2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

A Hardware Accelerator for Tracing Garbage Collection

Martin Maas*, Krste Asanović, John Kubiatowicz
University of California, Berkeley
{*maas,krste,kubitron*}@*eecs.berkeley.edu*

***Abstract*—A large number of workloads are written in garbage-collected languages. These applications spend up to 10-35% of their CPU cycles on GC, and these numbers increase further for pause-free concurrent collectors. As this amounts to a significant fraction of resources in scenarios ranging from data centers to mobile devices, reducing the cost of GC would improve the efficiency of a wide range of workloads.**

We propose to decrease these overheads by moving GC into a small hardware accelerator that is located close to the memory controller and performs GC more efficiently than a CPU. We first show a general design of such a GC accelerator and describe how it can be integrated into both stop-the-world and pause-free garbage collectors. We then demonstrate an end-to-end RTL prototype of this design, integrated into a RocketChip RISC-V System-on-Chip (SoC) executing full Java benchmarks within JikesRVM running under Linux on FPGAs.

Our prototype performs the mark phase of a tracing GC at 4.2x the performance of an in-order CPU, at just 18.5% the area (an amount equivalent to 64KB of SRAM). By prototyping our design in a real system, we show that our accelerator can be adopted without invasive changes to the SoC, and estimate its performance, area and energy.

***Keywords*-hardware accelerators; garbage collection; SoCs; language runtime systems; memory management;**

I. Introduction

A large fraction of workloads are written in managed languages such as Java, Python or C#. This includes server workloads in data centers, web applications in browsers, and client workloads on desktops or mobile devices. Garbage collection (GC) has been a challenge in these environments for a long time [1], [2]. On one hand, GC pauses cause applications to stop unpredictably, which results in long tail-latencies, stragglers and GUI glitches. On the other hand, GC is a substantial source of energy consumption: previous work has shown that applications spend up to 38% of their time in garbage collection, and that GC can account for up to 25% of total energy consumed (10% on average [3]).

It is therefore unsurprising that there has been a large amount of research on GC for the past 50 years [4]. Yet, despite a lot of progress, we have arguably still not come close to an *ideal* garbage collector. Such a collector would achieve full application throughput, support high allocation rates, introduce no GC pauses and would be efficient in terms of resource utilization (particularly energy consumption).

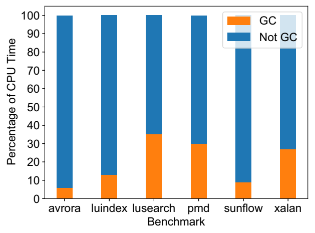
Instead, all existing collectors have to trade off three fundamental goals: (1) high application throughput, (2) high memory utilization, and (3) short GC pauses. Existing collectors can perform well on any two of these goals, at the cost of the third: *Stop-the-world* collectors introduce pauses but achieve high application throughput and memory utilization by stopping all threads and completing GC as quickly as possible on all cores of the machine. *Concurrent* collectors perform GC without pausing the application but lower application throughput since they must synchronize the application with the collector and utilize CPU cores for GC. Finally, *avoiding GC* altogether neither incurs pauses nor application slow-down but wastes memory.

Most collectors fall into none of these extreme points and instead strive to balance the three metrics. However, while they can shift work between different parts of the execution (e.g., perform GC interwoven with the application rather than during pauses), the work itself remains the same.

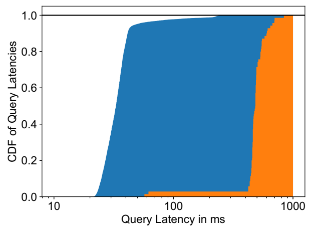
In this paper, we argue that we can build a collector that *fundamentally* improves these metrics by moving GC into hardware. While the idea of a hardware-assisted GC is not new [5]–[8], none of these approaches have been widely adopted. We believe there are three reasons:

1) **Moore's Law:** Most work on hardware-assisted GC was done in the 1990s and 2000s when Moore's Law meant that next-generation general-purpose processors would typically outperform specialized chips for languages such as Java [9], even on the workloads they were designed for. This gave a substantial edge to non-specialized processors. However, with the end of Moore's Law, there is now a renewed interest in accelerators for common workloads.
2) **Server Setting:** The workloads that would have benefitted the most from hardware-assisted garbage collection were server workloads with large heaps. These workloads typically run in data centers, which are cost-sensitive and, for a long time, were built from commodity components. This approach is changing, with an increasing amount of custom hardware in data centers, including custom silicon (such as Google's Tensor Processing Unit [10]).
3) **Invasiveness:** Most hardware-assisted GC designs were invasive and required re-architecting the memory system [8], [11]. However, modern accelerators (such as those in mobile SoCs) are typically integrated as memory-mapped devices, without invasive changes.

* Now at Google Brain (Contact: mmaas@google.com)



(a) CPU Time spent in GC



(b) Query latencies in Lucene

Figure 1: **Applications spend up to 35% of time in GC pauses, and GC causes long tail latencies.** Fig. (b) shows the query latencies in the *lusearch* benchmark processing 10 QPS. The long tail (note the log scale) is the result of GC; the colors indicate whether a query was close to a pause.

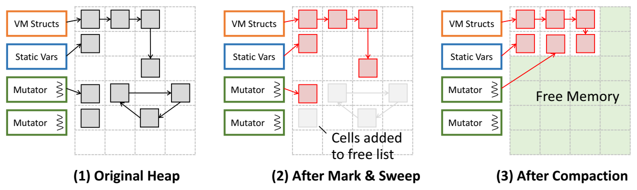


Figure 2: **Basic GC Operation**. The collector performs a traversal of the object graph, starting from roots such as static variables (arrows are references). Objects not reached are recycled, either by adding them to a free list or by compacting reachable objects into a smaller region.

We believe that for these reasons, the time is ripe to revisit the idea of a hardware-assisted garbage collector, to build a GC that can be pause-free, achieves high application throughput, high memory utilization, and is resource efficient.

This work takes a step in this direction. We present a hardware accelerator close to memory that is integrated into an SoC and performs garbage collection more efficiently than a CPU would, lowering GC's energy and performance impact. The accelerator is standalone but can be combined with light-weight changes to the CPU that reduce the cost of using the unit in a concurrent GC (by removing communication between CPU and GC from the execution's critical path and allowing the CPU to speculate over these interactions).

We first present our design in the most general form and describe possible incarnations. We then present an end-to-end prototype implemented in RTL that is integrated into a full SoC design and co-designed with modifications to the *JikesRVM* Java VM running on Linux. Executing this full hardware and software stack within FPGA-based simulation, we evaluate the design trade-offs of our accelerator, characterize its workload, and demonstrate that it can perform Mark & Sweep garbage collection 3.3× as fast as an in-order CPU, at 18.5% the on-chip area consumption. We also show that with a faster memory system, those speed-ups increase to 9.0× over the CPU for the tracing portion of the GC.

While the design is general, our prototype is integrated into a RISC-V RocketChip SoC. RocketChip is an open-source SoC generator that can target both FPGA and VLSI flows and has been used in commercial ASIC designs. Through this integration, we show that our accelerator is non-invasive and works efficiently within an existing SoC, running complete Java workloads, including DaCapo benchmarks.

We first present an example to demonstrate the problems arising from GC (Section II). We then give an overview on GC and describe the trade-offs of existing algorithms (Section III). Next, we present a general form of our collector design, describe its design space and introduce optional CPU modifications aimed at reducing the overheads of performing GC concurrently with the application (Section IV). Finally, we present a prototype of our proposed accelerator (Section V), show how it is integrated into the rest of the system and present a detailed evaluation of this prototype running in FPGA-based simulation (Section VI). We conclude by discussing implications of this research (Section VII) and related work (Section VIII).

II. Motivation

The performance overheads and tail-latency problems arising from GC have been widely measured and cited [3], [12]–[15]. To demonstrate these problems, we took several Java applications from the DaCapo benchmark suite [16] and ran them within the Jikes Research Virtual Machine [17] on a RISC-V-based system with an in-order *Rocket* processor [18] and JikesRVM's Mark & Sweep stop-the-world GC (Section VI shows more details on this setup).

Note that we present a specific design point and use systems that are not as optimized as server-grade JVMs and processors. However, as far as GC is concerned, our measurements are within the expected range [3], [19]. We therefore treat this setup as representative, and will point out differences to state-of-the-art systems where appropriate.

