Towards Hardware Accelerated Garbage Collection with Near-Memory Processing

Samuel Thomas*, Jiwon Choe*, Ofir Gordon†, Erez Petrank†, Tali Moreshet‡, Maurice Herlihy*, R. Iris Bahar§
*Brown University, †Technion Institute-Israel, ‡Boston University, §Colorado School of Mines

***Abstract*—Garbage collection is widely available in popular programming languages, yet it may incur high performance overheads in applications. Prior works have proposed specialized hardware acceleration implementations to offload garbage collection overheads off the main processor, but these solutions have yet to be implemented in practice. In this paper, we propose using off-the-shelf hardware to accelerate off-the-shelf garbage collection algorithms. Furthermore, our work is *latency* oriented as opposed to other works that focus on bandwidth. We demonstrate that we can get a $2\times$ performance improvement in some workloads and a $2.3\times$ reduction in LLC traffic by integrating generic Near-Memory Processing (NMP) into the built-in Java garbage collector. We will discuss architectural implications of these results and consider directions for future work.**

***Index Terms*—garbage collection, near-memory processing, benchmarking**

I. Introduction

Garbage collection (GC) is an automatic memory management protocol utilized by many high level programming languages [8], including Java, Python, Go, etc. These garbage-collected programming languages are popular among developers because manual memory management is difficult and error-prone, leading to memory bugs that are notoriously hard to overcome.

Unfortunately, garbage collection may utilize 10-30% of application CPU cycles, and the resulting overhead can be even more for memory-intensive edge-case workloads [20]. As such, prior work [3], [16], [20] has looked into hardware-accelerated garbage collection. This is an idea about as old as garbage collection itself [12], [17]. However, while these works provide strong theoretical solutions, none of them have been adapted in practice due to complex hardware that has highly specialized use cases. That is, none of the older proposals have been integrated into commodity or production hardware, for the solutions were not generalizable across various programming languages with different GC algorithms and implementation-level details [21], [22].

Nonetheless, prior work on alleviating GC bottlenecks have provided a fundamental understanding of GC behavior. In particular, the *marking phase* of the *mark-and-sweep* algorithm, which identifies the live objects (*i.e.*, uncollectible memory) in an application, is fundamentally pointer-chasing and thus subject to bottlenecks arising from frequent but irregular memory accesses.

To address this, we turn to *near-memory processing* (NMP). NMP describes a lightweight, simple compute unit being placed at the logic controller of a DRAM vault. In the NMP architecture, having these compute units physically near the memory allows for higher memory bandwidth and lower latency memory access. As such, NMP has recently reemerged as a promising solution to memory bottlenecks in pointer-chasing, data-intensive applications. Many prior works have delved into accelerating pointer-chasing operations through hardware and software redesigns with NMP [11], [13], [19]. They do so by exploiting *bandwidth* – a well explored property of NMP-style accelerators.

In this work, we aim to provide an exploratory analysis on accelerating garbage collection with NMP via *latency* benefits. We identify that the marking subroutine of the mark-and-sweep garbage collection algorithm exhibits this same pointer-chasing behavior. Work in [10], [11] provide precedent for accelerating pointer-chasing applications via NMP. In this work, we extend these themes to garbage collection – which is more complex and works at a higher level of abstraction.

This work specifically examines the effectiveness of offloading the pointer-chasing marking phase of garbage collection to *generic* near-memory compute units. Instead of developing idealistic custom hardware, we aim to exploit a generic NMP hardware similar to what is already in production [1]. We aim to take advantage of existing under-utilized hardware as opposed to developing new hardware with the hope that it will be deployed. While custom garbage collection hardware may lead to performance benefits in simulated environments, these proposed designs may not be realistic. Instead, we work from the overriding belief that *good solutions on unmodified hardware* can lead to implementations that can be easily adopted in off-the-shelf systems.

Our early results are promising. Depending on the evaluation configuration, offloading Java garbage collection's marking phase to near-memory compute units can reduce last-level cache misses by up to $2.3\times$ and improve overall benchmark performance by up to $2\times$. Although further investigation is needed in order to extend the benefits to a wider range of workloads and configurations, the analysis points to several promising directions for future work.

