Grail Quest: A New Proposal for Hardware-assisted Garbage Collection

Martin Maas Krste Asanović John Kubiatowicz
University of California, Berkeley

ABSTRACT

Many big data systems are written in garbage-collected languages and GC has a substantial impact on throughput, responsiveness and predicability of these systems. However, despite decades of research, there is still no "Holy Grail" of GC: a collector efficient enough to be unnoticeable to the application. Such a collector needs to achieve freedom from pauses, high GC throughput and good memory utilization, without slowing down application threads or using substantial amounts of compute resources.

In this paper, we present a proposal towards this elusive goal by reviving the old idea of moving GC into hardware. We discuss the trends that make it the perfect time to revisit GC in hardware and present the design of a hardware-assisted GC that aims to reconcile the conflicting goals. Our system is work in progress and we discuss design choices, trade-offs and open questions.

1. INTRODUCTION

A substantial portion of big data frameworks – and large-scale distributed workloads in general – are written in languages with Garbage Collection (GC), such as Java, Scala, Python or R. Due to their importance for a wide range of workloads, Garbage Collectors have seen tremendous research efforts for over 50 years. Yet, we arguably still don't have what has been called the "Holy Grail" of GC [1]: a pause-free collector that achieves high memory utilization and high GC throughput (i.e., sustaining high allocation rates), without slowing down the application or consuming substantial resources.

Many recent GC innovations have focused on the first three goals, and modern GCs can be made effectively pause-free at the cost of slowing down application threads and using a substantial amount of resources. However, these approaches oftentimes ignore a factor that is very important in warehouse-scale computers: energy consumption. Previous work [2] has shown that GC can account for up to 25% of energy consumption and 40% of execution time in common workloads. Worse, as big data systems are processing ever larger heaps, these numbers are expected to increase.

We believe that we can reconcile low pause times and energy efficiency by revisiting the old idea of moving GC into hardware. Our goal is to build a GC that simultaneously achieves high GC throughput, good memory utilization, pause times indistinguishable from LLC misses and energy efficiency. We build on an algorithm that performs well on the first three criteria but is resource-intensive [3]. Our key insight is that this algorithm can be made energy efficient by moving it into hardware, combined with several algorithmic changes.

We are not the first to propose hardware support for GC [3–7]. However, none of these schemes has been widely adopted. We believe that there are three reasons:

Garbage-collected languages are widely used, but they are rarely the only workload on a system. Systems designed for specific languages mostly lost out to general-purpose cores, partly due to Moore's law and economies of scale allowing these systems to quickly outperform the specialized ones. This is changing, with Moore's law slowing down and the advent of accelerators making it feasible to use chip area to improve common workloads, such as garbage-collected applications.

Most garbage-collected workloads run on servers (note that there are exceptions – for example, Android uses GC). Servers traditionally use commodity CPUs and the bar for adding hardware-support into such a chip is very high (e.g., it took a long time for HTM to get adopted). However, this is changing: cloud hardware and rack-scale machines in general are expected to switch to custom SoCs, which could easily incorporate IP to improve GC performance and efficiency.

Many proposals were very invasive and would require re-architecting of the memory system or other components [5, 7, 9]. We believe an approach has to be relatively non-invasive to be adopted. The current trend to accelerators and processing near memory may make it easier to adopt similar techniques for GC without substantial modifications to the architecture.

We therefore think it is time to revisit hardware-assisted GC. In contrast to many previous schemes, we focus on making our design non-invasive enough to be incorporated into a server or mobile SoC. This requires isolating the GC logic into a small number of IP blocks and limiting changes outside these blocks to a minimum.

In this paper, we describe our proposed design. It exploits two insights: First, overheads of concurrent GCs stem from a large number of small but frequent slowdowns spread throughout the execution of the program. We move the culprits (primarily barriers) into hardware to alleviate their impact and allow out-of-order cores to speculate over them. Second, the most resource-intensive phases of a GC (marking and relocation) are a poor fit for general-purpose cores. We move them into specialized units close to DRAM, to reduce power.

2. BACKGROUND

There exists an extensive body of work on GC (a general introduction can be found in [8]). We will now review work that specifically relates to our approach.

