Theme Article

A Hardware Accelerator for Tracing Garbage Collection

Martin Maas
Google Brain

Krste Asanovic and John Kubiatowicz
University of California Berkeley

***Abstract*—Many workloads are written in garbage-collected languages and GC consumes a significant fraction of resources for these workloads. We propose to decrease this overhead 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, 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 4.2× the performance of an in-order CPU, at just 18.5% the area. 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.**

A LARGE FRACTION of programs are written in garbage-collected languages, on servers (C#, Java, Python), web browsers (JavaScript), and mobile devices (Android). These workloads spend up to 35% of their CPU cycles on GC (10% on average for Java applications[1]), and this increases further for pause-free concurrent collectors. In other words: a significant fraction of all compute cycles globally are spent on the GC alone. If we could dramatically reduce its cost, we would therefore save a substantial fraction of cycles and energy, and improve the user experience by reducing GC pauses.

Digital Object Identifier 10.1109/MM.2019.2910509
Date of publication 23 April 2019; date of current version 8 May 2019.

However, despite several decades of work and significant GC improvements—including hardware support for GC on architectures such as Azul's Vega and the IBM z14—the situation has not improved substantially. Our work suggests that this is because CPUs are a fundamentally bad fit for GC (even with hardware support, the actual GC operation is usually still performed by the CPU).

We demonstrate how codesigned hardware can exploit the properties of the GC operation to perform it at 18× higher performance per area, even when compared to a wimpy CPU core (currently the best choice for GC[1]). Our proposal consists of a GC accelerator that is located close to the memory controller and integrated into an SoC like any other IP block, without requiring any

Published by the IEEE Computer Society

changes to the CPU or memory system. The accelerator is small (about the size of a 64KB cache), which makes it a cheap and low risk addition to existing SoCs.

In addition to the hardware and software techniques that make this accelerator possible, we make contributions on two fronts: First, we describe the design space associated with GC acceleration and show how it could be used for different types of collectors and settings. Second, we demonstrate both the efficiency and the noninvasive integration of our accelerator by building an end-to-end RTL prototype of the unit into a complete RISC-V System-on-Chip (using RocketChip,[2] an open-source SoC used in academia and industry). We evaluate this end-to-end design by mapping it to FPGAs, running a full software stack that includes Linux with drivers for our GC accelerator, a Java Virtual Machine with software modifications to support the unit, and full Java benchmarks. Using this setup, we demonstrate that the prototype performs the mark phase of the GC at 4.2× the performance of a CPU, at 18.5% its area (reclamation speed-up is 1.9× with two sweeper units).

The idea of hardware support for GC is not new, but past work typically focused on general CPU features that improve some aspect of GC (such as read barriers), instead of the actual GC operation. However, none of these techniques have been widely adopted.

MOTIVATION AND HISTORICAL PERSPECTIVE

GC is a significant problem that has been the topic of research for over 50 years but is still not "solved." At a fundamental level, existing GCs have to trade off among three metrics: pause time, application performance, and memory utilization (which is determined by the GC throughput). Arguably, current GCs can perform well on any two of these metrics but not on all of them simultaneously. While this has been a challenge for several decades, the problem is being exacerbated by current trends, particularly ever-growing heap sizes. Terabyte-sized heaps in servers are now feasible, and the amount of garbage collection work grows with the size of live memory. At the same time, an increasing number of latency-sensitive workloads are written in garbage-collected languages, making them sensitive to pauses. As such, a garbage collector that avoids pauses and utilizes the available memory, while maintaining high application performance, is becoming increasingly important yet harder to achieve. It seems unlikely that we will find another order-of-magnitude improvement of GC in software—we therefore think that the solution to GC has to be hardware.

The idea of hardware support for GC is not new, but past work typically focused on general CPU features that improve some aspect of GC (such as read barriers), instead of the actual GC operation. However, none of these techniques have been widely adopted. We think there are three reasons for this:

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, even on the workloads they were designed for. This gave a substantial edge to nonspecialized 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).
3) *Invasiveness:* Most hardware-assisted GC designs were invasive and required rearchitecting the memory system. However, modern accelerators (such as those in mobile SoCs) are typically integrated as memory-mapped devices, without invasive changes.