This paper is organized as follows: in Section II, we briefly review the mark-and-sweep algorithm used in garbage collection and provide motivation for the use of generic NMP architectures; in Section III, we describe the mechanism for offloading computation; in Section IV, we evaluate and analyze the architecture-level behavior of our approach across varying evaluation configurations; in Section V, we consider opportunities for future work based on our initial findings.

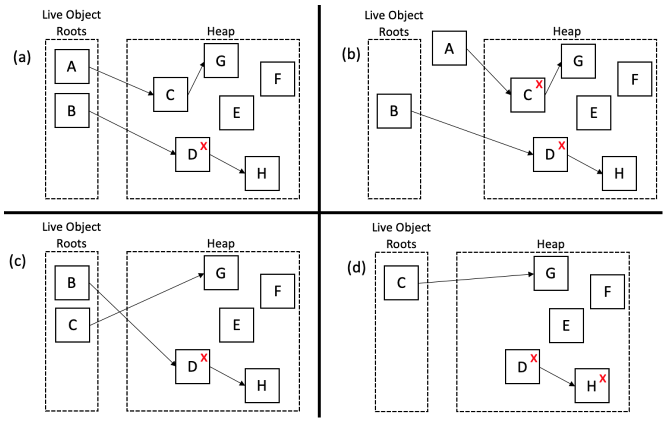

Fig. 1. Illustration of the marking phase in the mark-and-sweep GC algorithm. The stack of live object roots point to live objects in the heap. Objects with an x have been marked as live. Figure (a) shows the initial state of the heap. Figure (b) illustrates A being popped from the live object roots to mark referent item C. figure (c) shows C being added to the live object roots. Figure (d) shows the state after B has been popped from the live object roots, with H marked as live.

II. Background

Garbage collection often employs the *mark-and-sweep* method. Briefly described, let *roots* denote all pointers directly reachable by program threads. The *marking phase* starts by pushing all root pointers to the *mark-stack* (denoted as *live object roots* in Fig 1). Next, an iterative process of handling the mark-stack is executed until it is empty. In each iteration, (1) a pointer is popped from the mark-stack (depicted as A in Fig. 1b), (2) the referent object is scanned for pointers, and for each referenced object: (2a) it is marked as live (depicted as C in Fig. 1b) and (2b) if not previously marked, it gets pushed into the mark-stack (depicted as C in Fig. 1c). Once the stack is empty, the heap is *swept* clean of all objects that have not been marked. In Fig. 1, this would ultimately clear nodes E and F.

We look to utilize near-memory processing (NMP) to optimize garbage collection in Java. NMP architecture based on 3D-die stacking technologies have been explored as a workaround for computations with memory bottlenecks, including graph processing and generic pointer-chasing [2], [10], [11], [13]–[15], [19], [23]. NMP describes an architecture in which a lightweight processor is placed at the logic controller of the memory die to perform more complex computation than can be performed in a traditional memory controller. While NMP architectures do not cut down latencies inherent to DRAM, they reduce performance degradation and energy consumption caused by data movement on long and narrow off-chip interconnects

The benefits of using NMP have been shown to be two-fold. For one, pointer-chasing applications tend to follow pointers to random locations in memory and exhibit poor cache locality as a result. Instead, by using NMPs, the same computation can be performed without polluting the cache with pointers that will not be reused. The other benefit of NMP is that it exhibits a lower latency than CPUs in that there is no interconnect latency at play, which saves cycles. In our work, we work from the observation that the marking phase of garbage collection in Java exhibits similar behavior.

The pointer-chasing aspect of mark-and-sweep causes memory bottlenecks, which relevant prior work have tried to address by developing customized hardware accelerators. In particular, Maas *et al.* [20] designs specialized accelerators for both the marking and sweeping phases of GC to avoid cache pollution. They also describe an additional device driver that is maintained by the kernel so that any higher level programming language can utilize the underlying hardware. Charon [16] identifies certain fine-grained operations (*e.g.,* scanning the heap, copying objects for memory consolidation, etc.) as GC primitives and designs custom near-memory accelerators for each of these primitives. While these works have shown performance improvements, each required significant hardware modification and limited their evaluation to workloads with large objects in order to maximally exploit the high bandwidth benefits of NMP.

