Synthesized In-BRAM Garbage Collection for Accelerators with Immutable Memory

Martha Barker, Stephen A. Edwards, and Martha A. Kim
Department of Computer Science
Columbia University
New York, USA
{mbarker, sedwards, martha}@cs.columbia.edu

2022 32nd International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-7390-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/FPL57034.2022.00019

***Abstract*—Speed and ease of accelerator design is a growing need. High level programming languages have provided significant gains in the software world, but lag for hardware. We present a hardware implementation of a garbage collector that automates memory management, one of the major conveniences of modern software languages. Our garbage collector runs concurrently with the application it serves, using already-idle memory slots to do its work with little to no impact on performance. To achieve this, our collector exploits rapid synchronization that is straightforward in hardware but difficult in software. This synchronization enables the collector to interleave with fine pockets of idleness on a per-cycle, per-heap basis.**

Our collector typically incurred negligible overhead, only slowing the application to wait for collection to free memory when the heaps were so small that the collector could not keep pace with allocation. We also found that our concurrent collector performs best under an eager collection policy that collects garbage well before the application exhausts the available memory. Although this eager strategy performs more collection operations than strictly necessary, the application never pauses because our collector operates entirely in the background.

***Index Terms*—garbage collection, BRAM, heaps, accelerators, high-level synthesis**

I. Introduction

FPGAs are attractive acceleration platforms with recent commercial and cloud deployments. However, programming FPGAs remains a formidable engineering effort, demanding intimate knowledge of hardware design and the application. High-level synthesis (HLS) can greatly reduce this effort by compiling programs specified in a higher-level language, although it rarely matches hand-engineered design quality. Nevertheless, rapid design and deployment sit alongside performance as paramount concerns, as suggested by Google's use of HLS to design its video transcoding chip [1].

In modern programming languages, garbage collection simplifies the developer's task and eliminates bugs such as memory leaks and use-after-free errors. While today's HLS frameworks excel at optimizing arithmetic, loops, and regular memory accesses, memory management is not well supported.

Software garbage collectors typically impose a 10-35% slowdown depending on the application [2], [3], which hardware acceleration can sometimes reduce [3]–[6]. By definition, accelerators are faster than software and thus *even more sensitive* to interference from garbage collection. A 1ms collection pause on a 100ms software task is just 1% of runtime, but a 1ms pause on a 10ms accelerated task is a 10% slowdown.

We present a synthesized, hardware-only automatic garbage collector that operates concurrently with the accelerator, imposing a negligible slowdown on the accelerator (e.g., under 1%) when heaps are sufficiently large. Our collector integrates with FPGA accelerators synthesized from Haskell [7], [8] and operates on multiple, parallel heap memories. Each is implemented with FPGA BRAMs and can perform either an application or garbage collection operation each clock cycle.

When free memory dwindles below a given fraction of the total heap size, our garbage collector initiates collection. First, it performs a global memory fence, waiting for all in-flight memory operations to complete while prohibiting new ones. Then, the collector takes a single-cycle snapshot of all the "roots"—the pointers directly in the program's possession—and resumes normal application execution concurrently with collection. In our hardware collector, this operation is vastly less intrusive than is typical in software implementations.

Our hardware collector synchronizes rapidly with the application, allowing the collector to safely run in the background. The collector uses memory cycles that the application otherwise would not use, much like simultaneous multithreading [9] fills idle execution slots with instructions from other threads. To keep the memory allocator available to the application during collection, the allocator and collector maintain a shared list of free objects, which is easier to do in hardware.

Our collector is integrated into an FPGA HLS flow that relies on an immutable memory model, i.e., the contents of an object are set only when it is allocated. While this simplifies allowing the application to run during collection, our approach does not demand it. Existing techniques [4], [10] to manage heap mutations during collection could be used.

II. Garbage Collection Background

A garbage collector is an automated memory management tool that identifies and reclaims memory allocated to "dead" objects that are no longer accessible by the application, releasing application developers from having to manually free unused memory and the bugs that often accompany doing so. Jones [11] is the standard text on garbage collection.

