Fine-Grained Parallel Compacting Garbage Collection through Hardware-Supported Synchronization

Oswin Horvath and Matthias Meyer
Institute of Communication Networks and Computer Engineering
Universität Stuttgart
Stuttgart, Germany
{oswin.horvath, matthias.meyer}@ikr.uni-stuttgart.de

***Abstract*—Parallel garbage collection seeks to exploit the inherent parallelism of graph tracing by evenly distributing the set of objects in the heap among all available processing resources. Any straightforward implementation, however, suffers from prohibitive overheads since each access to the worklist of objects and to the objects themselves needs to be protected by synchronization, especially so in the case of compacting collectors. For this reason, known parallel collectors sacrifice a great deal of work distribution granularity and scalability to keep the synchronization costs acceptable.**

In this paper, we present a case study of a different approach. Our parallel compacting collector is based on Cheney's copying algorithm, employs a single worklist and distributes garbage collection work on an object-by-object basis. This way, it achieves well balanced work distribution and good scalability. To solve the synchronization problem, we introduce a low-cost multi-core garbage collection coprocessor and take advantage of hardware-supported synchronization.

We built an FPGA-based prototype with a single-core main processor supported by a multi-core garbage collection coprocessor. Measurement results show that an 8-core garbage collection coprocessor decreases the duration of garbage collection cycles by a factor of up to 7.4, while a 16-core configuration still achieves a factor of up to 12.1.

***Keywords*-Parallel Garbage Collection, Hardware Support, Object Based Processor Architectures, Synchronization**

I. Introduction

Modern programming languages such as Java, C# or Haskell rely on garbage collection (GC) because it offers three significant advantages over manual memory management [1]: First, GC relieves the programmer from the laborious and fault-prone task of memory management; second, GC prevents common severe programming errors (i.e. dangling references and memory leaks); third, GC is mandatory for modular software design.

The two fundamental classes of GC algorithms are reference-counting and tracing. As pure reference-counting algorithms incur severe runtime overheads and cannot collect cyclic data structures, most of today's algorithms belong to the latter class, i.e. they identify reachable objects by traversing the object graph in the heap, starting from a set of roots (i.e. processor registers and stacks). These algorithms keep objects that still have to be traced in a pool, usually organized as a list or a stack. An abstract description of tracing garbage collection follows:

```
for every object $o$ referenced by the roots do
    mark $o$
    add $o$ to $pool$
end for
while $pool$ is not empty do
    remove an object $o$ from $pool$
    for every unmarked child $c$ of $o$ do
        mark $c$
        add $c$ to $pool$
    end for
end while
```

Some tracing GC algorithms compact the heap, either in a separate compaction phase (mark-sweep-compact GC) or inherently while they trace the object graph (copying GC).

In today's multi-core systems, garbage collection faces a new challenge: As multi-threaded applications take advantage of many processors and allocate memory with a highly increased bandwidth, a single-threaded garbage collector will not be able to keep up and will threaten to become a serious performance bottleneck [2, 3, 4]. Therefore, it is imperative to parallelize garbage collection algorithms and to implement them in a multi-threaded way.

Parallel tracing garbage collection, however, faces three fundamental questions: (1) How is the collection work *decomposed* into tasks? (2) How are these tasks *assigned* to processes? (3) How are the processes *orchestrated* to prevent inconsistencies due to concurrent accesses? [5]

A natural answer to the first question is to define the scanning of a single object as a task, which effectively makes *pool* a task pool. The obvious answer to the second question is to let the algorithm dynamically assign these tasks to processes by granting all processes access to *pool*. This scheme promises that, at any given time, a sufficient number of tasks is available for execution. As a result, thanks to the shared nature of the object pool, this scheme achieves perfect work balancing in theory.

It is challenging to answer the third question, i.e. to exploit the concurrency and work-balancing potential of a fine-grained approach in an efficient way. All processes must synchronize their accesses to *pool* (add, remove) as well as to the object graph. As typical object sizes lie in the range of 10 to 50 bytes [6], the associated synchronization operations become so frequent that they render this approach prohibitively expensive on standard shared memory based platforms. As a result, known parallel collectors reduce the frequency of synchronization operations by sacrificing work distribution granularity as well as scalability to keep synchronization costs acceptable.

Rather than reducing the frequency of synchronization operations, we pursue a different approach and dramatically reduce the cost of synchronization. We achieve this by a specialized multi-core garbage collection coprocessor that takes advantage of hardware-supported synchronization to efficiently coordinate garbage collection tasks.

This paper is organized as follows: In Section II, we review the starting point of our study, Cheney's copying garbage collector. Section III discusses previous work on parallel tracing garbage collection. In Section IV, we derive a fine-grained parallel variant of Cheney's collector. In Section V, we introduce hardware-support that enables an efficient implementation of this algorithm. Finally, we present measurements obtained from our FPGA-based prototype in Section VI.

