CRAP: Collecting Resources Across different Processing levels

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

1. Introduction

Near-memory processing (NMP) has demonstrated promising properties to accelerate applications in several domains [13] including graph traversing/pointer-chasing applications [1], [8], [7], [12], [14], [6], [9]. They benefit from the fact that 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. This frees the host processor and caches from wasteful operations to perform more computationally efficient routines in the algorithm. While this direction is promising, existing work focuses on the data structure granularity.

In this work, we extend these themes to *garbage collection* – a much larger application with added complexities. Garbage collection is a tool provided by many high level programming languages that automatically frees dead objects in memory to ease the languages programmability for the user and help reduce memory leakage and memory corruption errors. We focus on garbage collection in Java.

Previous work has explored near-memory acceleration of the Java garbage collector via high-bandwidth memory [10]. Instead, we focus on certain routines of the garbage collection algorithm that exhibit pointer-chasing behavior. Garbage collection in Java is based on the *mark-and-sweep* algorithm. Briefly described, let *roots* denote all pointers directly reachable by program threads. The *marking phase* starts by pushing all root pointers to a "mark-stack." Next, an iterative process of handling the mark stack is executed until the mark-stack is empty. In each iteration, a pointer is popped from the mark-stack, the referent object is scanned, and it is marked as live. If not previously marked, then it gets pushed into the mark-stack. Once the stack is empty, the heap is *swept* clean of all objects that have not been marked.

Following pointers to random locations in the heap during the marking phase exhibits similar properties to graph-traversing problems. Given this, we propose utilizing NMP to accelerate the marking phase of garbage collection in Java. We modify the Java Developer Kit (JDK) to create a customized *worker process* to perform the marking phase of the garbage collection algorithm on NMP. Typically, the JDK requests a specialized process be started to perform the marking-phase. In our work, we propose that a similar process is started instead of an NMP core to reduce L2 traffic. This worker process operates in parallel with the host during garbage collection, so it is completely idle during normal execution. This implies that a user can additionally use the NMP to accelerate the application outside of garbage collection, so our model is conservative.

Our goal was to improve garbage collections in all conditions by utilizing NMP. Although our initial evaluation was promising, emulating the behavior of long-running programs didn't show the same promise. We hope to use this submission to begin a discussion about the properties of short-lived applications that benefit from NMP and under what circumstances they may extend to long-running applications. Given this, we will discuss the following:

1) We show that our initial evaluation was promising. By performing the marking phase of Java's garbage collection near memory, benchmark performance can be improved by 2x in short-lived programs. However, long-running programs do not see a significant improvement in performance from our technique.
2) We provide insights into why behaviors are different in long– and short-lived applications and why hardware exclusive solutions don't generalize.

2. Evaluation

To perform our evaluation, we use the h2 benchmark from the DaCapo benchmark suite [3], a standard Java garbage collection testing suite. Prior evaluation [5] 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. Our hardware configurations can be viewed in Table 1.

Our initial evaluation was promising, and its results are shown in Fig. 1a. Performance refers to running time in milliseconds, so lower is better. The evaluation involved modifying a single environment variable, *generational heap configuration*, to show scalability and proved to be promising. The figure shows that the NMP configuration can demonstrate up to a 2x improvement in performance.

We use different young and full heap sizes to show the scalability of our technique. Garbage collection in Java is inspired by the weak generational hypothesis, and young generation heap size and full heap size are used to trigger young and full collections. We perform our evaluation on three variant young and old generational heap configurations of the h2 benchmark: (1) default, (2) 250/50, (3) 250/100. The first value refers to the overall heap size (young generation size plus old generation size) in megabytes and the second number refers to the size of the young generation size in megabytes. Default refers to the default configurations of the JDK environment, which is 4GB and 256MB.

From an architectural perspective, the benefits in performance can be described by cache behavior. Fig. 2 shows that, although there are no significant differences in hit rates through the cache hierarchy, the number of requests to the

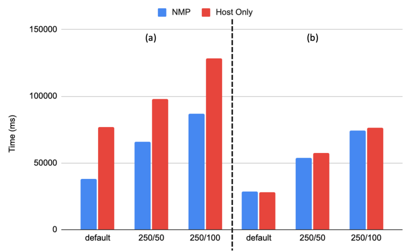

Figure 1: Running time (y-axis) of the h2 benchmark under different heap size configurations (x-axis). (a) shows performance without warm-up iterations and (b) shows performance with warm-up iterations

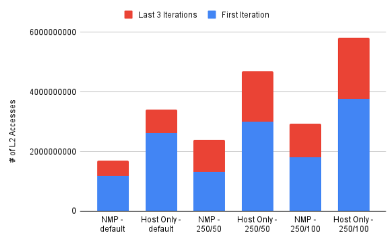

Figure 2: Number of requests for the L2 in each of the heap configurations by architecture. Most requests occur in the first warm-up iteration

L2 are 2.3x higher in the host-only configuration. That is, we identify that 55% of L2 activity in the h2 benchmark comes from the marking phase of garbage collection.