Our work is distinctly different from prior works in that we assume generic near-memory compute units. That is, these cores have no hardware components that are specialized to optimized garbage collection. Instead, all components of these cores can be utilized by general computation. We focus on appropriately partitioning the GC work between the on-chip host cores and the near-memory compute units and in turn aim to modify the mark-and-sweep algorithm accordingly to take advantage of NMPs latency benefits without polluting on-chip caches.

Java JDK Garbage Collection: While Java is not the only language that uses mark-and-sweep, our investigations are based on Java JDK's mark-and-sweep garbage collection. This garbage collection happens in a *stop-the-world* manner, meaning that application execution is stopped entirely while the garbage collection process runs. Garbage collection algorithms that run concurrently with the application have been proven to be too slow to be used in [7], so we do not consider them in our evaluation. Moreover, garbage collection in Java is triggered when heap space utilization exceeds 80% of its capacity. As such, larger heap sizes means that fewer collections will occur and overall application execution will ultimately be faster. However, because our approach aims to reduce the GC execution time, we intentionally reduce the heap size in order to trigger more collections and measure their performance appropriately.

More specifically, the JDK employs *generational* garbage collection, in which the heap is divided into a *young generation* heap and an *old generation* heap. This is based on the insight that objects are likely to be freed soon after their allocation. As such, objects are first placed in the smaller young generation heap. Objects that survive young generation collections are moved to the old generation heap, where collections are infrequent but incur longer pauses. In particular, it is in old generation collections (*i.e., full collections*) that the mark-and-sweep algorithm is used. Thus, in order to trigger more mark-and-sweep garbage collections for evaluation, we

TABLE I
EVALUATION FRAMEWORK CONFIGURATION.

Host Configuration	
Host cores	in-order processor (gem5 TimingSimpleCPU) 2GHz frequency, 1 thread/core
L1 cache	48kB icache, 32kB dcache, private 2-way set-associative LRU
L2 cache	1MB, shared, 8-way set associative LRU
NMP Core Configuration	
NMP cores	in-order single-cycle processor (gem5 TimingSimpleCPU), 2GHz frequency
L1 cache	48kB icache, 32kB dcache, private 2-way set-associative LRU
Memory Configuration	
2GB DRAM (gem5 DDR3_1600_3x3) t_{RP}: 13.75ns, t_{RCD}: 13.75ns, t_{CL}: 13.75ns, t_{BURST}: 3.2ns	

varied the young and old generation heap sizes.

III. METHODOLOGY

We perform our evaluation on a gem5 [4] full-system simulation extended from the ARM bigLITTLE configuration. ARM bigLITTLE describes a heterogeneous CPU architecture with high-powered and low-powered cores, and low-powered cores exhibit properties similar to NMP cores. In particular, little cores and NMP cores both are single-threaded, in-order cores with simple execution pipelines, and neither utilize the host's cache hierarchy. This makes the ARM bigLITTLE architecture particularly well suited to evaluate the ability of our approach to reduce cache pollution. Then, we modify the timing across the little core memory access pipeline so as to accurately model the memory access latency of a near-memory processing core. In particular, we remove interconnect latency, which was shown to have a significant impact on performance and energy consumption in [11], and reduce memory access latency by 5% to model NMP's faster access times due to proximity to memory.

We found this simulation model appropriate given the similar properties of ARM little cores and NMP cores in production. Furthermore, the JDK proved to be too complex for other state-of-the-art simulators that may more precisely simulate NMP, such as SMCSim [9]. Running these benchmarks on these simulators requires intensive changes to the simulators that would have had little impact on overall runtime performance or cache behavior results. Given this, ARM bigLITTLE provides a sufficient model for performing our evaluation. Full details of our hardware configurations can be viewed in Table I.