II. Cheney's Copying Collector

Copying collectors like Cheney's [7] divide the heap into two areas called semispaces. During a garbage collection cycle, all objects that are reachable from a set of roots are copied from one semispace (fromspace) to the other semispace (tospace). 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 fromspace object. If not, it

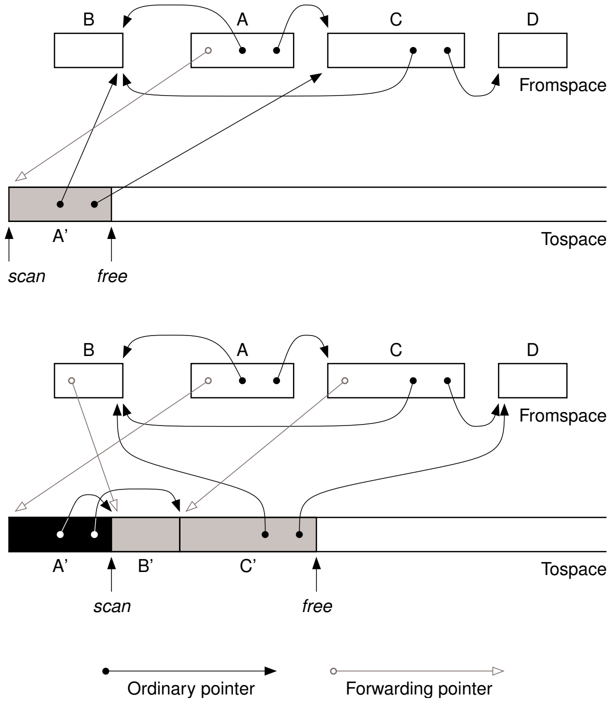


Figure 1. Cheney's sequential copying algorithm

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 [8]. 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.

III. Parallel Tracing Garbage Collection

To parallelize Cheney's algorithm, we have to synchronize accesses to the work pool (i.e. the objects between *scan* and *free*) and to the object graph (for marking objects as evacuated and installing or following forwarding pointers). The challenge is to do this efficiently. As all parallel tracing collector face the same challenge, we'll summarize some representative algorithms.

We first review how existing collectors tackle the question of how to distribute work units in a scalable way.

Halstead's parallel variant of Baker's copying collector [9, 10] statically assigns work to processes by partitioning the heap into process-local from- and tospaces. During a cycle, each process collects only its local heap partition and therefore does not interfere with the other processes' work. However, any algorithm that statically partitions the heap faces work balancing problems because of the unpredictable structure of the object graph.

Imai and Tick [11] extend an idea of Miller and Epstein [12] and distribute work in coarser-grained units in order to reduce contention. They dynamically partition the heap into chunks of constant size. At any given time, a process scans a single chunk and copies surviving objects to another chunk. The algorithm organizes references to chunks to be scanned as a shared stack, replacing object-level granularity by chunk-level granularity. The two major drawbacks of this approach are (1) fragmentation, canceling one of the main advantages of copying garbage collectors, and (2) the need for a dynamic auxiliary data structure apart from the heap.

Another class of algorithms, introduced by Ossia et al. [13], divides the collection work into work-packets, each containing references to a set of gray objects. Each process repeatedly removes a single packet from a shared packet pool, locally scans the objects referenced by this packet, and inserts packets with new gray references into the shared pool, thereby replacing object-level granularity by packet-level granularity.

Other algorithms replace the shared pool by multiple process-local pools. One example is Endo et al.'s [15] parallel variant of the Boehm-Demers-Weiser mark-sweep collector [14]. To avoid work imbalances, a process exposes some of the objects in its pool to the other processes. A process may steal some of these exposed objects when its own pool is empty. Flood [16] uses this work-stealing idea to parallelize both a copying and a mark-compact collector. One difference compared to Endo et al. is that other processes may directly access objects in all pools, not only a dedicated subset. Endo et al. report speedups of up to 11.1 on a 16-core machine, but only for the mark phase. Flood et al. achieve speedups for their compacting garbage collector of 3.9 on a 8-core machine.

Cheng and Blelloch [17] employ work-sharing: Every process owns a local pool that other processes cannot access. In order to distribute work, processes periodically push some objects to a shared pool. The drawback of this approach is that there is high contention in accessing this pool. The authors report a mean runtime overhead of the various synchronization operations of 37 %.

Wu and Li [18] use task-pushing to distribute work among processes: They introduce an object queue for every pair of processes (A, B). Process A pushes objects into the queue, while process B reads from the queue. These single-writer/single-reader queues can be implemented without heavy-weight synchronization primitives. They report speedups of up to 7.5 for the mark phase on a 16-core machine.

Parallel compaction basically uses the same idea to distribute work among multiple processes, i.e. objects or memory areas are combined to coarser-grained units, see e.g. [20, 21, 22].