There are two fundamental GC strategies: tracing and reference counting. Tracing collectors start from a

```
while (!q.empty()):
  node = q.dequeue(); if (!marked(node)):
    mark(node); q.enqueue(get_references(node))
```

Figure 1: Tracing is typically a BFS where the current frontier is kept in a mark queue *and per-object* mark bits *indicate whether an object has been visited. Every step, we take an object reference off the mark queue, identify all outgoing references stored in object fields and add them to the queue. Once the BFS has finished, the set of mark bits indicates the reachable objects.*

set of *roots* (such as static or stack variables), perform a BFS to determine all reachable objects and then recycle those that are not (Figure 1). Reference counting maintains reachability information on the fly but still requires a tracing backup collector to handle cycles. It can therefore be seen as an optimization.

We distinguish between stop-the-world and concurrent collectors. A stop-the-world collector requires to stop all *mutators* (parlance for application thread) and can only continue once GC has completed, while a concurrent collector operates in parallel to the mutators. Traditionally, concurrent collectors incur high overheads over stop-the-world collectors, as they have to keep the mutators and collector in sync using barriers[1].

Another fundamental distinction is whether or not a collector is relocating – whether or not it moves objects in memory. Many basic GC algorithms (such as Mark & Sweep) are non-relocating – memory therefore gets fragmented over time and locality is poor, which is why production environments often eschew non-relocating GCs. Many popular collectors therefore *compact* memory to reduce fragmentation, increase locality and enable fast bump pointer allocation (instead of free lists). However, relocation in concurrent collectors is very difficult.

2.1 Related Work

The closest to our work is Azul Vega [3], a commercial CPU specialized for Java applications. Vega uses the same basic GC algorithm, but executes most of it in software. Its main hardware support consists of a read barrier instruction that delivers a fast user-level trap to respond to relocation. Azul has since stopped producing hardware, implementing the read barrier in software on commodity CPUs. We believe that by moving much more of the algorithm into hardware, we can substantially improve over Vega in terms of energy efficiency, and potentially mutator and GC throughput. This is a different design point than Vega's, which appears to prioritize generality and flexibility.

10 years ago, Sun worked on a similar idea to ours [9], with specialized GC units close to memory and barriers support. However, it was never built or published.

[1]This is a different use of the term than in memory consistency or synchronization. A barrier is code that is added around certain reference operations; for example, a read barrier is executed whenever a reference is loaded from memory.

There exists work on GC coprocessors in the embedded and real-time space – our work differs in its focus on energy efficiency and server workloads [4]. Other work proposes support for barriers [4,10] and reference counting in hardware [5] (the latter is non-relocating and requires a backup tracing collector). There has also been an extensive body of work on *Java processors*, some of which contain features specific to GC [4].

Finally, a hardware tracing unit was presented by Bacon et al. [6]. However, this work was in the context of GC of Block RAMs on FPGAs, which is a special case and very different from conventional GC on DRAM.

3. GC ALGORITHM

Our design is a concurrent, relocating mark-compact collector. We believe this is the right design point for most modern server applications, as many of them are affected by GC pauses (even when short) and fragmentation is a problem for long-running workloads.

Our collector builds on the Pauseless GC algorithm from Azul [3]. Pauseless almost fully eliminates GC-related pauses (making them indistinguishable from OS context switches), but adds overheads to the mutators, decreasing application throughput. While this is oftentimes an acceptable trade-off for applications that are sensitive to GC-pauses, it may not be suitable for general deployment, in part due to the resulting power and resource overheads. We hypothesize this is a key reason why Azul's market share is not higher, despite arguably being the state-of-the art in concurrent collectors.

The Pauseless GC Algorithm: Pauseless GC has two key components: a mark and a relocation phase. The mark phase continuously performs BFS passes over the heap to produce a fresh set of *mark bits* that tell whether each object is reachable or not (multiple mark bits are maintained for each object). The relocation phase uses the most recent set of mark bits to pick pages in memory that are mostly garbage (i.e., unreachable objects) and compact them into a fresh page (page sizes are chosen to be large). The key idea of the algorithm is how to perform both of these phases concurrently with respect to each other and the mutators (Figure 2).

