Implementing Garbage Collection in Hardware

Yee, Evan. Author, *ECE, Duke University*, Lankford, Evan. Author, *ECE, Duke University*

***Abstract*—Garbage collection (GC) is typically a software-based process in managed languages that is responsible for freeing memory-allocated objects when a program no longer needs them. Even in multi-threaded or multi-core processors, the processor itself is typically in charge of running the GC program. Because of the linear nature of processors, a simple pipelined core might be poorly suited for GC. Accordingly, improving the efficiency of GC for workloads that spend large proportions of CPU cycles on GC can enhance processor performance and energy consumption.**

In this paper, we propose a dedicated hardware-based garbage collection module integrated into a RISC-V RocketChip SoC. By examining the performance and physical characteristics, we compare the trade-offs and potential utility of a dedicated hardware garbage collector for certain workloads. Based on the work of Maas et. al. [1], we designed our own GC unit that implements parallel marking and tracing with a block-based sweeper. We propose a new object layout, utilizing shared knowledge between the allocator and GC hardware. We completed functional verification using Verilator testbenches and synthesized our design using the Cadence Genus Synthesis Tool. Finally, we built a performance model from our microbenchmark results to compare to software-based GC on OpenJDK's JVM running workloads from the DaCapo Java benchmark suite.

We found an average performance improvement of 1.62x over software-based GC with an 11.3% increase in SoC surface area. We demonstrate that memory management is well-suited for parallel hardware acceleration and compare our performance, timing, power, and area to Maas's hardware.

***Keywords*-hardware accelerators; garbage collection; memory management, Java benchmarks;**

I. Introduction & Problem Statement

Today, many modern workloads are written in managed languages rather than compiled languages. One fundamental advantage of managed languages is the service of garbage collection (GC). Garbage collection is a process that allows the automatic freeing of unreachable memory-allocated objects. Typically, GC runs on a pipelined processor in software. However, because of the linear nature of processors, some research has found them to be ill-equipped to efficiently handle these potentially parallelizable memory operations [10]. For example, most objects in memory can be processed independently and could benefit from higher-throughput hardware. Moreover, traversal of object graphs like the heap is a recursive process that can visit each branch in parallel. Accordingly, researchers have investigated the utility of garbage collection hardware accelerators.

Maas et. al. [1] proposed a stop-the-world tracing GC hardware accelerator for RISC-V processors. Stop-the-world GC pauses the processor's application to perform memory management rather than running concurrently with the processor. It is typically triggered when a capacity threshold is reached in the memory heap.

By offloading garbage collection to a dedicated hardware module, Maas demonstrated that there is the potential for large improvements in workload performance. Fully implementing a hardware garbage collector requires modifications to CPU architecture, virtual machine memory management, memory hierarchy, and the actual SoC modules.

We built our GC hardware modules with an emphasis on high throughput while preserving simplicity. We specifically targeted Java workloads that spend large amounts of time in GC. Once GC is triggered in Java, the memory management system passes reference roots and memory blocks to our hardware rather than initiating software-based GC

To improve the efficiency of GC operations, we constructed a stop-the-world mark-and-sweep tracing garbage collector in Chisel alongside the in-order RISC-V Rocket core processor. We utilized the existing RocketCore framework with TileLink and HellaCache protocols for memory and cache accesses. We compiled our implementation into RTL modules and verified its behavior with Verilator testbenches.

Through synthesis and performance evaluation on a subset of the DaCapo Java benchmark suite, we hope to prove the functionality of our hardware accelerator and utility of hardware-based garbage collection.

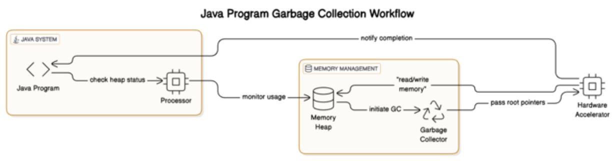


Figure 1: **Basic System Architecture:** Hardware is initiated by heap usage. Once complete, the CPU resumes program.

II. Related Work

There has been a fair amount of work in the hardware garbage collection space. Researchers have quantified the performance improvement of moving GC work into hardware [6]. One major topic that was mentioned in [1] and is the focus of other research is concurrent garbage collection [4]. As opposed to "stop-the-world" collection, concurrent GC runs in parallel to an application without pausing it. The speedups produced by concurrent GC come with complexity trade-offs that were unfeasible for this paper to explore in the given timeframe. For places like data centers, the pause times for stop-the-world GC can be massive and problematic [2-3]. However, stop-the-world GC still provides performance improvements over traditional GC [1]. Using Chisel [9] to build hardware modules was influenced by Maas' use of RocketChip, which implements the Rocket core [5] in Chisel.