After this summary of known approaches for work distribution, we are now going to review how previous work tackles the problems that arise when multiple processes are to access the object graph in a both synchronized and scalable way.

There are several publications that focus on reducing the contention a compacting collector faces when it accesses the target area. In Flood's copying collector [16], processes reserve regions in tospace ("local allocation buffers"). Then, they can consecutively evacuate multiple objects without further synchronization. The drawback of this approach is tospace fragmentation, which motivated the work of Petrank and Kolodner [19] that propose that each process pools multiple evacuations before it allocates a heap region of appropriate size.

In summary, all approaches to parallel garbage collection decouple collector processes by decreasing the granularity of a work unit and/or by partitioning the work list among processes. Similarly, accesses to the object graph are commonly performed at coarser granularities. Problems are (1) increased algorithm complexity, (2) fragmentation, (3) auxiliary dynamic data structures apart from the heap, and (4) poor load balancing.

IV. Parallelizing Cheney's Algorithm

We will now assume that, in contrast to the previously presented algorithms, we don't have to avoid synchronization. As a consequence, we can directly apply the "ideal" approach sketched in the introduction. In this section, we will derive a parallel variant of Cheney's algorithm. Gray objects represent tasks and are held in a centralized work list. We further assume that every object possesses a header that can hold mark state, object size, and forwarding pointers or backlinks.

Under these assumptions, synchronization is necessary to assure that:

1) *every gray object is assigned to exactly one process.* We achieve this by providing atomic access to the *scan* pointer.
2) *every object is only evacuated once.* We achieve this by providing atomic access to object headers.
3) *each object is assigned to an exclusive area in tospace.* We achieve this by providing atomic access to the *free* pointer.

All processes scan tospace in parallel. The following pseudo-code summarizes the corresponding main scanning loop:

```
with locked $scan$ do
  Read header of object $o$ at scan
  Increment $scan$ by object size
end with
for $ptr \in$ object $o$ do
  with locked header of $c := *ptr$ do
    Read header of $c$
    if $c$ not marked then
      with locked $free$ do
        Mark c
        Install forwarding pointer in header of $c$
        Install backlink to $c$ at $free$
        Increment $free$
      end with
    end if
  end with
  Replace $ptr$ in tospace copy of $o$
end for
```

The algorithm terminates when *scan* reaches *free* and when no process is currently scanning an object. To check for this condition, each process sets a busy flag while it executes the main scanning loop. Furthermore, each process atomically checks the state of all busy flags while it compares *scan* to *free*.

The algorithm exhibits the fixed lock ordering scheme $scan < header < free$: While a process locks a header, it never tries to lock *scan*; and while a process holds the *free* lock, it neither locks a header nor *scan*. As Habermann [23] shows, this ordering ensures that deadlocks cannot occur.

V. Implementation

A. Challenges

In shared memory based systems, all synchronization mechanisms ultimately rely on variables in shared memory. In contrast to regular data, lock variables are frequently accessed by many different processors. This causes a high amount of communication traffic to transfer data trough the memory hierarchy and to ensure global write ordering. Moreover, advanced processor features such as caches, write buffers, out-of-order and speculative execution make it even more costly to maintain coherent lock variables [24].

Lock accesses can negatively impact the performance of accesses to unrelated data. In particular, as data is transferred through the memory hierarchy at cache line granularity, cache flushes required for lock coherency will often remove unrelated data from the cache (false sharing). Another problem is that accesses to locks and data must frequently be ordered. This is e.g. required to assure that accesses to data structures protected by a lock variable are only performed within the corresponding critical section. The programmer can enforce such orderings only on a coarse level by expensive memory barriers. For example, such a memory barrier may force all stores preceding the barrier to complete before succeeding stores are permitted.

By employing a custom architecture, we can use a more efficient approach: As every process in the algorithm presented in Section IV, at any given time, holds a bounded number of locks only (*scan*, *free* and a single header), these locks can be held in registers. This allows us to efficiently access these locks and to isolate lock accesses from data accesses. Furthermore, we exploit the highly regular access patterns of GC algorithms to provide mechanisms for coherency and ordering between memory accesses of collector processes only where required.

B. Overview

Our implementation builds upon the architecture introduced in [25, 26, 27, 28, 29]. The main motivation for this architecture is fine-grained real-time garbage collection, i.e. to guarantee, for this first time, that GC pauses never exceed a couple of hundred clock cycles. This goal is achieved by a special GC coprocessor that is tightly coupled to a main processor to efficiently synchronize GC with application programs. To allow for this kind of efficient synchronization, objects and pointers must be known at the hardware level. For this purpose, the system's main processor implements an object-based memory model and strictly separates pointer from non-pointer data.

