers are distinguished from ordinary data by means of tags.

Gehringer and Chang [6] propose a cache coprocessor for conventional architectures that is intended to avoid unnecessary memory traffic. The coprocessor allocates objects in the cache without fetching garbage from memory and removes dead objects from the cache before they are ever written to memory. Since pointers are not differentiated from ordinary data, a conservative collection algorithm is applied.

Nilsen and Schmidt describe a garbage-collected memory module for hard real-time applications in a series of publications [e. g. 13, 14]. It is mainly composed of two memory banks and a local processor dedicated to garbage collection. Memory words are tagged in order to provide for pointer identification. Every memory bank is supported by a so-called object space manager (OSM) that is used to associate each memory access with the start address of the corresponding object. The start address is required to access the object header that contains vital information such as the size and the state of the object. Due to its complexity, the OSM has to be realized as a separate ASIC and is expected to consume almost as much chip area as the actual memory device itself. In respect of real-time capabilities, the authors report worst-case latencies for all memory load, store, and allocate operations in the order of 1 µs, with rare delays of typically 500 µs at the start of a garbage collection cycle.

Wise et al. [18] present a memory module realizing a hardware reference-counting heap. The module consists of the actual memory, a reference count memory, and a tag memory. The module automatically maintains a reference counter for each object. In order to reclaim cyclic data structures, the reference count collector is regularly assisted by a mark-sweep collector.

To conclude this section, the following observations should be noted: First, many architectures are dedicated to special programming languages. Second, with the exception of the expensive garbage-collected memory module, none of the described architectures targets hard real-time applications. Third, apart from early capability-based computers, all architectures for exact garbage collection use tags to identify pointers.

3 Architecture Definition

3.1 Design Goals

The architecture defined in this section is motivated by the following design goals: (1) ensure exact pointers without tags, (2) apply a general-purpose, RISC-style instruction set architecture for efficient implementation, and (3) do not rely on indivisible operations with execution times exceeding a few clock cycles.

3.2 Definition of Terms

Throughout the rest of this paper, the following definition of terms is presumed. *word:* a unit of data that can be moved to and from memory by a single processor instruction; *object:* a compound of memory words, whereby every word exclusively belongs to a single object; *pointer:* a word used to refer to an object; *null:* a unique pointer value used to refer to no object. The term "reference" is used as a synonym for pointer.

3.3 System Invariant

In order to ensure exact pointers, the architecture maintains the *system invariant* detailed below:

§1 every memory word or register is identified to be a pointer or not

§2 each pointer value is either null or uniquely associated with an existing object

3.4 Register Model

The architecture provides separate data register and pointer register sets (Figure 1). Data registers are used as general-purpose registers whereas pointer registers are exclusively used for referring to objects in memory. In order to comply with the system invariant, it must not be possible to write arbitrary values to pointer registers or to perform arithmetic operations on pointer registers.

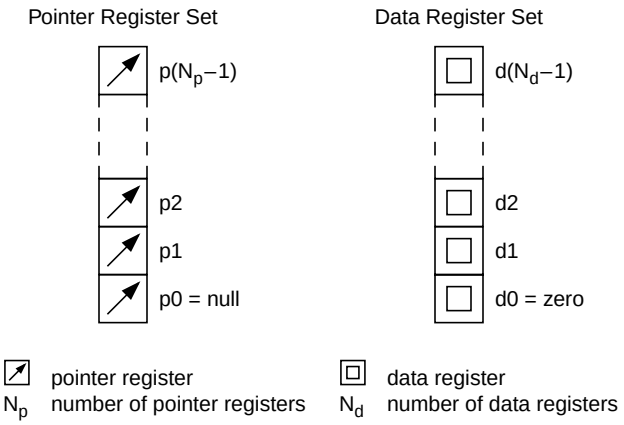


Figure 1 Register model

3.5 Object Model

The memory model of the processor architecture is object-based. An object is composed of a separate data area and a separate pointer area (Figure 2). The number of data words in the data area is referred to as the object's δ-*attribute*, the number of pointers in the pointer area as the object's π-*attribute*, respectively. The size of an object as described by the attributes is determined when the object is created and cannot be altered thereafter.

