# A Case Against Hardware Managed DRAM Caches for NVRAM based Systems

Mark Hildebrand*, Julian T. Angeles†, Jason Lowe-Power†, Venkatesh Akella*
*Department of Electrical and Computer Engineering
†Department of Computer Science
University of California, Davis
{mhildebrand, jtangeles, jlowepower, akella}@ucdavis.edu

***Abstract*—Non-volatile memory (NVRAM) based on phase-change memory (such as Optane DC Persistent Memory Module) is making its way into Intel servers to address the needs of emerging applications that have a huge memory footprint. These systems have both DRAM and NVRAM on the same memory channel with the smaller capacity DRAM serving as a cache to the larger capacity NVRAM in the so called 2LM mode. In this work we analyze the performance of such DRAM caches on real hardware using a broad range of synthetic and real-world benchmarks. We identify three key limitations of DRAM caches in these emerging systems which prevent large-scale, bandwidth bound applications from taking full advantage of NVRAM read and write bandwidth. We show that software based techniques are necessary for orchestrating the data movement between DRAM and PMM for such workloads to take full advantage of these new heterogeneous memory systems.**

## I. Introduction

Large scale machine learning and large scale graph analytics represent workloads of interest for high performance server in the forseeable future. Emerging machine learning models in NLP and recommendation engines (such as GPT3 [3] and DLRM [33]) can have over 100 billion parameters requiring hundreds of gigabytes to terabytes of memory for training. Similarly real world graphs can have hundreds of billions of edges, requiring hundreds of gigabytes to just store the graphs [36]. As a result, the cost of memory (DRAM) is becoming an important concern in datacenters and other high performance computing facilities dealing with large scale data analysis [15], [16].

To address this challenge Intel recently introduced Optane Data-Center Persistent-Memory-Modules (DC PMM), a non-volatile memory (NVRAM) technology based on phase change memory that can serve as a drop-in replacement for conventional DRAM [20]. While programmers can use the NVRAM as a main memory DRAM replacement using normal load and store instructions, the latency is 3× higher and the bandwidth is at least 60% lower than DRAM [51]. Traditionally, to hide high memory latency and limited bandwidth, computer architects have turned to hardware caches. In this tradition, Intel Cascade Lake systems implement a *DRAM* cache for the NVRAM. DRAM caches have been well studied in simulation [6], [7], [27]–[29], [31], [41]. These previous works have not taken all of the realistic implementation details (e.g., tracking "coherence" of request issued to NVRAM) leaving gaps between research proposals and the actual implementation.

In this work, we analyze the performance of an *actual implementation* of the DRAM cache in Intel's Cascade Lake based servers on workloads whose memory footprint *greatly exceeds* the capacity of DRAM. We first analyze the behavior of the DRAM cache with microbenchmarks to reverse engineer its design and understand pathological performance cliffs. It is well known that this DRAM cache is implemented as a direct-mapped [26], and we find that the tags are stored ECC bits of the DRAM DIMMs to limits the access overhead. However, we also find that in many cases there are extra DRAM accesses required to update the cache metadata (e.g., tag reads before writes) which can significantly decrease the performance of miss-heavy workloads. In fact, using microbenchmarks on real hardware, we find that a single demand request can require up to **5 memory accesses**.

After using microbenchmarks to understand the cache behavior and implementation, we analyze two memory capacity limited workloads: training large convolutional neural networks (CNNs) [21], [24], [47], [48] and graph analytics [17]. We show that in these realistic workloads, the DRAM cache *can hurt performance* even with a modest cache miss rate. We show that for the CNN workload, software management can increase performance by up to 3× over the DRAM cache. Furthermore, we show significant access amplification and bandwidth reduction for graph based workloads.

Fundamentally, we find three characteristics of this DRAM cache implementation which causes performance degradation for workloads with large working sets.

1) The direct-mapped, insert on miss cache is inflexible and many conflicts can increase the miss rate.
2) Under high miss rates, memory bandwidth is poorly utilized with extra bandwidth used for non-demand accesses (e.g., fills, writebacks, and tag checks).
3) For some workloads the data in the DRAM cache is *temporary* or *dead* from the program's perspective leading to wasted data movement.