Figure 1a shows the fraction of CPU time spent in GC pauses for different workloads. We confirm that workloads can spend up to 35% of their time performing garbage collection. Furthermore, Figure 1b shows the impact of garbage collection pauses on a latency-sensitive workload. We took the *lusearch* DaCapo benchmark (which simulates interactive requests to the Lucene search engine [20]) and recorded request latencies of a 10K query run (discarding the first 1K queries for warm-up), assuming that a request is issued every 100ms and accounting for coordinated omission [21]. This experiment shows that in the absence of GC, most requests complete in a short amount of time, but that GC pauses introduce stragglers that can be two orders of magnitude longer than the average request.

Long tail latencies result in a decrease in user experience and require replicating services to maintain the illusion of availability. Meanwhile, with data center energy consumption at 70 billion kWh in the U.S. for 2014 [22], even 10% of energy spent on GC translates into $100Ms per year.

III. Background

We first give a broad overview of the current state of garbage collection, and the design trade-offs involved.

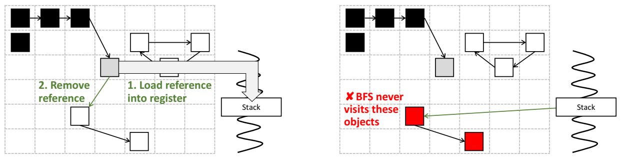

2. Remove reference
1. Load reference into register
Stack
✘ BFS never visits these objects
Stack

Figure 3: **Mutators can hide objects from a concurrently running GC pass.** This happens when removing a reference that has not yet been visited by the GC and loading it into a register.

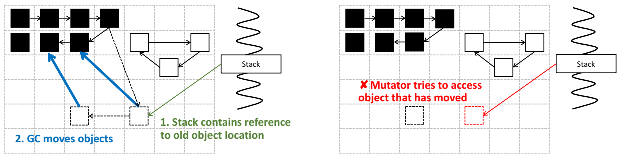

Stack
2. GC moves objects
1. Stack contains reference to old object location
✘ Mutator tries to access object that has moved
Stack

Figure 4: **Concurrent GC can move objects while mutators are accessing them.** If a mutator encounters a stale reference and uses it to access an object, it reads the old location, which may not be valid anymore.

A. Garbage Collection Overview

8 out of the 10 most popular languages use GC [23]. The garbage collector's task is to periodically free unused memory and make it available for allocation. In an object-oriented language, this means that the GC detects objects that are not reachable and frees their memory. There are two fundamental strategies for GC: tracing and reference-counting.

A tracing collector (Figure 2) starts from a set of *roots*, which are references (i.e., pointers) stored in static/global variables, the application threads' stacks and the runtime system. From these roots, the collector performs a graph traversal – typically a breadth-first search – and sets a *mark bit* in all objects it discovers. In a second step, all unmarked objects are freed, either by performing a *sweep* through the heap and adding the memory of all unmarked objects to a *free list*, or by *compacting* all marked objects into a new region (which can be done as part of the traversal) and freeing the old space. The latter requires a *relocating* collector (i.e., a GC that can move objects in memory, which means rewriting all references to objects that have been moved).

In contrast to tracing collectors, reference-counting systems add a counter to each object that is increased whenever a reference to it is added, and decreased whenever such a reference is deleted. Once the counter reaches zero, the object's memory can be reclaimed. As such a system cannot collect objects that form a cyclic reference graph, a backup tracing collector is required. Reference-counting can therefore be seen as an optimization to a tracing collector, but the fundamental task of tracing remains.

There are a range of variants of these basic strategies. *Generational GC* divides memory into a young and old generation; objects are allocated in the young generation, which is collected more frequently than the old generation. GCs can either be *stop-the-world* (stalling all application threads while the full collection is performed), *incremental* (limiting pause times by only executing a partial GC) or *concurrent* (running without stalling the application). In this work, we focus on non-generational tracing GC, both stop-the-world and concurrent. Most of our techniques would apply to generational GC as well, but would require extensions to track cross-generation references.

B. Concurrent Garbage Collection

Stop-the-world collectors are popular, as they are easy to implement and reason about. However, problems have been reported applying these collectors to modern workloads [12]–[14], [24]. Request latencies in data centers and applications alike are decreasing, and services now often have deadlines of under 1 ms [25], increasing the relative impact of stop-the-world pauses (which are often 10s or 100s of ms). Meanwhile, heap sizes are growing into the terabyte range; as GC work grows with the size of the heap, this can lead to pauses of over a minute [26], which is usually problematic.

Concurrent collectors (which run in parallel to the application without pausing it for more than a short amount of time) are therefore seeing renewed interest [27]–[29]. These collectors avoid pauses by running GC on some cores while the *mutators* (application threads) continue to run. This approach avoids pauses, but adds two sources of overhead:

- The collector still has to perform the same operation as a stop-the-world GC, but it is now running it on different cores, incurring overheads from interference in the memory system, such as moving cache lines.
- The collector and mutators now need to communicate to ensure that each maintains a consistent view of the heap. This is typically done through *barriers*, which are small pieces of code that are added to every reference/pointer operation by the compiler (not to be confused with barriers in memory consistency or GPUs).

These barriers can cause significant application slow-downs. As a point of reference, Oracle's newly announced concurrent ZGC collector [28] targets up to 15% slow-down compared to OpenJDK's G1 [30] collector, which itself is an incremental collector with (more lightweight) barriers.

To understand concurrent GC, it is crucial to understand these barriers. Their purpose is to address the two race conditions that can happen when running GC at the same time as the mutators. First, mutators can overwrite references as the collector is performing its graph traversal (Figure 3). This means that the mutator can load and overwrite a reference before the traversal has visited it, and therefore *hide* it from the collector. The collector will not mark the object and will thus free it even though it is still reachable. This problem can be avoided with a *write* barrier: every time a reference is overwritten, its previous value is added to the traversal's frontier (note that many variants of such barriers exist).

Second, a relocating collector may move objects in the heap that a mutator is accessing (Figure 4). This can lead to a mutator trying to access an object using an old reference. This can be avoided with a *read* barrier: When reading a

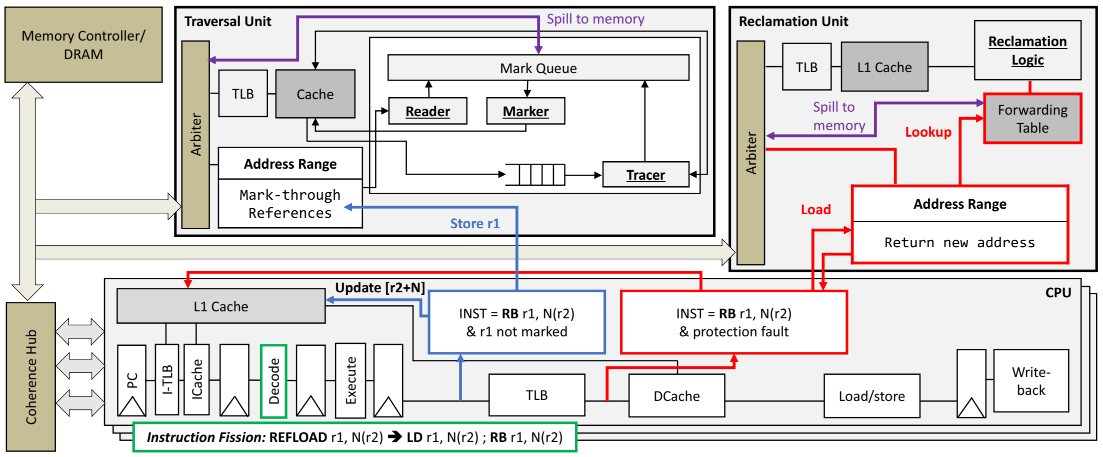


Figure 5: **Overview of our GC Design**. We introduce two units, a *Traversal Unit* that performs the GC's graph traversal (i.e., marking), and a *Reclamation Unit* that identifies and frees dead memory (i.e., sweeping). These units are connected to the on-chip interconnect, similar to other DMA-capable devices. Blue shows optional CPU modifications for concurrent GC. Red are changes for concurrent, relocating GC. Green shows an optional change to reduce code size by fusing the read barrier and the load instruction into a single REFLOAD instruction. All of these CPU changes are for performance and not fundamentally required. For simplicity, we show an in-order pipeline, but the changes could be implemented in an out-of-order core as well.

reference into a register, the barrier code checks that the object's location has not changed, and if it has, looks up the new location (once again, there are many variants of this).

It is these barriers that most existing hardware support for GC is targeting [6], [31], [32]. We now review this work.