In this paper, we apply the same approach of fine-grained synchronization in hardware to parallel garbage collection. Whereas we have to synchronize a single application with a single collector for concurrent garbage collection, we now synchronize multiple garbage collection processes. For this purpose, we designed a multi-core GC coprocessor (Figure 2) that consists of N microprogrammed cores, each executing a single process, one synchronization block (SB) that maintains the global synchronization state, and a memory access scheduler that enforces memory orderings. As our primary focus lies on parallelizing GC, the coprocessor currently stops the main processor for the whole collection cycle. However, as a next step, we intend to allow the multi-core coprocessor to run concurrently to the main processor.

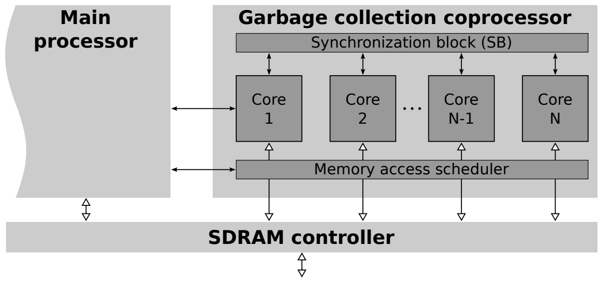


Figure 2. System overview

Each core of the coprocessor has a register file, two arithmetic logic units (ALUs), four buffers for asynchronous memory accesses, and a control unit that implements the garbage collection algorithm as a single microprogram. Each core performs up to two arithmetic-logic operations and can initiate up to four memory operations per clock cycle.

C. Synchronization

Scan and *free* are implemented as registers in the SB that can simultaneously be read by all cores. At most one core may modify each of these two registers during a clock cycle. For this purpose, the synchronization control block contains a lock for *scan* and a lock for *free*. The cores can acquire and release these locks via micro operations. If a core tries to acquire a lock that is currently held by another core, the SB will stall that core until the current owner releases the lock. In the case of multiple cores simultaneously claiming a lock, the SB applies a static prioritization scheme to determine which core acquires the lock next.

Each core owns a header lock register in the synchronization block. In contrast to the locks associated with *scan* and *free*, each core can only change the lock state of its own register. When a core tries to acquire a header lock, the SB compares the content of the core's header-lock register to the content of all other header lock registers in parallel. If it finds a match, the SB will stall that core. The case of multiple cores trying to lock the same header simultaneously is again resolved with a static prioritization scheme.

To implement the termination detection scheme described in Section IV, the synchronization block contains a register *ScanState*. Each core is assigned a busy bit in this register, and all cores can read this register simultaneously.

To ensure that no core enters the scan loop before Core 1 has initialized *scan* and *free* and that the main processor is not restarted before all buffers have been flushed, we implement a mechanism for barrier synchronization: Any micro-instruction can be marked as synchronizing. When a core executes such a micro-instruction, the SB stalls the core until all cores have reached a synchronizing micro instruction.

All these synchronization operations incur no clock cycle penalty in the uncontended case. In particular, a core acquires a free lock within one clock cycle and this acquisition is executed in parallel to other micro-operations. Similarly, a lock can be released by one core and reacquired by another core in the same cycle.

D. Object Layout and Memory Interface

Figure 3 shows the structure of objects in our system. Each object is partitioned into a pointer area of length π and a data area of length δ. π and δ as well as GC-related information such as mark state and forwarding pointers are referred to as attributes and stored in a two-word header.

Figure 4 shows the states that an object assumes during a GC cycle: The first picture shows the initial state of an object in tospace *(White)*. When evacuating the object, a collector process allocates an empty object frame in tospace and installs both a forwarding pointer in fromspace and

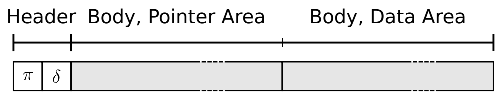


Figure 3. Structure of objects in memory

a backlink in tospace. Additionally, it sets a bit in the fromspace object header to indicate that the object has been evacuated *(Gray 1)*. Later, when *scan* reaches the object frame in tospace, a process sequentially copies the body from the fromspace original to the tospace copy *(Gray 2)*. As in Cheney's sequential algorithm, the process evacuates any unmarked fromspace objects that are discovered during this stage. Finally, when the process has completely copied the object, it writes π and δ into the header of the tospace copy *(Black)*.