The only way the mark phase can mistake a reachable object as unreachable is if a concurrently running mutator moves an unvisited reference from memory into a register and therefore "hides" it from the mark phase. Like other schemes, Pauseless GC solves this problem through a read barrier: whenever a reference is loaded into a register, it is also passed to the mark queue to be marked through (i.e., added to the BFS).

To avoid passing the same reference many times, the barrier is "self-healing": it tags the reference in its original memory location such that next time it is encountered, we know that it was already communicated to the marker. This works by using the MSB of references

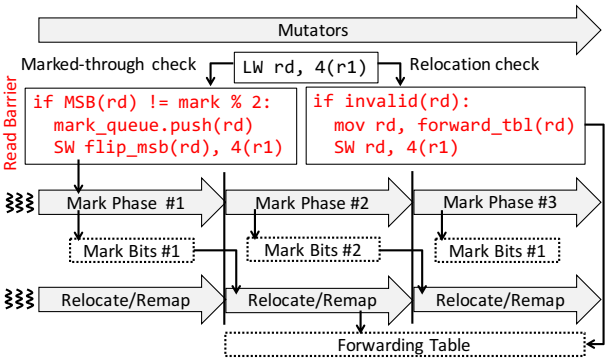


Figure 2: Overview of the Pauseless GC Algorithm. Red code indicates read-barriers following reference loads.

to store an NMT (not-marked-through) bit. The bit indicates whether we have already encountered this reference during the current mark. The barrier is only triggered if the bit does not match the current mark phase, and once it is triggered, it flips the NMT of the reference that triggered it. This way, the read barrier is only triggered once for each reference.

Azul proposed three ways of implementing the barrier: in software by interleaving it with the instruction stream, on the Vega platform (which delivers a fast user-level trap if the NMT bit is wrong), or reusing Virtual Memory (mapping all pages into the half of the virtual address space with the right NMT bit and trapping if the barrier is triggered). This results in a trade-off between either increasing code size, mutator and energy overheads, or incurring the cost of barrier traps.

The relocation phase uses the same mechanism: when an object is moved to another page, many places still contain stale references to the old location. To *remap* these references to the new location, another self-healing read barrier is used. When evacuating a page, it is first marked as protected in the page table, which will trigger the barrier when it is accessed. The relocation phase maintains a forwarding table outside the original page, which maps the old location of each object to its location in the new page. If a mutator tries to access the old page (due to a stale reference), it will trigger the read barrier, use the forwarding table to determine the new location of the object (potentially copying the object if it hasn't yet), and update the location where the reference that caused the trap came from.

The mark phase also remaps relocated references when it encounters them, to guarantee that all references to an object have been remapped after the next mark pass and the old page (and forwarding table) can be freed.

Challenges: By building on Pauseless GC, we hope to avoid correctness problems and focus on performance and efficiency (as it is very difficult to get a concurrent GC right). We address two challenges: 1) substantial compute resources are required to continuously trace the heap and relocate objects, and 2) mutator threads

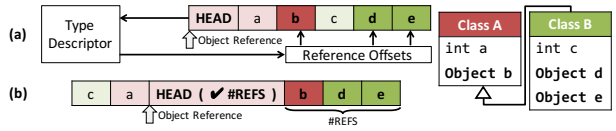


Figure 3: Traditional (a) vs. bidirectional (b) object layout.

incur overheads from barrier code or traps to stay in sync with the collector. These are the main areas we are aiming to improve with hardware support. On the software side, we propose a new memory layout, which allows our hardware to operate more efficiently.

Object Layout: The object layout has important implications for performance and energy consumption. Traditionally, language runtimes lay out fields sequentially, starting with a header, then all parent classes's fields and finally the class's fields itself (Figure 3a). This facilitates casting an object to its parent class. However, it means that references are interspersed throughout the object and the marker will have to look up a table to determine which fields contain references. While this causes little overhead on an out-of-order core, a different layout allows us to build a more efficient marker in hardware that requires much less area and power.

We use a *bidirectional* layout (Figure 3b) inspired by the Sable VM [11] (Sun patented a similar idea [9]). The header is in the middle of the object, all non-reference fields are to the left of the header, and reference fields are to the right (Figure 3b). The advantage of this layout is that the marker only needs to read and mark the header (which we extend with the number of references) and then read all references in a single stride.