While some of these characteristics may be alleviated in future hardware, we can use these three insights *on today's hardware* to improve the performance of heterogeneous memory systems. We present one example of a static software management technique which by managing the data movement

in software can mitigate many of these performance problems. In the future, we hope that the insights presented in this paper can influence the next era of DRAM cache development.

The rest of the paper is organized as follows. We start with the quick background on Intel's NVRAM technology and related work in the area of benchmarking NVRAM from recent literature. In Section III we present the details of our evaluation and validation framework. Section IV follows up with a detailed analysis of the DRAM cache in these systems. Next we use two representative case studies from deep learning and graph analytics to corroborate the findings from the microbenchmark experiments. We end the paper with a discussion of the results and the software based mitigation strategies in Section VII.

## II. Background and Related Work

Intel Optane DC (NVRAM)[1] is a phase-change based non-volatile memory [20]. These devices come in a dual in-line memory module (DIMM) form factor and have the same physical footprint as traditional DRAM DIMMs. Memory controllers in high-end Cascade Lake or newer Xeon processors are capable of managing both a DRAM DIMM and a NVRAM DIMM on the same memory channel. Since NVRAM resides on the memory bus, CPUs may read and write to these devices using normal load and store instructions.

NVRAM can be used in the so-called 2LM (also known as *memory mode* or *cached*) [26], where NVRAM act transparently as system memory. In this mode, system DRAM serves as a direct mapped cache for the non-volatile memory. NVRAM can also be used in the 1LM (or *app direct*) mode using the `ndctl`[2] tool to appear as regular devices that are mounted into the Linux file system. In this mode, all loads or stores to memory mapped regions on this device go directly to the NVRAM devices themselves.

There have been several efforts in research literature that focus on evaluating the system level performance of Optane DC [26], [39], [40], [43], [49], especially in comparison with DRAM. More recently, Wang et. al [50] developed a profiler and NVRAM simulator to model the microarchitecture of NVRAMs in general. However, to the best of our knowledge there has been no effort in trying *understand* the performance of DRAM caches in large scale NVRAM-based systems. However, the tools described by Wang [50] *could* be used for hardware/software codesign of DRAM caches in the future, building on the findings in this paper.

On the application front there has been work on the design of data structures and algorithms to mitigate the disadvantages of NVRAMs, chiefly the slower and asymmetric read/write latency and bandwidth [4], [13], [35], [38], [45]. Dhulipala et. al [13] and Gill et. al [17] evaluate the performance of large scale graph analytics on NVRAM based systems. These works focus on application performance evaluation and optimization but do not delve into the details of behavior of the DRAM cache (the 2LM mode) and why they do not work well on these applications. The goal of this work is to fill this gap. In fact, one could view Sage [13] as a software technique to mitigate the limitations of DRAM caches in NVRAM based systems as discussed in Section VI and Section VII

[1]In this paper we will use Optane DC and NVRAM interchangeably.

[2]https://docs.pmem.io/ndctl-user-guide/

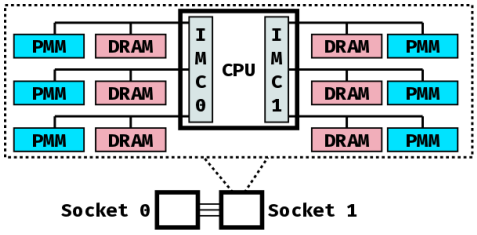



Fig. 1. Diagram of our test platform. Each socket has 192 GiB of DRAM and 3 TB of NVRAM spread across six memory channels.

