Investigating Hardware Caches for Terabyte-scale NVDIMMs

Julian T. Angeles
CS Dept., UC Davis
jtangeles@ucdavis.edu

Mark Hildebrand
ECE Dept., UC Davis
mhildebrand@ucdavis.edu

Venkatesh Akella
ECE Dept., UC Davis
akella@ucdavis.edu

Jason Lowe-Power
CS Dept., UC Davis
jlowepower@ucdavis.edu

1 INTRODUCTION

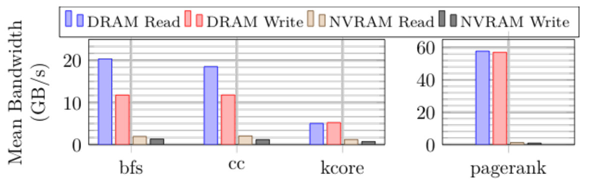
NVDIMMs based on 3DXpoint (such as Optane DC Persistent Memory Module) are emerging as an attractive option to address the needs of emerging applications that requires tens of terabytes of memory such as graph analytics and machine learning. To mitigate the increased latency and reduced bandwidth of NVDIMMs, in Intel systems, a smaller capacity DRAM serves as a cache to the larger capacity NVRAM in the so called 2LM mode (also known as *memory mode* or *cached*). Alternatively, the NVRAM devices can also be explicitly managed in the 1LM (or *app direct*) mode requiring application changes to take advantage of DRAM. Prior work has evaluated the system-level and device-level performance of Optane DC [4, 6]. Additionally, other works including Dhulipala et. al [1] and Gill et. al [2] evaluate the performance of large scale graph analytics on NVRAM based systems focusing on application performance evaluation and optimization but do not delve into the details of behavior of the DRAM cache (the 2LM mode) performance.

In the past, DRAM caches have been studied in the context of die-stacked systems [5] where the goal was to use a few gigabytes of stacked DRAM as a giant last-level cache mainly to overcome the bandwidth limitation of going off chip. However, the purpose of DRAM caches in a NVRAM based system is different. Instead of a few gigabytes, the DRAM cache in Intel's Cascade Lake systems can easily be 384 GB with 6 TB of backing main memory. This cache is about two orders of magnitude larger than previously studied DRAM caches. So, we are motivated by the question: how well do block-based die-stacked era DRAM caches work in NVRAM based systems and what are their limitations?

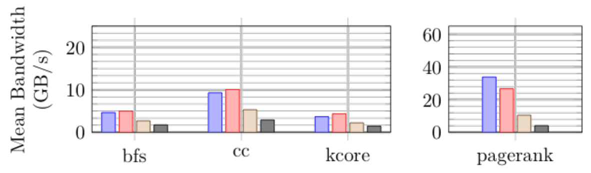
The goal of this work is to provide initial answers to this question by taking a deep dive into the performance of 2LM based systems using a real hardware and the built-in performance counters. We focus our study on large scale graph processing for two reasons. First, this represents a "growing" workload with today's large graphs already requiring many terabytes of RAM. Second, these workloads have irregular memory access patterns that are difficult to predict (e.g. with software managed approaches). We show that the current DRAM cache implementation (which is a naive direct mapped cache) performs poorly on graph workloads and does not take full advantage of the available bandwidth while generating a significant amount of unnecessary traffic. Further, we argue that designing hardware managed DRAM caches is an important problem that the computer architecture should address.

2 EXPERIMENTS

Our test machine is a two-socket Xeon server equipped with 24-core Cascade Lake engineering sample CPUs. Each socket has two integrated memory controllers (IMC) with three memory channels populated containing a 32 GiB DDR4 DRAM DIMM and a 512 GiB Optane DC DIMM for a total of 384 GB DRAM and 6 TB NVRAM.



(a) Performance of graph kernels on *kron30* which fits in DRAM



(b) Performance of graph kernels on *wdc12* which exceeds DRAM capacity

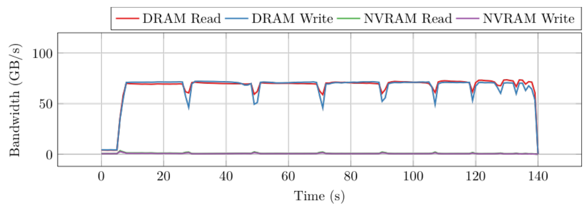
Figure 1: Average DRAM and NVRAM bandwidth.

All experiments were run using the shared memory graph analytics framework Galois. Gill et al. evaluated Galois, GBBS, and GAPBS on Optane and found that because of NUMA-aware memory allocation at the application level, kernel overhead avoidance, and asynchronous graph algorithms Galois performed the best [2]. Our evaluations consisted of 4 benchmarks from the lonestar suite: breadth-first search (bfs), connected components (cc), k-core decomposition (kcore), and pagerank-push (pr).

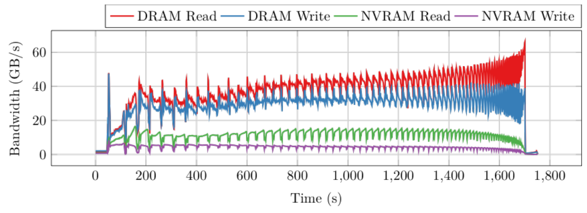
We used two realistic unweighted massive input graphs: wdc12, the largest publicly available graph, and kron30, a randomized scale free graph generated using a graph500 based kronecker generator. Each were chosen to highlight the differences between when a graph fit and did not fit in the DRAM cache. All measurements were gathered using hardware performance counters.