4. HARDWARE DESIGN

We observe that both the mark and relocation phase in the Pauseless GC algorithm are a poor fit for general-purpose cores, especially in terms of power. We also hypothesize that taking a trap for each triggered read barrier is inefficient (Figure 5), similar to the argument for refilling TLBs with a hardware page table walker.

For the mark's BFS traversal to be efficient, we need to keep as many memory requests in flight as possible to maximize memory bandwidth. While an out-of-order core is very good at this, it adds overheads in terms of power, since most of its logic, including instruction fetch, decode, issue window, reorder buffer, etc. are not required for a BFS. Further, the caching behavior of the mark phase is unfavorable for a general-purpose core: no data is ever reused except the mark bits (since we mark through every object once). However, caches cannot hold individual bits, leaving a choice of not caching the mark bits (using non-allocating loads) or caching the entire cache line (wasting space). The latter also pollutes the cache, slowing down mutators.

Analogously, the relocation phase is mainly a copy operation, which again requires no out-of-order proces-

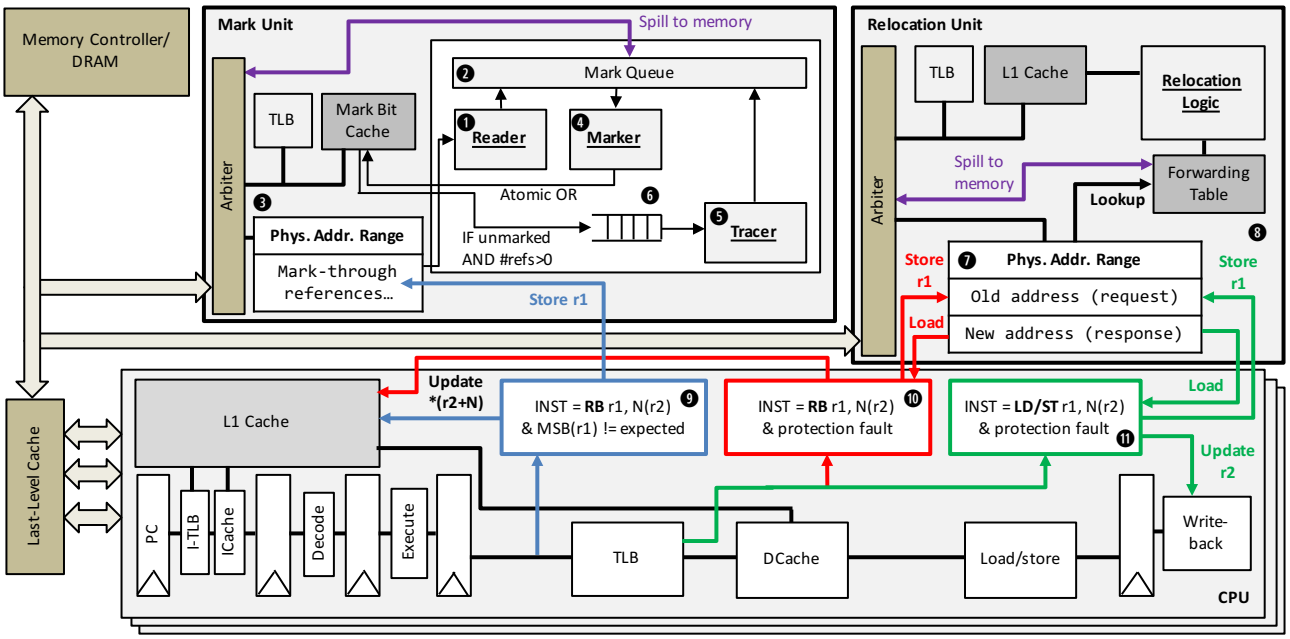


Figure 4: Overview of our GC hardware. We show our CPU changes in a 5-stage pipeline, but they also apply to out-of-order cores.

sor to perform and does not benefit from caching (and, in fact, can be parallelized well in hardware).

In addition, both phases benefit from being executed close to memory, reducing energy consumption from data movement between memory and the cores. We therefore implement these two phases in hardware, combined with minimal changes to the processor pipeline (Figure 4). We add a mark and a relocation unit that sit beyond the LLC, share the virtual address space with the process they are operating on (i.e., have their own TLBs) and are cache-coherent with the cores. Both units carve out a small range of the physical address space (❸ /❼) for communication with the CPUs.