We think that this makes it the perfect time to revisit hardware-assisted GC. We exploit these

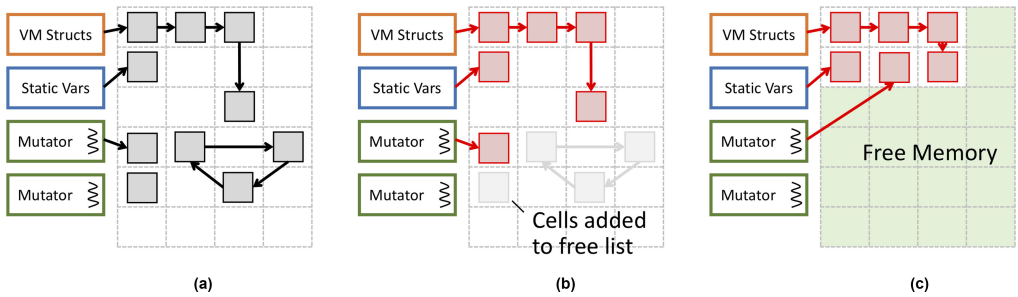

Figure 1. Basic GC operation. (a) Original heap. (b) After Mark & Sweep. (c) After compaction.

trends through a design that performs the GC operation in hardware while being noninvasive and general enough to be integrated into any SoC.

GARBAGE COLLECTION BACKGROUND

In object-oriented languages, the garbage collector periodically identifies objects that are no longer reachable by the program, and recycles them. Figure 1 shows an example: Reachability is defined by the object graph, where each node is an object and there is a directed edge from one node to another if there is a field in the first object that stores a reference to the second.

The most common type of GC in managed languages is tracing garbage collection—in fact, seven out of the ten most popular programming languages[3] use a tracing GC, either as their primary garbage collection mechanism or as cycle collector in a reference counting scheme. There are many variants of tracing collectors, but fundamentally, they all consist of two phases: 1) a traversal phase that starts from a set of roots (e.g., global variables or pointers stored on the stack) and traverses the object graph to identify and mark objects that are reachable; and 2) a reclamation phase that reclaims unmarked objects—e.g., by assembling them into a free list (known as "sweeping") or copying them into a new region (known as "compacting"). There are many variations of this theme, and some fold the two phases into one (e.g., scavenging GC). The phases can run in a stop-the-world pause or at the same time as the application (the latter type of GC is called "concurrent").

The key insight of our work is that both phases are a bad fit for traditional CPUs, as they are limited by keeping a large number of memory requests in flight, with little temporal locality. The CPU's sequential control flow and its load-store queue therefore limit performance, while both operations do not use most of the CPU's functional units or its area-intensive caches, hence wasting chip area. Similar to the argument for page table walkers, we therefore believe GC should be entirely performed in hardware to both increase the throughput of the operation and reduce the required area.

ACCELERATOR DESIGN

Our garbage collection accelerator is designed to be minimally invasive and requires no changes to the CPU or memory system. It is integrated into an SoC as an independent block, similar to other DMA-capable devices such as network adapters or flash controllers. The accelerator is connected to the existing cache-coherent interconnect, and all communication with the CPU occurs through this interconnect. To operate on the same virtual address space as the CPU, the accelerator has its own TLB and page table walker. As such, the application on the CPU perceives the GC accelerator as no different from another CPU performing the GC operation, which makes it easier to reason about correctness.

Internally, our accelerator consists of two units that implement the two phases of the GC operation: a traversal unit and a reclamation unit. We first show how these units work in a stop-the-world setting, and later discuss how the system would need to be adapted for a concurrent (pause-free) garbage collector.

Traversal Unit

Achieving high performance for the traversal phase while minimizing area is challenging. It

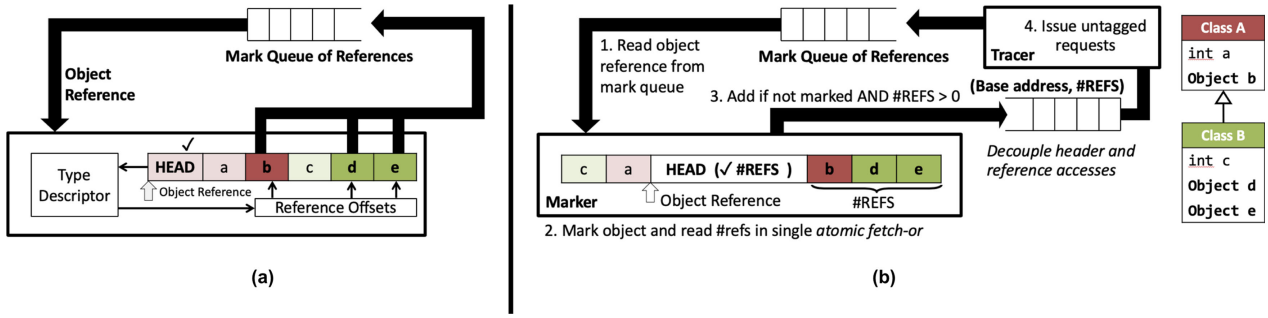

Figure 2. High-level overview of the traversal unit's operation. (a) Traditional traversal phase. (b) Traversal phase on the GC accelerator.

requires codesign across both hardware and software layers. We now describe the key insights for achieving these goals.