Poor Bandwidth Utilization with High Miss Rates. Our first observation is that when the working set of the application does not fit in the DRAM cache the bandwidth to both NVRAM and DRAM is severely underutilized. Figure 1a shows the average bandwidth for a graph which fits in the cache and Figure 1b shows the average bandwidth for a graph that exceeds the cache capacity. The graph which does not fit in the cache causes a 50% reduction in bandwidth for DRAM. This is due to three related problems: there are many cache misses which require waiting on high-latency NVRAM, many of the DRAM misses cause NVRAM writes, and the NVRAM accesses are fine-grained further increasing their latency.

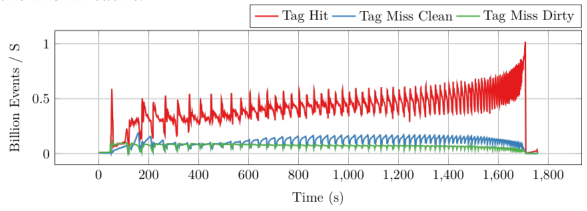
To further understand the cause of this bandwidth reduction, we captured the bandwidth over time and cache hit rate for the pagerank-push algorithm when the input graph fits or exceeds the capacity shown in Figure 3. Figure 2a shows the algorithm's bandwidth when its working set largely fits in the cache. Bandwidth is stable at 70 GB/s with roughly equal DRAM reads and writes. On the other hand, Figure 2b demonstrates the bandwidth of pagerank-push when its working set *does not* fit in the DRAM cache. Not



(a) **Bandwidth trace for *kron30*, which largely fits within the DRAM cache.**



(b) **Bandwidth trace for *wdc12*, which greatly exceeds the capcity of the the DRAM cache.**



(c) **Tag trace for *wdc12*.**

Figure 2: Detailed traces for the *pagerank-push* algorithm.

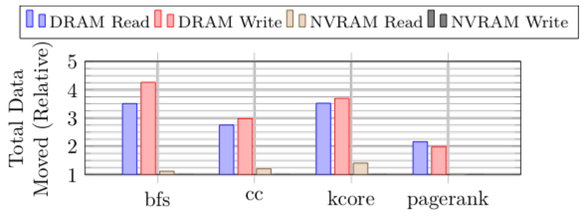


Figure 3: The total data moved when the input graph exceeds the capacity of the DRAM cache in 2LM relative to the NUMA baseline.

only is the average bandwidth significantly lower, but there is also an excess of DRAM reads coupled with heavy NVRAM traffic. The tag metrics shown in Figure 2c show the presence of both clean and dirty tag misses as well as the correlation between hit rate and DRAM bandwidth. As graph kernels often mutate graph data structures, this is especially a problem given the higher costs of writes than reads on NVRAM devices.

Significant unnecessary data movement. Our second main observation is that when using the DRAM as a cache of NVRAM the total data movement significantly increases. By comparing with the baseline data movement required by each algorithm, we find that the DRAM caches suffer from access amplification, generating unnecessary data movement and reducing the bandwidth utilization. To find these baselines, we configured the NVRAM regions on our two socket system as extra NUMA nodes. Since Galois uses a NUMA preferred policy, the threads on each socket will initially allocate memory on that socket's DRAM. When DRAM is exhausted, further allocations are serviced by NVRAM. By summing the traffic to DRAM and NVRAM, we can establish the baseline memory traffic required by each application, such that we can observe if and by how much unnecessary data is moved with DRAM caches.

Figure 3 shows the relative total amount of data moved in the NUMA and 2LM configurations for NVRAM. Since page migration was disabled, we observe the true demand accesses of the workload. Compared to the baseline, we see significant increase of data being moved across all graph kernels. We argue that the block-level movement used with DRAM caches contributes to this effect.

3 DISCUSSION

In this work, we show that the current DRAM cache implementation performs poorly for applications with a high miss rate. We also show that a miss in the DRAM cache generates significantly more data movement than is necessary. Further, we show that this causes performance degradation in bandwidth-limited workloads such as large graph analytics which is an important use case for NVRAMs since they have extremely large memory footprints.

Previous efforts have looked to mitigate these overheads by providing input from either the programmer or compiler. The authors of Sage [2] designed their graph framework specifically with NVRAM in mind. Their key approach is to (as much as possible) use NVRAM for read only data. Similarly, in prior work (AutoTM [3]), we showed that for static compute graphs such as static CNNs, that software data movement can provide a significant performance boost over hardware management. AutoTM achieves a 1.88×, 2.24×, and 3.10× speedup over 2LM for Inception v4, ResNet 200, and DenseNet 264 respectively [3].

That being said, both examples require explicit data management by software. As future research, we present a challenge to the community: **how do we design a solution that is both implicit (like hardware caches) and high performance (like explicit data movement)?** This mechanism likely requires flexibility for a variety of different workloads with different data resuse patterns and the ability to track coarse granularity data movement to reduce the amount of metadata necessary. Another interesting research direction is to specialize the cache designs for specific workloads, such as graph processing or training deep neural networks. We are excited to see the potential future of this space.

REFERENCES

[1] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proceedings of the VLDB Endowment* 13, 9 (2020).
[2] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1304–1318.
[3] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *ASPLOS 2020*.
[4] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714
[5] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *MICRO 2014*.
[6] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jisen Zhano. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *MICRO 2020*.