A True Hardware Read Barrier

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

ABSTRACT

Read barriers synchronize compacting garbage collection and application processing in a simple yet elegant way. Unfortunately, read barrier checks are expensive to implement in software, and even with hardware support, the clustering of read barrier faults irregularly impairs application progress to an unacceptable extent. For this reason, read barriers are often considered unsuitable for hard real-time systems.

In this paper, we introduce a novel hardware read barrier design for an object-based RISC architecture. The design integrates read barrier checking and, for the first time, read barrier fault handling directly into a processor pipeline.

Our system handles read barrier faults within 20 clock cycles on average. Despite fault clustering, all application programs we have run on our prototype show minimum mutator utilizations of more that 55% within arbitrary time intervals of only 1 ms. Thanks to this property, our system facilitates worst case estimates for tasks with very short response times, thereby paving the way for garbage collection in embedded systems with extremely fine-grained real-time requirements.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – *Memory management (garbage collection)*

General Terms Design, Algorithms, Languages, Measurement, Performance, Experimentation

Keywords Real-time garbage collection, object-based processor architecture, read barrier, hardware support

1. INTRODUCTION

In real-time systems, garbage collection must be performed concurrently with application processing, either by interleaving the collector with the application on the same processor, or by running the collector on a separate processor. In any case, it is imperative to prevent applications from interfering with the collector's work and to ensure that objects are not prematurely reclaimed. To accomplish this, either read or write barriers supervise any pointer accessed by an application.

While write barriers are less expensive than read barriers, they do not allow garbage collectors to move objects for compaction. Therefore, they are mainly used with generational or mark-sweep collectors. In contrast, read barriers realize a higher degree of protection and allow the collector to relocate objects. Regrettably, read barriers are considerably more expensive.

The first read barrier was introduced by Baker [1]. Since then, researchers have invested a lot of effort to reduce the cost of read barriers, and this paper follows this tradition.

To start with, Section 2 introduces Baker's original read barrier and discusses more recent related work. Next, Section 3 gives an overview of a system with hardware support for garbage collection, including an object-based RISC processor and a coprocessor for garbage collection. This system serves as the basis for the work described in this paper. Thereafter, Section 4 evaluates the costs of the system's initial read barrier, Section 5 proposes a novel read barrier design, and Section 6 presents measurement results from our prototype. Finally, Section 7 discusses related architectures, and Section 8 provides a conclusion and identifies potential for future work.

2. BAKER'S READ BARRIER

Basically, Baker's algorithm represents a real-time extension to the non-recursive copying algorithm introduced by Cheney [4]. For this reason, we first describe Cheney's algorithm and then proceed to the enhancements added by Baker. Finally, we discuss the problems of Baker's algorithm with respect to real-time performance in general as well as the issues with Baker's read barrier in particular.

2.1 Cheney's Copying Collector

Copying collectors like Cheney's divide the heap into two areas called semispaces. During collection, all objects that are reachable from a set of roots are copied from one semispace *(fromspace)* to the other semispace *(tospace)*. This way, copying collectors inherently compact the heap. At the beginning of a garbage collection cycle, Cheney's collector flips the roles of fromspace and tospace and initializes two pointers called *scan* and *free* to point to the bottom of tospace. Next, it evacuates all objects referenced by the root set from fromspace to tospace (Figure 1, upper diagram, assuming that only A is referenced by the root set). During evacuation, the collector just copies the contents of an object, so the pointers inside the tospace copy still refer to the original objects in fromspace. After evacuation, the garbage collector advances *free* and overwrites the first word in the fromspace object with a forwarding pointer to the tospace copy. Although it is safe to overwrite the contents of evacuated objects in fromspace, Cheney's algorithm requires at least one bit per object to distinguish evacuated objects from objects that have not yet been visited by the collector.

After the collector has evacuated all objects referenced by the root set, it successively scans tospace locations pointed to by *scan*. If the collector encounters a pointer, it checks whether the corresponding object has already been evacuated by the collector. If so, it overwrites that pointer with the forwarding pointer found in the

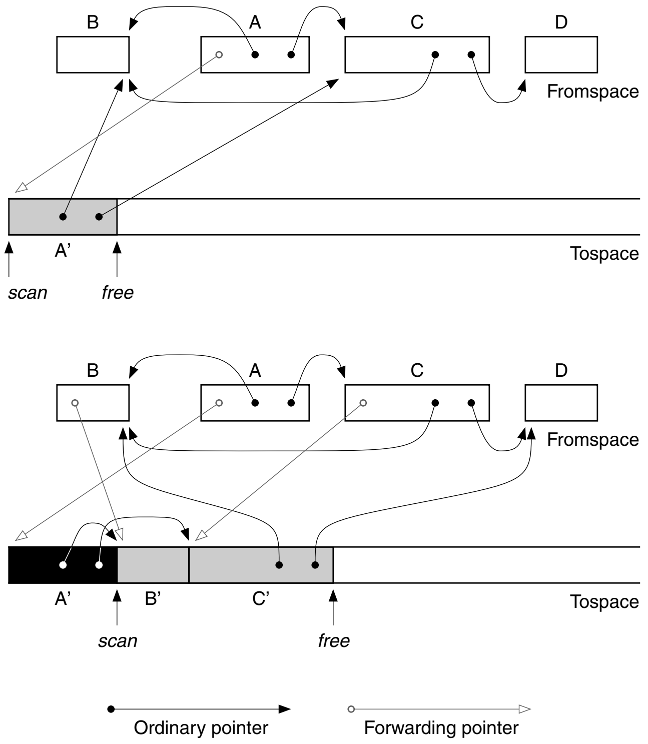


Figure 1: Cheney's non-recursive copying algorithm

fromspace object. If not, it evacuates the corresponding object as described above and updates the pointer to refer to the tospace copy (Figure 1, lower diagram). In this way, the collector consecutively replaces all pointers to fromspace objects with pointers to their tospace copies. The algorithm terminates as soon as *scan* catches up with *free*.

For illustration purposes, object states during garbage collection are often described by Dijkstra's tricolor abstraction [6]. In this abstraction, *black* indicates that the collector has finished with an object for the current garbage collection cycle, *gray* indicates that the collector has not finished with the object or, for some reason, has to visit the object again, and *white* indicates that the object has not been visited by the collector. Applying Dijkstra's tricolor abstraction to Cheney's algorithm, tospace objects below *scan* are black, objects between *scan* and *free* are gray, and unevacuated objects in fromspace are white.

Strictly speaking, the term garbage collection is not at all appropriate to describe copying collectors like Cheney's. Rather than collecting garbage, a copying collector just picks live objects and saves them to tospace. This way, it only collects useful data and effectively ignores any garbage. As a resulting benefit, the cost of copying garbage collection is solely proportional to the amount of live data. In contrast, the cost of mark-sweep collection is proportional to the size of the heap.

2.2 Baker's Extensions

Cheney's collector suspends application processing for the entire duration of a garbage collection cycle. It is therefore referred to as a "stop-the-world" collector and as such unsuitable for interactive systems or real-time applications. To break this restriction, Baker extended Cheney's algorithm and allows the application to proceed during garbage collection. In this context, the application program is usually referred to as the *mutator*, because it changes the graph of objects while the collector is traversing the heap.

According to Baker's proposal, the mutator allocates new objects from the top end of tospace while the collector compacts surviving objects into its bottom end. As a consequence, all new objects are allocated black. In order to prevent new objects from overwriting surviving objects, an appropriate scheduling mechanism has to ensure that the garbage collector completes a cycle before newly allocated objects reach the *free* pointer.

Baker's algorithm faces a multiple-reader, multiple-writer coherency problem. If both the mutator and the collector are allowed to read and write the heap without 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 that the mutator loads from memory. Whenever the mutator is about to access a pointer to a white object (i. e. 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 either case, the barrier replaces the fromspace pointer with a tospace pointer, and so the mutator exclusively sees pointers to tospace. Thanks to this tospace invariant maintained by the read barrier, the mutator can never disrupt the garbage collector by installing a white pointer into a black object, and so the garbage collector will never lose sight of unprocessed objects.

Usually, the read barrier is realized by a compiler that inserts a short code sequence after each pointer load. This sequence checks whether the loaded pointer refers to a white object. In this case, subsequently referred to as a *read barrier fault,* the read barrier code calls a routine (the *read barrier fault handler*) that converts the white pointer to a gray or black pointer as described above.

Baker's algorithm is extremely elegant and considerably simplifies implementations. Without the read barrier, the mutator would always have to deal with two different identifiers for a single object. Thus, simple operations like the comparison of two pointers would become difficult to implement. In contrast, the presence of the read barrier and the tospace invariant enable the mutator to uniquely identify each object by means of its address in tospace.

2.3 Baker's Algorithm and Real-Time

Incremental garbage collectors like Baker's regularly pause the mutator, either to perform an increment of garbage collection work, or to synchronize the mutator with the collector, such as by the read barrier. Hard real-time systems, however, require correct responses to environmental changes within a specified amount of time. For this reason, real-time garbage collectors must always leave a guaranteed fraction of processing time to the mutator. To quantify this property, Cheng and Blelloch introduce the term *minimum mutator utilization* (MMU) as the fraction of time that the mutator is guaranteed to run within a given time quantum [5].