4.1 The Mark Unit

The mark unit consists of three parts. After launching a mark phase, the *Reader* ❶ first loads all roots into the on-chip *Mark Queue* ❷. It communicates with the CPUs through a range of the physical address space ❸. This range is cacheable and each CPU can write to it to send addresses of reachable objects to the mark unit. The reader polls the range until it has received all roots (CPUs terminate root list with a special word). Roots are collected in software, an infrequent operation that can be implemented without stalling [12].

The mark queue is implemented as an on-chip SRAM, and is expected to be small (a few KB). There is a design space in that the smaller the queue, the more often its middle part has to be spilled to memory.

Once the roots have been loaded, the *Marker* ❹ and the *Tracer* ❺ perform the mark phase. The marker is responsible for taking an object pointer from the mark queue, sending out an atomic fetch-and-or request to mark and read the object's header and, if the object was unmarked before and has at least one outgoing reference, put it into a *Trace Queue* ❻ that stores pairs of object pointers and the number of references associated with them. The tracer then takes elements from this queue and issues read requests to load the references within the object into the mark queue as they return.

This design decouples the different types of memory accesses necessary for the mark phase. If the tracer is busy copying one long object, the marker can run ahead and queue up additional objects. Similarly, if the marker is busy marking objects that have been marked before, the tracer can work through the remainder of the trace queue. As such, the marker and tracer work together to maximize memory bandwidth. This design is enabled by our memory layout, since it allows us to fetch all information that the tracer needs to do the copying in the same memory request as the mark.

In addition to the basic design, the tracer has to check for relocated references as they are added to the mark queue and remap them (Section 4.3). Furthermore, based on the observation that only the mark bits need to be cached, the mark unit can implement a bit cache as an optimization (e.g., using the design from [10]).

4.2 The Relocation Unit

The relocation unit's task is to continuously 1) find pages that are mostly garbage and should therefore be evacuated, 2) build a side-table ❽ of forwarding pointers, 3) protect the original page in the page table, and 4) move objects over to the new page. At the same time, the relocation unit receives requests from CPUs when they need to find the new location of an object ❼. If

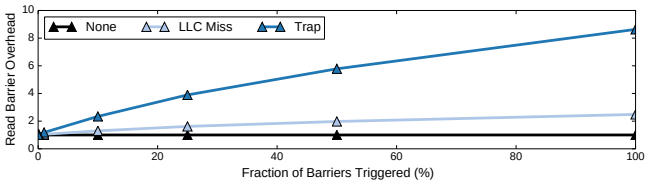


Figure 5: Ballpark estimate of overheads from handling barriers through traps vs. in hardware. We ran quicksort on an Intel i7-920 and added a mock "barrier" to each array read that, when triggered, either forced an LLC miss (to simulate the latency of our hardware) or a user-level trap. Each would read from an in-memory table with little locality and write the result into a variable. We enable user-level traps by running on Dune [13].

the data has already been relocated, the relocation unit will respond immediately with the new address, otherwise it will relocate the object and then respond. CPUs hence always get a response with the new location, but it may be delayed (which is seen by the CPU as a long memory load). One advantage over the software approach is that this makes concurrency much simpler.

Like the mark queue, the forwarding table may have to be spilled to memory, but part of it can be cached by the relocation unit. This gives rise to a large design space (e.g., whether to use a CAM or hash table).

Once the relocation unit has finished relocating a page and remapping has completed, it frees the physical memory and writes the addresses of available blocks into a free-list in memory, which can be accessed by conventional TLAB allocators on the CPUs.

4.3 CPU Modifications

Changes to the CPUs are confined to a similar read barrier mechanism as Pauseless GC. However, we fully implement the barrier in hardware instead of triggering a user-level trap. While only a few cycles, these traps can be frequent, redirect the instruction stream (dropping out-of-order state) and cannot be speculated over. Note that the latter is also true for software barriers. A hardware barrier has the best of both worlds: it avoids both the cost of trapping and the energy and resource overheads from implementing the barrier in software. To show the potential gain over no hardware support, Figure 5 shows the overhead if we used traps on current CPUs (Vega's fast traps target the same problem).