Barrier Implementations: Barriers span a wide design space that trades off fast-path latency (e.g., if a read barrier's object has not moved), slow-path latency (e.g., if the barrier gets triggered because the object has moved), the instruction footprint and how it maps to the underlying microarchitecture (i.e., how well it can be interleaved with existing code). As barriers are very common operations – they typically have to be added to every reference read and/or write operation – these design decisions have a substantial impact on application and collector performance. Fundamentally, there are three basic approaches:

1) Compile the code for checking the barrier condition into the instruction stream and branch to a slow-path handler whenever the barrier triggers. One special case of this is a forwarding table (which has no fast-path).
2) Reuse the virtual memory system to fold the barrier check into existing memory accesses and incur a trap if it triggers. For example, the OS can unmap pages whose objects have been compacted into a new space, which means that any accesses using old references will raise a page fault. The trap handler can then look up the new location and fix the old reference.
3) Introduce a barrier instruction in hardware. The semantics of these instructions vary, but they typically have similarities to the second approach and raise a fast user-level trap if the barrier triggers.

This leads to a trade-off between invasiveness, programmability and performance. Most existing designs choose option (1), minimizing invasiveness by operating solely in software (e.g., the G1 collector [30] and Go's concurrent GC [29]). Options (2) and (3) have been used in systems such as IBM's z14 [33] or Azul's Vega [6]. For example, the Pauseless algorithm [6] relied on a barrier instruction that raises fast user-level traps in the slow-path case, while IBM's *Guarded Storage Facility* [34] protects regions and raises a trap when loading a pointer to them. These designs are invasive as they change the CPU, but maintain programmability. While they speed up the fast path, the slow path may still result in an expensive pipeline flush or instruction stream redirect.

IV. GC Accelerator Design

We take a complementary approach. Even if barriers account for 15% overhead, the main cost in terms of CPU time is still the actual GC work. Our insight is that GC is a bad fit for CPUs, and regular enough to be executed in hardware (a similar argument as for page-table walkers). By offloading GC to a small accelerator that takes up a fraction of the area and energy of a CPU core, we can reduce the impact of the GC work (at the cost of programmability).

While the idea of a GC accelerator is not entirely new [5], [7], [31], we are not aware of any work that has explored this idea in the context of full-stack high-performance SoCs. Our proposed accelerator could be used in either a stop-the-world setting (freeing up CPU cores to run other applications), or could be combined with barriers to be used in a pause-free collector. We also propose optional CPU modifications to reduce the cost of these barriers, but they are not required.

Motivation: Our design is motivated by one key insight: CPUs are a bad fit for GC operations. 75% of time in a Mark & Sweep collector is spent in the mark phase (Figure 15).

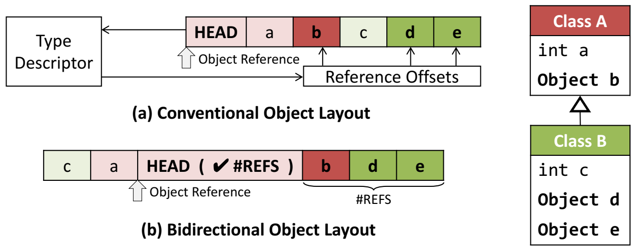


Figure 6: **Bidirectional Object Layout**. To support inheritance, objects are laid out with the fields of all parents first, followed by the class's own fields. This results in reference fields interspersed throughout the object. In a bidirectional layout, the header is placed in the middle, all reference fields are stored on one side of the header, and all non-reference fields on the other. This layout still supports inheritance, but identifies reference fields without any extra accesses.

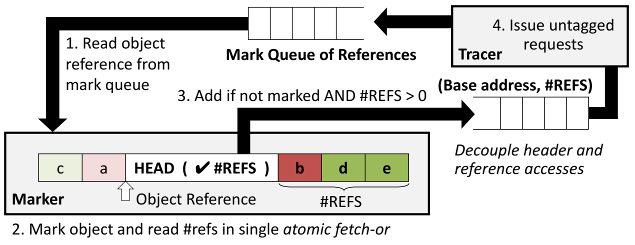

1. Read object reference from mark queue
Mark Queue of References
4. Issue untagged requests
Tracer
3. Add if not marked AND #REFS > 0
(Base address, #REFS)
Decouple header and reference accesses
c
a
HEAD (✔ #REFS)
b
d
e
Marker
Object Reference
#REFS
2. Mark object and read #refs in single atomic fetch-or

Figure 7: **Traversal Unit Operation**. The marker removes objects from the on-chip mark queue, uses a single atomic memory operation (AMO) to mark the header word and receive the number of outbound references, and (if the object has not been marked yet), enqueues it to the tracer queue. The tracer takes elements from this queue and copies the object's references into the mark queue using untagged memory requests.

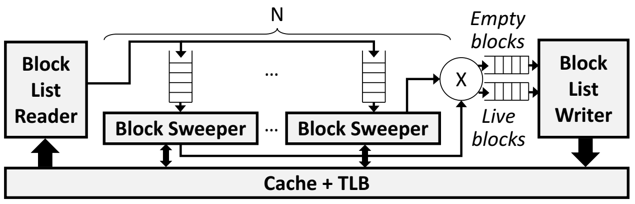


Figure 8: **Reclamation Unit**. Blocks are read from a global block list, distributed to blocks sweepers that reclaim them in parallel, and then written back to the respective free lists of empty and (partially) live blocks.

At each step of this operation, the collector takes an element out of the mark queue/frontier, marks it, and (if it has not been marked yet) copies all outbound references back into the mark queue. To make this operation efficient, the system needs to keep as many requests in flight as possible.

On an out-of-order CPU, this number is limited by the size of the load-store queue and the instruction window [35]. For this reason, modern GCs are often parallel, using multiple cores in the machine to speed up the graph traversal. However, this is not making efficient use of most of the hardware in a CPU core (even when using wimpy cores [3]). The operation does not use most functional units, and cannot make effective use of caches (as each object is only copied once, and the only temporal locality that can be exploited is for mark bits – i.e., one bit per cache-line). Similarly, sweeping and compaction are embarrassingly parallel operations (as each page can be swept independently), and only touch every word once, also making them a poor fit for the CPU.

We claim that the same operations could be executed much more efficiently in a small, energy-efficient accelerator that is located next to the memory controller (Figure 5). Our design has two parts, a *Traversal Unit* that performs the mark phase, and a *Reclamation Unit* that sweeps the memory and places the resulting free lists into main memory for the application on the CPU to use during allocation.

A. The Traversal Unit

The traversal unit implements a pipelined version of the graph traversal described in the previous section. There are three ideas that, taken together, enable our traversal unit design to outperform a CPU by 4.2× at 18.5% the area.

I. Bidirectional Object Layout: GC does not benefit much from caches, and the traversal unit can therefore save most of this area and power. However, if one were to use a cacheless accelerator design with an unmodified language runtime system, the performance would be poor [19]. The reason is that when copying the outbound references of an object back into the mark queue, the collector has to identify which fields contain these references. While this can be achieved using specialized functions or tag bits on architectures that support them [36], most runtime systems use a layout where the object's header points to a type information block (TIB), which lists the offsets of the reference fields (Figure 6a).

This approach works well on systems with caches, since most TIBs are in the cache. However, it adds two additional memory accesses per object in a cacheless system. To address this, we use a *bidirectional layout* (Figure 6b). This layout was used by SableVM [37] as a means to improve locality, but its benefits on CPUs have been limited [38]. However, we found that such a layout helps greatly on a system without caches, as it eliminates the extra accesses. Its access pattern (a unit-stride copy) is beneficial as well.

While this approach requires adapting the runtime system to target our accelerator, the changes are invisible to the application and contained at the language-runtime level.

II. Decoupled Marking and Copying: With the bidirectional layout in place, we can store the mark bit and the number of references in a single header word, which allows us to mark an object and receive the outbound number of references in a single *fetch-or* operation. On a CPU, a limited number of these requests can be in flight, due to the limited size of the load-store queue. Since the outcome of the mark operation determines whether or not references need to be copied, this limits how far a CPU can speculate ahead in the control flow (and causes expensive branch mispredicts).

On the traversal unit, the number of outstanding requests can be larger but is still limited by the number of available

MSHRs. If the unit encounters a large number of objects without outbound references or that have already been marked, our effective bandwidth is limited by the mark operation. Similarly, if there are long objects, we are limited by copying (or "tracing") the references of these objects.

We therefore decouple the marking and tracing from each other. Our traversal unit consists of a pipeline with a *marker* and a *tracer* connected via a tracer queue (Figure 7). If a long object is being examined by the tracer, the marker continues operating and the queue fills up. Likewise, if there are few objects to trace, the queue is drained. This design allows us to make better use of the available memory bandwidth than a control-flow-limited CPU could (Section VI).

III. Untagged Reference Tracing: While the marker needs to track memory requests (to match returning mark bits to their objects), the order in which references are added to the mark queue does not affect correctness. The tracer therefore does not need to store request state, but can instead send as many requests into the memory system as possible, and add responses to the mark queue in the order they return. This increases bandwidth utilization (Figure 16).