Tracing and reference counting are the two main approaches. Ours is a tracing collector, which starts from the application's *roots*—the set of pointers held directly in variables (our roots reside in buffers throughout the

1946-1488/22/$31.00 ©2022 IEEE
DOI 10.1109/FPL57034.2022.00019

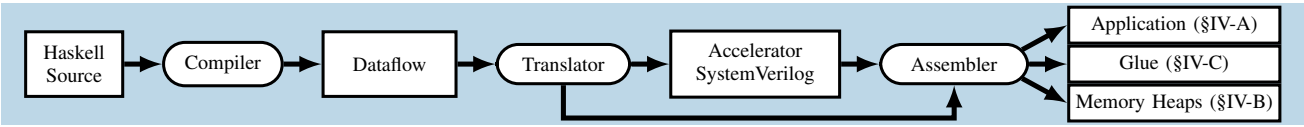

Fig. 1. Accelerator synthesis toolchain. The Haskell program is compiled into SystemVerilog through a dataflow IR. The assembler synthesizes per-type garbage collected heaps (Section IV-B) connected to the application with glue logic (Section IV-C) according to type information and designer parameters.

microarchitecture)—then traverses the object graph to *mark* all objects reachable from roots. When marking is complete, the *mark and sweep* variant of a tracing collector does a linear scan to reclaim the storage of all unmarked objects.

By contrast, reference counting maintains a per-object count of references, recursively frees an object when its count reaches zero, and usually requires more bookkeeping.

Collectors often pause the application to avoid the confusion of a changing reference graph, a *stop-the-world* approach. Unfortunately, this often produces unacceptable pauses. *Concurrent collectors* avoid such pauses by allowing the application to continue during collection, but are challenging to implement efficiently because of the careful synchronization required between the application and collector.

III. Related Work

Bacon et al. [4], [10] present one of the few garbage collectors that, like ours, is hardware and coexists with an accelerator on an FPGA. Their collector is also concurrent and collects multiple inter-connected heaps. They observed zero mutator stall cycles with heaps between 1.1 and 1.7 times the minimum necessary heap size, but did so on memory traces from a Java application; it is less clear what their performance overhead would be for an accelerator. Our garbage collector is integrated with a Haskell-to-Hardware compiler and we have measured its overhead on six accelerators being virtually zero.

Termination detection is a key challenge of multi-heap garbage collectors. Bacon et al. detect termination by waiting for the maximum cycles required for a pointer to travel between heaps; we support undetermined delays between heaps. Furthermore, Bacon et al. connect every pair of heaps; we only connect heaps that reference each other. Together, we believe these differences make our approach more scalable.

Hardware accelerators for software garbage collection are less like our work. Maas et al. [3] propose a tracing garbage collection accelerator that employs small CPU modifications. Their accelerator achieves high memory bandwidth by using a bidirectional object layout to identify reference fields and decoupling marking and tracing. Unlike our collector, software collects roots and processes state such as the page-table base register and provides them to the accelerator. Jang et al. [5] accelerate key primitives in the ParallelScavenge GC algorithm on the near-memory logic layer of a 3D stacked DRAM. Tang et al. [12] accelerate three common GC functions that consume nearly 50% of collection time and add CPU instructions.

Researchers have proposed alternative hardware assists for GC. The hardware reference counting of Joao et al. [6] stores the reference count in object headers and caches updates to reduce memory accesses. Srisa-an et al. [13] store a limited reference count for each object in their Active Memory Processor. These works identify and reclaim some dead objects in hardware, reducing the frequency of software collections. GC co-processors move collection off of the main core. The Integrated Hardware Garbage Collector of Garcia et al. [14] is closely coupled with the processor to support modern languages in embedded systems. Meyer [15] develops an on-chip GC co-processor paired with a special purpose processor for embedded systems. Schmidt and Nilsen's Garbage Collected Memory module [16] is a near-memory co-processor accessed over the memory bus. In a CPU, hardware read barriers reduce the synchronization cost of concurrent collectors. Azul Systems' Click et al. [17] build a custom system for concurrent garbage collection including a read barrier instruction. Meyer [18] implements both a hardware read barrier and handler directly in the pipeline of his GC co-processor.