Like Azul, we reuse the virtual memory system to implement the barrier. Our key difference is the way we handle stale references in registers. Read barriers are only added when reading a reference into a register; the challenge remains how to handle the case where an object was relocated afterwards. Pauseless handles this by requiring mutators to reach a global safepoint and scrub all references from their registers before relocation can begin. However, global safepoints cause pauses (although not just for GC). An alternative would be to not use explicit read barriers and trap when accessing a relocated object's data, but then we could not self-heal

```
LD rd, 4(r1) # Load ref from field in r1
RB rd, 4(r1) # Read barrier (might fix up [r1+4])
SD $0, 8(rd) # Use ref, store a value into a field
```

Figure 6: Code to load and use a reference on our architecture

the reference, as we do not know where it originally came from (plus we would lose temporal locality).

We get around these problems by using a hybrid: in addition to the read barrier, we trap on accesses stemming from stale references and self-heal the address stored in the register used for the access.

Altogether, our modifications to the CPU restrict themselves to adding a `RB rd, N(rs)` read barrier instruction that checks whether `rd` contains a valid reference loaded from `rs` and if not, fixes the reference and stores the updated value back to address `rs+n`. Figure 6 shows how it is used. We then add logic to the pipeline to intercept and handle the three cases from above:

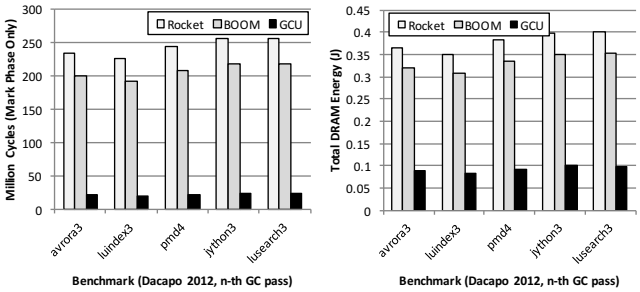
❾ **NMT fault**: When executing a read barrier and the MSB is wrong, write the reference to the buffer that is used to communicate reachable addresses to the mark unit, flip the mark bit, fix the original address and continue. When the buffer's cache line is evicted, the mark unit sees the update as it owns that physical memory. When the mark queue is empty, that memory is polled directly (which ensures that we see all references).

❿ **Relocation fault**: When executing a read barrier and we get a protection fault, the object is being relocated and we need to determine its new location. To do so, we write the old address to a physical address under the control of the relocation unit and then load from another. The load will stall until the relocation unit responds (reusing the cache coherence protocol) and when it completes, we write the location back to the original location associated with the barrier. This is better than trapping, since the out-of-order processor can speculate over this stall, and does not have to discard any out-of-order state. Note that the mark unit needs the check for every pointer it puts into the mark queue.

⓫ **Stale reference fault**: When not at a barrier and we get a protection fault, request the new location from the relocation unit as in the previous case but fix up the value in the original register rather than memory.

4.4 Summary

Our design has several advantages. It offloads the phases that are not a good fit for general-purpose cores to accelerators close to memory. It also allows the CPU to speculate over triggered barriers. Finally, the approach is very non-invasive, as it reuses the existing coherence network and only adds moderate changes to the CPU. There is, of course, an enormous design space associated with each aspect of this proposal, and we are planning to evaluate a wide range of these trade-offs.



(a) Completion Times *(b) DRAM Energy*

Figure 7: Comparison between Mark Unit (with 120K item mark queue), Rocket in-order and BOOM OOO core (16/32KB L1D caches) at 1 GHz. DRAM: 256MB DDR3, 667MHz, 8 banks.

5. DISCUSSION AND PROGRESS

Evaluating our design is challenging, as system performance and power is determined by very fine-grained interactions between components. As such, high-level simulators are unlikely to give credible results. At the same time, cycle-accurate architectural simulators are too slow to run full Java applications for minutes of simulated time and large numbers of threads.

The established standard for evaluation of GC research are the Jikes RVM [14] and the Dacapo benchmarks [15]. Dacapo is designed to stress the memory management layer with a set of real-world workloads, and Jikes provides state-of-the-art GCs as a baseline. We therefore decided to build on this infrastructure.