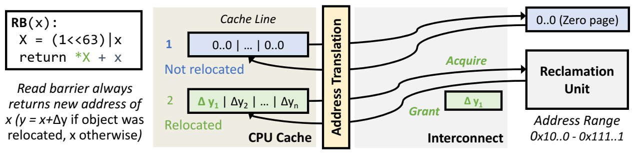
B. *The Reclamation Unit*

Taken together, the previous three strategies enable the traversal unit to achieve a higher memory bandwidth than the CPU, at a fraction of on-chip area and power. The reclamation unit achieves the same for the sweeping operation. While this operation is highly dependent on the underlying GC algorithm, it typically involves iterating through a list of blocks and either (1) evacuating all live objects in a block into a new location (for relocating GC) or (2) arranging all dead objects into a free list (for non-relocating GC).

Each of these operations can be performed with a small state machine, and can be parallelized across blocks. We therefore designed a unit that iterates through all blocks and parallelizes them across a set of *block sweeper* units that each reclaim memory in a block independently and then return the block to either a list of free or live blocks (Figure 8). As each unit is negligibly small, a large part of the design is the cross-bar that connects them.

C. *Runtime System Interactions*

Using the unit in a stop-the-world setting requires minimum integration beyond the new object layout. The runtime system first needs to identify the set of roots (which can be done in software without stalling the application [39]) and write them into a memory region visible to the accelerator. It also has to inform the unit where in memory the allocation regions are located, as well as configuration parameters (e.g., available size classes). Beyond this, the unit acts autonomously and the runtime system polls a control register to wait for it to be ready. Note that no modifications to the CPU or memory system are required. Instead, the unit acts as a memory-mapped device, similar to a NIC.



```
RB(x):
 X = (1<<63)|x
 return *X + x
```

Figure 9: **Read Barrier.** The barrier checks object references and ensures they point to their new location, whether or not they have been moved. By relying on the existing coherence protocol, the functionality can be implemented in hardware without changing the CPU or memory system.

D. *Concurrent Garbage Collection*

Our design can be integrated into a concurrent GC without modifying the CPU. While not implemented in our prototype, we propose a novel barrier design that eliminates instruction stream redirects for *both* the fast and slow path. Our insight is that by "hijacking" the coherence protocol, barriers can be implemented without traps or branch mispredicts.

Write Barrier: When overwriting a reference, write it into the same region in memory that is used to communicate the roots. The traversal unit writes all references that are written into this region to the mark queue.

Read Barrier: For a relocating collector (where the reclamation unit moves objects in memory), the read barrier needs to check whether an object has moved and get the object's new location if it has. Many relocating GCs operate on large pages or regions, and invalidate all objects within the same page at a time (e.g., Pauseless GC [6]). They then compact all objects from these pages into new locations, keeping a forwarding table to map old to new addresses.

We propose adding a new range to the physical address space that nominally belongs to the Reclamation Unit but is not backed by actual DRAM (Figure 9). We then steal one bit of each virtual address (say, the MSB), mapping the heap to the bottom half of the virtual address space. Whenever we read a reference into a register, we add instructions that take the address, flip this special bit, read from that location in virtual memory and add the result to the original address.

By default, we map the top half of this virtual address space to a page that is all zeros. All these loads hence return 0 (i.e., the object has not moved). However, when we are relocating objects on a page, we map the corresponding VM page to the reclamation unit's physical address range instead. The unit then sends out *probe* messages across the interconnect to take exclusive ownership of all cache lines in this page. This means that whenever a thread tries to access an object within this page, the CPU needs to acquire this cache line from the reclamation unit. When responding to the request, the unit computes the deltas between the new and the original addresses for the objects in the cache line. It then releases the cache line, the CPU reads from it and adds the value to the original address, updating it to the new value. This communication only has to happen once, as the cache line is in the cache when the object is accessed again.

E. Optional CPU Extensions

While these barriers avoid traps and are semantically more similar to loads that might miss in the LLC, they double the TLB footprint and introduce additional cache pressure (as well as instruction overheads). We therefore sketch out optional CPU changes that may allow hiding most of these overheads as well. We have not implemented these changes, but are considering them for future work.

The challenge is the read barrier (the write barrier check can be folded into the read barrier by checking whether the loaded reference was visited yet and writing it into the mark queue otherwise [6]). The read barrier is problematic since it needs to complete before the reference it is guarding can be used. Pauseless GC has shown that this barrier check can be made very efficient by folding it into the virtual memory system [6]: pages that are evacuated are marked as invalid in the page table and the barrier raises a trap when these pages are accessed. However, these traps can be very frequent if churn is large (resulting in *trap storms* when many pages are freshly invalidated), and are expensive as they require flushing the pipeline and cannot be speculated over.

We propose a change to the CPU by introducing a new `REFLOAD` instruction, which behaves like a load, but always returns the new address of an object. Internally, this instruction will be split into a load and a read barrier (`RB`) instruction. The read barrier can be implemented through the previously described virtual-memory trick, but the page fault from the TLB is intercepted and transformed into a load from the reclamation unit's address range. The load is then added to the load-store queue and can be speculated over like any other load (the GC unit will not release the cache line until it has looked up the new location). This means that the only effect of the GC are loads that may take longer, but traps and pipeline flushes are eliminated.

V. Implementation

We now describe our specific implementation of this general design. We implemented an RTL prototype of the GC unit within a RocketChip SoC [18]. The unit is integrated with the JikesRVM Java VM [17], which is popular in academic managed-language research. We evaluate our prototype in a stop-the-world setting, but as previously described, it could be used in a concurrent collector as well. Figure 10 shows an overview of how our design is integrated into the system.

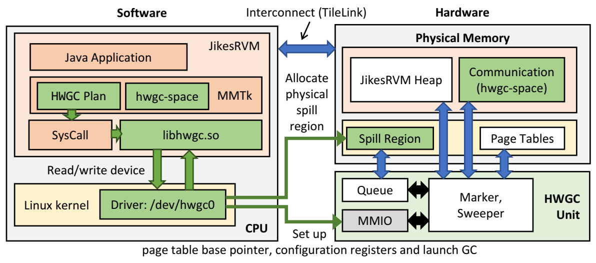


Figure 10: **System Integration**. Green boxes refer to components we added to the system, yellow is the OS and orange represents JikesRVM.

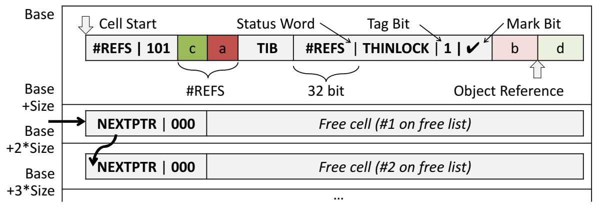


Figure 11: **JikesRVM Integration**. Memory is divided into blocks, which are split into equisized cells (each field represents a 64b word). Cells either include objects or free list entries. References in Jikes point to the second object field to facilitate array offset calculation.

A. JikesRVM Modifications

We use a port of JikesRVM to RISC-V [40], which enabled us to co-design language runtime system and hardware:

MMTk: We implemented a new *plan* in Jikes's MMTk GC framework [41]. A plan describes the spaces that make up the heap, as well as allocation and GC strategies for each of them. We base our work on the `MarkSweep` collector, which consists of 9 spaces, including large object space, code space and immortal space. Our collector traces all of these spaces, but only reclaims the main MarkSweep space (which contains most freshly allocated objects). The other spaces, such as the code space, are still managed by Jikes, but there is no fundamental reason they could not use the GC unit.

Root Scanning: We modify the root scanning mechanism in Jikes to not write the references into the software GC's mark queue but instead write them into a region in memory that is visible to the GC unit (`heap-space`).

Object Layout: We modified the object layout to implement a bidirectional scheme (Section V-C). We found 34 unused bits in the header's status word (Figure 11) – we use 32 of these bits to store the number of references in an object (for arrays, we set the MSB of these 32 bits to 1 to distinguish them). The remaining two bits are used for the mark bit and for a tag bit that we set to 1 for all live cells (this is useful for the reclamation unit). Furthermore, we also replicate the reference count at the beginning of the array, which is necessary to enable linear scans through the heap.

Jikes's Mark & Sweep plan uses a segregated free list allocator. Memory is divided into blocks, and each block is assigned a size class, which determines the size of the cells that the block is divided into. Each cell either contains an object or a free list entry, which link all empty cells together.