Bidirectional Object Layout: At a high level, a typical traversal operation performs a breadth-first search (BFS). It keeps a frontier (mark queue) of pointers and, in every step, takes out an element, marks the object at that location, identifies all object fields that contain pointers, and copies them back into the mark queue [see Figure 2(a)]. In software, this is typically implemented as a loop that looks up a type descriptor in the object's header, which lists the fields that contain references (these fields are usually interspersed through the object—to support inheritance, objects must typically start with the fields belonging to parent classes, followed by the children).

On a CPU, this works well, as the type descriptor is typically in the cache. However, our accelerator strives to avoid using cache area. We therefore need to identify the pointer fields without this extra lookup. We hence change the language runtime system to internally use a bidirectional object layout,[4] which is an old idea that was originally proposed to improve locality on CPUs but has seen limited success and adoption there. In a bidirectional object layout, the header (i.e., base of the object) is placed in the middle, and all reference fields are placed to one side, while all the non-reference fields are to the other. We found that while this change does not make much difference on a CPU, it is a perfect fit for an accelerator. Now, instead of having to do an extra lookup, we can simply store the number of references and the mark bit in the same header word, and mark the object and identify the number of references in a single fetch-or operation.

Decoupled Marking and Copying: The BFS traversal on the CPU is a nested loop, which means that we can only keep as many requests in flight as we can speculate ahead in the control flow. In our unit, we want to issue as many memory requests as possible, up to one every cycle the memory system is ready. To do so, we pipeline the operation [see Figure 2(b)]. One stage, the *Marker,* is taking a pointer off the mark queue every cycle and sends out a memory request to set the object's mark bit and fetch the number of references. If the mark bit was not set and there is at least one reference field, those references need to be copied into the mark queue. This is done by the *Tracer*, which copies pointers back into the queue for those objects that were identified by the marker. To maximize memory bandwidth, we pipeline these two units and decouple them using a queue: When the marker encounters a large number of already marked objects, the tracer does not stall but can still perform work from the queue, and if the tracer is copying an object with many references (e.g., a pointer array), it does not stall the marker until the queue fills up.

Untagged Memory Requests: The final insight is how to perform the memory requests themselves. GC does not rely on the order in which the graph is visited, and we therefore do not need to maintain ordering between the memory requests. This means that we can reduce the amount of on-chip state to store per memory request in flight. In fact, the tracer does not need to keep *any* state; it simply sends requests into the memory system and

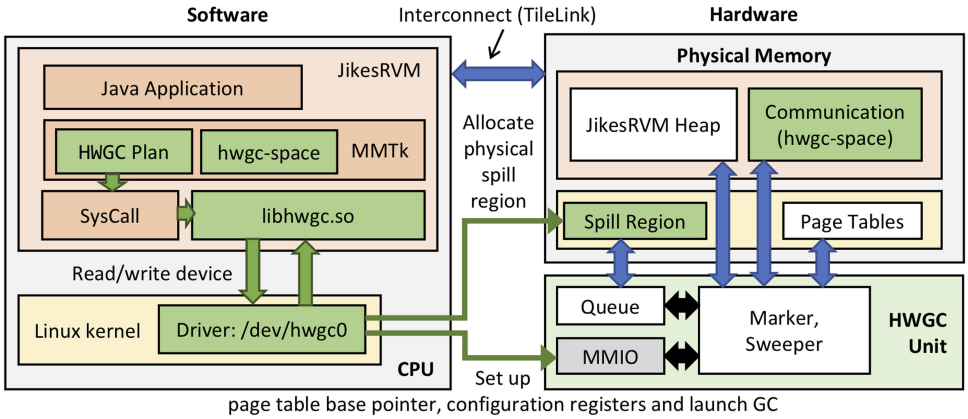

Figure 3. System-level overview of the prototype.

adds them to the mark queue in the order they return. The marker, on the other hand, needs to store a tag and the base address for each request it sends into the memory system; however, this is much less state than on a CPU, which would need one miss status holding register (MSHR) for each request in flight and store information about ordering, the type of request, etc. As a result, we save a large fraction of the on-chip state and can keep a larger number of requests in flight.