Our evaluation extends beyond the initial evaluation to more accurately reflect the behavior of long-running applications with *warm-up iterations*. Fig. 3 shows the impact of using warm-up iterations on performance with and without NMP. In Java, modules and classes are loaded and compiled dynamically at runtime to the virtual environment using just-in time (JIT) compilation. As such, it is standard practice to use these warm-up iterations to avoid measuring these extraneous behaviors, as they are less common in the typical use case of long running applications. The number of warm-up iterations varies in prior work from one [4] to 20 [11]. The lack of communal uniformity in determining the number of warm-up iterations to use is problematic for comprehensive evaluation [15], so we use their "cookbook" approach for a more up-to-date analysis of the DaCapo benchmarks, because each iteration may take a few seconds to a few minutes on a host machine. In our work, we use four warm-up iterations.

Fig. 1b demonstrates the impact of warm-up iterations on running time with the host only and NMP configurations – lower is better. The figure shows that the difference in performance between the host only and NMP configurations is relatively insignificant. This result alone may not be particularly exciting, but we can utilize it to give us a bigger insight to the fundamental properties of Java programs from an architecture perspective. We found that the cache hit rates are consistent across each of the host only and NMP configurations. The hit rate in the data cache was consistently about 95% and the L2 hit rate was consistently about 50%, though some NMP configurations saw hit rates closer to 60%. Instead, Fig. 2 demonstrates that the architectural

TABLE 1: Evaluation framework configuration.

Host Configuration	
Host cores	1 in-of-order processors (gem5 [2] 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	1 in-order single-cycle processor/vault (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	

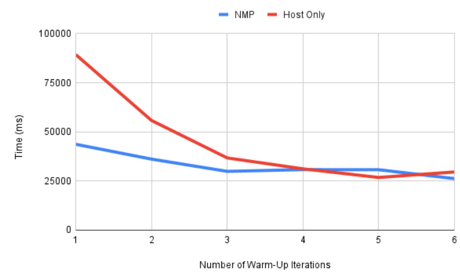

Figure 3: Running time in milliseconds versus number of warm-up iterations of on the h2 benchmark in host only and NMP configurations.

explanation for the difference in performance comes from the fact that about 75% of L2 traffic occurs during the first iteration of the benchmark with four warm-up iterations.

In summary, our analysis of the marking phase of garbage collection on NMP demonstrated that 55% of L2 traffic in the first benchmark iteration results from the marking phase. As such, using NMP can give a 2x improvement in performance under these circumstances. However, with warm-up iterations there is little to no improvement in performance. We found that about 75% of L2 traffic occurs during the first iteration in our evaluation.

3. Discussion

We believe that a further understanding of the Java runtime could lead to interesting insights into hardware-aware modifications to the software. Furthermore, these insights could be utilized to exploit the cases where NMP can benefit garbage collection. This project led us to speculate about the importance of evaluating short-lived applications. For instance, are the properties of these benchmarks relevant in long-running applications where modules are loaded dynamically over long periods of time? We hope that this work can act as the beginning of this conversation, and that its underlying themes can be relevant to future work in this area.

We began this work interested in optimizing garbage collection in Java with NMP because of the algorithmic properties that these algorithms exhibit. However, we now find ourselves with larger questions about Java in general. Although our findings may not bear fruit for the integration of new NMP-aware garbage collection algorithms, we firmly believe that there is a place for NMP-awareness in the JDK.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Paulo Cesar Santos, Francis Birck Moreira, Aline Santana Cordeiro, Sairo Raoní Santos, Tiago Rodrigo Kepe, Luigi Carro, and Marco Antonio Zanata Alves. Survey on near-data processing: Applications and architectures. *Journal of Integrated Circuits and Systems*, 16(2):1–17, 2021.

[14] 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.

[15] Lisa Wu and Martha A Kim. Acceleration targets: A study of popular benchmark suites. In *The First Dark Silicon Workshop, DaSi*, volume 12, page 25. Citeseer, 2012.