A garbage collector for real-time systems must satisfy two conditions to achieve a given minimum of mutator utilization. First, the duration of all pauses must be bounded by a constant, and this constant must be smaller than the considered time quantum. Second, pauses must not be too heavily clustered. Unfortunately, Baker's algorithm fails to meet these conditions in multiple respects. The following paragraphs identify the main issues with the algorithm and briefly summarize more recent work that aims at resolving or ameliorating these issues.

Work-based scheduling. Baker couples the garbage collector to the mutator and performs a certain amount of garbage collection at each allocation. This way of interleaving the mutator with the collector is often referred to as work-based scheduling. Using work-based scheduling, however, the garbage collection overhead is clustered with allocations and does not smooth out over a garbage collection cycle. For this reason, more recent algorithms [2] favor time-based scheduling over work-based scheduling and interleave the mutator with the collector based on fixed time intervals. This way, they decouple the garbage collector from the mutator's allocation behavior and evenly distribute the garbage collection overhead over time, resulting in superior mutator utilization.

Incremental stack processing. According to Baker's algorithm, the garbage collector must process the root set in an indivisible fashion. Since root sets usually include a program stack of potentially unbounded size, the time needed to process the root set is unbounded as well. To solve this problem, one option is to scan the stack incrementally. In this case, however, the mutator has to extend the read barrier to stack accesses, including accesses to local variables that are currently not available in registers. Because of the high frequency of stack accesses, the overhead of this option is usually considered too high. Other methods to address the root set problem include stacklets [5] and root set copies in the heap [14]. Yet, these methods are expensive to implement and introduce a number of new problems. In summary, efficient incremental stack processing is still an open issue.

Incremental compaction. Baker's original collector copies objects atomically when it evacuates them from fromspace to tospace. As a consequence, the duration of pauses caused by copying depends on the size of objects and is not bounded by a small constant. To address this issue, Baker proposes a refined version of his algorithm, but credits Steele for the primary idea [1]. This refined version does not copy objects during evacuation. Instead, it merely allocates an empty object frame in tospace and uses a backlink to doubly link the tospace frame to the fromspace original. It is not until the garbage collector's scan pointer eventually reaches an empty object frame in tospace that the garbage collector actually copies the corresponding object. The cost for this sort of lazy copying, however, has to be paid by the mutator. Whenever the mutator accesses an object, it has to check whether the corresponding object is gray, and if so, determine the current state of garbage collection. If the requested field inside the object has already been copied, the mutator has to access the object's tospace copy. If not, it has to follow the backlink in order to access the fromspace original. In contrast to read or write barriers, this elaborate procedure is required for pointer and non-pointer loads and stores. Without hardware support, the resulting overhead is generally considered prohibitive.

To avoid the cost of Steele's fine-grained synchronization, real-time garbage collectors in software do not directly support arbitrarily large objects. Instead, they either compose all objects of fixed-sized blocks and organize these as linked lists or trees [14], or they split only arrays into fixed-sized arraylets [2]. In either case, the mutator has to replace all appendant loads and stores through access sequences that follow chains of pointers. However, these access sequences may cause notable runtime and code size overhead. Furthermore, the decomposition of objects requires storage for auxiliary pointers.

2.4 Issues with Baker's Read Barrier

In the last section, we have summarized three general problems that more or less apply to any real-time garbage collector. In this section, we will concentrate on the cost of Baker's read barrier and of the tospace invariant it preserves.

In software, read barriers are realized by compiler-inserted code sequences. As such, read barriers cause three major problems. First of all, they inflate the program code and, in doing so, decrease the efficiency of an instruction cache. Second, the execution of the read barrier code results in considerable runtime overhead. Finally, compiler-supported synchronization mechanisms like read barriers introduce a strong dependency between compiled programs and a particular version of a garbage collector. Therefore, major updates of the garbage collector are likely to necessitate the recompilation of any application program.

In principle, all these problems apply to write barriers in the same way as to read barriers. However, since pointer loads are considerably more frequent than pointer stores, the overhead of read barriers is much higher than that of write barriers. Although Zorn found that the total runtime overhead of read barriers can be less than 20%, he also observed that read barriers typically double the size of compiled programs [16]. Therefore, while write barriers are considered practicable in software, it is often argued that Baker-style read barriers require hardware support for adequate performance [8].

The worst property of Baker's read barrier, however, is that read barrier faults are unevenly distributed over time. After a flip, all pointers are still white, so the majority of pointer loads will trigger read barrier faults. As a consequence, the pauses caused by the read barrier are clustered at the beginning of a garbage collection cycle and seriously degrade mutator utilization. Ultimately, this clustering of read barrier pauses is caused by Baker's tospace invariant. This invariant provides the illusion that garbage collection completes right after a flip. The more this illusion departs from reality, the more work has to be performed by the barrier to maintain it.

2.5 Alternatives to Baker's Read Barrier

The clustering of read barrier faults is tightly correlated with the tospace invariant of Baker's algorithm. For this reason, it is widely accepted that it is necessary to relax the invariant for adequate real-time performance. In this regard, the most prominent extension to Baker's algorithm has been proposed by Brooks [3]. Brooks basically replaces Baker's read barrier by a memory indirection and a write barrier. For that purpose, he requires each object to contain an additional field for an indirection pointer. Initially, this indirection pointer refers to the object itself, but as soon as the collector evacuates an object, the indirection pointer is updated to point to the copy in tospace.

Without a read barrier, the mutator is now able to see both fromspace and tospace pointers. To nevertheless ensure that the mutator always works with the proper copy of an object, any reference to an object must unconditionally follow the indirection pointer in the object's header. To furthermore prevent the mutator from writing white pointers into black objects, Brooks complements the indirection with a write barrier. On a barrier fault, this write barrier performs essentially the same operations as Baker's read barrier.

Compared to Baker's read barrier, Brooks' approach shows three advantages that ultimately result in a better balanced distribution of synchronization pauses. First, the indirection is unconditional, and so its costs do not depend on the current state of garbage collection. Second, write barrier checks are less frequent than read barrier checks. Finally, white pointers require black objects to actually trigger write barrier faults. While there are indeed many white pointers at the beginning of a garbage collection cycle, there is only a small number of black objects. Reversing the colors, the same is true near the end of a cycle. As a consequence, write barrier faults are not clustered at either end of a garbage collection cycle. Instead, the fault rate can be expected to show a flat maximum somewhere near the middle of a cycle.

The benefit of the better balanced write barrier fault rate, however, has to be paid by a costly memory indirection. Regrettably, the overhead of this indirection is particularly high since it needs to be resolved on every object access. Furthermore, the code for following indirections involves interdependent load-load and load-store sequences that are especially expensive in processor pipelines because the latency of load instructions is usually higher than that of standard arithmetic instructions.

Bacon et al. [2] found that the runtime overhead of a Brooks-style memory indirection can be reduced to an average of 4% and to a maximum of 10% by "eagerly" resolving indirections and by aggressively applying compiler optimizations. According to their proposal, the mutator immediately follows the indirection whenever it loads a pointer from memory. This way, their algorithm effectively converts Brooks' indirection procedure into an unconditional read barrier and at the same time restores a part of Baker's tospace invariant, namely that pointers to gray objects always refer to tospace copies. As a benefit of the read barrier, the mutator no longer has to repeatedly follow the same indirection when it dereferences the same pointer over and over again. Instead, the read barrier resolves the indirection only once, and the loaded pointer can subsequently be dereferenced without additional costs.

Unfortunately, Bacon et al.'s approach faces two problems. First, the unconditional read barrier complicates garbage collection since it does not prevent the mutator from seeing white objects. Therefore, whenever the garbage collector evacuates an object, it has to search through all registers and the entire program stack in order to update all pointers to that object with a pointer to the corresponding tospace copy. This way, the unconditional read barrier merely trades some increase in read barrier performance for a considerable amount of additional work imposed on the collector.

The second problem is caused by compiler optimizations. To reduce the cost of the read barrier to a tolerable degree, Bacon et al. heavily rely on the compiler to remove part of the read barrier code by applying optimizations such as common subexpression elimination. As a negative side effect of these optimizations, however, regions of optimized code may no longer be interrupted by the garbage collector. Considering time-based scheduling, this restriction is particularly disturbing since it requires additional synchronization to protect critical regions that have been artificially created by optimizations.

3. A HARDWARE-SUPPORTED APPROACH

Most problems associated with Baker's algorithm in particular and with real-time garbage collection in general are inherent to software. In software, any kind of synchronization must be serialized and implemented by compiler-inserted instruction sequences. This synchronization includes read and write barriers, memory indirections, and mechanisms for incremental compaction such as the decomposition of large objects or Steele's extension to Baker's algorithm.

Hardware has the potential to efficiently synchronize the mutator with garbage collection in parallel to instruction processing. However, to thoroughly exploit this potential, a system requires the knowledge of objects and pointers in hardware. For this reason, we have proposed an object-based processor architecture especially designed for hardware-supported garbage collection [9]. This architecture completely abstracts from memory management at the assembly language level. Thanks to this abstraction, implementations dispose of the largest possible degree of freedom to effectively support garbage collection in hardware, reaching from elementary synchronization circuits to fully-featured garbage collection in hardware.

The work we present in this paper is based on an actual implementation of our object-based processor architecture [9, 10]. While this implementation solves most of the aforementioned problems of real-time garbage collection, we identified a serious problem with respect to the read barrier. To start with, this section provides an introduction to the architecture and our initial implementation. In the remaining sections, we address the identified problem by a novel read barrier design and evaluate its benefits.