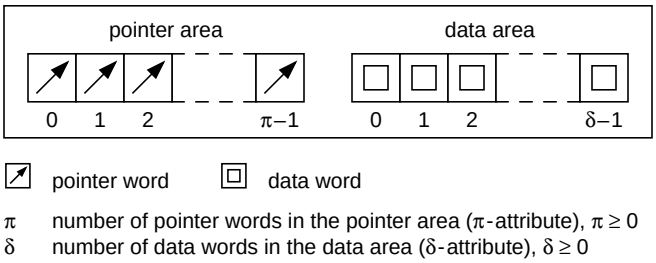


Figure 2 Object model

3.6 Instruction Set

The architecture definition comprises pointer-related instructions only, 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

Dynamic memory management is entirely abstracted by the processor architecture. A special *allocate object* instruction is used for creating a new object and for associating 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 object reference becomes visible to the program.

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 proposed architecture relies on garbage collection at the processor level.

Load and Store

Load and store instructions are used to access words inside an object. Different load and store instructions for accessing pointer and data words are provided: the *load data* and *store data* instructions exclusively move data words between an object's data area and data registers, while the *load pointer* and *store pointer* instructions exclusively move pointers between an object's pointer area and pointer registers, respectively.

Load and store instructions identify the memory word to be accessed by means of a pointer register holding the reference to the object and a positive integer index. To compute this index, an implementation may provide different "indexing modes" (the equivalent to "addressing modes" on conventional architectures) using constant offsets, 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 may be possible to access pointers by means of out-of-range data load instructions or vice versa. For this reason, the architecture has to perform bounds checks. If a bounds check fails, the load or store instruction is not permitted to complete, and an *index out of bounds exception* is raised. For similar reasons, the attempt to access memory by means of a pointer register holding null is cancelled by a *null pointer exception*.

Query Object Attributes

The attributes of an object can be queried by means of two *query attribute* instructions.

Copy and Compare Pointers

In contrast to the multitude of register-to-register instructions that may be provided 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.

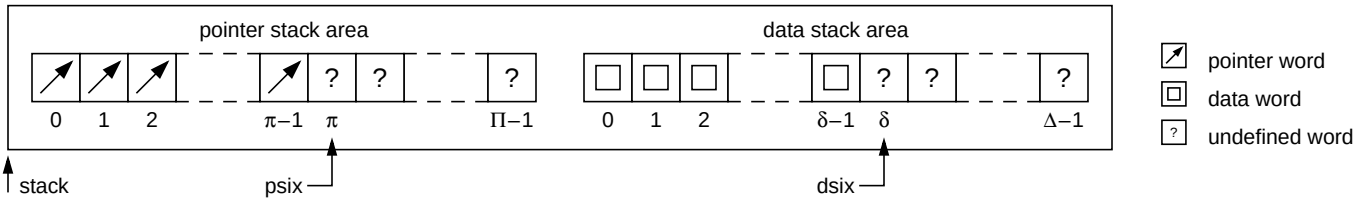
3.7 Program Stack

Because of the unstructured and highly dynamic nature of program stacks, they traditionally constitute the greatest challenge with respect to pointer identification. For this reason, the realization of program stacks within the scope of the proposed architecture deserves special attention.

The stack object, like every object, is composed of a data area and a pointer area and can thus be thought of as two separate stacks (Figure 3). A pointer register is reserved to hold the reference to the stack object. In each of the two areas, a stack index is used to divide the respective area into the actual stack and an unoccupied area, whereby the stack index refers to the first unoccupied location. The two stack indices are designated as *data stack index (dsix)* and *pointer stack index (psix)*, and each is held in a data register reserved for this purpose.

However, if the stack object is treated like an ordinary object, the system cannot differentiate whether a pointer belongs to the actual pointer stack or to the unoccupied pointer area. Since every word in the pointer stack area is identified as a pointer, there may be many pointers within the unoccupied area that point to objects that are no longer needed. 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, resulting in unpredictable amounts of floating garbage.