IV. Garbage Collector Design

Our garbage collector works in tandem with a hardware accelerator capable of taking atomic snapshots of the roots. We describe these structures below.

A. Accelerator Synthesis Flow

Figure 1 illustrates how we synthesize our accelerators with garbage collection. We extended the Haskell-to-Hardware compiler of Townsend et al. [7], [8], which targets algorithms with complicated control and irregular memory access patterns that rely on an immutable, garbage-collected heap.

The compiler synthesizes accelerators from Haskell programs by transforming them into a dataflow network, optimizing the network, and then synthesizing the network into SystemVerilog. The compiler dismantles recursion and polymorphism [7] and adds buffers to the dataflow network to break what would become combinational cycles [8].

Dataflow nodes communicate data tokens that encode integers and Haskell's algebraic data types (tagged unions). The hardware employs latency-insensitive channels with back pressure [19]. Bit vectors hold data that typically includes a tag to indicate how to interpret the fields in the rest of the vector, typically consisting of integers and references to other objects encoded as pointers—per-type word addresses. Channels are strongly typed in that the compiler knows the possible interpretations of each vector, including any pointers.

Like Haskell's, our heap is immutable: an object's value is set only when the object is created. While the maximum live object count is the same for mutable and immutable memory, the immutable model results in more garbage because updating

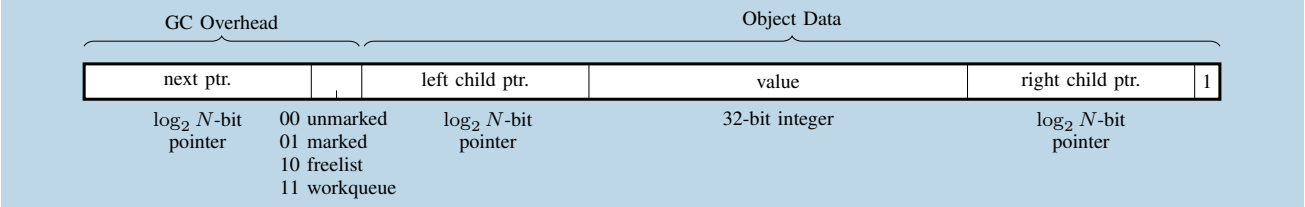

Fig. 2. Layout of a binary tree node in memory. The application sees the Object Data bits: a value field, two child pointers, and a flag indicating the object is a branch or leaf. GC Overhead bits are a pointer field used to indicate the next object in the work or free list and two object status bits.