3.1 System Overview

Figure 2 shows a structural overview of our initial system. The main processor shown in the left of the figure implements the object-based processor architecture mentioned above. It is complemented by a small, low-cost coprocessor dedicated to and specialized for garbage collection. For efficient synchronization, the main processor and the garbage collection coprocessor are tightly coupled and realized on the same chip. A memory controller provides separate ports for both the main processor and the coprocessor. Thanks to this configuration, the inherently non-local behavior of garbage collection does not disrupt cache locality. At its external interface, the device behaves like a standard uniprocessor and interfaces to standard memory devices.

3.2 Main Processor

3.2.1 Architecture

Basically, the architecture of the main processor combines object-based memory addressing with a modern RISC-style instruction set to allow for efficient pipelined implementations. Instead of plain memory addresses, load and store instructions use object/index pairs to access memory. To distinguish pointers from ordinary non-pointer data, the architecture strictly separates pointers from non-pointer data. Furthermore, the architecture ensures the integrity of pointers, i.e. pointers are always guaranteed to be either *null* or to be uniquely associated with an existing object.

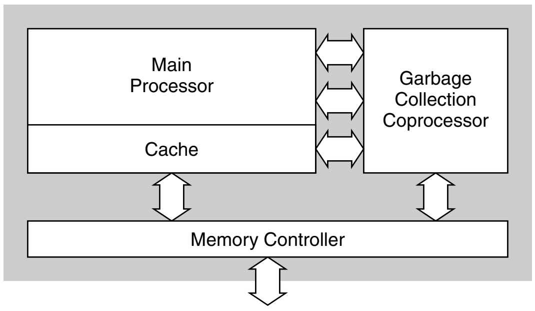


Figure 2: System overview

Objects in memory are composed of a separate pointer area and a separate data area (Figure 3). Two attributes describe the size and the partitioning of an object. The π-attribute specifies the number of bytes in the pointer area, and the δ-attribute specifies the number of bytes in the data area, respectively. The attributes are stored in an attribute header that remains completely invisible to assembly language programs.

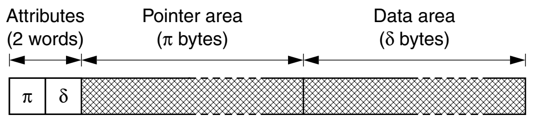


Figure 3: Structure of objects in memory

Load and store instructions for pointers implicitly target an object's pointer area, while load and store instructions for data implicitly target its data area. This way, each object effectively offers two independent index spaces, each starting with zero. To protect the integrity of objects and pointers, range checks ensure that load and store instructions never violate the bounds of the respective area.

The architecture has a dedicated allocate instruction for the creation of new objects. In contrast, there is no explicit mechanism for the deletion of objects. Instead, the architecture relies on an "invisible" garbage collector to reclaim memory behind the scenes, thereby providing the illusion of infinite memory.

3.2.2 Implementation

Despite the architecture's object-based nature, the implementation is based on a straightforward pipelined RISC design that is extended to efficiently handle objects and attributes (Figure 4).

The register set in the decode stage includes 16 pointer registers and 16 data registers. In the execute stage, the ALU performs operations targeting data registers, the PGU (Pointer Generation Unit) performs operations targeting pointer registers (such as allocate object or copy pointer), and the AGU (Address Generation Unit) generates addresses for the cache in the subsequent memory stage. For simplicity, this cache is designated as a data cache even though it contains ordinary data and pointers.

Whenever a load or store instruction dereferences a pointer to access memory, the AGU requires the attributes of the corresponding object for address generation and for range checking. For this reason, each pointer register is supplemented by attribute registers. Whenever a pointer register holds a non-null value, the corresponding attribute registers hold the attributes of the object to which the pointer refers. In this way, the effort for dereferencing a pointer register is as low as address generation in conventional architectures. Range checking does not involve any time overhead since it occurs in parallel to address generation.

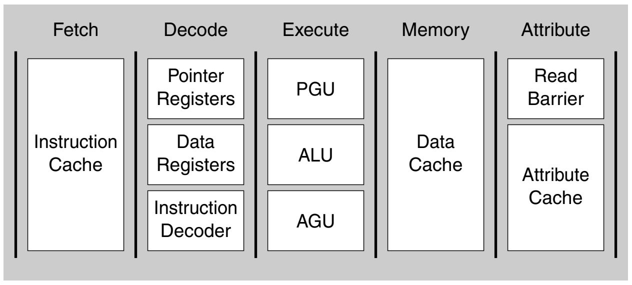


Figure 4: Basic structure of the main processor pipeline

Whenever a load pointer instruction loads a non-null value from memory, it must also load the associated attribute registers. 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.

3.3 Garbage Collection Coprocessor

The garbage collector is realized as a microcoded coprocessor. In contrast to usual microcoded processors, it does not offer a universal set of microcoded machine instructions. Instead, it realizes the complete garbage collection algorithm as a single microprogram. As a benefit, the coprocessor does not require any memory bandwidth for instruction fetching.

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 two burst registers that basically correspond to single cache lines. To transfer the contents of the burst registers from and to memory, the coprocessor takes advantage of efficient burst modes offered by modern memory devices.

Essentially, the coprocessor realizes a Baker-style copying collector 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. To do so, it sets a gray-bit in the π-attribute, saves the δ-attribute to tospace, and overwrites the δ-attribute with Cheney's forwarding pointer (Figure 5). In tospace, it initializes the field for the π-attribute with Steele's 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.

3.4 Hardware-Supported Synchronization

Our system efficiently synchronizes garbage collection and application processing in various ways. To ensure cache coherency, the garbage collector is able to inspect the main processor's caches

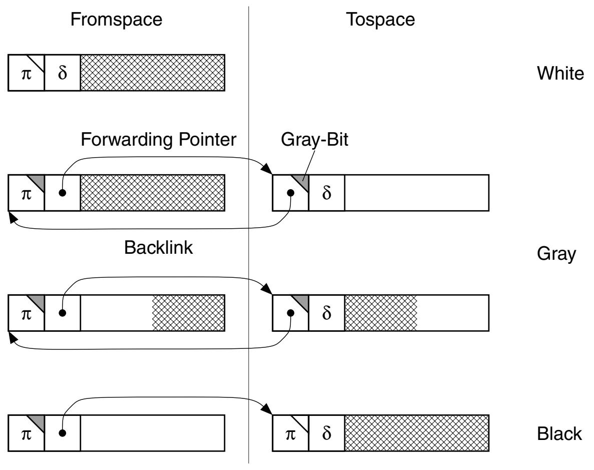


Figure 5: Object states during garbage collection

and to flush individual cache lines if necessary. Furthermore, a cache line locking mechanism guarantees exclusive access to objects. Finally, the garbage collection coprocessor may temporarily suspend the main processor to protect critical regions in the microcode, e.g. for root set processing.

In the context of this paper, two tightly related synchronization mechanisms are of particular interest: the mechanism for incremental compaction and, as a matter of course, the read barrier. They are subsequently described in some greater detail.

3.4.1 Incremental Compaction

Steele's extension to Baker's algorithm allows the collector to copy objects incrementally. As a consequence, the mutator must always decide whether the 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 Steele's backlink 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 (see Figure 1, lower diagram), 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 (see Figure 5). 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.

3.4.2 Read Barrier

The knowledge of pointers in hardware enables the processor to realize the read barrier as simple as by two comparators that check whether a loaded pointer refers to fromspace. If so, the processor suspends the corresponding instruction and signals an interrupt to the garbage collection coprocessor. Thereupon, the coprocessor invokes the read barrier fault handler in its microcode. As soon as the handler completes, the main processor restarts the suspended load pointer instruction. This time, the instruction loads a tospace pointer and passes the read barrier check.

To realize a read barrier in the way just described, a standard processor would have to delay each load pointer instruction until the read barrier check completes. Our processor, however, fully pipelines the load pointer instruction by means of the attribute stage after the usual memory stage. As a benefit, the read barrier is able to check loaded pointers while the same pointer simultaneously accesses the attribute cache (see Figure 4). In this way, the processor performs the read barrier check completely in parallel to standard instruction execution.

Whenever the read barrier detects a fromspace pointer, the corresponding load pointer instruction is effectively delayed until the garbage collector's fault handler completes. Thanks to Steele's extension to Baker's algorithm, the handler has to perform a strictly bounded number of basic operations only. In the worst case, it evacuates an object, which comprises the following basic operations: (1) load the pointer that triggered the read barrier, (2) load the attributes of the corresponding object, (3) advance *free*, (4) overwrite the fromspace attributes with π and the forwarding pointer, (5) initialize the tospace attributes with the backlink and δ, and (6) overwrite the initial pointer with the previous value of *free*, totalling two memory reads and three memory writes. If the object has already been evacuated, the procedure includes two memory reads and one memory write.

The hardware read barrier supervises any pointer loaded from memory, including those in program stacks. The performance penalty caused by this globalization is only small. Because of the high locality of stack accesses, most pointers read from the stack are likely to have only recently been written, and so the read barrier will not fail in the majority of these cases. Thanks to the read barrier globalization, though, the garbage collector is now able to process program stacks in an incremental way.