One possible solution to this problem consists in explicitly storing null to 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



stack pointer
stack reserved pointer register, e.g. p1 = stack

stack indices
psix pointer stack index, reserved data register, e.g. d1 = psix
dsix data stack index, reserved data register, e.g. d2 = dsix

stack attributes
Π max. number of pointer words in pointer stack area
Δ max. number of data words in data stack area
π number of occupied words in pointer stack area, $0 \le \pi \le \Pi$
δ number of occupied words in data stack area, $0 \le \delta \le \Delta$

Figure 3 Stack object

many pointers are to be removed from the stack at the same time (e.g. the deallocation of a stack frame at the end of a subroutine).

Due to its constant size, an ordinary object is actually not adequate to realize a program stack. As a better approach, the architecture takes the dynamic size of a stack into account. This can be accomplished by describing the program stack object by two pairs of attributes, one pair (π, δ) for the actual stack size and a second pair (Π, Δ) for the maximum stack size. In this way, the π-attribute is equivalent to the value of *psix*, and the δ-attribute is equivalent to the value of *dsix*. The stack attributes Π and Δ are held in system registers that are not visible to user programs. In respect to pointer identification and the system invariant, only pointers at indices below π will be considered pointers.

Memory locations inside the stack are accessed by ordinary load and store instructions. Values can be removed from a stack by decreasing the value of the respective stack index by means of ordinary arithmetic instructions. There is, however, a problem related to pushing pointer values onto the pointer stack. If it is permitted to store the pointer before the pointer stack index is incremented, §1 of the system invariant will be offended in between the two instructions, as the pointer just stored is not identified as a pointer. If, on the other hand, the pointer stack index is incremented prior to storing the pointer, §2 will be offended in between the two instructions, since a word identified as a pointer does not contain a valid pointer. This problem is solved by providing a pointer store instruction (or, alternatively, an indexing mode that can be used with a pointer store instruction) that both stores a pointer at the first unoccupied pointer stack location and increments the pointer stack index in an indivisible fashion (*push pointer*). Furthermore, this must be the only instruction permitted to increment the pointer stack index. Any attempt to increment the pointer stack index by means of other instructions results in a *psix increment exception*.

3.8 Special Objects

Constant Objects

Assuming the instruction set introduced so far, the only way to access memory is through pointers, and the only instruction to actually create new pointers is the allocate instruction. However, it should also be possible to access constant data that exists as part of the program code before a program is started. Examples for constant data include constant strings and compiler generated structures such as branch tables and type descriptors.

One way to accomplish this is to describe program code including constant data as a large, immutable program object with no pointer area. But this approach suffers from two major drawbacks: First, constant data and ordinary objects cannot be accessed in a uniform way. For example, accessing a constant string within the program object will differ from accessing an ordinary object representing a dynamically created string. Second, it will not be possible to use pointers to build up constant data structures such as tables containing references.

To overcome these shortcomings, *constant objects* are introduced. A constant object is an immutable object stored as part of the program code or as part of a special constant memory area. There is, however, a problem in creating pointers to constant objects (*constant pointers*), as this includes the conversion of a constant object's address into a pointer, thus substantially threatening the system invariant. For this reason, operations on constant pointers are restricted in that they may only be used for read accesses, and that accesses through constant pointers are confined to the area intended for constant objects. As a consequence, the pointer area of a constant object can contain constant pointers only. If the rules stated above are offended at runtime, appropriate exceptions will be raised.

A pointer to a constant object is created by the *create constant pointer* instruction. Constant objects will be distinguished from ordinary objects by means of a φ-attribute that is introduced in order to distinguish different kinds of objects.

Static Objects

In many systems, separate program stacks are used for different operating modes (e.g. user mode, supervisor mode). Moreover, multithreaded environments require separate program stacks for each thread of execution. All these stacks are usually arranged apart from a garbage-collected heap and managed by an operating system.

In order to permit manually managed memory areas outside of the heap, the presented architecture proposes *static objects*. Static objects can only be created in supervisor mode and are laid out in a dedicated memory area. Static objects are identified by the φ-attribute, and pointers to static objects (*static pointers*) may never become visible to user programs.

Uninitialized Objects

In order to satisfy the system invariant, each pointer in a newly created object has to be initialized before the corresponding allocate instruction is permitted to complete. Therefore, the execution time for the allocate instruction is not bounded by a small constant. This is not acceptable for hard real-time applications.