At a high level, we aim to pin the marking protocol to NMP cores in our simulation. However, achieving this end is non-trivial. In order to do so, we modify the Java garbage collector in the JDK version 14. In particular, we isolate the marking phase of the mark-and-sweep algorithm in order to run it on the NMP cores. This entails creating a specialized `workGroup` – a special garbage collection process object in the JDK source. We then leverage the fact that our assumed architecture allows the operating system (OS) to see the NMP core as a core in the simulated environment to offload computation by pinning it to NMP cores, but that the Java runtime environment only has explicit knowledge of host cores.

The ability of the OS to manage the host processors and NMP cores solves difficult problems in NMP research that are tangential to offloading computation. For example, coherence between host and NMP cores remains an open problem that is distinct from partitioning work across the heterogeneous architecture, but is difficult to navigate without the help of the OS.

By implementing this protocol in software, our approach is not limited by the ability to implement software primitives in hardware. This enables our approach to take advantage of general-purpose hardware without any invasive architectural changes. Furthermore, this means that we can extend our approach to newer versions of garbage collection in the JDK and other high level programming languages that might exhibit different semantics. Supporting this flexibility is important given that software frequently evolves – for example, JDK version 14 had stable changes pushed almost every month during its release.

IV. EVALUATION

In our evaluation, we use two configurations: *host-only* and *NMP*. The host-only configuration models a conventional system with on-chip processors, L1 caches and a shared L2 cache. The NMP configuration incorporates NMP cores into the host-only configuration. In the NMP configuration, there is a single near-memory core (*NMP core*) equipped with an L1 instruction and data cache. We use the h2 benchmark from the DaCapo benchmark suite [5], which is a standard Java garbage collection testing suite. Prior evaluation [7] of the DaCapo suite has demonstrated that the h2 benchmark is among the most memory intensive benchmarks in the suite and is well suited for evaluating full collections. When using the DaCapo benchmark suite, it is standard practice to run warm-up iterations prior to collecting measurement data. This is done in order to avoid measuring extraneous behavior such as dynamic loading and compiling of modules and classes, which rarely happen in the typical execution of long-running applications. The number of warm-up iterations varies in prior work from one [6] to 20 [18]. In our preliminary evaluation, we found that the performance improvements gained by increasing the number of warm-up iterations from four to 20 is less than 5%. We perform evaluations with and without warm-up iterations. For evaluations with warm-ups, we run three warm-up iterations and report the measurements that we obtain from the fourth unless the discussion explicitly states otherwise.

A. Performance

Our initial evaluation is promising, and its results are shown in Fig. 2a. Performance refers to runtime in milliseconds, so lower is better. The evaluation involved varying the young and old generation heap sizes to show scalability and promise of our approach. We used the following old/young heap size configurations: (1) 4GB/256MB (JDK default), (2) 250MB/50MB, and (3) 250MB/100MB. By utilizing smaller old generation

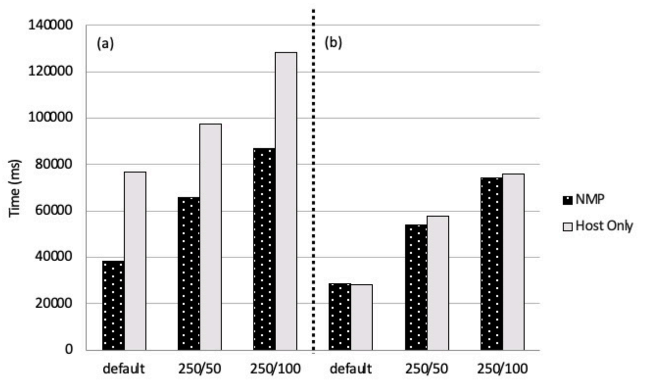

Fig. 2. Running time (y-axis) of the h2 benchmark under different heap size configurations (x-axis). (a) shows performance without warm-up iterations and (b) shows performance with warm-up iterations.