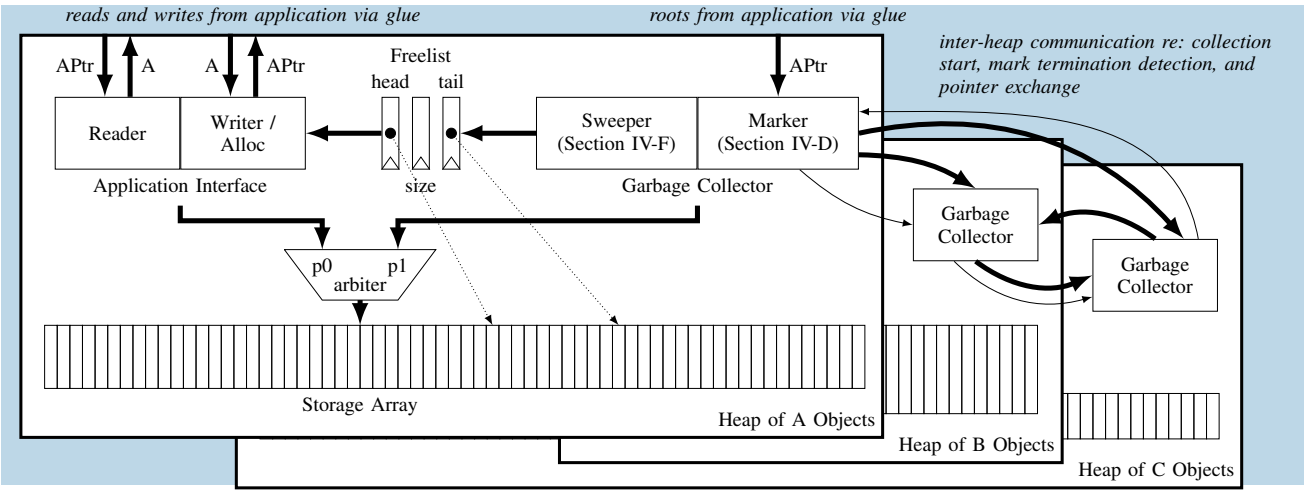

Fig. 3. Parallel heap architecture. Each type in the application has a dedicated heap that operates independently from the other heaps *except* during garbage collection, when the heaps share pointers discovered during mark and communicate to detect mark completion.

an object creates a new copy, keeping the live object count the same, but generating garbage out of the old copy. The dataflow network interacts with the heap through two primitives: *write* operates like C++'s *new* operator: it takes the data for a particular type of object (typically tag and field), allocates and stores it in memory, and returns a pointer to the object, whose data may be later retrieved by passing the pointer to *read*.

B. Heap Architecture

Our toolchain synthesizes an application-specific memory architecture in which each type has its own heap. The bit-width of each heap is set by the bit-width of the objects it stores and garbage collection metadata; the number of objects per heap is set by the designer. Figure 2 depicts the encoding of a binary tree node. The encoding, and hence the number of bits used for each object's fields, is automatically synthesized and is type- and application-specific.

Figure 3 shows the heaps we synthesize for an application that manipulates A's, B's, and C's. Each heap has a object storage array that is accessed by the application through the heap's read/write interface and implemented with pipelined FPGA BRAM with logic designed to fill the pipe and tolerate its latency. In each cycle, an arbiter that favors the application grants access to either the application or the collector. Thus, while there is free memory, the application operates uninterrupted and the collector works in the background.

Objects that reference each other require special consideration in our multi-heap setting. Our static typing policy ensures we know at compile time which objects could potentially reference objects in other heaps. Consider an application with three types A, B, and C, where A objects contain B and C pointers. During garbage collection, when the marking process encounters an object of type A, marking will have to then mark the B and C objects it refers to, which are in separate heaps. Our heaps include such communication channels.

C. Root Collection

The parallelism available in hardware allows us to take an atomic root snapshot in a single cycle. Every buffer that holds a type that may contain a "pointer." may contain a root. We augment each such buffer with an additional "root register" that can be compelled by a global "snap" signal to capture all pointers, if any, held in the buffer. The root registers of each type connect into a shift register leading to that type's heap.

When the free list in any heap dwindles below a threshold, collection is triggered and back pressure on the request channels blocks new memory operations. Once pending memory requests have completed, the root registers are loaded, and the hold on new memory operations is lifted. The application can continue normal execution and marking begins.

D. Marking

Marking begins by appending roots to a work list linked via the "next ptr" field (see Figure 2). Each object's status field indicates whether it is marked, is in the work queue, or is free. Marking then traverses the object graph breadth-first by popping an element from the work list then appending each of its unmarked children. The only departure from a standard mark algorithm is that pointers can come from roots, be locally discovered, or be discovered while marking another heap.

To avoid deadlock, each marker adds discovered references to the appropriate work list before popping the next object. Thus, as long as each marker has sufficient internal buffering for the maximum number of discoverable references per object, the marker can always drain them and make progress.

To maintain correctness during concurrent operation, objects allocated during marking are presumed live and allocated *marked*. This is a conservative assumption that ensures no live objects are reclaimed. At worst, an object may survive one additional collection. Marking is complete when all mark engines have run out of objects to mark.