In order to provide for interruptible allocate instructions, *uninitialized objects* (strictly speaking *incompletely initialized objects*) are introduced. Uninitialized objects will be created if and only if allocate instructions are suspended due to interrupts. In order to observe the system invariant, pointers to uninitialized objects may never be dereferenced. Like static and constant objects, uninitialized objects are identified by the φ-attribute.

3.9 Summary

Figure 4 summarizes the pointer-related instructions defined by the architecture and categorizes them on whether they read, write, or dereference pointer registers. The register that is read, written, or dereferenced in each case is shown in bold face.

The processor architecture supports four different kinds of objects: dynamic (i.e. ordinary), constant, static, and uninitialized. The φ-attribute is used to distinguish between object kinds and may hold a value taken from the set *{dyn, const, stat, uini}*. Dynamic and uninitialized objects reside in the heap, static objects in a static area, and constant objects in the area designed for program code and constant data. Since static and uninitialized objects are restricted to supervisor mode, they are referred to as *system objects*.

read pointer registers		
store pointer	sp	py[index] := **pz**
copy pointer	cpp	px := **pz**
compare pointers	cmpp	dx := **py**,**pz**

write pointer registers		
allocate object	alc	**px** := dy,dz
load pointer	lp	**px** := py[index]
copy pointer	cpp	**px** := pz
create constant pointer	ccp	**px** := dy

dereference pointer registers		
load data	ld	dx := **py**[index]
load pointer	lp	px := **py**[index]
store data	sd	**py**[index] := dz
store pointer	sp	**py**[index] := pz
query π-attribute	pattr	dx := **py**
query δ-attribute	dattr	dx := **py**

dx, dy, dz *data registers*
px, py, pz *pointer registers*
index *indexing mode expression*

Figure 4 Classification of pointer-related instructions

Concerning garbage collection, the four object kinds can be defined on how they are to be treated by a compacting (i.e. moving) garbage collector. Ordinary dynamic objects have to be scanned for pointers and must be moved for compaction. Static objects have to be scanned for pointers, too, but they must not be moved. In contrast, uninitialized objects have to be moved during compaction, but they must not be scanned since they may contain invalid pointers. Finally, constant objects must neither be scanned nor moved by a garbage collector. It is remarkable that, although motivated by rather practical reasons, the four identified object kinds completely fill the space spanned by the "scanned" and the "moved" properties (Figure 5).

object kind			**scanned?**	**moved?**
user objects	dynamic	(φ = dyn)	yes	yes
	constant	(φ = const)	no	no
system objects	static	(φ = stat)	yes	no
	uninitialized	(φ = uini)	no	yes

Figure 5 Object kinds

4 Processor Implementation

This section presents the implementation of a processor conforming to the proposed architecture that has been realized in order to demonstrate that the architecture can be realized in an efficient way. The processor can be operated without a hardware garbage collector. Extensions in support of concurrent hardware garbage collection are described in the next section.

4.1 Object Layout

For the implementation described in the following, a word size of 32 bit is assumed. Memory is byte-addressable in order to provide for byte and half-word accesses within the data area. The implementation slightly changes the definition of π and δ as they describe the number of bytes rather than the number of words in the respective area.

An object in memory is composed of two header words holding the object's attributes followed by the pointer area and the data area (Figure 6). For efficiency reasons, objects are double-word aligned. As the address of an object is a multiple of eight and π is a multiple of four, three bits in a pointer word and two bits in the word holding π can be used for tagging purposes during garbage collection and for encoding the φ-attribute.

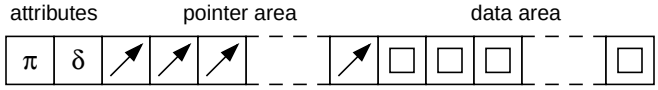


Figure 6 Object layout

4.2 Processor Pipeline

The implementation is based on a straightforward pipelined RISC design [7] that is extended to efficiently handle objects and attributes (Figure 7). The register set comprises 16 data registers and 16 pointer registers. Instructions targeting data registers are processed by the ALU, instructions targeting pointer registers (e. g. allocate) are processed by the PGU (pointer generation unit). The AGU (address generation unit) generates addresses for accessing the cache in the memory stage. Because this cache serves the same purpose as a data cache in conventional architectures, it is designated as "data cache" even though it contains ordinary data and pointers.