B. Integration into RocketChip

Our accelerator is implemented in RocketChip, a RISC-V SoC generator managed by the *Free Chips Project* [42]. RocketChip can target both FPGA and ASIC flows and has been taped out more than a dozen times, including in commercial products [43]. By implementing our design as

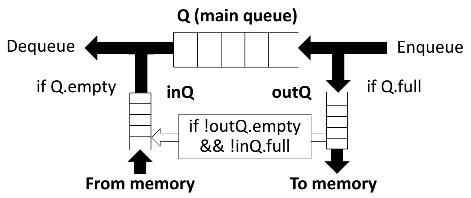


Figure 12: **Mark Queue Spilling**. When the main mark queue (*Q*) fills up, requests are redirected to *outQ*, which is written to memory. When *Q* empties, these requests are read back through *inQ*.

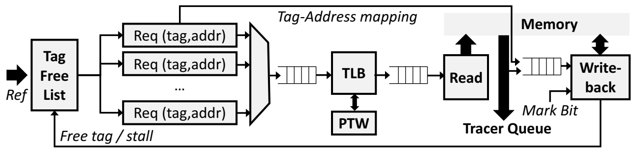


Figure 13: **Marker**. Instead of using a cache with MSHRs, we manage our own requests, as they are identical and unordered.

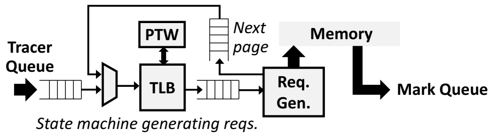


Figure 14: **Tracer**. The *request generator* walks the reference section and issues the largest requests it can based on the current alignment.

an end-to-end prototype in the context of a real SoC, we show that it is non-invasive and can be easily integrated into an existing design. This approach also enables us to perform cycle-accurate evaluation using an FPGA-based simulation platform, and gather area and power estimates (Section VI).

RocketChip is written in Chisel [44], a hardware description language embedded in Scala. Chisel is not an HLS tool but operates directly at the RTL level. By being embedded in Scala, it makes it easier to write parametrizable generators. RocketChip also provides a library of highly parameterizable components such as cores, devices, caches and other functional units. Devices and cores are connected together using a shared-memory interconnect called TileLink, which automatically negotiates communication contracts between endpoints using the *Diplomacy* framework [45].

To integrate a new device into this system, we created a new Chisel module which we register as a TileLink client and connect it to the system bus (which is the RocketChip equivalent of the Northbridge). We also connect a set of memory mapped (MMIO) registers to the periphery bus (Southbridge), for configuration and communication with the CPU. Diplomacy automatically infers the necessary protocol logic, wires the module into the SoC and produces a device-tree structure that can be used to configure Linux drivers.

C. Traversal Unit

The traversal unit closely follows the design in Figure 5. We explored several versions of the marker and tracer: one connects to a shared 16KB data cache, one partitions this cache among the units and one directly talks to the TileLink interconnect. As the GC unit operates on virtual addresses, we added a page-table walker and TLB (the PTW is backed by an 8KB cache, to hold the top levels of the page table).

At the beginning of a GC, a *reader* copies all references from the *hwgc-space* into the mark queue. Then, the marker and tracer begin dequeuing references from their respective input queues and put their results into their output (the queues exert back-pressure to avoid overflowing, and marker and tracer can only issue requests if there is space). This repeats until all queues and the hwgc-space are empty.

Mark Queue Spilling: As the mark queue can theoretically grow arbitrarily, we need to spill it to memory when it fills up. Figure 12 shows our approach. We add two additional queues, *inQ* and *outQ*. A small state machine writes entries from *outQ* into a physical memory range not shared with JikesRVM, and reads entries from this memory into *inQ* if there is space (and *outQ* is empty). We always give priority to the main queue, but if it is full, we enqueue to *outQ* (when it is empty, we dequeue from *inQ*). When *outQ* reaches a certain fill level, we assert a signal that tells the tracer to stop issuing memory requests, to avoid *outQ* from filling up. If there are elements in *outQ* and free slots in *inQ*, we copy them directly, reducing the number of memory requests. By prioritizing memory requests from *outQ*, we avoid deadlock.

Marker: We started out with a design that sends AMOs to a non-blocking L1 cache. However, this limits the number of requests in flights (a typical L1 cache design has 32 MSHRs). MSHRs operate on 64B cache lines, and need to store an entire request. In contrast, all requests in the marker are the same, operate on less than a cache line, and do not need to be ordered. We therefore built a custom marker that talks to the interconnect directly (Figure 13). Instead of full memory requests, we only hold a tag and a 64-bit address for each request, translate them using a dedicated TLB, send the resulting reads into the memory system and then handle responses in the order they return. For each response, we then issue the corresponding write-back request to store the updated mark bit and free the request slot (we can elide write-backs if the object was already marked).

Tracer: We built a custom tracer that can keep an arbitrary number of requests in flight. After translating the virtual address of the object, it enters a *request generator*, which sends `Get` coherence messages into the memory system. Our interconnect supports transfer sizes from 8 to 64B, but they have to be aligned. If we need to copy 15 references (15×8 bytes) at `0x1a18`, we therefore issue requests of transfer sizes 8, 32, 64, 16 (in this order). Note that we need to detect when we hit a page boundary; in this case, the request is interrupted and re-enqueued to pass through the TLB again.

Address Compression: We operate on 64-bit pointers, but runtime systems do not typically use the whole address space. For example, our JikesRVM heap uses the upper 36 bit of each address to denote the space, and the lowest 3 bit are 0 because pointers are 64-bit aligned. Many runtime systems also use bits in the address to encode meta-data, which may be safely ignored by the GC unit. Our design provides a general mechanism to exploit this property: before enqueuing a reference to the mark queue, it can be mapped to a smaller number of bits using a custom function (the reverse function is applied when the object is dequeued). We demonstrate this strategy by compressing addresses into 32 bits, which doubles the effective size of the mark queue and halves the amount of traffic for spilling. Runtime systems with larger heaps may use a larger number of bits instead (e.g., 48).

Mark Bit Cache: Most objects are only accessed once, and therefore do not benefit from caching (Figure 21a). However, we found that there are a small number of objects that are an exception to the rule: about 10% of mark operations access the same 56 objects in our benchmarks. We therefore conclude that a small mark bit cache that stores a set of recently accessed objects can be efficient at reducing traffic. Dynamic filtering is another technique that has been shown to be effective in similar scenarios [32].

D. Reclamation Unit

In our prototype, we implement the simplest version of the reclamation unit, which executes a non-relocating sweep. As such, it does not require the forwarding table or read-barrier backend from Section IV-D. Each block sweeper receives as input the base address of a block, as well as its cell and header sizes. It then steps through the cells linearly.

The unit first needs to identify whether a cell contains an object or free list entry (recall Figure 11). It reads the word at the beginning of the cell – if the LSB is 1, it is an object with a bidirectional layout. Otherwise, it is a next-pointer in a free cell or a TIB (for an object without references, or an array). Based on each of these cases, we can calculate the location of the word containing the mark bit, which then allows us to tell whether the cell is free (i.e., the tag bit is zero), it is live but not reachable (the tag bit is one, the mark bit is not set) or contains a reachable object (both bits are 1). In the first two cases, we write back the first word to add it to the free list, otherwise we skip to the next cell.

E. Putting It All Together

With the JikesRVM modifications and the GC unit in place, the missing part is to allow the two to communicate. This happens through a Linux driver that we integrated into the kernel. This driver installs a character device that a process can write to in order to initialize the settings of the GC unit, initiate GC and poll its status. When a process accesses the device, the driver reads its process state, including the page-table base register and status bits, which are written to memory-mapped registers in the GC unit and used to configure its page-table walker. This allows the GC unit to operate in the same address space as the process on the CPU.

Processor Design (Rocket In-Order CPU @ 1 GHz)	
Physical Registers	32 (int), 32 (fp)
ITLB/DTLB Reach	128 KiB (32 entries each)
L1 Caches	16 KiB ICache, 16 KiB DCache
L2 Cache	256 KiB (8-way set-associative)
Memory Model (2 GiB Single Rank, DDR3-2000)	
Memory Access Scheduler	FR-FCFS MAS (16/8 req. in flight)
Page Policy	Open-Page
DRAM Latencies (ns)	14-14-14-47

Table I: **RocketChip Configuration**

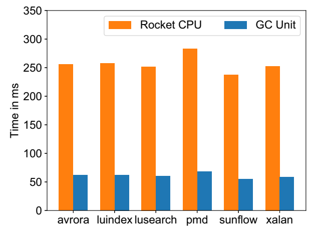
When the driver is initialized at boot time, it allocates a *spill region* in physical memory, whose bounds are then passed to the GC unit. This region has to be contiguous in physical memory and we currently allocate a static 4MB range by default (a full implementation could dynamically allocate additional memory). To communicate with the driver, we also extend JikesRVM with a C library (`libhwgc.so`). Our MMTk plan uses Jikes's SysCall foreign function interface to call into this C library, which in turn communicates with the driver to configure the hardware collector (e.g., setting the pointer to the `hwgc-space`), and to initiate a new collection. By replacing `libhwgc`, we can swap in a software implementation of our GC, as well as a version that performs software checks of the hardware unit (or produces a snapshot of the heap). This approach helped for debugging.