III. Research Questions

Our project broadly focuses on the trade-offs between hardware-dedicated and software-based GC. Specifically, our project assesses both the feasibility and utility of developing a dedicated GC unit in hardware. A pipelined processor has hardware limitations that make efficient GC difficult. Accordingly, we hope to quantify the potential impact of a hardware unit on workload performance, power consumption, and area. Additionally, we intend to compare our hardware and design choices to Maas' hardware to investigate potential improvements and the impacts of different object layouts and structures.

We intend to study how various Java workloads might benefit from hardware-accelerated GC. Using the DaCapo benchmark suite, we hope to establish the types of application workloads that might be best suited for our mark-sweep stop-the-world garbage collector.

More broadly, we also hope to offer a solution for how to build a hardware accelerator for garbage collection. Aside from the performance and physical ramifications, studying GC designs will help us propose what the unit will look like strictly in hardware logic.

To briefly summarize, our project investigates how to design hardware-based GC. Moreover, it evaluates performance of hardware-based GC compared to software-based GC and establishes the area and power of such a module. Finally, we examine the use cases for hardware-based GC and explore the ramifications of alternative hardware design choices.

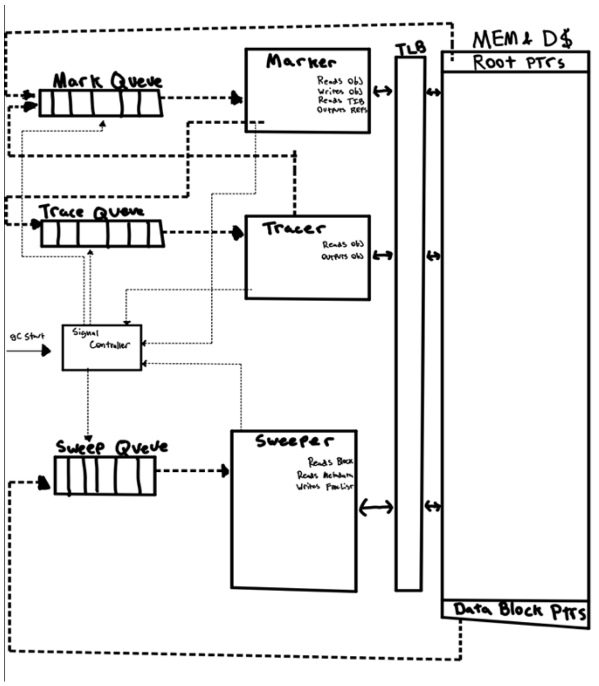


Figure 2: **GC Wrapper Architecture:** a block diagram of the modules and data structures that are wrapped within our unit.

IV. Proposed Solution

A. Overview

We elected to construct our hardware unit through a modular approach, separating the responsibility and functionality of our mark-sweep collector into submodules wrapped in a single controller file. This controller file is responsible for the I/O signals, memory pointer management, and providing a clean top-level module for synthesis. The four modules communicate through various input registers, control signals, and queues that run in parallel via separate FSMs.

This GC unit is built adjacent to the in-order Rocket core on the RocketChip SoC. It does not disrupt the serial processing of the CPU. However, it does interface with the memory and the cache hierarchy directly. The GC utilizes HellaCache and TileLink for memory I/O and requires TLB and PTW modules to handle address translation.

Our hardware unit is initiated by the virtual machine memory management system through a signal that goes high when the memory heap reaches a certain utilization threshold during runtime. We also simplify our hardware complexity through some important assumptions about memory. Our first assumption is that the root object pointers for a program are easily accessible and can be passed to our GC unit with minimal computation. We also assume a traditional object header layout that includes a TIB pointer as well as three bits for information about marking, free list status, and validity. The second half of the header is reserved for a pointer to the next element in a free list. For a 40-byte object, this totals to roughly a 29% minimum memory overhead for object memory, with even more being used for our TIB data and our block metadata. Finally, we assume that the memory allocator accurately updates the necessary metadata regarding each memory block such as cell size, block size, and free list head. We assume this metadata is listed at the start of each memory block.