E. Mark Termination Detection

Termination detection is challenging because while a marker may appear to be done, a remotely-discovered pointer may restart it. Termination occurs only once the last marker has completed and there are no inter-heap pointers in flight. We adapted our algorithm from Chandy-Lamport [20]: we connect the heaps in a simple ring around which they send "done check" messages. A cycle counter in each heap records the time when it "runs out" of work. At that point, a heap initiates a check by sending a positive done check message to the next heap which contains the initiating heap's ID and the time stamp of local completion. When a heap receives a done check, if it is done marking and finished before the heap initiating the check, it propagates the positive done check to the next heap. If instead it still working, or ran out of pointers later than the initiating heap, it propagates a negative done check message. The cycle completes when the initiating heap receives a done check message with its own ID. If the message is still positive, then all types have finished and the check initiator was the last to finish, indicating the mark phase is complete.

F. Sweeping

To reclaim unmarked garbage, each heap performs a linear sweep of its address space. Unmarked objects are added to the free list and marked objects are unmarked. We allow allocations to proceed during sweep by either allocating an object marked or determining that it will not be swept. During the sweep, each heap maintains a pointer S to the current position of the linear sweep. At all times, objects in memory below S have already been swept and are allocated normally; objects above S have not been swept and are therefore allocated marked guaranteeing the object survives sweep. If the sweeper and allocator collide, allocation will occur before sweeping, so the object is allocated marked.

TABLE I
BENCHMARK APPLICATIONS

Application	Size (LoC)	Heaps	Rate (Obj/Cyc)
Bandwidth	176*	5	varies
MergeSort	48	6	0.073
TreeSort	49	7	0.073
Reflect	49	6	0.085
RedBlack	124	6	0.079
Map_RedBlack	129	6	0.075
TSP	100	8	0.001
Clustering	382	19	0.071

*Lines of SystemVerilog code; all others are lines of Haskell code

V. EVALUATION

Our evaluation quantifies the performance overhead of our collector by comparing its performance to an (unrealizable) ideal of execution without garbage collection.

We find our run-in-the-background collector imposes less than a 1% performance overhead when the heaps are "big enough" (often 2.5× the working set size), the collector is set to start collecting before the heap fills up, and the application does not fully use the available memory bandwidth. Our evaluation aims to characterize the design space of systems that are provisioned just enough to achieve such low overhead.

We measure performance as the total number of execution cycles, which we gathered from the cycle-accurate simulator Verilator. Our experiments indicate that the clock frequency does not change for all implementations of the same application. Application slowdown is the percentage of additional cycles taken beyond that necessary for the ideal collector (simulated by making the heaps so large as to avoid ever needing collection); a 100% slowdown means the application running with our collector took twice as many cycles as the application would with an ideal collector.

A. Synthesis Results

Our accelerators and synthesized garbage collectors are heavily pipelined to maintain high clock frequencies regardless of size, which our experiments confirm. To evaluate operating frequency and area, we synthesized our collectors in Xilinx Vivado targeting a Zynq Ultrascale XCZU7CG-FBVB900-1LV-I FPGA with 230400 LUTs and 11Mb of BRAM.

Area: We synthesized the memory and garbage collector without an application, implemented a single heap containing linked-list type objects, and swept its capacity from 2^8 to 2^{17} objects. This increased LUT and register needs only modestly since only the size of the pointer objects and memory multiplexing logic needed to change. The largest collector required fewer than 1200 LUTs.

Frequency: The clock frequency of the synthesized garbage collector and memory with no application ranged from 270 MHz to 350 MHz across all the heaps we considered. The frequency decreased modestly for larger heaps, likely due to BRAM wiring. In any case, our collector never limited application frequency: we synthesized accelerators with and without GC and found working frequency unchanged.