VI. Evaluation

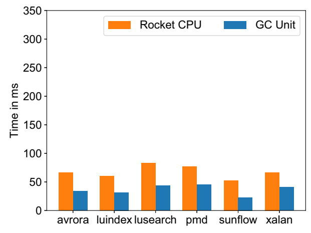
To evaluate our design, we ran it in FPGA-based simulation, using a simulation framework called FireSim [46] on Amazon EC2 F1 instances. FireSim enabled us to run cycle-accurate simulation at effective simulation rates of up to 125 MHz (on Xilinx UltraScale+ XCVU9P FPGAs). While our target RTL executes cycle-accurately on the FPGA, FireSim provides timing models for the memory system (in particular, DRAM and memory controller timing). This framework is similar to DRAM simulators such as DRAMSim [47], but runs at FPGA speeds and uses a FAME-style approach to adjust target timing to match the models [48].

Table I shows the configuration of our RocketChip SoC and the parameters of the memory model that we are using. We compare against an in-order *Rocket* core, which is comparable to a little/wimpy core in a BIG.little setup. We think this baseline is appropriate: A preliminary analysis of running heap snapshots on an older version of RocketChip's BOOM out-of-order core with DRAMSim showed that it outperformed Rocket by only around 12% on average [49]. This may be surprising, but limited benefits of out-of-order cores for GC have been confirmed on Intel systems [3].

The goal of our prototype and evaluation is (1) to demonstrate the potential efficiency of our unit in terms of GC performance and area use, (2) characterize the design space of parameters, and (3) show that these benefits can be gained without invasive changes to the SoC. While we integrated our design into RocketChip, our goal is not to improve this specific system, but instead understand high-level estimates and trade-offs for our GC unit design.



(a) Mark Phase



(b) Sweep Phase

Figure 15: **GC Performance**. On average, the GC Unit outperforms the CPU by a factor of 4.2× for mark and 1.9× for sweep.

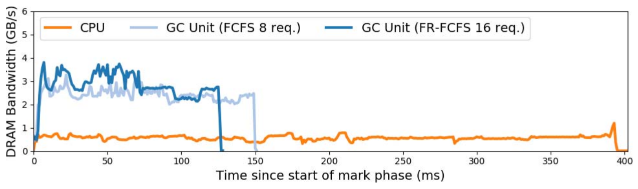
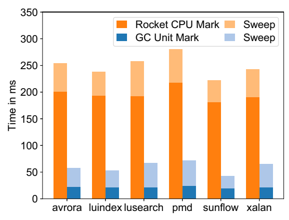
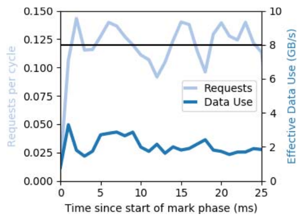


Figure 16: **Memory Bandwidth.** Measured for the last GC pause of the *avrora* benchmark, based on 64B cache line accesses.



(a) Mark/Sweep Performance



(b) Memory Bandwidth

Figure 17: **GC Performance with 1 cycle DRAM and 8 GB/s bandwidth**. As some requests are smaller than cache lines, the amount of usable data is smaller than the theoretical peak bandwidth.

A. *Garbage Collection Performance*

Methodology: We evaluate performance using the subset of DaCapo benchmarks [16] that runs on our version of JikesRVM (not specific to RISC-V). We use the small benchmark size on a 200MB maximum heap and average across all GC pauses during the benchmark execution.

Our JikesRVM port does not include the optimizing JIT compiler. As Jikes JIT-compiles itself, this would have resulted in a slow baseline of the CPU version of the GC. We therefore rewrote Jikes's GC in C, compiling it with -O3 and linking it into the JVM using the same `libhwgc.so` library that we use to communicate with our hardware unit.

Overall Performance: Our baseline GC unit design contains 2 sweepers, a 1,024 entry mark-queue, 16 request slots for the marker, 32-entry TLBs and a 128-entry shared L2 TLB. This configuration outperforms Rocket by 4.2× on mark and 1.9× on sweep (Figure 15).

Memory Bandwidth: Figure 16 shows the source of this gain: our unit is more effective at exploiting memory bandwidth, particularly during the mark phase. This was confirmed by experimenting with different memory scheduling strategies: While Rocket was insensitive to the configuration, we found that our performance was significantly improved changing from FIFO MAS to FR-FCFS and increasing the maximum number of outstanding reads from 8 to 16.

Potential Performance: While the previous experiment showed a specific design point with a realistic memory model, we want to fundamentally understand how much memory bandwidth our unit can exploit if it was given a faster memory system. We therefore replaced our model with a latency-bandwidth pipe of latency 1 cycle and bandwidth 8 GB/s. In this regime, we outperform the CPU by an average of 9.0× on the mark phase (Figure 17a). We believe that this is similar to the speed-ups we could see in a high-end SoC with higher memory bandwidth and lower latency. Note that the limited speedup for the sweep phase is based on using only two sweepers, and can be increased (Section VI-B).

Instrumenting our unit, we found that our TileLink port is busy 88% of all mark cycles. Figure 17b shows that this translates to a request being sent into the memory system every 8.66 cycles. With 64B cache lines, one request every 8 cycles would be the full bandwidth of the 8GB/s system, but as our requests are less than 64B in size, we sometimes exceed this limit. Using small requests also means that, depending on the memory system, we may not be able to use all 8 GB/s (we consume a maximum 3.3 GB/s of data).

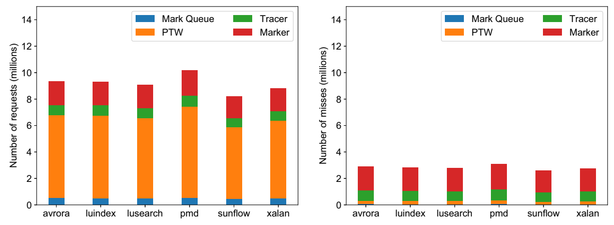
Limits/Impact of TLBs: To understand what prevents our baseline from reaching this 9.0× speedup, we instrumented the unit to record sources of stalls. We also compared to preliminary simulations on DRAMSim with 8 banks and no virtual memory, which showed 8.5× speedup over Rocket.

One bottleneck are TLB accesses in marker and tracer: as the TLB and page table walker are blocking, TLB misses can serialize execution. Future work should therefore introduce a non-blocking TLB that can perform multiple page-table walks concurrently while still serving requests that hit in the TLB. As the unit is pipelined, there is also an opportunity to use bigger multi-cycle TLBs, which might reduce TLB pressure and improve area, as they can use sequential SRAMs.

B. *Impact of Design Parameters*

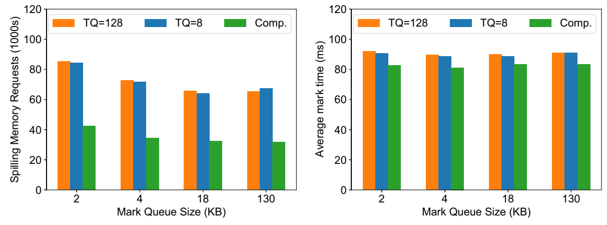
Cache Partitioning: As described in Section V-C, we started with a design that had a small, shared cache. We found that this performed barely better than the CPU. Figure 18a shows why: 2/3 of requests to the cache are from the page-table walker (as the mark phase has little locality and therefore introduces a large number of TLB misses).

This creates a lot of contention on the cache's crossbar, effectively drowning out requests by other units. This led us to apply cache partitioning: The PTW benefits from a small 8KB cache to hold part of the page table, while the mark queue and sweeper access memory sequentially and therefore only need 2 cache lines. Meanwhile, the marker and tracer can connect to the interconnect directly. Another advantage of this setup is that we can remove features from caches



(a) L1 Cache Requests (b) Memory Requests

Figure 18: **Traversal Unit Memory Requests**. In order to reduce contention, we partition the L1 cache into smaller caches.



(a) Spilling Memory Requests (b) Mark Performance

Figure 19: **Mark Queue Size Trade-Offs**. Sizes include *inQ/outQ* and we show numbers for two different tracer queue (TQ) sizes, as well as with compressed references.

that are not needed (e.g., the mark queue only operates on physical memory and therefore does not need a TLB).

The result is shown in Figure 18b: In terms of memory requests that are sent into the actual memory system, marker and tracer now dominate (which is the intention, as these are the units that perform the actual work).