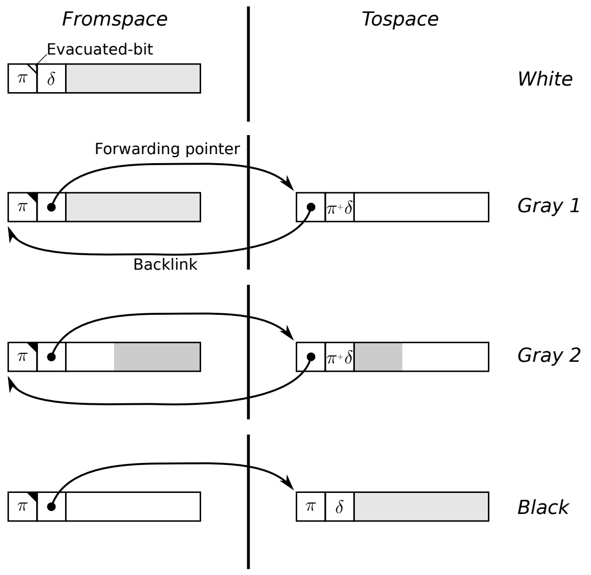


Figure 4. Object states

As we design the coprocessor specifically for garbage collection, we can exploit the highly regular access patterns of GC for the design of the memory interface and the memory access scheduler. In contrast to body accesses that are highly sequential, header accesses show no spatial locality. Furthermore, while headers in from- and tospace are accessed multiple times and potentially by different cores, only one core accesses an object's body areas, and accesses each word exactly once.

We differentiate between those two access types and provide each core with separate load and store ports for headers and bodies. The actual transfers between main memory and the four buffers of each core are handled asynchronously: Each core may initiate a transfer at any time and will only stall when it tries to write to a store buffer while the previous store is not complete or when it tries to read from a load buffer while the corresponding load is not complete. In total, the memory interface allows up to $4 \times N$ pending requests, using a split-transaction scheme for a high degree of latency tolerance.

The memory access scheduler enforces an ordering between memory operations only where required. In particular, as there is no overlap between headers and bodies, the memory access scheduler and the controller handle header and body accesses completely independently of each other.

Each word in the body area in both fromspace and tospace is either written or read exactly once during a collection cycle. Furthermore, these accesses are independent of each other. Therefore, no ordering has to be enforced for body accesses. The memory access scheduler must only ensure that loads are not performed before all stores of the previous collection cycle have been committed. For this purpose, it simply flushes all buffers at the end of a GC cycle.

The situation is different for object headers: Headers in fromspace are written exactly once (when the object is grayed) and read at least once (immediately before the object is grayed and whenever a parent of the object is scanned), potentially by a different core. Headers in tospace are first written during evacuation, then read once (potentially by a different core) during scanning and finally written a second time to blacken the object. The memory access scheduler orders header loads and stores by ensuring that a header load is delayed whenever there is a store pending for the same location. This is achieved by a simple comparator array. No logic is necessary to enforce write ordering, as the locking protocol ensures that there is only one writer for each header.

Finally, we introduce an optimization to accelerate accesses to tospace headers. In our algorithm, *scan* can only be advanced after the size of the object at *scan* is known, i.e. after its tospace header has been read. Therefore, these accesses can become a bottleneck. However, as gray tospace headers are read in exactly the same order as they are written, we buffer them in an on-chip header FIFO. As long as the number of gray objects does not exceed the capacity of this queue, no memory accesses are required for header accesses.

E. Coordination with the Main Processor

Core 1 stops the main processor when the current semi-space is full and restarts the main processor when the collection cycle has finished. Core 1 also accesses the main processor's registers in order to access the root set. Furthermore, Core 1 flushes the main processor's caches at the beginning of a collection cycle and restarts the main processor when all GC store buffers are empty.

VI. Experimental Results

A. Measurement Platform

To demonstrate the feasibility and efficiency of our approach, we developed an FPGA-based prototype based on the system presented in [26]. The protoype board contains an Altera Stratix II FPGA (EP2S130 [30]), a standard DDR-SDRAM module and various peripheral devices, including an Ethernet interface for file access via NFS. The entire prototype is synchronously operated at 25 MHz.

The main processor is realized as a statically scheduled 3-way multiple-issue explicitly parallel 32-bit RISC, with 8K instruction cache, 8K data cache, and 2K header cache. The garbage collection coprocessor consists of up to 16 cores, each including a microcode memory of 180 words with 96 bit each. The coprocessor's header FIFO has a capacity of up to 32k entries. The 16-core configuration of the coprocessor requires approximately twice the chip area of the main processor.

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

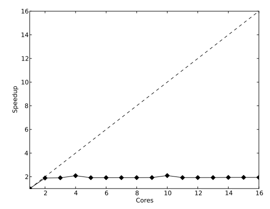
On the software side, we have developed a static Java compiler that translates standard Java bytecode to the main processor's native machine code. Moreover, we realized a subset of the Java class libraries supporting text-based, single-threaded applications in order to facilitate the execution of representative programs. As our implementation currently does neither support multiple threads nor reflection, we had to limit the benchmarks to the programs mentioned in the next section.