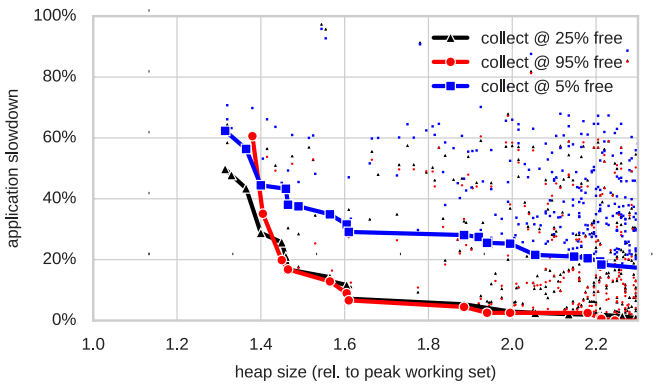

Fig. 4. Garbage collector overhead in cycles on MergeSort as a function of heap size and collection threshold. An eager garbage collection trigger lets it run continuously in the background. Sufficient heap space prevents stalls by providing enough room for the collector to outpace allocations.

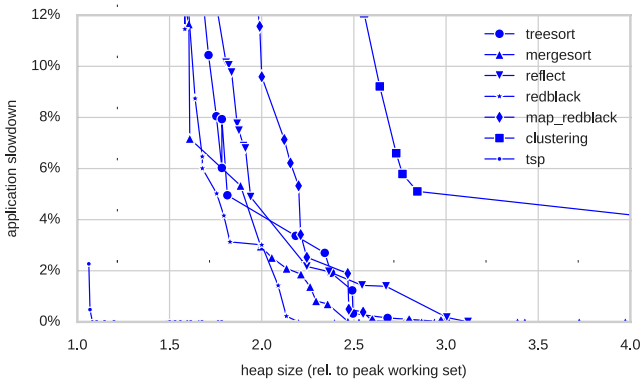

Fig. 5. Garbage collector overhead in cycles as a function of heap size. Each application has a heap size where the collector can run fully in the background.

B. Applications

We ran experiments on the eight applications listed in Table I. Haskell naturally supports applications with irregular memory access patterns and recursion which our benchmarks exploit. We synthesized most applications from Haskell programs using the toolchain depicted in Figure 1; we manually coded *Bandwidth* in SystemVerilog.

Bandwidth is a micro-benchmark that generates high levels of memory traffic. It reads and writes from memory at specified rates while building and destroying list data structures.

MergeSort recursively sorts an integer list by separating it into even- and odd-indexed elements, sorting those, and then merging. *TreeSort* sorts a list by building and flattening a binary search tree. *Reflect* builds a binary search tree then flips the left and right children of each node. *RedBlack* and *Map_RedBlack* build balanced red-black trees then RedBlack swaps the left and right children of each tree node; Map_RedBlack rewrites the tree ten times, adding a constant to each tree node each pass. *TSP* implements a nearest-neighbor heuristic for the traveling salesman problem starting from a matrix of towns and the distances between them. This is computationally intensive, but generates little garbage, which demonstrates the unobtrusive nature of our collector on applications with a small memory footprint. *Clustering* implements K-means clustering. Used in a range of machine learning applications, it first constructs a K-d tree from a list of 2D points then performs the clustering algorithm.

C. Performance Overhead vs Heap Size

Our experimental results for *MergeSort* in Figure 4 show the performance overhead of our collector drops below 1% compared to an ideal collector as the heap sizes begin to exceed about twice the size of the maximum working set. We gathered this data with each heap sized randomly between the maximum working set for its type and an over-provisioned maximum. The horizontal axis of Figure 4 shows the heap size in objects compared to the maximum working set of the application, which we found empirically using memory access traces from each application. The heaps on the left side of the figure are just barely large enough to run the application; heap space increases towards the right. Unlike stop-the-world collectors, concurrent collectors require more heap space than the working set to accommodate allocations during collection.

On the far left of Figure 4, the garbage collection overhead is as high as 40%, which corresponds to the case where the heap is just barely large enough to accommodate the application. Here, the heap requires frequent garbage collection and the application is using memory at a rate faster than the collector recycles it. As heap capacity increases, the collection overhead drops as the application uses memory at a rate equal to or lower than the collection rate, indicating that the collector is able to scavenge unused memory bandwidth and still recycle garbage fast enough that the application can always allocate.