3.5 Summary

The system described so far solves all problems with respect to real-time performance addressed in section 2.3. First, it performs garbage collection truly concurrently with application processing and thereby eliminates the need to interleave the collector with the mutator on the same processor. Second, the globalization of the read barrier enables the garbage collector to scan program stacks in an incremental way. Third and finally, the hardware implementation of Steele's lazy copying permits efficient incremental compaction. Thanks to all these fine-grained synchronization mechanisms, the system bounds the duration of any garbage collection related pause to a maximum of 500 clock cycles [10].

Aside from these real-time features, the system shows additional qualities with respect to modularity and robustness. The abstraction of garbage collection at the assembly language level renders both compilers and compiled programs completely independent of a particular garbage collection method. Even more, garbage collection does not depend on any kind of compiler support whatsoever, be it for pointer identification or for synchronization. Finally, the system introduces the robustness stemming from garbage collection and fine-grained memory protection at the machine code level.

4. THE COST OF THE READ BARRIER

In the system described in the last section, by far the most frequent synchronization pauses are caused by read barrier faults. All other pauses are extremely rare and can effectively be ignored [10]. For this reason, we subsequently focus on read barrier faults.

Thanks to hardware support for the read barrier and incremental copying, the penalty incurred by a single read barrier fault is rather low. As stated in section 2.3, however, it is not adequate to exclusively consider the length of pauses to evaluate real-time performance. Therefore, we will now study the distribution of read barrier faults over time in order to determine their actual influence on mutator progress and utilization.

4.1 Prototype and Measurement Platform

For our performance measurements, we are using an FPGA-based prototype of the system described in Section 3. The main processor is statically scheduled and issues up to three instructions in a clock cycle. It has an elaborate branch prediction unit, an 8K instruction cache, an 8K data cache, and an attribute cache with 256 entries. The garbage collection coprocessor is integrated on the same device and occupies less than 20% of the chip area. Main memory is composed of standard SDRAM modules. Processor, SDRAM and various standard peripherals are synchronously operated at 25MHz. On the software side, a static Java compiler translates standard Java bytecode to the processor's native machine code. The compiler includes a code scheduler that rearranges instructions in order to take advantage of the processor's parallel execution units and to hide instruction latencies as far as possible.

For accurate measurements, we must be able to observe the state of various signals inside the processor at every single clock cycle. To accomplish this, we integrated a measurement port into the processor that routes 32 arbitrary internal signals to a dedicated Gigabit Ethernet interface. The resulting data stream of 800Mbit/s is written to three hard disks in parallel and analyzed offline.

4.2 Distribution of Read Barrier Pause Lengths

To evaluate the cost of a single read barrier fault, we first determine the length of all pauses caused by the fault handler. In doing so, we start to measure from the moment that a load pointer instruction triggers the read barrier until the moment that the restarted instruction completes. The measured durations include a pipeline flush (5 cycles), a potential attribute cache miss, and penalties possibly incurred by other instructions that are executed in the pipeline at the same time. For this reason, the measurements are somewhat pessimistic, and the actual costs are slightly smaller.

Figure 6 shows the distribution of pause lengths that we obtained from the "database" benchmark (SPEC JVM98). With respect to overall read barrier behavior, this benchmark represents the worst case of all applications we have examined.

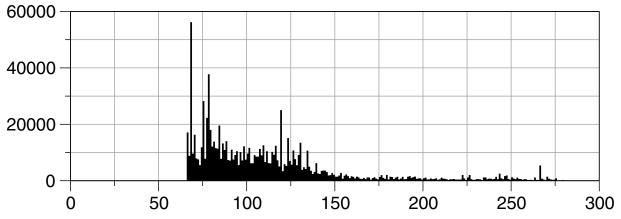


Figure 6: Distribution of pause lengths – number vs. duration in clock cycles (microcoded fault handler, benchmark "database")

The distribution has a mean value of 111.6 clock cycles, but shows a wide variation from approx. 60 to 300 clock cycles. This high degree of variation has several causes. First, the handler must distinguish two separate cases and either evacuate an object or read a forwarding pointer. Second, the coprocessor protects small critical regions of uninterruptible code to avoid access conflicts to shared resources. Third, any memory access must first inspect the main processor's caches and potentially wait until the processor completes a cache line flush. Fourth and finally, memory access times vary because the memory controller has to insert refresh cycles on a regular basis.

4.3 Influence on Minimum Mutator Utilization

As mentioned before, while it is absolutely necessary to bound the duration of all pauses, it is even more important to examine their distribution over time. To do so, we first divide time into units of 500 clock cycles each. This granularity seems appropriate since the duration of the longest synchronization pauses is in the same order of magnitude. Then, we measure the number of read barrier stall cycles within each time unit.

The results from running our worst-case benchmark "database" are shown in Figure 7. These results are even worse than we anticipated. Severe bursts of read barrier faults stall the processor pipeline for up to 99.6% within some time units. Even more serious, units with a high percentage of stalls are extremely clustered in time. Examining a sliding window of 5ms (i.e. all possible inter-

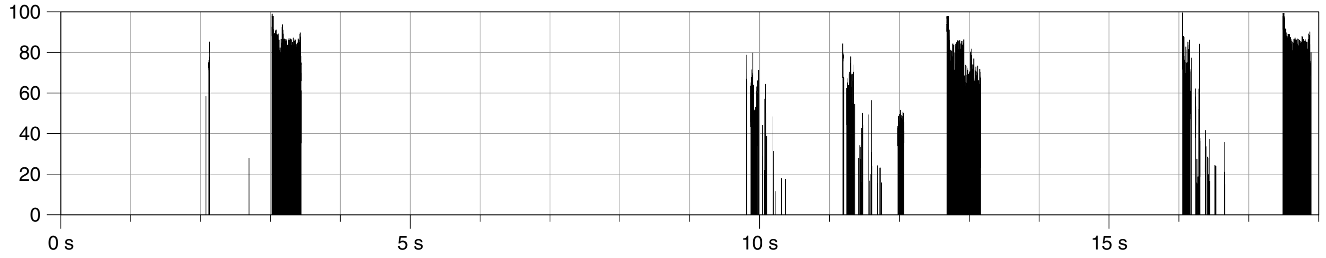


Figure 7: Percentage of read barrier stall cycles within intervals of 500 clock cycles (microcoded fault handler, benchmark "database", first 18 seconds)

vals comprising 250 time units or 125000 clock cycles), we found that the maximum percentage of stall cycles per window still amounts to 91.7%, which corresponds to a minimum mutator utilization of 8.3% at a time quantum of 5 ms.

5. A TRUE HARDWARE READ BARRIER

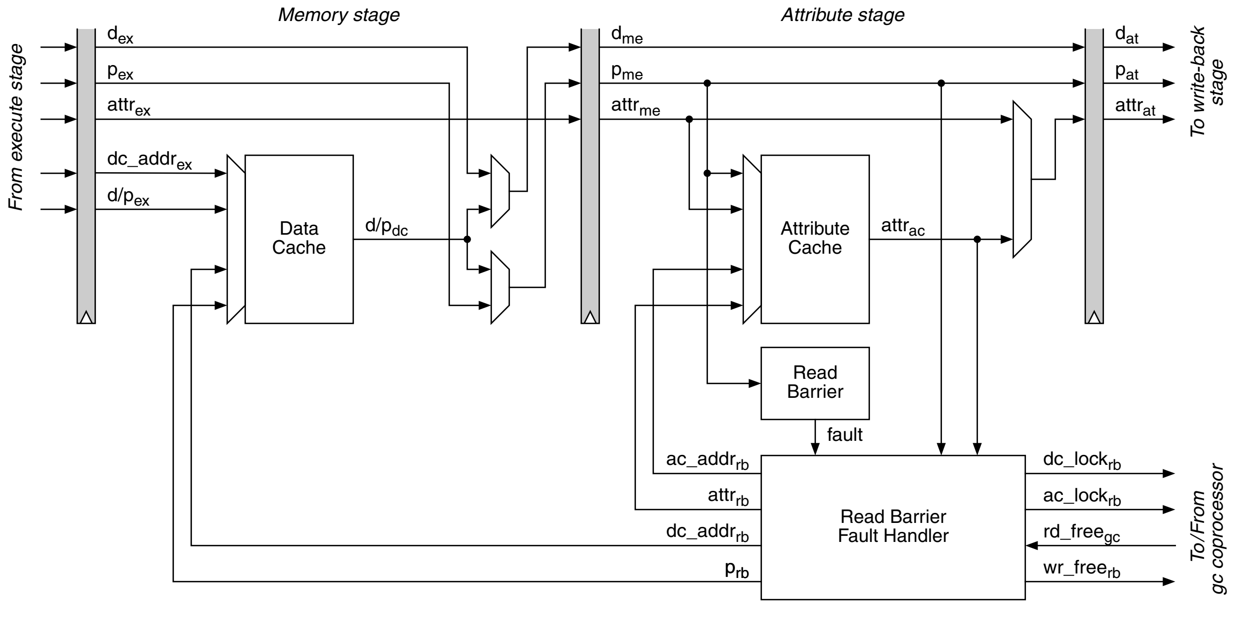
Although the last section covers a worst-case scenario, the results are disappointing. To improve the situation, the obvious alternative is to replace Baker's read barrier by a variant of Brooks-style indirections and write barriers. However, the resulting lack of the tospace invariant considerably complicates garbage collection and will raise the cost and complexity of a hardware-based implementation. For this reason, we propose an entirely opposed approach. Our basic idea is to live with the clustering of read barrier faults and to instead increase the efficiency of the fault handler as much as possible.