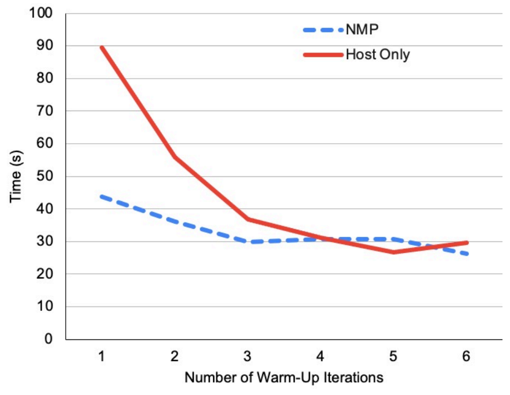

Fig. 3. Running time in milliseconds versus number of warm-up iterations on the h2 benchmark in host only and NMP configurations with the default heap size configurations.

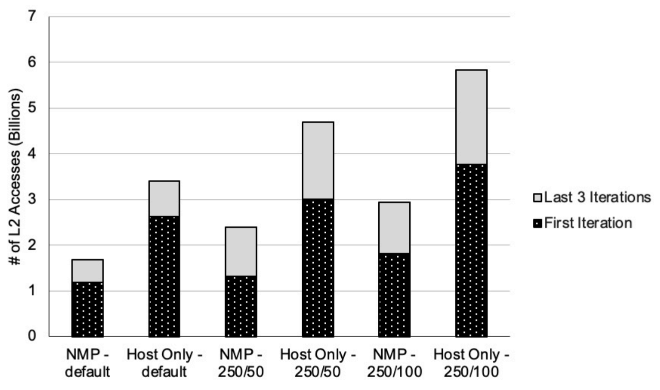

Fig. 4. Number of requests for the L2 in each of the heap configurations by architecture. Most requests occur in the first warm-up iteration.

heap sizes in configurations (2) and (3), more mark-and-sweep full collections were incurred overall. No warm up iterations were used in this initial evaluation. The figure shows that the NMP configuration can demonstrate up to a 2x improvement in performance (which occurred in the default JDK configuration).

We see that the 250MB/100MB configuration is consistently slower than the 250MB/50MB configuration. This is largely due to the fact that young garbage collections will take longer in configurations where the young generation size is bigger (as there are more objects in the heap), and our approach does not modify young generation collections. In the 250MB/50MB heap size configuration, we see a 48% improvement in overall performance of the NMP-based solution relative to the unmodified host-only baseline. In the 250MB/100MB heap size configuration, we see a 47% improvement in overall performance by the NMP-based solution. The takeaway from this result is that full collections of the young and old generation heaps are the bottleneck to performance rather than having a large number of young generational collections. Fine-tuning heap-sizes for optimal performance is difficult, and is beyond the scope of this project. However, we see that the performance improvement with our approach is consistent across each heap configuration without warm-up iterations. As such, we use these results to further our intuition, but otherwise largely focus on the default heap size configuration.

Fig. 2b shows the impact of warm up iterations on performance. The results demonstrate that the differences in performance are much less significant between the NMP and host only configurations with warm ups. Fig. 3 shows the impact on performance as the number of warm-up iterations increases in the h2 benchmark with and without NMP in the default heap size configuration. As previously described, there is 2× difference in performance without any warm up iterations, but this difference is reduced over time. In fact, the difference in performance becomes negligible across the two configurations after three warm up iterations.

We see a greater impact of our approach on performance without warm-up iterations. This is largely attributed to differences in application behavior. In general, evaluating performance on a cold-start tends to induce more cache misses. Seeing as our approach optimizes L2 misses, it makes sense that our approach performs better in configurations where there are more L2 misses. We evaluate and discuss this in more detail with a more complete sensitivity analysis in Sec. IV-C.

B. Cache Traffic

At the architecture level, we find that L2 traffic is the primary culprit for differences in overall behavior. Fig. 4 shows that more than 77% of overall L2 accesses occur in the first warm-up iteration. This implies that the majority of cache traffic occurs during the marking phase of full generational garbage collection. The NMP provides the most benefit when the application exhibits a large number inefficient cache accesses. Seeing as we only move the marking phase of full generation collections to the NMP, it is implied that the difference in L2 cache accesses between the host only and NMP configurations are due to L2 accesses from this procedure. In our evaluation, we also measure how many of these accesses to the L2 come in the first warm-up iteration relative to the rest of the benchmark's execution.