B. Hardware Modules

1. GC Wrapper

This module acts as the central controller for our hardware GC. It coordinates and instantiates the Marker, Tracer, and Sweeper modules, managing initiation, execution, and data movement. It operates through three different FSMs, one of which starts by loading root object addresses from a dedicated place in memory into a mark queue. The inputs into the wrapper are a start signal, the address of the reference roots in memory, and the address of the memory block pointers. The wrapper notifies the memory management system when GC has finished, which occurs when all the pointer queues are empty and the sweeper is idle.

Our wrapper allows the Marker, Tracer, and Sweeper to process memory independently, supporting the parallelization of GC. The wrapper pops addresses from our Mark Queue for the Marker and adds relevant outbound references from Marker to our Trace Queue. Similarly, the wrapper is responsible for popping addresses from the Trace Queue for the Tracer and passing relevant addresses to the mark queue. The wrapper is also responsible for initiating the sweeper and passing it the necessary block pointers that it loaded into the Sweep Queue. Our Mark Queue, Trace Queue, and Sweep Queue are each 1000 entries in size.

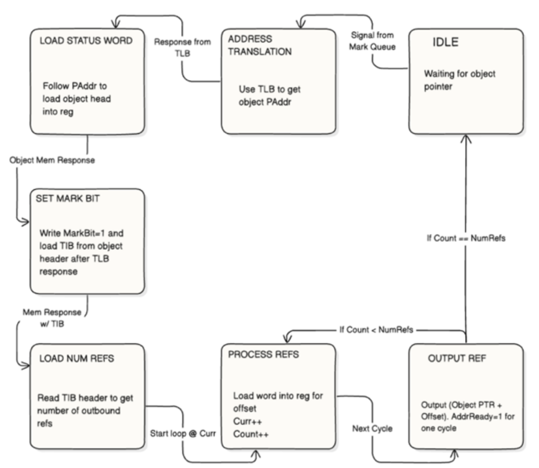

LOAD STATUS WORD
Follow PAddr to load object head into reg
Response from TLB
ADDRESS TRANSLATION
Use TLB to get object PAddr
Signal from Mark Queue
IDLE
Waiting for object pointer
Object Mem Response
SET MARK BIT
Write MarkBit=1 and load TIB from object header after TLB response
If Count == NumRefs
Mem Response w/ TIB
If Count < NumRefs
LOAD NUM REFS
Read TIB header to get number of outbound refs
PROCESS REFS
Load word into reg for offset
Curr++
Count++
OUTPUT REF
Output (Object PTR + Offset). AddrReady=1 for one cycle
Start loop @ Curr
Next Cycle

Figure 3: **Marker State Machines:** The logic for Marker to visit active objects.

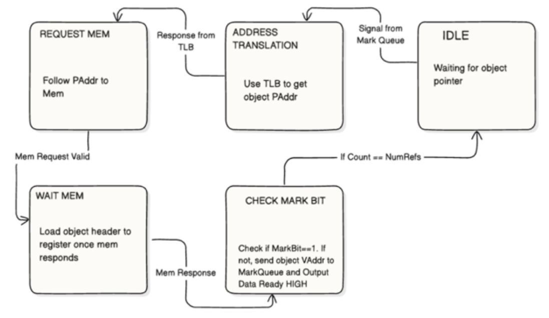

REQUEST MEM
Follow PAddr to Mem
Response from TLB
ADDRESS TRANSLATION
Use TLB to get object PAddr
Signal from Mark Queue
IDLE
Waiting for object pointer
Mem Request Valid
If Count == NumRefs
WAIT MEM
Load object header to register once mem responds
Mem Response
CHECK MARK BIT
Check if MarkBit==1. If not, send object VAddr to MarkQueue and Output Data Ready HIGH

Figure 4: **Tracer State Machines:** The logic for Tracer to visit potentially cyclic references.

2. Marker

This module performs the actual marking for GC. It takes in a start flag from the wrapper as well as a virtual address of an object to visit. We designed our Marker to translate this address using the TLB module and interface. The Marker then loads the word at the physical address of the object in question. This word is the first half of the object header. Following the "conventional" object layout [1], we read the first 32 bits from this register value as a virtual address of the Type Information Block (TIB) which contains the relevant outbound reference metadata for the object. Moreover, the marker sets the LSB of this register to 1 (the Mark Bit) and saves the word back into memory. This design of the Marker differs from Maas [1], who elects for a "bidirectional" object layout and added complexity for performance improvements.