Analyzing the original read barrier, we found that most of the overhead is caused by the fact that the main processor and the garbage collection coprocessor expensively communicate via main memory. First, the main processor's caches flush the respective pointer and attributes to memory. Next, they are reloaded by the coprocessor, processed, and stored back to memory. Eventually, the processor restarts the instruction and loads the pointer and attributes back to its caches. Ultimately, this overhead is nothing but a consequence of a locality issue: A pointer that triggers the read barrier, along with its attributes, is local to the mutator and not local to the garbage collector.

To exploit this insight, we decided to completely move the read barrier fault handler from the garbage collection coprocessor to the main processor and to directly integrate it into the pipeline. If the required pointer and attributes are already available in the caches, all that the fault handler actually will have to do is to check the color of the object, to advance *free*, and to store the new pointer and, in case of an evacuation, two new sets of attributes. This procedure is so simple that it is possible to implement the entire fault handler directly in hardware.

Figure 8 illustrates the integration of the read barrier fault handler into the pipeline. The read barrier, instead of interrupting the garbage collection coprocessor, now activates the read barrier fault handler circuit. Thereupon, the handler locks the corresponding cache lines in order to prevent the garbage collector from doing the same. In case the fault handler and the collector lock the same cache line at the same time, the coprocessor withdraws its lock and gives way to the fault handler. Next, the handler reads and updates the garbage collector's *free* pointer. A simple locking mechanism ensures that this happens in a consistent way. Finally, the handler writes the updated pointer to the data cache and, if necessary, two sets of attributes to the attribute cache. In doing so, it can simultaneously write to both caches. Since a read barrier fault flushes the pipeline, the fault handler has exclusive access to the caches. In the way just described, the fault handler efficiently profits from the processor's caches and avoids any direct memory access.

A somewhat subtle side effect of handling read barrier faults in the processor pipeline itself is that the attribute cache now has to deal with fromspace pointers. While the original read barrier suppresses fromspace attribute cache misses, the hardware fault handler requires the cache to resolve them since it depends on the attributes and the forwarding pointer in the fromspace object. Hence, the attribute cache now contains both tospace and from-



Signals

d, p	Data word, pointer word	dc_addr	Data cache address
attr	Attributes (π, δ, and p_2)	ac_addr	Attribute cache address
rd_free	Read port for GC's free pointer	dc_lock	Data cache lock
wr_free	Write port for GC's free pointer	ac_lock	Attribute cache lock

Indices

ex	From execute stage	dc	From data cache
me	From memory stage	ac	From attribute cache
at	From attribute stage	rb	From rb fault handler
		gc	From gc coprocessor

Figure 8: Integration of the hardware read barrier fault handler into the main processor pipeline

space records, whereby the latter actually "misuse" a cache line's backlink entry for the forwarding pointer.

6. MEASUREMENT RESULTS

To quantify the advantage of the new read barrier design, we repeat the measurements described in Sections 4.2 and 4.3, again using our worst case application "database" as the first benchmark.

The new distribution of pause lengths is shown in Figure 9. Pauses now vary from approx. 5 to 50 clock cycles with a mean value of 19.3 clock cycles only. Compared to the initial read barrier handler, the pauses are a significant factor of almost 6 shorter.

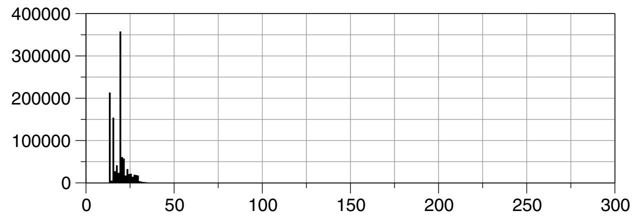


Figure 9: Distribution of pause lengths – number vs. duration in clock cycles (hardware fault handler, benchmark "database")

Considering this substantial reduction, it is interesting to see how this advantage translates to the distribution of read barrier stall cycles over time. The result for "database" is shown in Figure 10. As a first observation, the shape of the graph is very similar to the graph obtained with the initial microcoded handler (Figure 7). The height of the bursts, however, is much lower. Regarding a time quantum of 5 ms, the worst case percentage of read barrier stalls drops from 91.7% to 41.9%, which corresponds to an increase in minimum mutator utilization by a factor of 7.

To confirm these initial findings, we have repeated the measurements for a total of seven different applications that we successfully compiled for our prototype. To assess the gain in minimum mutator utilization for various time quantums, we determined the worst case percentage of stall cycles for various time intervals (Table 1). Assuming a time quantum of 1 ms, for example, the minimum mutator utilization with the original fault handler is less than 10% in three of the cases. In contrast, the hardware fault handler always achieves more than 55%. Consequently, a hard real-time system with response times in the order of 1 ms must respect a worst case overhead of a factor of 10 with the original handler, but only a factor of 2 with the hardware handler.

For reference, we have assembled additional measurement data in the Appendix, including some general benchmark characteristics and a comparison of all distribution graphs for all benchmarks.

It should be noted that, as before, all measurements are conservatively pessimistic. Since the pipeline processes many instructions when a read barrier fault occurs, it is not always possible to tell whether a particular stall cycle is actually caused by the fault handler itself. If, for example, a load pointer instruction is immediately followed by a load instruction that causes a data cache miss, the corresponding pipeline stall, although unrelated, will affect the duration of the measured pause. Compared to the initial handler, the resulting pessimism is even larger for the hardware handler since the overall duration of pauses is much shorter.

Table 1: Worst case percentage of read barrier stall cycles – microcoded handler (mc) vs. hardware handler (hw) for different time intervals

		compress	cup	database	javac	javacc	jflex	jlisp
Basic time unit of	mc	91.0	99.6	99.6	99.8	99.6	99.6	99.6
500 clock cycles (20 µs)	hw	23.2	55.2	54.6	69.0	46.8	67.4	61.2
Sliding window over	mc	32.1	89.9	93.8	98.6	80.7	93.9	96.6
10 time units (200 µs)	hw	5.2	44.9	45.6	61.1	22.4	40.8	25.6
Sliding window over	mc	9.1	84.2	92.8	93.6	75.1	92.1	79.4
50 time units (1 ms)	hw	1.3	35.4	43.2	44.9	16.0	20.6	13.3
Sliding window over	mc	2.6	82.9	91.7	77.5	61.1	43.4	38.2
250 time units (5 ms)	hw	0.4	30.6	41.9	16.2	9.7	7.8	5.5
Sliding window over	mc	0.6	82.3	88.6	50.3	19.8	16.6	8.6
1250 time units (25 ms)	hw	0.1	28.4	37.9	13.2	2.6	2.3	1.6

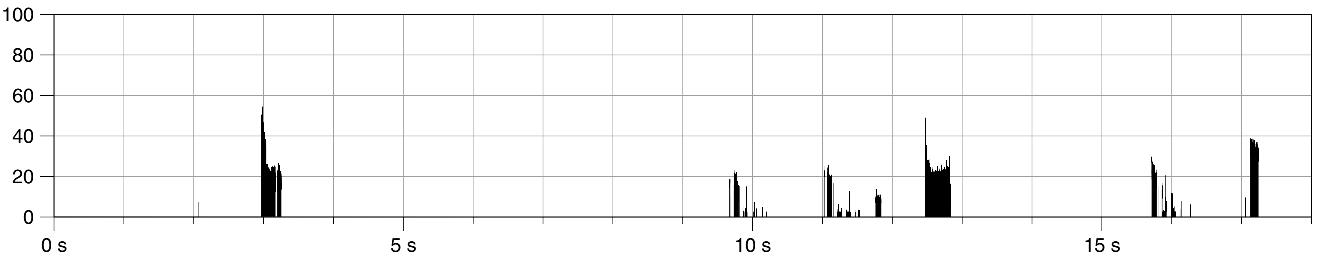


Figure 10: Percentage of read barrier stall cycles within intervals of 500 clock cycles (hardware fault handler, benchmark "database", first 18 seconds)

7. RELATED ARCHITECTURES

Hardware support for read and write barriers was introduced by language-directed architectures in the 1980s. Aside from typical CISC architectures like the Symbolics 3600 [11], examples also include pipelined RISC designs like SOAR [15] and SPUR [7]. All these architectures include read or write barriers in hardware, but the corresponding exception handlers are realized in software. By the end of the 1980s language-directed architectures were superseded by less specialized commodity architectures that offered better performance at lower cost.

In the 1990s, a number of researchers proposed active memory modules that were designed to work with standard microprocessors. The best known of these modules is the garbage-collected memory module (GCMM) of Nilsen and Schmidt [13]. Their module accommodates the actual memory devices, a private microprocessor, and a number of custom circuits, including two elaborate CAM-like devices. An arbiter within the module manages accesses from the external processor and realizes a read barrier as well as Steele-style incremental compaction. Unfortunately, this arbiter is merely drafted in the form of abstract flowcharts [12], and the system was never realized as a prototype. Also, the hardware costs for the memory module are prohibitive, particularly for most embedded applications. Regarding performance, the module's data throughput is considerably inferior to that of standard memory, especially when compared with modern, burst-oriented devices. Furthermore a significant overhead is caused by communicating the location of pointers from the main processor to the module, most notably regarding stack operations. Ultimately, most of these problems are not related to garbage collection itself, but result from the loose coupling between main processor and module.