Impact of Mark Queue Size: The mark queue is the largest data structure of our unit and we assumed that its size has a major impact on performance. Figure 19 shows that the size has an impact on the amount of spilled data. However, spilling accounts for only $\approx$ 2% of memory requests.

We were surprised to find that the mark queue's impact on overall performance is small. The reason is that most of the parallelism in the heap traversal exists at the beginning: The queue fills up, almost all of this data is spilled into memory and in the steady state that follows, enqueuing and dequeueing occur at about the same rate, which means that the queue stays mostly full without much spilling.

We can therefore make the queue very small (e.g., 2 KB) without sacrificing performance. An interesting trade-off is that we could throttle the tracer to match the dequeueing rate of the mark queue. As every reference in the tracer queue expands to multiple references in the mark queue, this would help manage the amount of spilling.

Mark Bit Caching: A small number of objects account for 10% of all memory accesses (Figure 21). Storing the most recently seen references therefore helps reduce the number of marks requests. The largest gain per area can be achieved with a small cache (<64 elements). At the same time, we found this to not have a substantial impact on the mark performance (but this may change closer to peak bandwidth).

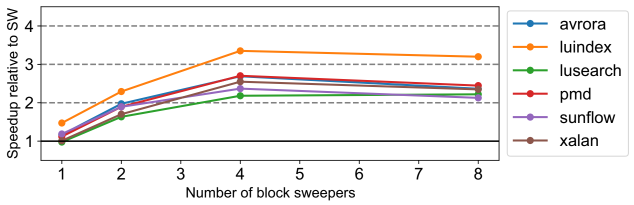
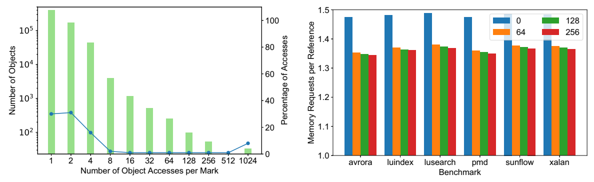


Figure 20: **Scaling the number of block sweepers.** Performance is reported as speed-up relative to the software implementation.



(a) Access Frequencies (b) Effect of Caching

Figure 21: **Impact of Mark Bit Cache**. 56 objects account for 10% of accesses (8th GC of *luindex*), and we can filter these with a cache.

Mark Queue Compression: Figure 19 shows that our compression scheme from Section V-C reduces spilling by a factor of 2. Note that this scheme compresses to 32b – real implementations would likely need to preserve at least 48b.

Sweeper Parallelism: Figure 20 shows how additional block sweepers improve sweep performance. We found that we scale linearly to 2 sweepers but that beyond this point, speed-ups start to reduce. At 8 sweepers, the contention on the memory system starts to outweigh the benefits from parallelism. 4 sweepers outperform the CPU by 2-3×. The optimal number depends on the system design.

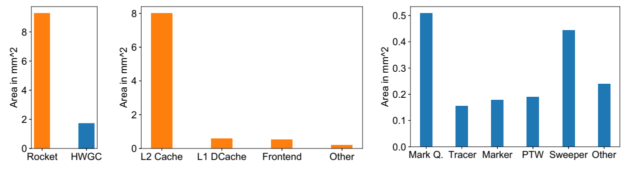
C. Area & Power Synthesis Results

We ran our design through Synopsys Design Compiler in topographical mode with the SAED EDK 32/28 standard cell library [50], using a VLSI flow developed at Berkeley called HAMMER [51]. This provides us with ballpark estimates of area and power numbers. Figure 22 shows that our GC unit is 18.5% the size of the CPU, most of which is taken by the mark queue. This is comparable to the area of 64KB of SRAM. Note that this is comparing to a small CPU – the trade-off would be much more pronounced in a server or mobile SoC, where a block of this size is negligible.

To estimate energy, we collected DRAM-level counters for the GC pauses in Figure 16 and ran them through MICRON's DDR3 Power Calculator spreadsheet [52]. Power numbers for the GC unit and processor were taken from Design Compiler. Using these power numbers and execution times, we calculate the total energy, broken into mark and sweep (Figure 23). Without activity counters, these results are not exact, but we conclude that the overall energy for the GC unit will likely improve over the CPU (by 14.5% in our results).

VII. Discussion & Future Work

Our unit is small enough to be added to any SoC. Its potential 9× speed-up would translate to application speed-



(a) Total (b) Rocket CPU (c) GC Unit

Figure 22: **Area**. Estimated using Synopsys DC with the SAED EDK 32/28 standard cell library. Note that Rocket is a small CPU.

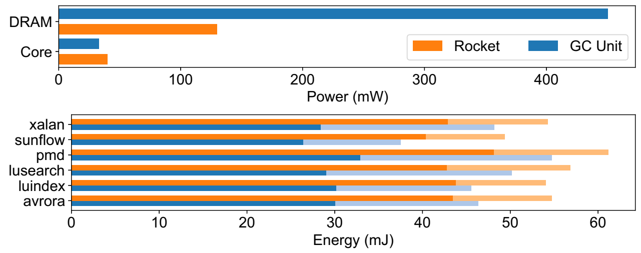


Figure 23: **Power and Energy**. Due to its higher bandwidth, the GC Unit's DRAM power is much higher, but the overall energy is still lower.

ups of 9% if an average of 10% of CPU cycles were spent on GC (31% if it was 35% [3]). Further, the accelerator not only decreases on-chip area/power, but also overall energy. Finally, our design is simple (3,122 lines of Chisel code), reducing verification effort. We now discuss several additional aspects:

Context Switching: The setup and context switch time could be made very low: if the device was mapped in user space and streams the roots as it executes (rather than copying them at the start), the minimum overhead would be equivalent to transferring less than 64B into an MMIO region, which is negligible compared to a mark pass. Flushing the accelerator's state would be similar to a context switch on a CPU.

Heap Size Scalability: Due to the lack of caching, most of the unit's operation is independent of the heap size. However, a larger heap size would increase the pressure on TLB and PTW cache (but not more than for a CPU). As discussed in Section VI-A, the TLB is currently a bottleneck, but large heaps could use superpages instead of 4KB pages.

Supporting a general object layout: While the bidirectional layout helps performance, it is not fundamental to our approach and forcing runtimes to adapt it to support our unit is limiting. A more general accelerator could support arbitrary layouts by replacing the marker with a small RISC-V microcontroller (only implementing the base ISA). We could then load a small program into this core which parses the object layout, schedules the appropriate requests and enqueues outgoing references for the tracer.

Bandwidth Throttling: Our GC unit aims to maximize bandwidth, potentially interfering with applications on the CPU. This interference could be reduced by communicating with the memory controller to only use residual bandwidth.

Proportionality and Parallelism: The accelerator bandwidth could potentially be increased by replicating units. Switching these units on and off would allow a concurrent GC to throttle or boost tracing, depending on memory pressure in the application. This can improve energy consumption [53].

Supporting multiple applications: Our current design only supports one process at a time, but the same unit could perform GC for multiple processes simultaneously, by tagging references by process and supporting multiple page tables.

Page faults: The JVM currently has to map the entire address space. A more general system might handle page faults as well (likely by forwarding them to the CPU).

VIII. Related Work

Hardware support for garbage collection is not a new idea, and there has been a large body of work throughout the years. Most commercial products focus on barriers, including Azul Vega [6], [54] and IBM's Guarded Storage Facility [34]. There is also standalone work on barriers [31], [32].

There has been work on hardware support for reference counting [8], which is orthogonal to our system and reduces the number of times a tracing collector needs to be invoked. GC has also seen interest in the real-time and embedded systems communities, particularly in the context of Java processors [9], [31], [55]. Some of them introduced GC features such as non-blocking object copying [56]. Bidirectional object layouts have been used in this context as well [55].

There have been several proposals for hardware units performing garbage collection. For example, the IBM Cell processor could use its SPE accelerators for GC [57]. Further, Wright et al. at Sun proposed a hardware-assisted GC that relied on an object-aware memory architecture [5], [11]. The design provided a fully concurrent GC but required changes to the memory system. Finally, Bacon et al. presented a stall-free GC unit for FPGA on-chip memory, eliminating all pauses [7]. However, this work was in the context of BRAMs on FPGAs, which are dual-ported and have predictable timing.

Some other accelerators have similarities to our general approach, but for different types of workloads. For example, walkers [58] accelerate hash index lookups for databases, while Mallacc [59] accelerates memory allocation.

IX. Conclusion

We introduced the design of a hardware accelerator for GC that can be integrated into a server or mobile SoC and performs GC for the application on the CPU. The unit can be implemented at a very low hardware cost (equivalent to 64KB of SRAM), generalizes to stop-the-world and concurrent collectors and does not require modifications to the SoC beyond those required by any DMA-capable device.