Reclamation Unit

While traversal accounts for the majority of cycles of the garbage collection operation, we offload the reclamation phase as well. In contrast to the traversal operation, the reclamation operation is typically embarrassingly parallel. It can be parallelized by dividing the heap into blocks, performing a linear scan through each block, and either assembling all unmarked objects in the block into a free list, or compacting (i.e., copying) all marked objects into a new region. To support such operations efficiently, the reclamation unit consists of a number of parallel "block sweepers," which are state machines that each operate on one block at a time. The number of such state machines is a configuration parameter and is explored in our original paper.[5]

RISC-V-BASED PROTOTYPE SYSTEM

While the previous section describes the high-level design, there are many details on how to make the design work in practice and how to integrate it into a full system. At the microarchitectural level, our unit can take advantage of the fact that most of our memory requests are of the same type and do not have to maintain ordering constraints between them, which allows us to save area that would be spent on MSHRs on a traditional CPU. Another challenge is how to represent the mark queue: In our design, we keep the middle of the queue in main memory, with head and tail sections cached on-chip. The size of the on-chip portion is a design parameter.

We implemented our design in a RISC-V RocketChip SoC[2] (Figure 3 shows an overview). RocketChip is a framework for generating SoC designs that can target both FPGAs and ASICs. It supports a range of different processor designs and IP blocks, combined with generators for interconnects and caches. RocketChip has been used in over a dozen tape-outs, both in academia and industry. To evaluate our design with a full software stack, we ported the Yocto Linux distribution generator to RISC-V, in order to provide the necessary dependencies to execute a full JVM. We then ported the JikesRVM research Java virtual machine[6] to RISC-V, including its nonoptimizing JIT compiler. This full software stack allowed us to perform an end-to-end evaluation of the system, running Dacapo benchmarks.[7]

The accelerator is implemented as a DMA-capable device and communicates with the CPU through the cache-coherent on-chip interconnect. It has its own TLB and page-table walker, which allows it to operate on the same virtual address space as the application on the CPU. The JVM communicates with the unit through shared memory and a Linux kernel driver, and we modified the runtime system to implement

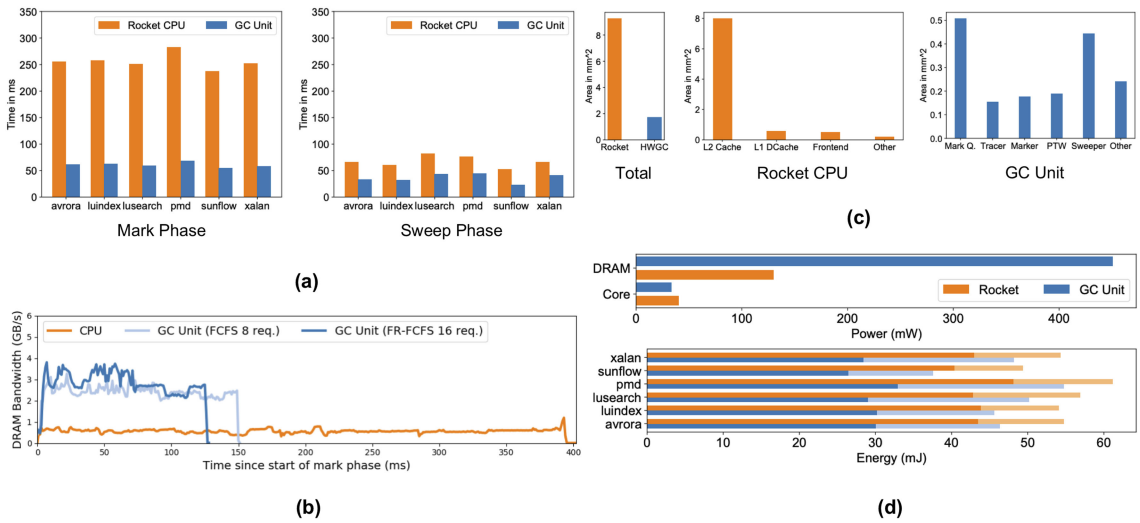

Figure 4. Performance, area, and power of the garbage collection accelerator. (a) GC performance. (b) Area. (c) Memory bandwidth. (d) Power and energy.

our object layout and add a new GC scheme that uses the unit. At initialization time, the driver allocates a physical region for the unit which is used to spill the content of the mark queue when it fills up. When the JVM triggers a GC pass, our new GC scheme will instead call into a C library, which communicates with the Linux driver through a character-based device interface. The driver will write information such as the page table base register and the base address of a shared memory region within the JVM into memory-mapped registers within the GC accelerator. During the rest of the GC operation, communication between the JVM and the accelerator primarily occurs through this region, which removes the driver from the critical path. In particular, the JVM scans all of its threads for roots and writes them into the memory region (while this operation could be offloaded as well, we refrain from doing so in our prototype).