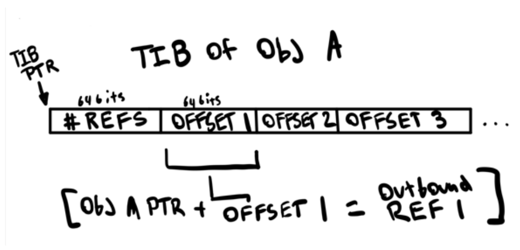


Figure 5: **TIB Layout:** The number of outbound references is followed by the offset for each reference

With the TIB pointer for the object in question, the Marker also translates this address using the TLB. We then load the word at the corresponding physical address into a register. This 64-bit value tells us how many outbound references this object contains. We then iterate through each word adjacent to this physical address, which provides the offset value of each outbound reference. The Marker adds these offset values to the original virtual address of the object in question and outputs these for the Tracer to visit, along with a data-ready bit.

More broadly, the Marker is responsible for indicating to the Sweeper if an object in memory is still reachable by the program. If so, the Mark Bit will be high and this cell in memory will not be freed. Marker traverses the object tree starting at the root nodes until it has marked every outbound reference and sends nodes to Tracer to ensure it does not conduct duplicate visits to the same object.

3. Tracer

The Tracer module traverses all the outbound references from an object that are popped from the Mark Queue. These outbound addresses are provided from the Trace Queue, which intakes addresses that were found from an object's TIB (handled in Marker). The purpose of the Tracer is to ensure cyclic references are not re-visited by the Marker. As inputs, Tracer needs a virtual address from the Trace Queue as well as a start signal from the wrapper. The tracer translates the virtual address into a physical address via the TLB. It then loads the word (first half of the 128-bit header) from this address into a register and assesses if the LSB (Mark Bit) is 1. If so, this object has already been visited by the Marker and does not need to be marked again (a cyclical reference graph likely exists somewhere in the program). However, if the LSB is 0, the virtual address is outputted by the Tracer along with a data ready bit so the address can be added to the Mark Queue.

The actual implementation of the Tracer operates through an FSM where we are either waiting for a new address, translating addresses, loading words, and conditionally sending data out.

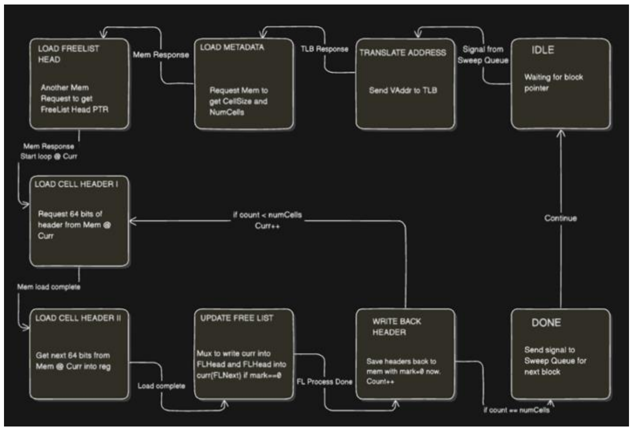

LOAD FREELIST HEAD
Another Mem Request to get FreeList Head PTR
Mem Response
LOAD METADATA
Request Mem to get CellSize and NumCells
TLB Response
TRANSLATE ADDRESS
Send VAddr to TLB
Signal from Sweep Queue
IDLE
Waiting for block pointer
Mem Response Start loop @ Curr
LOAD CELL HEADER I
Request 64 bits of header from Mem @ Curr
if count < numCells Curr++
Continue
Mem load complete
LOAD CELL HEADER II
Get next 64 bits from Mem @ Curr into reg
Load complete
UPDATE FREE LIST
Mux to write curr into FLHead and FLHead into curr(FLNext) if mark==0
FL Process Done
WRITE BACK HEADER
Save headers back to mem with mark=0 now. Count++
if count == numCells
DONE
Send signal to Sweep Queue for next block

Figure 6: **Sweeper State Machine:** Sweeper reads block metadata then iterates through the cells.