By implementing a prototype of this design in the context of a real SoC, we demonstrate that the integration effort is manageable, and that the unit takes up 18.5% the area of a small CPU, while speeding up GC by 3.3× overall (4.2× for marking). At this low cost, we believe that there is a strong case to integrate such a device into SoC designs.

Acknowledgments

We thank Ben Keller for collaborating on a precursor of this work. We also want to thank Sagar Karandikar, Howard Mao, Donggyu Kim and the entire FireSim group for their work on the simulation infrastructure we used, Colin Schmidt for his help with RocketChip, and Edward Wang for supporting us in using HAMMER. Special thanks is owed to David Biancolin for developing the on-FPGA memory models used for our evaluation and supporting our experiments. Finally, this work would not have been possible without the RocketChip infrastructure developed by many authors, and we want to especially thank Chris Celio, Henry Cook, Yunsup Lee and Andrew Waterman for their help and feedback throughout this project. We also want to thank the many people who have given feedback to this project throughout the years, including Steve Blackburn, Tim Harris, Kathryn McKinley, Philip Reames, Ian Rogers, Gil Tene, Mario Wolczko and the anonymous reviewers. Research was partially funded by ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, LGE, NVIDIA, Oracle, and Samsung.

References

[1] E. Moss, "The Cleanest Garbage Collection," *Commun. ACM*, vol. 56, no. 12, pp. 100–100, Dec. 2013.

[2] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.

[3] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software," in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[4] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sep. 1996.

[5] G. Wright, "A Hardware-Assisted Concurrent & Parallel GC Algorithm," 2008.

[6] C. Click, G. Tene, and M. Wolf, "The Pauseless GC Algorithm," in *Proceedings of the 1st International Conference on Virtual Execution Environments*, 2005.

[7] D. F. Bacon, P. Cheng, and S. Shukla, "And then There Were None: A Stall-free Real-time Garbage Collector for Reconfigurable Hardware," *Commun. ACM*, vol. 56, no. 12, pp. 101–109, Dec. 2013.

[8] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible Reference-counting-based Hardware Acceleration for Garbage Collection," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[9] M. Schoeberl, "JOP: A Java Optimized Processor," in *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Nov. 2003, pp. 346–359.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[11] G. Wright, M. L. Seidl, and M. Wolczko, "An object-aware memory architecture," *SCP*, vol. 62, no. 2, pp. 145–163, 2006.

[12] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz, "Trash Day: Coordinating Garbage Collection in Distributed Systems," in *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[13] D. Terei and A. Levy, "Blade: A Data Center Garbage Collector," *arXiv:1504.02578 [cs]*, Apr. 2015.

[14] I. Gog, J. Giceva, M. Schwarzkopf, K. Viswani, D. Vytiniotis, G. Ramalingan *et al.*, "Broom: Sweeping out Garbage Collection from Big Data systems," in *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[15] N. Artyushov, "Revealing the length of Garbage Collection pauses," https://plumbr.eu/blog/garbage-collection/revealing-the-length-of-garbage-collection-pauses.

[16] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur *et al.*, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the 21st Conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2006.

[17] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng *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.

[18] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio *et al.*, "The Rocket Chip Generator," EECS Dept, UC Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[19] M. Maas, P. Reames, J. Morlan, K. Asanovic, A. D. Joseph, and J. Kubiatowicz, "GPUs As an Opportunity for Offloading Garbage Collection," in *Proceedings of the 2012 International Symposium on Memory Management*, 2012.

[20] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Greenwich, CT, USA: Manning Publications Co., 2010.

[21] G. Tene, "How NOT to Measure Latency," https://www.infoq.com/presentations/latency-response-time, 2016.

[22] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey *et al.*, "United States Data Center Energy Usage Report," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-1005775, 2016.

[23] S. Cass, "The 2017 Top Programming Languages," https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages, Jul. 2017.

[24] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "Yak: A High-Performance Big-Data-Friendly Garbage Collector," in *12th Symposium on Operating Systems Design and Implementation*, 2016.

[25] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[26] E. Kaczmarek and L. Yi, "Taming GC Pauses for Humongous Java Heaps in Spark Graph Computing," 2015.

[27] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform*, 2016.

[28] P. Liden, "CFV: New Project: ZGC," http://mail.openjdk.java.net/pipermail/announce/2017-October/000237.html, Wed Oct 25 19:45:23 UTC 2017.

[29] "Go GC: Prioritizing low latency and simplicity - The Go Blog," https://blog.golang.org/go15gc.

[30] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th International Symposium on Memory Management*, 2004.

[31] M. Meyer, "A True Hardware Read Barrier," in *Proceedings of the International Symposium on Memory Management*, 2006.

[32] T. Harris, S. Tomic, A. Cristal, and O. Unsal, "Dynamic Filtering: Multi-purpose Architecture Support for Language Runtime Systems," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[33] C. Jacobi and A. Saporito, "The Next Generation IBM Z Systems Processor," *Hot Chips 29 Symposium*, 2017.

[34] "How Concurrent Scavenge using the Guarded Storage Facility Works," https://developer.ibm.com/javasdk/2017/09/25/concurrent-scavenge-using-guarded-storage-facility-works/.

[35] S. Beamer, K. Asanovic, and D. Patterson, "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server," in *2015 International Symposium on Workload Characterization*, 2015.

[36] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis *et al.*, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceeding of the 41st International Symposium on Computer Architecuture*, 2014.

[37] E. M. Gagnon and L. J. Hendren, "SableVM: A Research Framework for the Efficient Execution of Java Bytecode," in *In Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2000.

[38] D. Gu, C. Verbrugge, and E. M. Gagnon, "Relative Factors in Performance Analysis of Java Virtual Machines," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

[39] W. Puffitsch and M. Schoeberl, "Non-blocking Root Scanning for Real-time Garbage Collection," ser. JTRES '08.

[40] M. Maas, K. Asanovic, and J. Kubiatowicz, "Full-System Simulation of Java Workloads with RISC-V and the Jikes Research Virtual Machine," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[41] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and Water? High Performance Garbage Collection in Java with MMTk," in *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[42] Y. Lee, "Free Chips Project: A nonprofit for hosting open source RISC-V implementations, tools, code," 5th RISC-V Workshop, 2016.

[43] J. Kang, "SiFive FE300 and low-cost HiFive Development Board," 5th RISC-V Workshop, 2016.

[44] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing Hardware in a Scala Embedded Language," in *Proceedings of the 49th Design Automation Conference*, 2012.

[45] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic Design Patterns: A TileLink Case Study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[46] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *Proc. of the 45th International Symposium on Computer Architecture*, 2018.

[47] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: A Memory System Simulator," *Comput. Archit. News*, vol. 33, no. 4, pp. 100–107, Nov. 2005.

[48] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovic, and D. Patterson, "A Case for FAME: FPGA Architecture Model Execution," in *Proceedings of the 37th International Symposium on Computer Architecture*, 2010.

[49] M. Maas, K. Asanovic, and J. Kubiatowicz, "Grail Quest: A New Proposal for Hardware-assisted Garbage Collection," in *Workshop on Architectures and Systems for Big Data*, 2016.

[50] "Synopsys SAED_EDK32/28_CORE Databook," Tech. Rep. Version 1.0.0, 2012.

[51] E. Wang, "HAMMER: A Platform for Agile Physical Design," Master's thesis, EECS Dept, UC Berkeley, 2018.

[52] "Micron System Power Calculator Information," https://www.micron.com/support/tools-and-utilities/power-calc.

[53] A. Hussein, A. L. Hosking, M. Payer, and C. A. Vick, "Don't Race the Memory Bus: Taming the GC Leadfoot," in *Proc. of the 2015 International Symposium on Memory Management*.

[54] G. Tene, B. Iyengar, and M. Wolf, "C4: The Continuously Concurrent Compacting Collector," in *Proceedings of the International Symposium on Memory Management*, 2011.

[55] T. B. Preußer, P. Reichel, and R. G. Spallek, "An Embedded GC Module with Support for Multiple Mutators and Weak References," in *Architecture of Computing Systems 2010*.

[56] M. Schoeberl and W. Puffitsch, "Non-blocking Object Copy for Real-time Garbage Collection," in *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2008.

[57] C.-y. Cher and M. Gschwind, "Cell GC: Using the Cell synergistic processor as a garbage collection coprocessor," in *Proceedings of the 4th International Conference on Virtual Execution Environments*, 2008.

[58] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th International Symposium on Microarchitecture*.

[59] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating memory allocation," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*.