B. Measurement Results

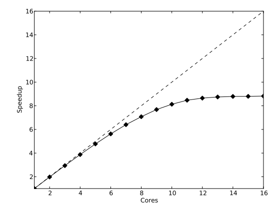
In our experiments, the heap size had little to no influence on the measurement results regarding synchronization overhead and scalability. Therefore, we dimensioned the heap according to a rule of thumb and chose twice the minimal heap size.

In a first experiment, we determined the speedup in garbage collection time for various numbers of GC cores (Figure 5). As the base for these comparisons, we used a configuration where only one core is enabled. Because synchronization operations incur no clock cycle penalty in the uncontended case, this single-core configuration performs like the original sequential implementation of Cheney's algorithm.

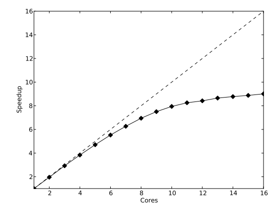
Most applications scale reasonable well. However, two benchmarks (*compress* and *search*) don't show any significant speedup. For these simple benchmarks, object-level parallelization does not offer enough parallelism, as the corresponding object graphs show highly linear structures (compare [31]). To quantify a benchmark's degree of object-level parallelism (or the lack thereof), we measured the number of clock cycles during which *scan* equals *free*. In



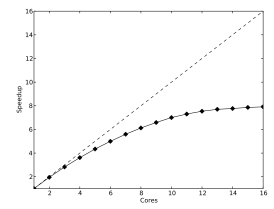
(a) compress



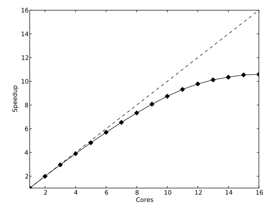
(b) cup



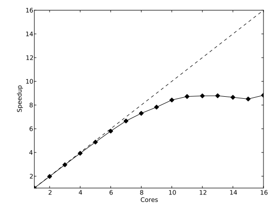
(c) db



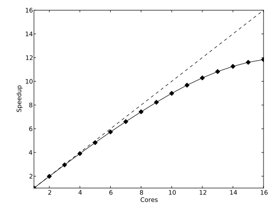
(d) javac



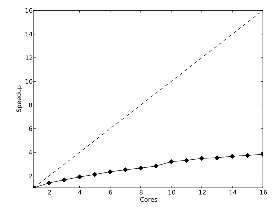
(e) javacc



(f) jflex



(g) jlisp



(h) search

Figure 5. Scaling behavior

Table I
FRACTION OF CLOCK CYCLES DURING WHICH WORK LIST IS EMPTY

Application	1 core	2 cores	4 cores	8 cores	16 cores
compress	0.01 %	0.15 %	98.58 %	99.43 %	99.72 %
cup	0.00 %	0.01 %	0.02 %	0.04 %	0.10 %
db	0.00 %	0.01 %	0.02 %	0.03 %	0.06 %
javac	0.00 %	0.01 %	0.01 %	0.03 %	0.08 %
javacc	0.15 %	0.57 %	1.35 %	3.06 %	5.34 %
jflex	0.02 %	0.05 %	0.13 %	5.48 %	35.35 %
jlisp	0.10 %	0.27 %	0.61 %	1.34 %	2.59 %
search	0.06 %	73.74 %	98.75 %	99.53 %	99.76 %

these clock cycles, there are no gray objects available for processing. Tab. I lists the fraction of these clock cycles relative to the total number of clock cycles. The table shows that *jflex* also suffers from limited object-level parallelism, yet to a lesser extent.

Next, we determined the mean amount of time each core is stalled because of memory accesses and synchronization. In Tab. II, the column "Total" lists the mean number of clock cycles per collection cycle, while the remaining columns list both the absolute and relative number of stall cycles.

Generally, few stalls are caused by synchronization operations. One exception is *javac*, where a high amount of conflicting header accesses occur, which indicates that a few objects are referenced by many objects. We hope to improve our implementation by reading the mark bit without prior acquisition of the header lock and by attempting a locking read only if the mark bit is cleared. The second exception is *cup*, where the header FIFO overflows and the resulting memory accesses prolong the critical section protected by the *scan* lock.

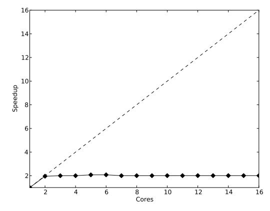
Finally, we present some preliminary results concerning the influence of the processor-memory interface on the scalability of the coprocessor. Our prototype processors operate at a clock rate of 25 MHz, whereas the DDR-SDRAM needs to be operated at a clock rate of at least 100 MHz. Therefore, the relation of processor speed to memory latencies and memory throughput are not very typical for non-prototype systems. In our system, the memory access latency is in the range of a few clock cycles, compared to up to hundreds of clock cycles in non-prototype systems. Therefore we performed another experiment where we added an artificial latency of 20 clock cycles to each memory access. Figure 6 shows that this increased latency significantly improves scalability for all benchmarks that offer a sufficient degree of object-level parallelism. The reason for this counter-intuitive behavior is that the higher the memory latency, the higher the fraction of time each core is stalled. Consequently, more cores are required to exhaust the available memory bandwidth.