4. Sweeper

This module is responsible for adding memory cells to the Free List based on the previous operations by the Marker and

Tracer. The module intakes a virtual address that points to a memory block, as well as a start signal by the wrapper. This sweeper required some important design choices / assumptions about memory to implement. Physical memory is divided into blocks, each of which contains several cells. Each cell corresponds to one object. The actual assignment of an object to a cell is the responsibility of the memory allocator. We require our allocator to provide some key pieces of metadata about our memory blocks. The allocator is responsible for determining the size of each block, the number of cells in each block, and the size of any cell in that block. Also, it is responsible for providing a physical pointer to each of these blocks to our wrapper, which will subsequently pass these pointers to our Sweeper in the form of a physical address.

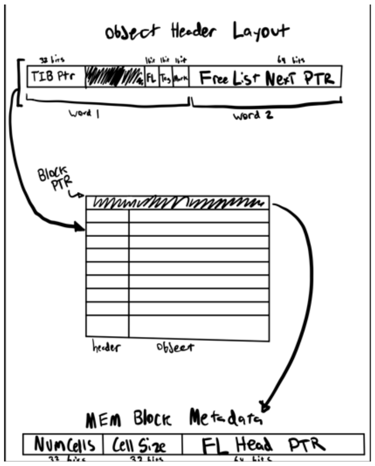


Figure 5: **Object and Block Metadata Layout:** 2 words for object header and block metadata, which starts every block.

Additionally, we assume that the first 128 bits of each block contain this metadata, along with the head pointer of the Free List for the given block. The first 128 bits of each cell contain the object header for the object that resides in this cell. The first 32 bits of the header are the TIB pointer as previously discussed. Bit 64 is the Mark Bit. Bit 65 is the Tag Bit, which tells our Sweeper if this is a live object. Bit 66 is the FL Bit, which tells us if this cell already resides within the Free List for a given block. Bits 0-63 are the FLNext pointer, which is a pointer to the next element in the linked-list-based Free List.

From our memory and block design, our Sweeper can operate in a simplified hardware module. The Sweeper takes a start signal from the wrapper along with a block virtual address pointer. The Sweeper loads the first two words from the corresponding physical address into two separate registers that provide us with the block metadata. The Sweeper then uses the block size, cell size, and number of cells to iterate through the block. On each iteration, the sweeper will check the FL Bit, Tag Bit, and Mark Bit for a '010' combination. This indicates that there is a live object here, it's not currently in the free list, and is no longer reachable by the program. The Sweeper performs these checks via bit masks and basic muxes. Upon identifying a '010' combination, the sweeper saves the current Free List Head Pointer in the FLNext field for the cell in question and sets the cell pointer as the new Free List Head Pointer.

The Sweeper also sets the mark bit of every cell header it visits to zero to reset for subsequent GC iterations. After iterating through the entire block, the Sweeper indicates to the wrapper that it is ready for a new block pointer and repeats the process until the Sweep Queue is empty. An important advantage of the Sweeper's design is that we can have multiple Sweepers running at once, operating on separate blocks in memory to free cells in a highly parallel manner. The Sweeper also operates through an FSM that is controlled by the input signals and number of cells.

V. Evaluation

A. Methodology

After implementing each of the previously mentioned hardware modules in Chisel on top of the Rocket core, we began evaluation of their functionality and performance. We started with functional validation of our unit and submodules using microbenchmarks. Then, we conducted synthesis to determine the physical characteristics of our hardware. Next, we used our cycle-accurate simulation results from functional validation to create a performance model. Finally, we performed benchmarking with Java workloads to compare our hardware's performance with a software-based tracing collector.

B. Functional Validation

For functional validation, we wrote a driver to convert our Chisel HDL modules to Verilog RTL. After producing our Verilog modules, we utilized Verilator to create cycle-accurate C++ behavioral models of each. We wrote C++ microbenchmarks for both debugging and verification of our submodules' functionality. This involved designing input signals and simulating memory and TLB responses.

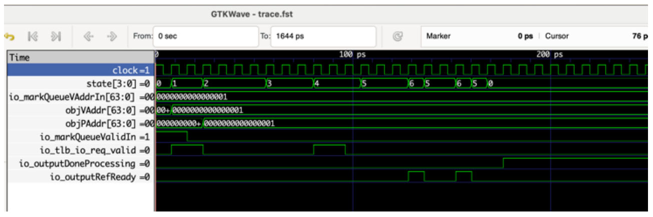


Figure 7: **GTKWave Trace**: Image of an example GTKWave output from a Marker testbench to verify outputs and intermediate signals as well as cycle counts.

Careful examination of GTK wave traces allowed us to verify the behavior of each submodule. Additionally, these traces provided us with cycle counts on a per-object or per-reference basis for GCWrapper, Marker, Tracer, and Sweeper. We were able to infer this data from microbenchmark clock and output signals.