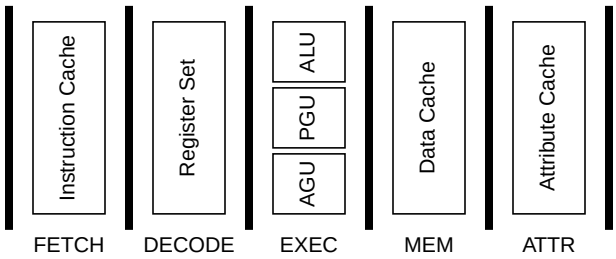


Figure 7 Pipeline

Before a pointer can be used to access an object, the object's attributes are needed for range checking and, in case of a data access, for address generation as well. For this reason, every pointer register is supplemented by attribute registers. Whenever a pointer register contains a non-null value, the corresponding attribute registers hold the attributes of the object the pointer register is referring to. In this way, the effort for dereferencing a pointer register is as low as address generation in conventional architectures. Range checking in itself constitutes no performance penalty since the required comparison is performed in parallel with address generation.

Whenever a pointer is loaded from memory, the associated attribute registers must be loaded as well. This is accomplished by means of an additional pipeline stage after the usual memory stage. This stage is designated as the attribute stage and features an attribute cache in order to allow for attribute accesses without performance penalty in the common case.

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

5 Garbage Collector Implementation

5.1 System Overview

The hardware garbage collector is realized as a separate, microcoded coprocessor (Figure 8). The memory controller provides separate ports for both the main processor and the garbage collector. Due to this configuration, the inherently non-local behavior of the garbage collector is not disrupting cache locality.

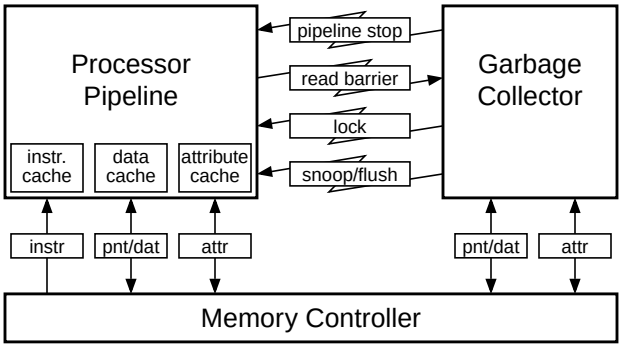


Figure 8 System overview

5.2 Garbage Collection Algorithm

The implemented garbage collection algorithm is derived from the incremental copying algorithm introduced by Baker [2]. Whenever an object is evacuated from fromspace to tospace, only space is reserved in tospace instead of actually copying the object. In doing so, the π-attribute of the object is tagged and the δ-attribute is saved to the tospace copy and then overwritten with a forwarding pointer (fp) (Figure 9). In tospace, the field for the π-attribute is initialized with a backlink (bl) to the fromspace original. When the collector traverses the yet empty object in tospace, each pointer in the corresponding fromspace object is scanned by either evacuating the referenced object or, if the object has already been evacuated, by reading the forwarding pointer. The resulting tospace pointer is written to the tospace object. Subsequently, the data area is copied and the tospace object is blackened by replacing the tagged backlink with the untagged π-attribute.

The garbage collector starts a collection cycle whenever the amount of available memory falls below an adjustable threshold. For real-time behavior, some memory headroom is required to make sure that the collection cycle terminates before the mutator runs out of memory.

5.3 Synchronization and Concurrent Object Copying

Synchronization of processor and garbage collector occurs on different levels (Figure 8). The garbage collector ensures cache coherence by snooping the data and attribute caches and by flushing the according cache line in case of a snoop hit. Exclusive access to objects is enforced by a cache line locking mechanism.