VII. CONCLUSIONS AND FURTHER WORK

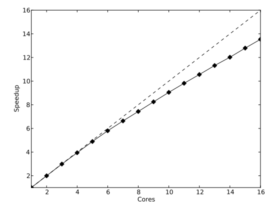
We demonstrated the efficient implementation of a fine-grained parallel compacting garbage collector by taking advantage of hardware-supported synchronization. In particular, we showed that this kind of synchronization does not impair the scalability of the collector.

Our experiments show that two remaining issues limit scalability: (1) Limited object-level parallelism and (2) limited memory bandwidth.

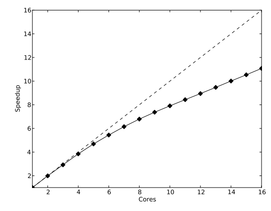
Therefore, we are currently investigating improvements that allow us (1) to distribute work at a finer granularity than object-level granularity, e.g. at the granularity of cache lines, and (2) to make better use of the available memory bandwidth, e.g. by header caches in conjunction with an optimized header FIFO.



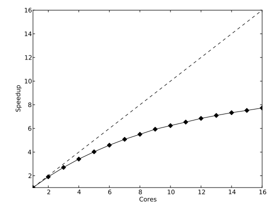
(a) compress



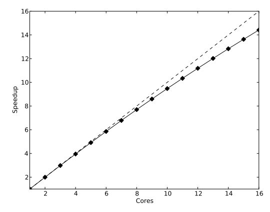
(b) cup



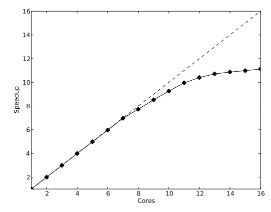
(c) db



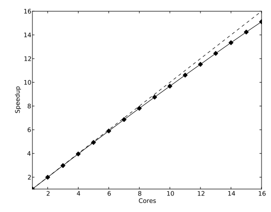
(d) javac



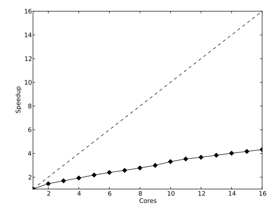
(e) javacc



(f) jflex



(g) jlisp



(h) search

Figure 6. Scaling behavior (more realistic memory latency, see text)

Table II
CLOCK CYCLE DISTRIBUTION (FOR 16 CORES)

Application	Total	Scan-lock stall	Free-lock stall	Header-lock stall	Body load stall	Body store stall	Header load stall	Header store stall
compress	4735060	113 (0.00 %)	4 (0.00 %)	38 (0.00 %)	75023 (1.58 %)	14626 (0.31 %)	2821 (0.06 %)	0 (0.00 %)
cup	3251965	341040 (10.49 %)	2940 (0.09 %)	7917 (0.24 %)	493847 (15.19 %)	4074 (0.13 %)	1254764 (38.58 %)	337 (0.01 %)
db	1089535	20633 (1.89 %)	893 (0.08 %)	1195 (0.11 %)	232208 (21.31 %)	6174 (0.57 %)	360913 (33.13 %)	0 (0.00 %)
javac	2141803	19067 (0.89 %)	1019 (0.05 %)	629596 (29.40 %)	235314 (10.99 %)	4442 (0.21 %)	560618 (26.18 %)	0 (0.00 %)
javacc	542825	18289 (3.37 %)	340 (0.06 %)	837 (0.15 %)	101272 (18.66 %)	2900 (0.53 %)	153939 (28.36 %)	0 (0.00 %)
jflex	411784	1517 (0.37 %)	96 (0.02 %)	208 (0.05 %)	55538 (13.49 %)	3809 (0.93 %)	44618 (10.84 %)	0 (0.00 %)
jlisp	37247	724 (1.94 %)	30 (0.08 %)	161 (0.43 %)	5468 (14.68 %)	243 (0.65 %)	10527 (28.26 %)	0 (0.00 %)
searchA	5916511	113 (0.00 %)	4 (0.00 %)	41 (0.00 %)	64849 (1.10 %)	15542 (0.26 %)	2953 (0.05 %)	0 (0.00 %)

In a next step, we plan to combine parallel garbage collection (presented in this paper) and real-time garbage collection (presented in our previous work) in order to realize a fine-grained parallel and real-time garbage collector.

REFERENCES

[1] R. E. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: Wiley, Jul. 1996, with a chapter on Distributed Garbage Collection by R. Lins.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, Jun. 2004, pp. 25–36.