In this evaluation, we observe two key takeaways. For one, we find that the NMP approach is very effective at reducing the number of L2 accesses in the first iteration of benchmark execution. As demonstrated in comparing default heap configurations, we see that there are approximately 2.4 times more first warm-up iteration L2 accesses in the host only configuration compared to the NMP configuration. This shows that the marking phase of full generation collections significantly dominates L2 cache accesses in baseline protocols. This trend is consistent across all heap sizes, which suggests that application behavior in a cold-start requires lots of L2 accesses as a consequence of garbage collection. We also see that the number of L2 accesses is reduced in the rest of benchmark execution by as much as 81% (in the 250MB/100MB configuration), but the number of accesses as a whole is much lower. As such, the impact of our technique will not be as apparent.

The other takeaway from this evaluation is that an overwhelming majority of L2 accesses occur in the first warm-up iteration across all heap configurations. On average, we observe that 61% of L2 accesses occur in the first warm-up iteration. As such, the impact of our technique will be most visible in this iteration.

More generally, these large differences in cache traffic for the L2 in the host largely describe the benefits of utilizing NMP. That is, the cache hierarchy in the host is not saturated by requests for the L2 during the marking phase of garbage collection. Performing these operations elsewhere in the architecture means that these caches will be subject to less pollution. This should ultimately help application performance.

C. LLC Variations

We vary the L2 size in order to provide a sensitivity analysis of our approach with warm-up iterations. In particular, we look to demonstrate the efficacy of our approach – namely, providing a fast alternative for workloads with many L2 accesses – while controlling for warm-up iterations and heap sizes. That is, we evaluate our technique with the default heap configuration and with four warm-up iterations – circumstances when the L2 misses are minimized. Our approach optimizes for L2 misses, and we aim to use this section to confirm our initial hypothesis that applications that use the L2 inefficiently will benefit from our approach.

Fig. 5 demonstrates the differences in performance between NMP and host only configurations. We measure for overall running time, so lower is better. In the configuration where the L2 is set to 512kB, we see the fewest number of L2 misses and the differences in performance between the NMP and host only configuration is less than 1%. As L2 cache size decreases, the number of L2 misses and the impact of the NMP-based approach both increase. That is, we see a 12% performance improvement in the NMP configuration versus the host only configuration with a 64kB L2.

This evaluation demonstrates that our approach is effective at optimizing for L2 misses. It highlights that the performance benefits demonstrated in Fig. 2a were a result of reducing the impact of inefficient utilization of the L2. However, this coupled with the reduction in performance impact in Fig. 2b also leads to the conclusion that utilizing warm-up iterations also significantly improves the efficiency by which the L2 is utilized. In general, measuring performance after warm-up iterations avoids measuring cache behavior from a cold-start in which caches are not used as efficiently.

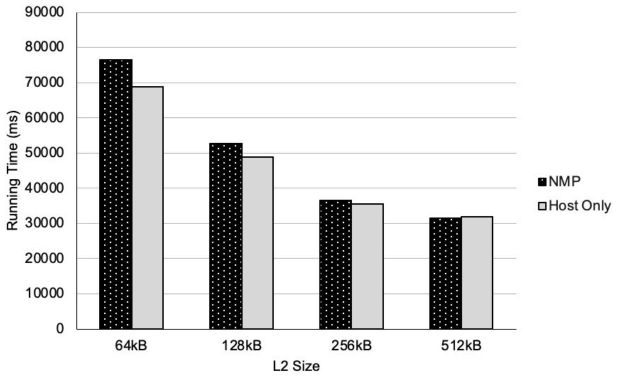

Fig. 5. Performance results of the h2 benchmark in the host only and NMP configurations with varying sizes for the L2 with four warm up iterations in the default heap configuration.