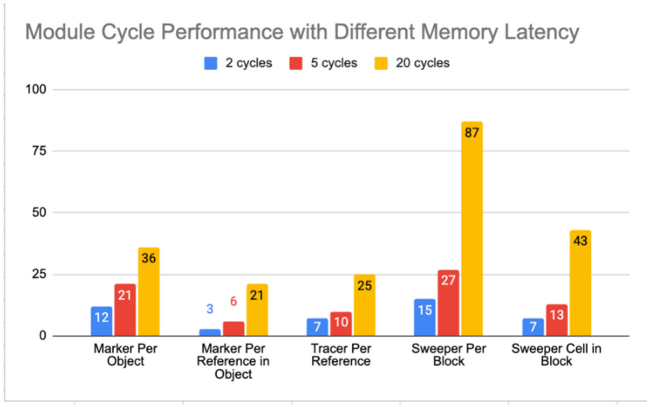


Figure 8: **Cycle-Accurate Simulations**: Cycle counts for each submodule on a per object, reference, and block basis.

We designed our microbenchmarks to give us cycle predictions on a per object, per reference, per block, and per cell basis. These numerical results form the foundation of our performance model that will be discussed in Section V-D. Importantly, we provide cycle counts for our submodules for varying degrees of memory latency depending on prefetching accuracy. Evidently, the sweeper module is most impacted by higher memory latency. The most important independent variables for GC performance are the heap size and the number of objects and references in program memory.

C. Physical Characteristics

To conduct area, power, and timing analysis of our RTL design, we synthesized using the Cadence Genus Synthesis Tool. We did this using TSMC PDKs for a 65 nm technology node. Our design passed static timing analysis (STA) with a clock frequency of 500 MHz.

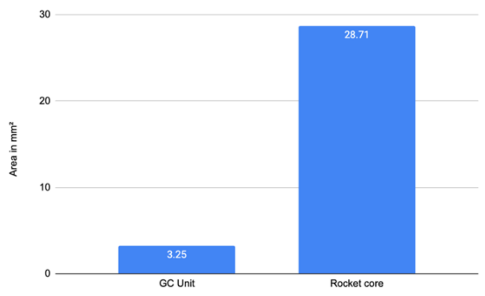


Figure 9: **Area Comparison**: Total area of GC unit in comparison to Rocket core. The Rocket core was synthesized with SAED PDKs for a 28nm technology node, so the area was converted to its 65nm technology equivalent [11].

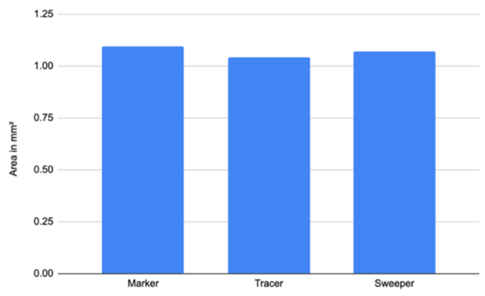


Figure 10: **Area Breakdown**: Area of each submodule of our GC unit.

As seen in *Figure 9*, our GC unit had an overall area of 3.25mm². This corresponds to ~11% of the Rocket core total area, supporting the feasibility of integrating this unit into a RocketChip SoC. *Figure 10* conveys that the Marker, Tracer, and Sweeper submodules had similar areas around 1 mm², most of which is reserved for their respective queues. *Table 1* shows a physical comparison between our hardware unit to Maas's [1]. Our unit was smaller, but both units were less than 20% of the area of the Rocket core. The power for our hardware unit was 359 mW which was less than the Maas unit [1] and the overall power for the Rocket core.

Table 1: Area Comparison

	Rocket core	Maas Unit [1]	Our GC Unit
Technology (nm)	28	28	65
Factor [11]	3.3x	3.3x	1x
Area (mm²)	8.7	1.8	3.25
Converted Area (mm²)	28.71	5.94	3.25