Despite the experiences in the 1980s, language-directed architectures nowadays experience a resurrection in the form of Java processors for embedded applications. However, these architectures concentrate on bytecode execution and offer marginal to no support for real-time garbage collection.

Regarding this condensed overview, the architecture we propose is the first universal processor architecture with a RISC-style instruction set that thoroughly exploits the potential of language-independent hardware support for garbage collection.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a novel hardware read barrier that is directly integrated into a processor pipeline. In contrast to previous work, our read barrier handles read barrier faults completely in hardware and completes in less than 20 clock cycles on average. Consequently, the cost of a read barrier fault descends to the same order of magnitude as that of an ordinary cache miss.

Despite common belief, we have shown that it is possible to maintain the simplicity and elegance of Baker's original tospace invariant without seriously degrading mutator utilization. For a time quantum as short as 1 ms or 25000 clock cycles, all the applications we have examined show a minimum mutator utilization of more than 55%. Thanks to this property, it is feasible to estimate worst case execution times for hard real-time applications.

Our current implementation within a statically scheduled in-order processor is very well suited for low-cost embedded systems. In more elaborate out-of-order processors, however, it is not reasonable to stall the processor on a cache miss or to flush the processor pipeline on a read barrier fault. Notwithstanding, we expect that our approach will fit well or even better into the concept of out-of-order processors. Since these processors completely decouple instruction issue from instruction completion, it will be possible to encapsulate all garbage collection related synchronization into the load/store unit. This way, a read barrier fault merely looks like a slow memory access and will not affect the execution of unrelated instructions. To exploit these opportunities, we are currently investigating an out-of-order implementation of our processor architecture.

REFERENCES

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

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

[3] Brooks, R.A.: Trading data space for reduced time and code space in real-time garbage collection on stock hardware, *Proc. ACM Symposium on LISP and functional programming,* August 1984, pp. 256–262.

[4] Cheney, C.J.: A nonrecursive list compacting algorithm, *Comm. ACM,* vol. 13(11), Nov. 1970, pp. 677–678.

[5] Cheng, P.; Blelloch, G.E.: A parallel, real-time garbage collector, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation,* Jun. 2001, pp. 125–136.

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

[7] Hill, M.; et al.: Design decisions in SPUR, *IEEE Computer,* vol. 19(11), Nov. 1986, pp. 8–22.

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

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

[10] Meyer, M.: An on-chip garbage collection coprocessor for embedded real-time systems, *Proc. IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications,* Aug. 2005, pp. 517–524.

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

[12] Nilsen, K.D.: *Memory Cycle Accountings for Hardware-Assisted Real-Time Garbage Collection,* Tech. Report 91-21, Dep. of Computer Science, Iowa State University, Nov. 1992.

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

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

[15] Ungar, D.; et al.: Architecture of SOAR: Smalltalk on a RISC, *Proc. 11th Ann. Symp. on Computer Architecture,* Jun. 1984, pp. 188–197.

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

APPENDICES

A. General benchmark characteristics

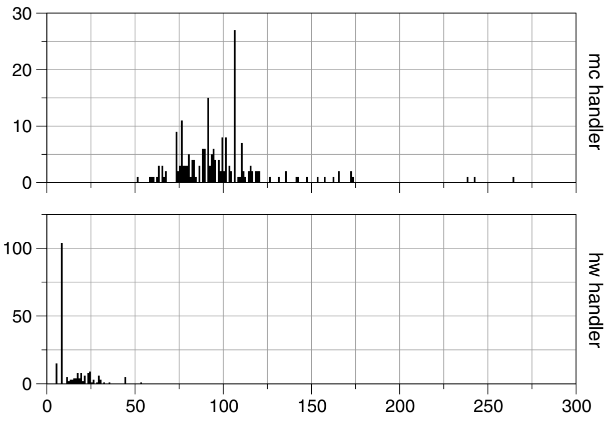
		compress	cup	database	javac	javacc	jflex	jlisp
Semispace size	absolute	10M	19M	16M	8.25M	5M	3.5M	192K
	relative	~140%	~220%	~160%	~150%	~260%	~170%	~150%
Total program runtime	mc	47.7s	53.0s	262.7s	39.2s	20.3s	25.6s	61.7s
	hw	47.5s	52.4s	261.3s	37.1s	20.2s	25.6s	61.2s
Number of read barrier faults	mc	215	174,168	930,872	168,260	17,530	9,624	112,936
	hw	207	207,631	1,118,045	191,296	17,401	9,755	123,524
Average percentage of read barrier stall cycles	mc	0.00	1.44	1.57	1.91	0.40	0.20	0.93
	hw	0.00	0.32	0.31	0.35	0.06	0.02	0.11

Programs were run in "typical minimum" memory configurations that (a) avoid mutator starvation and (b) keep the collector busy.
Semispace sizes are given both absolute and relative to the smallest possible size required for stop-the-world collection.
(mc: microcoded read barrier fault handler; hw: hardware read barrier fault handler)