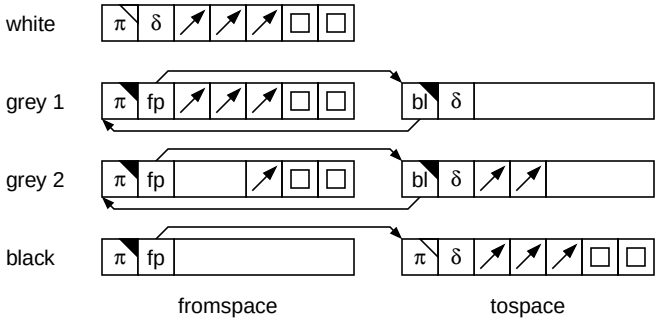


Figure 9 Object states

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 is able to stop the processor pipeline for root set scanning. Since stacks are treated as static objects that are incrementally processed, only the pointer registers need to be scanned atomically. For any of the described synchronization mechanisms, it is guaranteed by design that the duration of pauses they may cause never exceeds a small constant in the order of a couple of hundred clock cycles.

In order to support concurrent object copying, a backlink entry is added to every pointer register and to every attribute cache line. If the attributes of a grey object are loaded from memory, the corresponding backlink is loaded as well. Whenever the processor is about to access a grey object, the AGU determines whether the tospace pointer or the backlink is to be used for address generation. In this way, either the fromspace original or the tospace copy of the corresponding object is accessed.

6 Experimental Results

Processor and hardware garbage collector have been described in VHDL and realized by means of a single advanced programmable logic device (Altera APEX 20K1000C). The garbage collector occupies approximately 20% of the chip area. The prototype operates at 25 MHz and features an 8K instruction cache, an 8K data cache and a 2K attribute cache. Furthermore, an experimental computer system based on the processor has been built up. Standard SDRAM modules are used for main memory. A static Java bytecode compiler and a subset of the Java class libraries supporting text-based applications have been implemented in order to facilitate the execution of representative programs.

Figure 10 shows the results from running several “real-world” applications on the prototype (*jflex:* a scanner generator, generating a scanner for the Java language; *cup:* a parser generator, generating a parser for the Java language; *javac:* Sun’s Java compiler, compiling itself; *jlisp:* a simple Lisp interpreter, solving a puzzle). Each application is run with different semispace sizes, ranging from the smallest possible size (100%) to a virtually infinite size. To provide a basis for comparison, all test cases are executed with a non-incremental software collector (sw) and with the concurrent hardware garbage collector (hw). Apart from the type of garbage collector, the environment is exactly the same in both cases. In Figure 10, a table entry contains the execution time in seconds, the garbage collection time overhead in relation to the case without any garbage collection activities (infinite semispace

		100%	110%	125%	150%	200%	infinite
jflex	**sw**	43.4 (+20.8%)	40.7 (+13.2%)	39.4 (+9.5%)	38.2 (+6.2%)	37.4 (+3.9%)	36.0 (+0.0%)
100% = 2043K	**hw**	38.6 (+7.4%) ✗	37.0 (+2.8%) ✗	36.3 (+0.9%) ✗	36.1 (+0.3%) ✓	36.0 (+0.1%) ✓	36.0 (+0.0%) ✓
cup	**sw**	576.7 (+766.8%)	128.4 (+92.9%)	95.9 (+44.2%)	83.5 (+25.5%)	75.5 (+13.4%)	66.5 (+0.0%)
100% = 8707K	**hw**	406.1 (+510.4%) ✗	107.8 (+62.0%) ✗	85.3 (+28.2%) ✗	72.5 (+9.0%) ✗	67.1 (+0.8%) ✓	66.5 (+0.0%) ✓
javac	**sw**	114.8 (+133.0%)	69.6 (+41.3%)	62.1 (+26.1%)	57.5 (+17.1%)	54.1 (+9.9%)	49.3 (+0.0%)
100% = 5639K	**hw**	93.3 (+89.4%) ✗	57.3 (+16.3%) ✗	51.5 (+4.6%) ✗	50.3 (+2.2%) ✓	49.8 (+1.2%) ✓	49.3 (+0.0%) ✓
jlisp	**sw**	108.5 (+14.4%)	105.1 (+10.8%)	102.3 (+7.8%)	100.0 (+5.3%)	98.0 (+3.2%)	94.9 (+0.0%)
100% = 129K	**hw**	96.5 (+1.7%) ✗	95.9 (+1.0%) ✗	95.5 (+0.6%) ✓	95.2 (+0.3%) ✓	94.9 (+0.0%) ✓	94.9 (+0.0%) ✓

Figure 10 Measurement results (application execution time in seconds vs. relative semispace size)

size), and, in case of the hardware collector, a symbol that indicates whether pauses occurred due to mutator starvation ("✗" for pauses, "✓" for real-time behavior).