EVALUATION

We evaluate the system using an FPGA simulation framework called FireSim,[8] which allows us to run the entire SoC design cycle accurately on AWS F1 FPGAs in the cloud, while running FPGA-based timing models of DRAM and the memory system to simulate the performance that we would see if the design was implemented as an ASIC running at 1 GHz. This setup produces cycle-accurate results while running the simulation at 125 MHz.

Our baseline is a Rocket in-order CPU with a 256-KB L2 cache. This is representative of the type of wimpy CPU that has been shown to be efficient for garbage collection. Note that previous research[1] showed that out-of-order CPUs, while faster, are not the best tradeoff point for GC (a result we confirmed in preliminary simulations). Compared to our baseline, our accelerator achieves 4.2× the mark performance as the CPU, and a 1.9× speed-up for the sweep phase with two sweepers [see Figure 4(a) and (b)].

However, the performance of the accelerator is limited by the memory system (the SoC design we used only had a relatively small amount of memory bandwidth). We therefore studied the limits of scaling our system as the memory system bandwidth increases. We repeated our experiments with a memory system where every request returns after one cycle, and found that while the CPU cannot exploit most of the additional bandwidth, the accelerator can, leading to a mark speed-up of 9.0× over the CPU. This indicates that the speed-ups in a realistic memory system will be higher, likely between 4.2×and 9.0×.

We also estimate the unit's area by running the design through Synopsys Design Compiler with the SAED EDK 32/28 educational standard-cell library [see Figure 4(c)]. Compared to the CPU, the unit occupies 18.5% of the CPU's area, a size equivalent to a 64 KB cache. Note that these numbers are conservative and that there is ample

room for further improvements on both performance and area (we describe some of these opportunities in our original paper[5]). Finally, we estimate power consumption [see Figure 4(d)] and conclude that the accelerator reduces the energy of the GC operation by 15%.

CONCURRENT GARBAGE COLLECTION

While our prototype is implemented as a stop-the-world collector, we also show a solution to the problem of using the design for concurrent GC without changing the CPU. The key challenge of a concurrent garbage collector is that the application threads (or "mutators") are running in parallel with garbage collection, which means that the GC can move objects while the application is trying to access them and *vice versa*.

While our prototype is implemented as a stop-the-world collector, we also show a solution to the problem of using the design for concurrent GC without changing the CPU.

This problem is typically addressed through constructs called "barriers" (not to be confused with barriers in memory consistency or parallel computing). In this context, a barrier refers to a small amount of code that is compiled into the instruction stream and guards a reference operation; for example, a read barrier is compiled into every read of a reference into a register, and checks whether the object that is being loaded has been moved. Since barriers need to be added for every reference access, they can incur a significant cost both in terms of execution time and instruction overhead.

Barriers typically have a fast path ("the object has not been moved") and a slow path. Existing hardware support for GC has focused on improving the barrier fast path, e.g., by combining it with virtual memory checks.[9] However, when the barrier encounters the slow path, it needs to redirect the instruction stream to a handler, either through a branch or a trap. Both cases will cause the CPU pipeline to be flushed, which is expensive.

Therefore, we propose (but do not prototype) a mechanism that never flushes the pipeline. The semantics of our proposed barrier are those of a load that always returns the new location of an object, whether or not the object has moved. This means that the CPU can speculate over the barrier in the same way as over any other load, without flushing the pipeline. We propose two different ways of implementing this barrier, one that modifies the CPU and one that uses virtual memory tricks to achieve the same behavior and does not modify the CPU. We refer to the full paper for details.[5]

EXTENSIONS FOR REAL-WORLD ADOPTION

While our prototype could be directly integrated into a real-world system, we believe that there are several extensions that should be considered for real-world adoption:

- *Multiple Processes:* 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. This would allow amortizing the GC unit across multiple workloads and exploiting more of the available bandwidth.
- *Programmability and 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—and even algorithms—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 throttle the accelerator dynamically and only use residual bandwidth not used by the application.