V. Discussion

In this work, we propose and implement a means to reduce the negative impact of Java's mark-and-sweep algorithm on efficient L2 cache utilization. We offload this costly computation to NMP in order to avoid accessing the cache hierarchy. This approach is designed to be minimally invasive by taking advantage of emerging technologies through *software modification*. By doing so, we demonstrated that we can improve performance by 2× in benchmarks without warm-up iterations, and we attribute this performance improvement to the 2.3× reduction in L2 traffic. This work shows that the pointer-chasing nature of the mark-and-sweep algorithm does not utilize the L2 efficiently without warm-up iterations. We address this by offloading the computation to a component of the microarchitecture that does not negatively impact on-chip caches. The performance advantage of this approach is two-fold – (1) application cache accesses should hit more frequently as important data will not be evicted prematurely, and (2) the mark-and-sweep algorithm will not have to wait for cache miss latency in addition to memory access latency on values that likely will not be in cache. Our evaluation shows that the impact of our technique is a function of L2 utilization in host only configurations, so using warm-up iterations reduce the impact of our technique.

We believe there is an open opportunity to evaluate whether or not cache inefficiency in early warm-up iterations is a consequence of cold caches or application behavior. In particular, Java programs execute inside the Java Runtime Environment (JRE), which performs many tasks at program invocation – such as just-in time compilation, dynamic module loading, etc. The JRE as a whole is a complex software structure, and coming up with a more precise view of its behavior throughout the duration of an application would lend itself to other interesting architectural insights. We believe that a further understanding of the JRE could lead to interesting insights into hardware-driven modifications to the software. This could help extend our techniques to other applications as well, in Java and otherwise, that exhibit similar program behavior.

To begin that discussion, we observe that one potential contributing factor is the cold-start of the L2 cache in the first warm-up iteration. While the first warm-up iteration may not be of particular interest to users looking to optimize long-running applications, such as databases or hosting an application on a web server, there may be use cases that exhibit similar behaviors. For example, if future work deems dynamic module loading the primary contributing feature of garbage collection during first warm-up iteration, then it makes sense to perform similar offloading in applications that are bound by dynamic module loading.

Improving performance in garbage collection remains an important issue, in that long pauses due to full generation collections stops the execution of the application in order to execute automatic memory management. We believe that prior work has produced highly effective specialized hardware to accelerate specific garbage collection primitives, but the ever-changing nature and wide breadth of software makes it unlikely that these solutions will ever be fully adapted. Instead, our technique attempts to be minimally invasive, and it is flexible to be applied on other parts of the application and to alternative use cases. As a result, we hope this work leads to future opportunities in re-designing high-level software protocols to better take advantage of emerging hardware and its ability to improve performance without invasive hardware changes.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments. This work was supported by NSF grant 1909715.

REFERENCES

[1] Upmem. https://www.upmem.com. Accessed: 2022-03-11.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[3] David F Bacon, Perry Cheng, and Sunil Shukla. And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. *ACM SIGPLAN Notices*, 47(6):23–34, 2012.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[6] Stephen M Blackburnα, Robin Garnerβ, Chris Hoffmannγ, Asjad M Khanγ, Kathryn S McKinleyδ, Rotem Bentzurε, Amer Diwanζ, Daniel Feinbergε, Daniel Framptonβ, Samuel Z Guyerη, et al. The dacapo benchmarks: Java benchmarking development and analysis. 2006.

[7] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. A performance study of java garbage collectors on multicore architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 20–29, 2015.

[8] Stephen Cass. The 2018 top programming languages. *IEEE Spectrum*, 31:1, 2018.

[9] Jiwon Choe and Erfan Azarkhish. Brown-smcsim. https://github.com/jiwon-choe/Brown-SMCSim, 2022.

[10] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *The 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 321–332, 2022.

[11] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. Concurrent data structures with near-data-processing: An architecture-aware implementation. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 297–308, 2019.

[12] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56, 2005.

[13] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. Accelerating linked-list traversal through near-data processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 113–124, 2016.

[14] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32. IEEE, 2016.

[15] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. A heterogeneous pim hardware-software co-design for energy-efficient graph processing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 684–695. IEEE, 2020.

[16] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae 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*, pages 726–739, 2019.

[17] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[18] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 3–14, 2017.

[19] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245, 2017.

[20] Martin Maas, Krste Asanović, and John Kubiatowicz. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 138–151. IEEE, 2018.

[21] Thomas Perl. Python garbage collector implementations cpython, pypy and gas, 2012.

[22] Tony Printezis. Garbage collection in the java hotspot virtual machine, 2005.

[23] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.