[3] F. Xian, W. Srisa-an, and H. Jiang, "Garbage collection: Java application servers' Achilles heel," vol. 70, no. 2–3, Feb. 2008.

[4] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: Evaluation methodology for the 21st century," *Communications of the ACM*, vol. 51, no. 1, pp. 83–89, 2008.

[5] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998.

[6] S. Blackburn, R. Garner, K. S. McKinley, A. Diwan, S. Z. Guyer, A. Hosking, J. E. B. Moss, and D. Stefanović, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the Twenty-First ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. ACM SIGPLAN Notices 41(10), Portland, OR, USA, Oct. 2006.

[7] C. J. Cheney, "A non-recursive list compacting algorithm," *Communications of the ACM*, vol. 13, no. 11, pp. 677–8, Nov. 1970.

[8] 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, pp. 965–975, Nov. 1978.

[9] H. G. Baker, "List processing in real-time on a serial computer," *Communications of the ACM*, vol. 21, no. 4, pp. 280–94, 1978, also AI Laboratory Working Paper 139, 1977.

[10] R. H. Halstead, "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, Oct. 1985.

[11] A. Imai and E. Tick, "Evaluation of parallel copying garbage collection on a shared-memory multiprocessor," Institute for New Generation Computer Technology, ICOT technical report TR-650, May 1991.

[12] J. S. Miller and B. S. Epstein, "Garbage collection in multischeme," in *Proceedings of the US/Japan workshop on Parallel Lisp on Parallel Lisp: languages and systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 138–160.

[13] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, "A parallel, incremental and concurrent GC for servers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. ACM SIGPLAN Notices 37(5), Berlin, Germany, Jun. 2002, pp. 129–140.

[14] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[15] T. Endo, K. Taura, and A. Yonezawa, "A scalable mark-sweep garbage collector on large-scale shared-memory machines," in *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[16] C. Flood, D. Detlefs, N. Shavit, and C. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Proceedings of the First Java Virtual Machine Research and Technology Symposium*. Monterey, CA, USA: USENIX, Apr. 2001.

[17] P. Cheng and G. Blelloch, "A parallel, real-time garbage collector," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. ACM SIGPLAN Notices 36(5), Snowbird, UT, USA, Jun. 2001, pp. 125–136.

[18] M. Wu and X.-F. Li, "Task-pushing: a scalable parallel GC marking algorithm without synchronization operations," in *IEEE International Parallel and Distribution Processing Symposium (IPDPS) 2007*, Long Beach, CA, Mar. 2007.

[19] E. Petrank and E. K. Kolodner, "Parallel copying garbage collection using delayed allocation," *Parallel Processing Letters*, vol. 14, no. 2, Jun. 2004.

[20] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein, "An efficient parallel heap compaction algorithm," in *Proceedings of the Nineteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. ACM SIGPLAN Notices 39(10), vol. 39, no. 10. New York, NY, USA: ACM, Oct. 2004, pp. 224–236.

[21] H. Kermany and E. Petrank, "The Compressor: Concurrent, incremental and parallel compaction," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. ACM SIGPLAN Notices 41(6), M. I. Schwartzbach and T. Ball, Eds., Ottawa, Canada, Jun. 2006, pp. 354–363.

[22] M. Wegiel and C. Krintz, "The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural support for programming languages and operating systems*. Seattle, WA, USA: ACM Press, 2008, pp. 91–102.

[23] A. N. Habermann, "Prevention of system deadlocks," *Commun. ACM*, vol. 12, no. 7, pp. 373–ff., 1969.

[24] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[25] M. Meyer, "A novel processor architecture with exact tag-free pointers," in *2nd Workshop on Application Specific Processors*, San Diego, CA, 2003, pp. 96–103.

[26] M. Meyer, "An on-chip garbage collection coprocessor for embedded real-time systems," in *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hong Kong, China, Aug. 2005, pp. 517–524.

[27] M. Meyer, "A true hardware read barrier," in *ISMM '06: Proceedings of the 5th international symposium on Memory management*. New York, NY, USA: ACM, 2006, pp. 3–16.

[28] S. Stanchina and M. Meyer, "Mark-sweep or copying? "a best of both worlds" algorithm and a hardware-supported real-time implementation," in *Proceedings of the Sixth International Symposium on Memory Management*, G. Morrisett and M. Sagiv, Eds. Montréal, Canada: ACM Press, Oct. 2007, pp. 173–182.

[29] S. Stanchina and M. Meyer, "Exploiting the efficiency of generational algorithms for hardware-supported real-time garbage collection," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 713–718.

[30] Altera, "Stratix II device family data sheet," May 2007.

[31] F. Siebert, "Limits of parallel marking collection," in *Proceedings of the Seventh International Symposium on Memory Management*, R. Jones and S. Blackburn, Eds. Tucson, AZ, USA: ACM Press, Jun. 2008, pp. 21–29.