The overhead we report for each heap capacity value (horizontal position) represents the minimum observed overhead for that particular heap size—a Pareto frontier. An open problem in our technique is how best to allocate space among multiple heaps. We assume that an accelerator will be given a heap memory area budget, but that it will fall to the designer to allot it among the heaps, depending on the desired trade off between memory overhead and performance. Heap allocation is important because an under-provisioned heap can become a bottleneck by forcing more collections than if it had more capacity. Absent an effective algorithm for finding the best relative allocations, we show the best configurations from a randomized design space exploration.

D. Performance Overhead vs Collection Threshold

Figure 4 also shows it is best for our collector to run continuously in the background rather than to minimize the number of collection events. We find that our collector slows the application less when it collects eagerly. Collection is triggered when the number of free objects in any heap drops the *collection threshold*. At a threshold of 5%, the collector waits until the heap is nearly full, whereas a 95% threshold leads to almost constant collection.

This result further demonstrates our collector's "run-in-the-background" character. This may seem counter-intuitive since

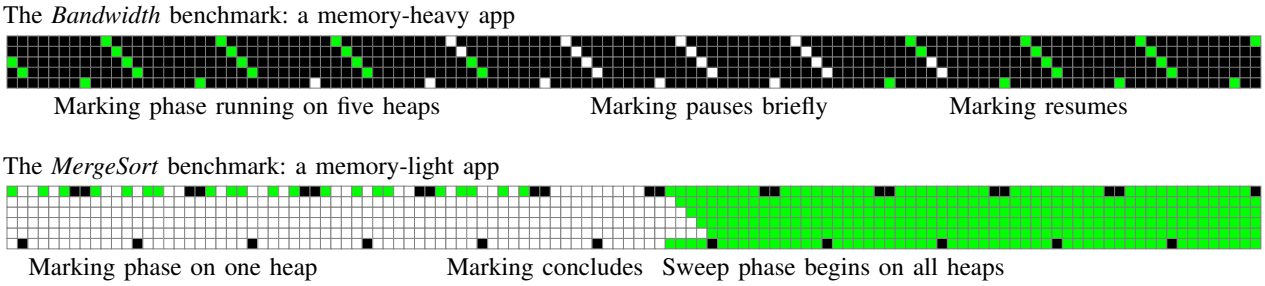

Fig. 6. Memory accesses over time. (top) When the app memory utilization is high, GC proceeds more slowly, but it is still able to use gaps in application memory accesses to make progress. (bottom) When there are more memory slots available the collector uses however many it needs, from just a few to all.

a higher threshold triggers more collections, but overhead from the collector arises when the application has to wait for the collector to free additional objects. When garbage collection is eager, there is enough free memory for the application to continue making progress while the garbage collector works in the background. Triggering collection later, the application is more likely to consume all of the available heap space before the collector has reclaimed dead objects.

Our experiments show that above a certain size, the heaps are large enough for our concurrent collector to achieve idyllic behavior: collection is triggered early and often enough to avoid application pauses due to an out-of-memory situation and there is sufficient unused memory bandwidth to accommodate this larger-than-necessary number of collections. The overhead never drops to exactly zero in these experiments because there is a small cost for the memory barrier before the roots are captured, but this cost is negligible.

Figure 5 shows the overhead of our collector for various heap sizes of all the applications. Here, we selected a 25% collection threshold to balance eagerness and excessive collection. For all applications except *TSP* and *Clustering* our collector quickly drops to having negligible overhead.

Clustering is our largest and most complicated application with the heap divided among 19 data types. Its collection overhead does eventually become negligible, but not until 6.8× the maximum working set. The garbage collection overhead on *TSP* is near zero, even at the minimum viable heap size, because *TSP* is not memory-intensive due to the distance matrix, which remains live for the duration of the application dominating the working set. With little generated garbage, the application never has to pause for the collector to catch up.

E. Memory Bandwidth Overhead

To verify that the collector is filling empty memory slots, we collected the cycle-by-cycle memory usage traces shown in Figure 6. Time runs from left to right, with each row corresponding to a heap. White squares are unused memory slots, black are the application, and green are the collector.