We believe that all of these extensions are compatible with our proposed design.

LONG-TERM VISION: GC-OBLIVIOUS SYSTEMS

In this article, we provide a blueprint for what a real-world GC accelerator could look like. We believe that this kind of accelerator could become a standard component in all server, desktop, and mobile chips, due to its low area cost, noninvasive design, and the pervasiveness of GC. The fact that server designs such as the IBM z14 have already adopted more invasive but limited GC hardware points to the willingness of manufacturers to do so. One could also see intermediate points where the accelerator would be implemented as an add-on card connected through QPI or PCIe, or is implemented on the FPGA of integrated FPGA+CPU platforms.

We envision that future systems could have a GC accelerator that completely eliminates garbage collection from software and is invisible to the application. Such an accelerator would be fully concurrent (i.e., never pause the application), adjust its bandwidth consumption dynamically to never interfere with the application, and not need to run any code on the CPU.

While our prototype looks at a specific design of such an accelerator, our overarching long-term vision is broader. We envision that future systems could have a GC accelerator that completely eliminates garbage collection from software and is invisible to the application. Such an accelerator would be fully concurrent (i.e., never pause the application), adjust its bandwidth consumption dynamically to never interfere with the application, and not need to run any code on the CPU. In this setting, the application on the CPU would be under the illusion that there is a constant supply of available memory and never needs to schedule garbage collector threads, work around GC pauses, or incur unexpected slowdowns due to barriers. We call such systems "GC-oblivious," and we think that our collector design and the techniques introduced in our paper are a first step toward such a system.

CONCLUSION

We introduced a hardware accelerator for GC that can be implemented at a very low hardware cost (equivalent to 64 KB of SRAM) and does not require modifications to the SoC beyond those required by any DMA-capable device. At this low cost, we believe that there is a strong case to integrate such a device into any SoC design.

ACKNOWLEDGMENT

This work was done at the University of California, Berkeley. It was supported in part by ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, LGE, NVIDIA, Oracle, and Samsung.

REFERENCES

1. 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 *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 225–236.
2. K. Asanovic *et al.*, "The rocket chip generator," *EECS Dept.*, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016.
3. S. Cassand P. Bulusu, "Interactive: The top programming languages 2018," *IEEE Spectrum*. [Online]. Jul. 31, 2018, https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018
4. E. M. Gagnon and L. J. Hendren, "Sable VM: A research framework for the efficient execution of java bytecode," in *Proc. Java Virtual Mach. Res. Technol. Symp.*, 2001, pp. 27–40.
5. M. Maas, K. Asanovic, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *Proc. 45th Annu. Int. Symp. Comput. Architecture.*, 2018, pp. 138–151.
6. B. Alpern *et al.*, "The jikes research virtual machine project: Building an open-source research community," *IBM Syst. J.*, vol. 44, no. 2, pp. 399–417, 2005.
7. S. M. Blackburn *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," *ACM Sigplan Notices*, vol. 41, no. 10, pp. 169–190, 2006.
8. S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, and Q. Huang, "Firesim: FPGA-accelerated

cycle-exact scale-out system simulation in the public cloud," in *Proc. 45th Ann. Int. Symp. Comput. Architecture.*, 2018, pp. 29–42.

9. C. Click, G. Tene, and M. Wolf, "The pauseless GC algorithm," in *Proc. 1st ACM/USENIX Int. Conf. Virtual Execution Environ.*, 2005, pp. 46–56.

Martin Maas is a research scientist at Google, where he is a member of the Google Brain team. His research interests span language runtimes, computer architecture, systems and machine learning. He has a PhD in computer science from the University of California, Berkeley. This work was done while he was a graduate student at UC Berkeley. Contact him at mmaas@google.com.

Krste Asanovic is a professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and the Association for Computing Machinery (ACM). Contact him at krste@berkeley.edu.

John Kubiatowicz is a professor of electrical engineering and computer science at the University of California at Berkeley. His specialties include computer architecture, operating systems, and networking. His research interests include speculative approaches for computer design, such as quantum, biological, and autonomic computing, as well as issues in internet-scale systems design, namely security, privacy, and denial-of-service resilience. He has a PhD in electrical engineering and computer science from MIT and a dual BS in electrical engineering and physics, as well as an MS in electrical engineering and computer science from MIT. Contact him at kubitron@cs.berkeley.edu.