B. Distribution of pause lengths (number vs. duration in clock cycles)

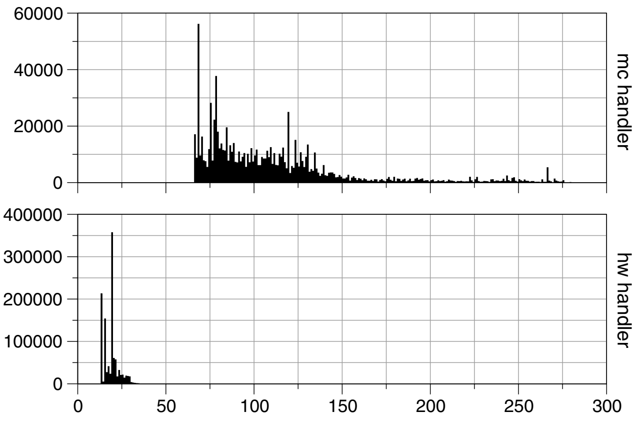
compress

mean$_{mc}$ = 104.5 – mean$_{hw}$ = 14.8



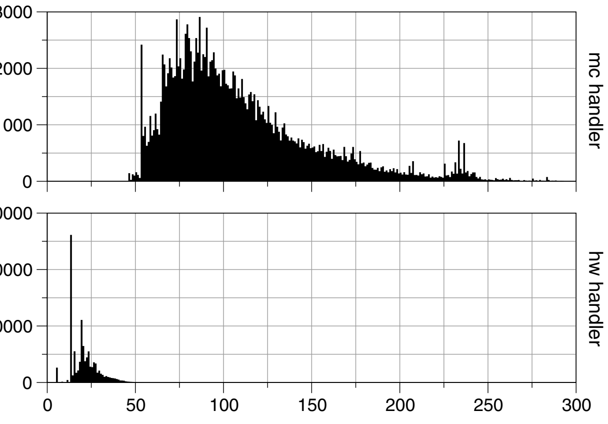
database

mean$_{mc}$ = 111.6 – mean$_{hw}$ = 19.3



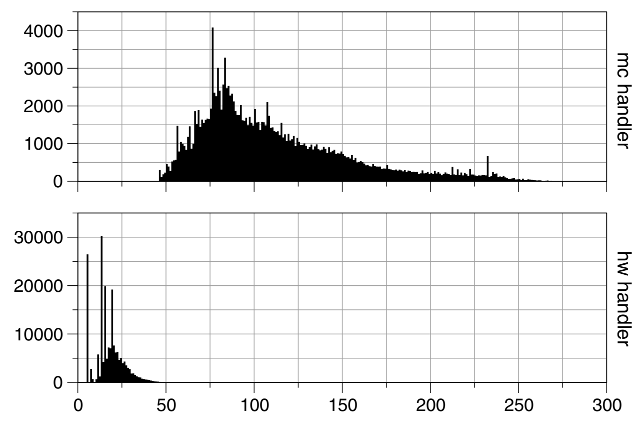
cup

mean$_{mc}$ = 110.2 – mean$_{hw}$ = 21.3



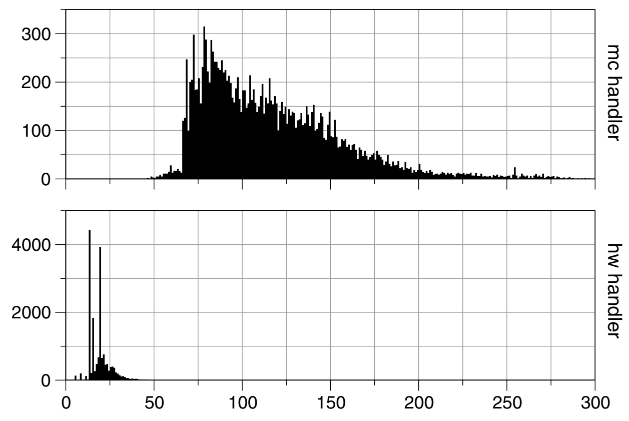
javac

mean$_{mc}$ = 112.1 – mean$_{hw}$ = 18.1



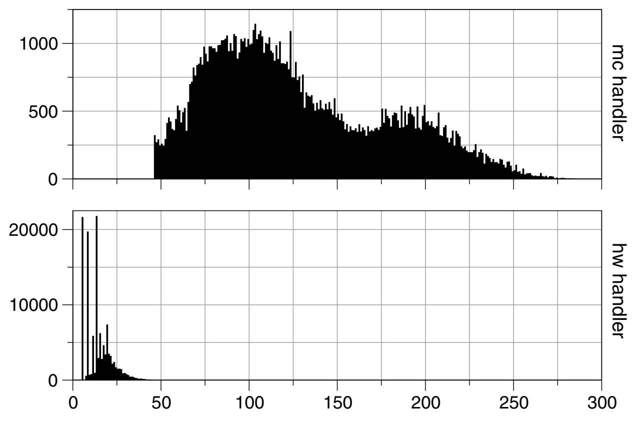
javacc

$\text{mean}_{mc} = 117.2$ – $\text{mean}_{hw} = 19.4$



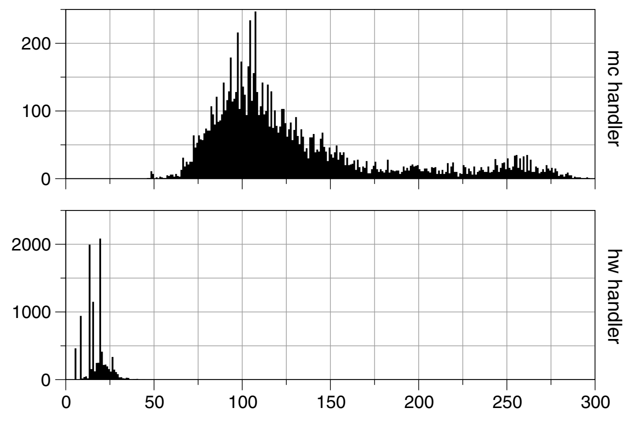
jlisp

$\text{mean}_{mc} = 128.4$ – $\text{mean}_{hw} = 14.9$



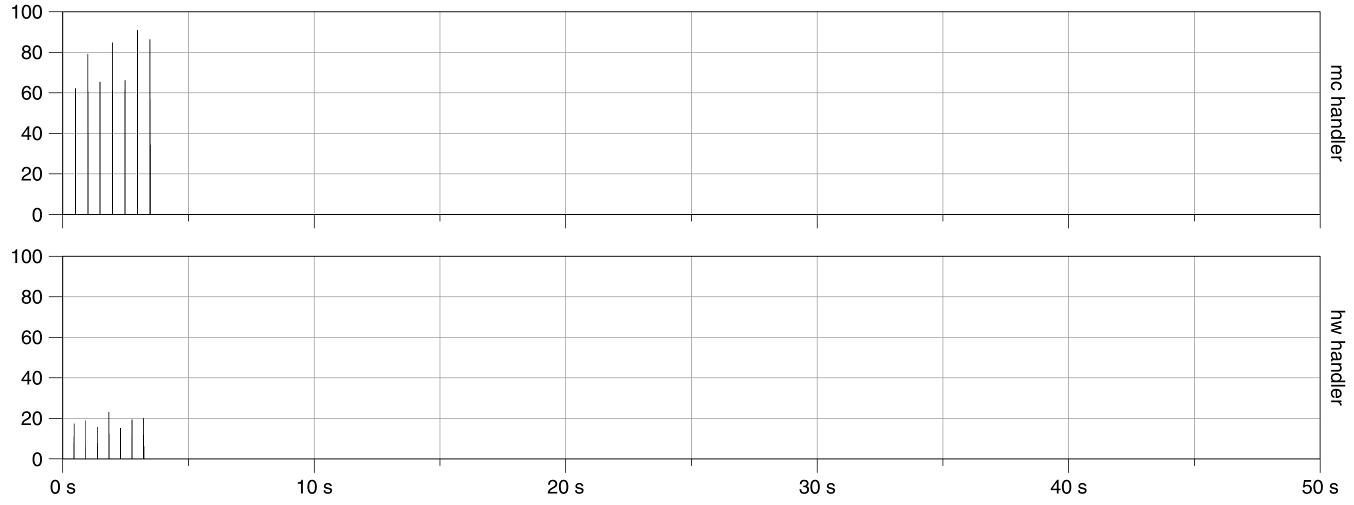
jflex