All applications that have been examined make extensive use of heap allocated memory. In the case of *jflex* and *jlisp*, most objects die relatively young. This is 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, depending on the application, real-time behavior can be achieved by semispace sizes that are 25% to 100% larger than the required minimum. Under relevant operating conditions, the overhead caused by the hardware collector can be as little as a few percent or less. In any case, the real-time hardware collector is significantly more efficient than the non-incremental "stop-the-world" software collector.

7 Conclusions and Further Work

This paper introduces a novel RISC processor architecture that hides the details of dynamic memory management at the processor level and ensures exact pointers without the need for tags in order to provide a basis for hardware-supported real-time garbage collection. An efficient implementation of a processor conforming to the proposed architecture and a copying hardware garbage collector are presented. Performance measurements on the prototype show that the real-time hardware garbage collector will almost have no noticeable effect on application programs if there is a reasonable amount of memory headroom available.

In the future, it is planned to explore the potential of generational garbage collection schemes for the presented architecture in order to further reduce the cost of garbage collection with respect to memory and time overhead.

References

[1] O. Agesen, D. Detlefs, E. Moss: "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[2] Henry G. Baker: "List Processing in Real Time on a Serial Computer", *Communications of the ACM, Volume 21, Number 4, pp. 280-294, April 1978.*

[3] R. P. Colwell, E. F. Gehringer, E. D. Jensen: "Performance Effects of Architectural Complexity in the Intel 432", *ACM Transactions on Computer Systems, Vol. 6, No. 3*, 1988.

[4] 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, Vol. 21, No. 11*, 1978.

[5] A. Diwan, E. Moss, R. Hudson: "Compiler Support for Garbage Collection in a Statically Typed Language", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[6] E. F. Gehringer, J. M. Chang: "Hardware-Assisted Memory Management", *OOPSLA'93 Workshop on Memory Management*, 1993.

[7] J. L. Hennessy, D. A. Patterson: "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann Publishers*, 1996.

[8] M. Hill et al.: "Design Decisions in SPUR", *IEEE Computer, Vol. 19, No. 11*, 1986.

[9] R. Jones, R. Lins: "Garbage Collection: Algorithms for Automatic Dynamic Memory Management", *John Wiley & Sons*, 1996.

[10] H. M. Levy: "Capability-Based Systems", *Digital Press*, 1984.

[11] D. A. Moon: "Symbolics Architecture", *IEEE Computer, Vol. 20, No. 1*, 1987.

[12] G. J. Myers: "Advances in Computer Architecture", *John Wiley & Sons*, 1978.

[13] K. D. Nilsen, W. J. Schmidt: "A High-Performance Hardware-Assisted Real-Time Garbage Collection System", *Journal of Programming Languages, Vol. 2, No. 1*, 1994.

[14] W. J. Schmidt: "Issues in the Design and Implementation of a Real-Time Garbage Collection Architecture", *Dissertation, Iowa State University*, 1992.

[15] F. Siebert: "Hard Real-Time Garbage Collection in the Jamaica Virtual Machine", *6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[16] D. Ungar, R. Blau, P. Foley, D. Samples, D. Patterson: "Architecture of SOAR: Smalltalk on a RISC", *11th Annual Symposium on Computer Architecture*, 1984.

[17] I. Williams, M. Wolczko: "An Object-Based Memory Architecture", *Fourth International Workshop on Persistent Object Systems*, 1991.

[18] D. S. Wise, B. Heck, C. Hess, W. Hunt, E. Ost: "Research Demonstration of a Hardware Reference-Counting Heap", *Lisp and Symbolic Computation, Vol. 10, No. 2*, 1997.

[19] B. Zorn: "Barrier Methods for Garbage Collection", *Technical Report CU-CS-494-90, University of Colorado*, 1990.