In contrast to most existing work, we implement our GC into a real SoC. This allows us to evaluate it at high clock rates on an FPGA and achieve much higher fidelity than a simulator, including detailed energy models [16]. We use *Rocket Chip* from UC Berkeley [17], which provides a customizable in-order core (*Rocket*) and out-of-order core (*BOOM*) as well as an SoC generator that can generate flexible topologies including cores, caches, accelerators, etc. Rocket Chip has been taped out eleven times and supports a large software ecosystem (we are in the process of porting Jikes).

We now show preliminary, unscientific numbers from our work-in-progress prototype running in RTL-level simulation with DRAMSim. As our Jikes port is not complete yet, we take heap snapshots from several Dacapo benchmarks running on x86 and rewrite them to our object layout. We first focussed on the mark unit. We implemented it in RTL and integrated it into Rocket Chip. Figures 7 and 8 demonstrate the impact of a dedicated mark unit relative to executing an (untuned) implementation of the mark phase on traditional CPUs. The main benefit is a much better utilization of memory bandwidth, which substantially reduces *overall* DRAM and core energy. Note that we report DRAM energy (as it currently dominates) and that we do not spill the mark queue into memory yet. With spilling, the mark queue size can be decreased to 10s of KBs.

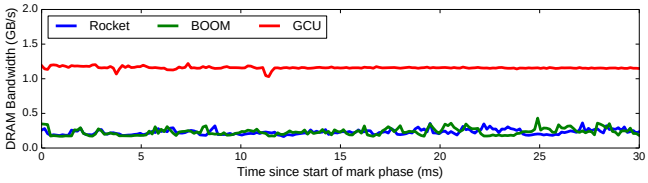


Figure 8: DRAM bandwidth over time (avrora3 benchmark).

There are many open questions and we are hoping to investigate some of them as part of our research:

1. Is there hardware support for root scanning and generational GC? Azul's C4 collector is generational and Sun's Maxwell project has worked on both [9].
2. Can the system be made more general by making it configurable to handle parts in software if necessary?
3. Would it be possible to support other memory layouts? (e.g., with a programmable mark unit)
4. How are multiple processes supported?
5. We focus on object-oriented languages. What would it take to support functional languages?

Conclusion: We think it is time to revisit hardware support for GC to build a pause-free collector that is much more energy efficient than the state of the art. We believe our design is promising step in this direction and are implementing our GC in a real system.

References

[1] E. Moss, "The cleanest garbage collection: Technical perspective," *CACM*, vol. 56, p. 100, Dec. 2013.
[2] T. Cao *et al.*, "The yin and yang of power and performance for asymmetric hardware and managed software," ISCA '12.
[3] C. Click *et al.*, "The pauseless gc algorithm," VEE '05.
[4] M. Meyer, "A true hardware read barrier," ISMM '06.
[5] J. A. Joao *et al.*, "Flexible reference-counting-based hardware acceleration for garbage collection," ISCA '09.
[6] D. F. Bacon *et al.*, "And then there were none: A stall-free real-time garbage collector for reconfigurable hardware,"
[7] G. Wright, M. L. Seidl, and M. Wolczko, "An object-aware memory architecture," *SCP*, vol. 62, no. 2, 2006.
[8] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
[9] M. Wolczko, G. Wright, and M. Seidl, "Methods and apparatus for marking objects for garbage collection in an object-based memory system." US Patent 8,825,718.
[10] T. Harris, S. Tomic, A. Cristal, and O. Unsal, "Dynamic filtering: Multi-purpose architecture support for language runtime systems," ASPLOS '10.
[11] E. M. Gagnon *et al.*, "SableVM: a research framework for the efficient execution of java bytecode," JVM '00.
[12] W. Puffitsch and M. Schoeberl, "Non-blocking root scanning for real-time garbage collection," JTRES '08.
[13] A. Belay *et al.*, "Dune: Safe user-level access to privileged cpu features," in *OSDI '12*.
[14] B. Alpern *et al.*, "The jikes research virtual machine project: Building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–417, 2005.
[15] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*.
[16] D. Kim *et al.*, "Strober: Fast and accurate sample-based energy simulation for arbitrary rtl," in *ISCA '16*.
[17] K. Asanovic *et al.*, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, UC Berkeley, Apr 2016.