The top trace is our Bandwidth micro-benchmark, which makes heavy use of memory. The collector is near the beginning of the mark phase, it pauses briefly in the middle of this trace, possibly because of a "bubble" in the arriving roots, then resumes marking. The highly periodic pattern of the application's memory accesses is not interrupted during the bubble, confirming that our collector is able to share memory bandwidth with the application without interfering.

The bottom trace depicts our MergeSort benchmark at a point where the application is using just two of the heaps and the collector is transitioning from marking to sweeping. The start of the trace (left) shows the mark phase finishing on the top heap while the other heaps have completed marking and are waiting to start sweeping. At about the halfway point, all heaps begin sweeping, which saturates the unused memory bandwidth. As with the memory-heavy benchmark, this change does not interfere with the application's progress which continue their periodic behavior.

The previous experiment shows the garbage collector is able to fill idle memory slots regardless of the application memory utilization, but that application memory utilization effects the collection overhead as the garbage collector can only operate in the background if there are idle memory slots.

VI. Conclusion

We presented a synthesized concurrent hardware garbage collector for FPGA-based accelerators. Our collector exploits rapid synchronization to fill otherwise idle memory slots on multiple, parallel memory heaps. We find that an eager collector that runs early and often paired with reasonably provisioned heaps allows the collector to run almost completely in the background. This brings the benefits of automated memory management to HLS, further reducing development time.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their time and feedback. This work was supported by NSF Award Number 1162124 and Google and Qualcomm Faculty Awards.

REFERENCES

[1] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. S. Iv, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. suk Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D. He, C. R. Ho, R. W. Huffman, E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V. M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. Wu, "Warehouse-scale video acceleration: co-design and deployment in the wild," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 600–615.

[2] 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 *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 225–236.

[3] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 138–151.

[4] D. F. Bacon, P. Cheng, and S. Shukla, "And then there were none: A stall-free real-time garbage collector for reconfigurable hardware," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 23–34, 2012.

[5] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, "Charon: Specialized near-memory processing architecture for clearing dead objects in memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 726–739.

[6] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the 36th annual international symposium on computer architecture*, 2009, pp. 418–428.

[7] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 7686. [Online]. Available: https://doi.org/10.1145/3033019.3033027

[8] S. A. Edwards, R. Townsend, M. Barker, and M. A. Kim, "Compositional dataflow circuits," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3274280

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 392–403.

[10] D. F. Bacon, P. Cheng, and S. Shukla, "Parallel real-time garbage collection of multiple heaps in reconfigurable hardware," *ACM SIGPLAN Notices*, vol. 49, no. 11, pp. 117–127, 2014.

[11] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[12] J. Tang, S. Liu, Z. Gu, X.-F. Li, and J.-L. Gaudiot, "Achieving middleware execution efficiency: hardware-assisted garbage collection operations," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1101–1119, 2012.

[13] W. Srisa-an, C.-T. Lo, and J.-M. Chang, "Active memory processor: A hardware garbage collector for real-time java embedded devices," *IEEE Transactions on Mobile Computing*, vol. 2, no. 2, pp. 89–101, 2003.

[14] A. A. García, D. May, and E. Nutting, "Integrated hardware garbage collection," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5, jul 2021. [Online]. Available: https://doi.org/10.1145/3450147

[15] M. Meyer, "An on-chip garbage collection coprocessor for embedded real-time systems," in *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*. IEEE, 2005, pp. 517–524.

[16] W. J. Schmidt and K. D. Nilsen, "Performance of a hardware-assisted real-time garbage collector," *ACM SIGPLAN Notices*, vol. 29, no. 11, pp. 76–85, 1994.

[17] C. Click, G. Tene, and M. Wolf, "The pauseless gc algorithm," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005, pp. 46–56.

[18] M. Meyer, "A true hardware read barrier," in *Proceedings of the 5th international symposium on Memory management*, 2006, pp. 3–16.

[19] B. Cao, K. A. Ross, M. A. Kim, and S. A. Edwards, "Implementing latency-insensitive dataflow blocks," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 179–187.

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.