D. Performance Evaluation

We selected workloads from the DaCapo Java benchmark suite to evaluate our hardware unit against software-based GC. Due to the extremely invasive nature of GC operations, full integration testing and live evaluation of a hardware garbage collector requires modification of a virtual machine's memory management system. For our research, end-to-end evaluation also requires enforcing cache coherence protocols and full integration into the RocketChip SoC to run the Java workloads. These complex steps were outside the scope of our project. Therefore, we took an alternative approach to evaluation of our hardware unit. Using our cycle-accurate functional validation results from Section V-B, we were able to develop a model for time spent per GC operation on a given application. The following equations were devised for cycle and time calculations:

$$Mark\ Cycles\ (MC) = B\left(Max((M1 * X + M2 * Y), (T * Y))\right)$$

B: Overlap factor, M1: marker object cycles, M2: marker reference cycles, T: tracer cycles, X: total objects, Y: total references

$$Sweep\ Cycles\ (SC) = \frac{(S1 * Z + S2 * U)}{N}$$

S1: sweeper block cycles, S2: sweeper cell cycles, Z: blocks, U: total cells, N: number of sweeper units

$$GC\ Time = \frac{(MC + SC)}{f}$$

f: frequency

We benchmarked total and average time spent in GC for three different workloads from the DaCapo suite. These workloads were selected due to their high GC time. We ran the workloads using OpenJDK's Java VM with the MarkSweepGC collector enabled. To evaluate these workloads with our performance model, we tracked the average number of objects, object size, and object references in the 1GB heap across all GC operations. Using the clock frequency verified with STA in Section V-C, we were able to obtain performance results for our hardware unit on these workloads.

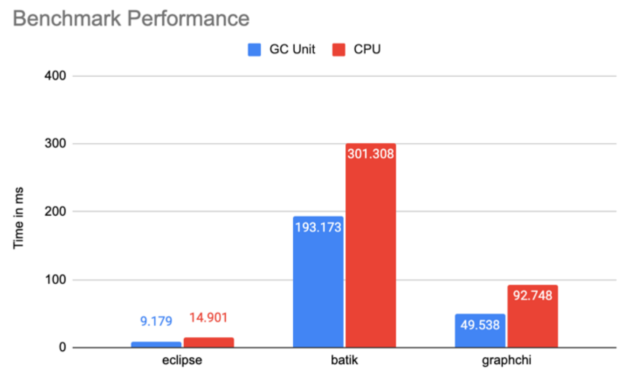


Figure 11: **Benchmark Performance**: Average time per GC operation (mark-sweep) across DaCapo workloads using 1GB heap.

In *Figure 11*, we can see the average time per GC operation for our hardware unit compared to the software-based GC. We find an average performance speedup factor of 1.68x across the three workloads, demonstrating a clear performance gain. The *batik* workload was especially GC-intensive, as it spent 53% of total application runtime in GC. With a performance speedup factor of 1.55x, our GC unit provided a 23.2% overall application speedup for *batik*. The other two applications spent significantly lower amounts of time in GC but still saw application speedups with our GC unit.

VI. Discussion

With an average speedup factor of 1.68x for the three DaCapo benchmarks discussed in Section V-D, the performance utility of our GC unit is immediately evident. As expected, the performance benefit of a hardware-based garbage collection is more pronounced for the software programs that have a larger GC overhead. Although the Maas hardware unit was tested with different workloads in the DaCapo suite, their hardware achieved roughly a 3.3x speedup. Evidently, our GC hardware is less efficient than the Maas GC. Although both our hardware and Maas's demonstrate the effectiveness of running GC in hardware, it is important to examine the differences between both hardware units.

The total area for our GC hardware is 3.25 mm^2, while Maas's hardware is 5.94 mm^2 (after conversion from 28nm to 65nm technology [11]). Therefore, our performance inefficiency is partially compensated for by a smaller surface area. With our smaller surface area, we have a lower power draw—as expected. We presume that a larger and more complex garbage collector could achieve results like the Maas hardware.

Table 2: Trade-offs

	Maas Unit [1]	Our GC Unit
Technology (nm)	28	65
Area (mm^2)	1.8	3.25
Power (mW)	460	460
Speedup Factor	3.3x	1.68x

We offer some potential explanations for the discrepancies between the two hardware accelerators. Firstly, our hardware does not enforce any cache coherence protocols whereas Maas does. Secondly, our hardware was synthesized using TSMC PDKs while Maas is synthesized with more conservative SAED PDKs. This likely accounts for some of the area difference between the accelerators. Additionally, Maas' GC unit used a larger queue size, which directly impacts area. Third, we assumed a conventional object header layout and TIB structure, while Maas utilized a bidirectional object layout and modified TIB to achieve faster results [1]. Finally, we did not actually implement a PTW and TLB for our modules while Maas did, which likely contributed to our smaller size.