$\text{mean}_{mc} = 130.8$ – $\text{mean}_{hw} = 17.6$

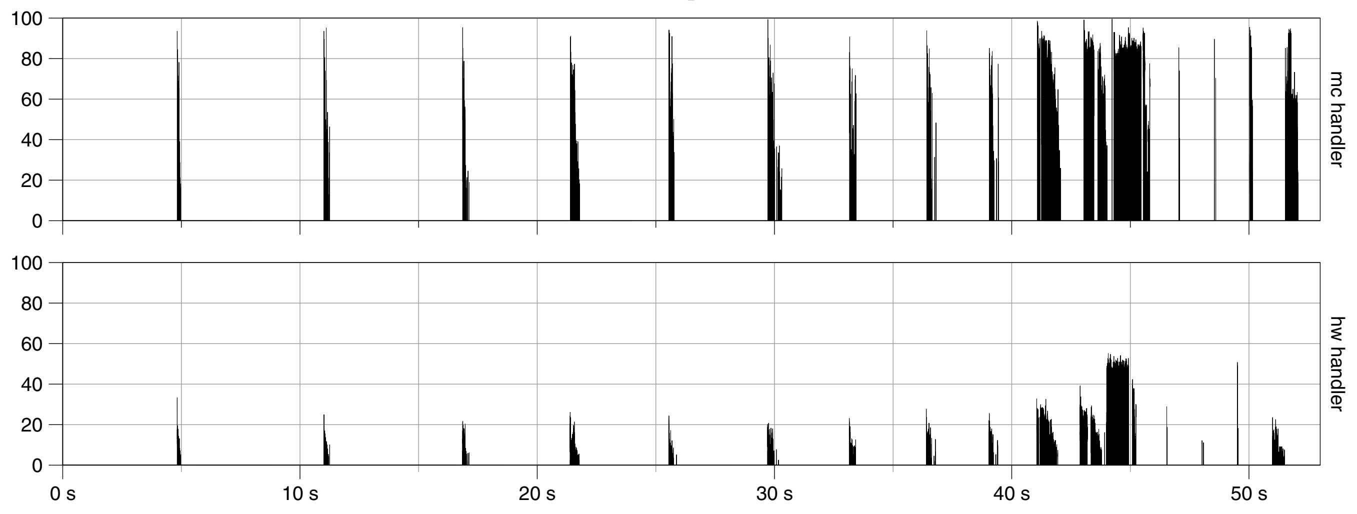


C. Percentage of read barrier stalls over time (within intervals of 500 clock cycles)

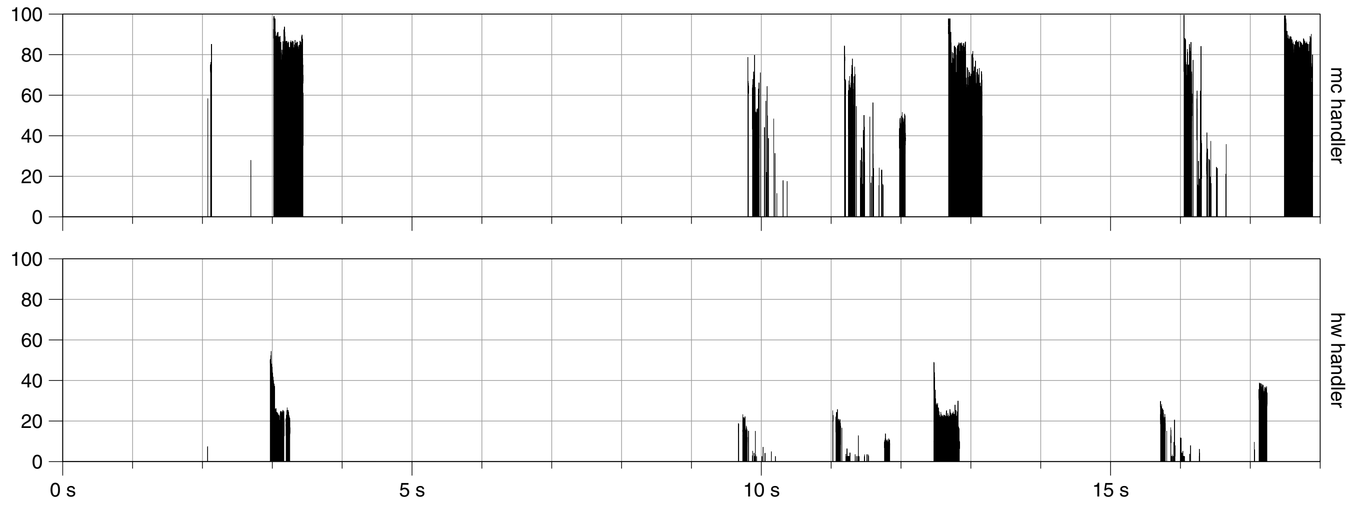
compress



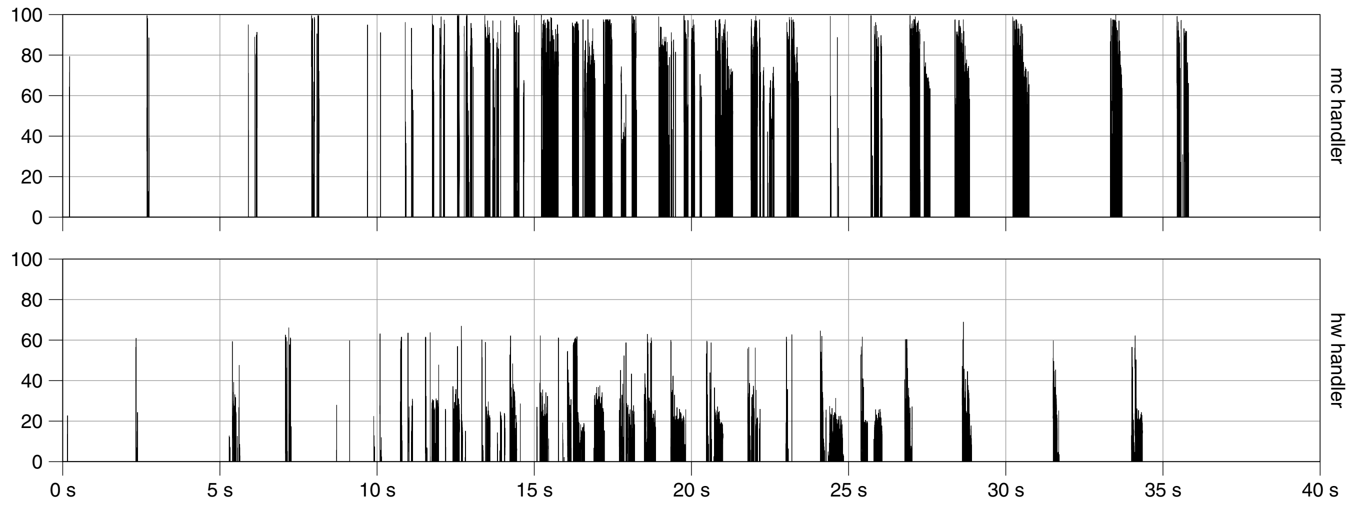
cup

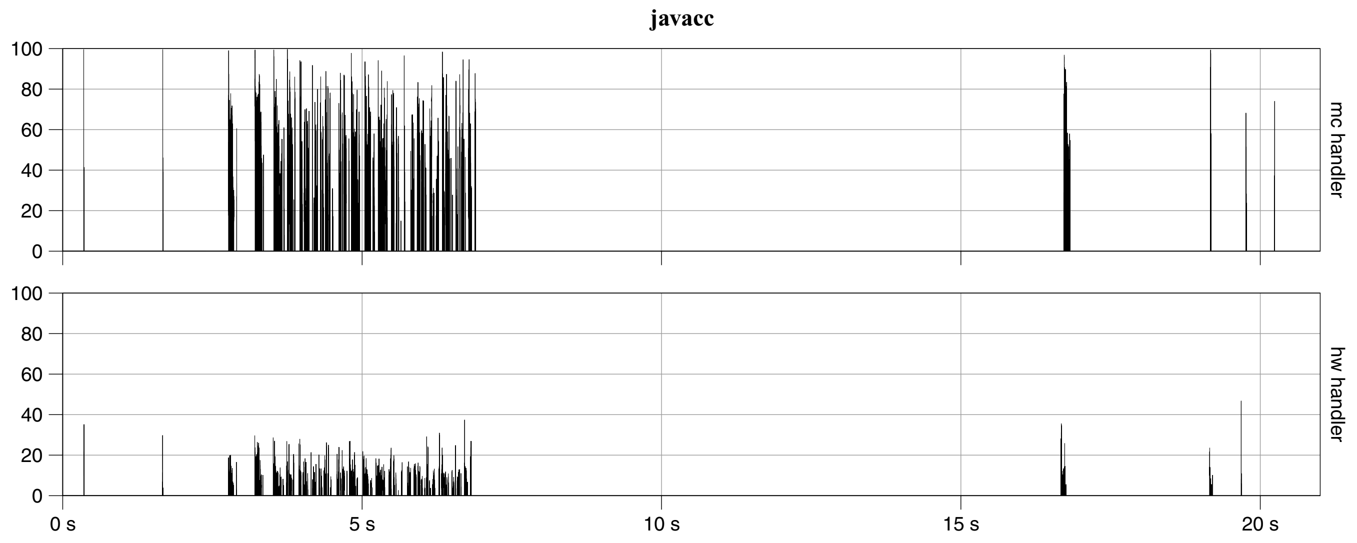


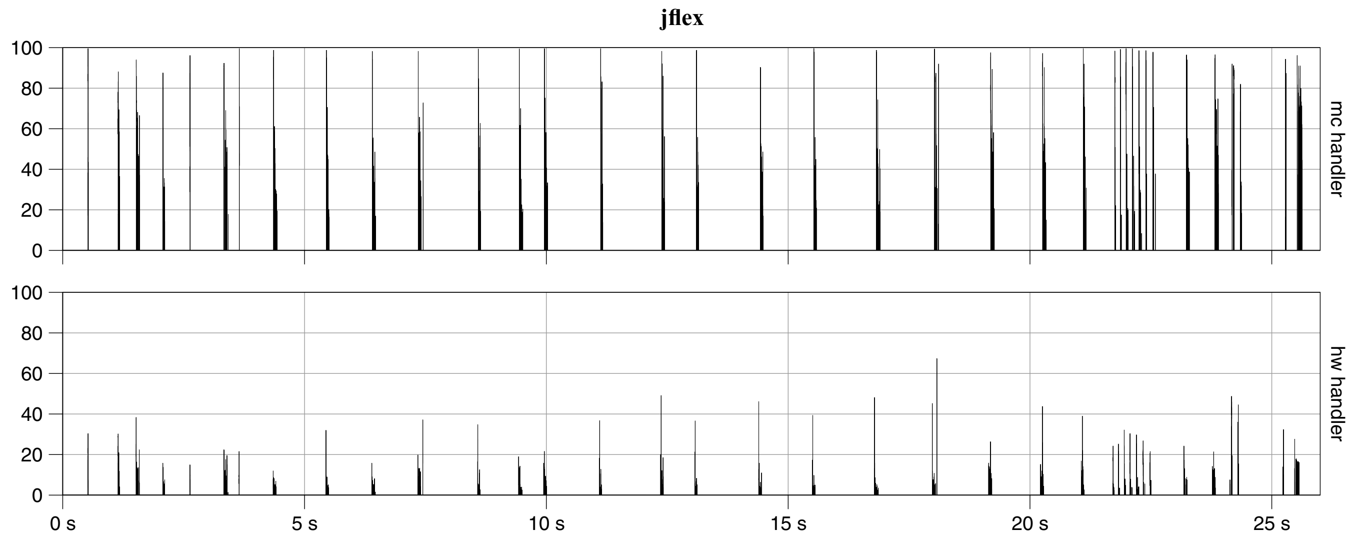
database (first 18 seconds)

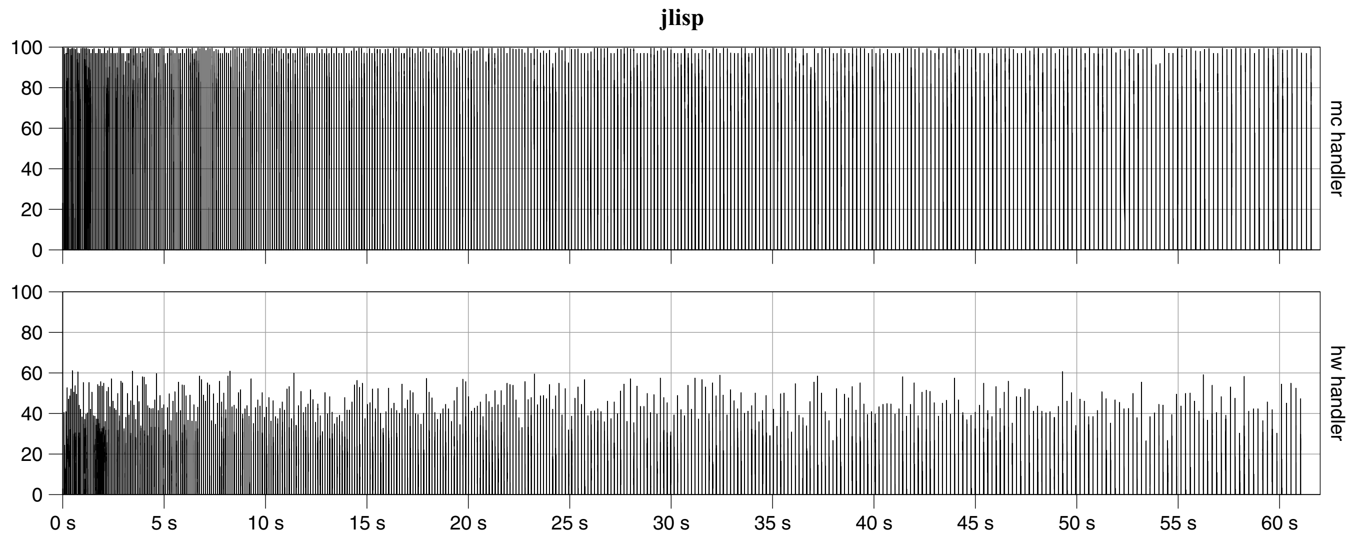


javac




javacc
mc handler
hw handler
0 s
5 s
10 s
15 s
20 s
0
20
40
60
80
100


jflex
mc handler
hw handler
0 s
5 s
10 s
15 s
20 s
25 s
0
20
40
60
80
100


jlisp
mc handler
hw handler
0 s
5 s
10 s
15 s
20 s
25 s
30 s
35 s
40 s
45 s
50 s
55 s
60 s
0
20
40
60
80
100