Nevertheless, both units demonstrate the effectiveness of offloading stop-the-world tracing GC to a dedicated hardware accelerator. Adjusting the object layout and mark-sweep logic to reduce memory accesses, as well as adding additional sweeper units for greater parallelization, could increase the performance gain from our hardware unit. Further testing to better optimize our queue size could also improve performance. Finally, examining additional Java workloads to test our hardware with high or low cyclic reference counts could also be indicative of potential limitations in our design.

In terms of practical usage, fully integrating our hardware into the RocketChip SoC to run actual Java workloads with a modified Java VM would be the next step. Another important avenue of exploration is concurrent GC with our hardware design and measuring the added complexity it would introduce.

VII. Conclusion

We implemented a hardware accelerator designed for GC that can be integrated into an SoC at a relatively low hardware cost. It specifically implements tracing, stop-the-world GC for the Java applications running on the CPU. The hardware unit takes up 11% of the area of the Rocket core and provides a speedup of 1.68x. For managed languages, GC can consume large percentages of workload runtimes, causing significant performance slowdowns. These results make a strong case for offloading the GC process onto a dedicated hardware accelerator.

More broadly, this paper supports the ongoing research of application-specific hardware acceleration for parallelizable processes. As managed languages continue to dominate modern software development, the computing industry demands higher throughput and faster memory management. For hardware that prioritizes performance over size and power, such as personal computers or servers, incorporating garbage collectors in an SoC might prove effective.

References

[1] Maas, Martin, Krste Asanović, and John Kubiatowicz. "A Hardware Accelerator for Tracing Garbage Collection." Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18), June 2018, pp. 66–79.
[2] 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.
[3] E. Kaczmarek and L. Yi, "Taming GC Pauses for Humongous Java Heaps in Spark Graph Computing," 2015.
[4] E. Moss, "The Cleanest Garbage Collection," Commun. ACM, vol. 56, no. 12, pp. 100–100, Dec. 2013.
[5] 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.

[6] J. A. Joao, O. Mutlu, and Y. N. Patt, “Flexible reference-counting-based hardware acceleration for garbage collection,” International Symposium on Computer Architecture, Jun. 2009.

[7] N. Artyushov, “Revealing the length of Garbage Collection pauses,” https://plumbr.eu/blog/garbage-collection/revealingthe-length-of-garbage-collection-pauses.

[8] 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.

[9] 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.

[10] 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.

[11] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74-81, 2017.

APPENDIX A

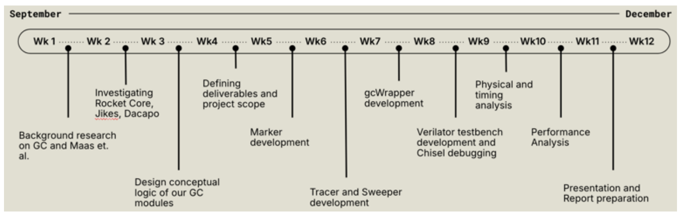
DIVISION OF LABOR

Both Evan Yee and Evan Lankford researched the tools and relevant work for this project. Evan Yee took the lead on building the marker and tracer Chisel modules and Evan Lankford took the lead on building the sweeper Chisel module. Both members collaborated on building the garbage collection wrapper module. Evan Yee focused on building testbenches for each module in Verilator. Evan Lankford focused on synthesis and analyzing physical characteristics. Both members worked on benchmarking performance against software GC. Both members were also responsible for documentation and paper writing.

APPENDIX B

TIMELINE

Our 12-week timeline was split into major milestones that are listed below.


September
December
Wk 1
Wk 2
Wk 3
Wk4
Wk5
Wk6
Wk7
Wk8
Wk9
Wk10
Wk11
Wk12
Background research on GC and Maas et. al.
Investigating Rocket Core, Jikes, Dacapo
Design conceptual logic of our GC modules
Defining deliverables and project scope
Marker development
Tracer and Sweeper development
gcWrapper development
Verilator testbench development and Chisel debugging
Physical and timing analysis
Performance Analysis
Presentation and Report preparation

Milestone 1: Research and relevant work
Milestone 2: Design GC hardware
Milestone 3: Implement GC Hardware in Chisel modules
Milestone 4: Functional Validation with Verilator
Milestone 5: Synthesis and Physical Characteristics
Milestone 6: Performance Evaluation & Benchmarking
Milestone 7: Final Report