## III. Evaluation Methodology and Validation

### A. Test System

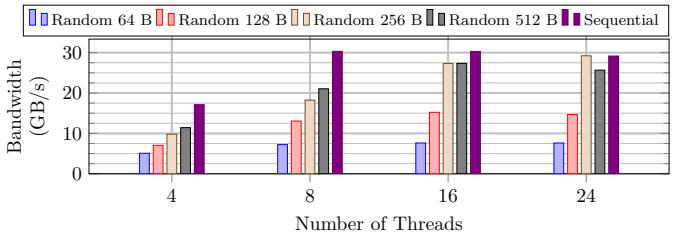
Our test machine is a two-socket Xeon server (illustrated in Figure 1) equipped with 24-core Cascade Lake engineering sample CPUs. The CPU on each socket is equipped with two integrated memory controllers (IMC), each with three memory channels. Integrated memory controllers are responsible for performing the actual reads and writes to DRAM and NVRAM. Each memory channel is populated with a 32 GiB DDR4 DRAM DIMM and a 512 GiB Optane DC DIMM.

### B. Evaluation Methodology

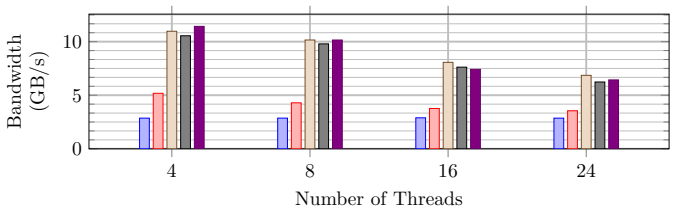
To test the basic bandwidth performance of DRAM and NVRAM, both in 1LM and 2LM, we made a custom open source benchmark generator[3] written in Julia [2]. The generator uses Julia's metaprogramming and just-in-time compilation to generate custom low overhead load and store loops. Memory can be accessed either sequentially or pseudo-randomly. When accessed pseudo-randomly, we ensure that each addresses is touched exactly once (i.e. no repeats) using a maximum length Linear Feedback Shift Register to generate array indices. Furthermore, for pseudo-random iteration, access granularity ranges from 64 B to 512 B. We found sequential iteration is largely indifferent to access granularity, so only a single result for sequential access is reported. For these experiments, we used read-only, write-only, and read-modify-write operations. We explore both *standard* or *nontemporal* instructions for all stores. *Nontemporal* stores bypass the on-chip cache, allowing us to directly study the behavior of LLC writes to the memory controller. Data is partitioned evenly across threads when multithreading is used.

To measure DRAM and NVRAM traffic, we use uncore hardware performance counters located in each IMC. These counters capture column access strobes (CAS) for DRAM reads and writes. The Cascade Lake generation added IMC counters for NVRAM read and write requests, and 2LM tag statistics including tag hit, tag miss clean, and tag miss dirty,

[3]https://github.com/darchr/KernelBenchmarks.jl




(a) Read bandwidth using *standard* load instructions.




(b) Write bandwidth using *nontemporal* store instructions.

Fig. 2. Bandwidth to 6 interleaved 512 GiB NVRAM DIMMs.

which will be explained in more detail later. Results from the hardware performance counters are validated with the expected data movement and benchmark wall clock time.

Each benchmark was executed on a quiet system. Unless otherwise specified, all six Optane DC DIMMs are configured as a single interleaved set and experiments are run on socket 1 to avoid NUMA overheads.

### C. NVRAM Performance Results

The results obtained here are in line with observations made by other researchers [18], [26], [39], [43]. We highlight results that are relevant to our upcoming discussion in Section IV on the 2LM DRAM cache. Since read and write bandwidth to Optane DC is asymmetric, we will consider these separately. Figure 2a shows the read bandwidth of six interleaved 512 GB NVRAMs under varying thread counts. Sequential bandwidth scales with the number of threads up to a maximum 30 GB/s with 8 threads, at which it stops increasing. This result is slightly different than the 39 GB/s reported in other works [26] because our system uses 512 GiB DIMMs instead of 128 GiB or 256 GiB DIMMs. The 512 GiB DIMMs provide a maximum read bandwidth of 5.3 GB/s read bandwidth per DIMM while the others provide 6.8 GB/s [9].

Figure 2b demonstrates the write bandwidth of NVRAM when using *nontemporal* stores. In addition to bypassing the on-chip cache, *nontemporal* stores do not need a Read-For-Ownership (RFO), a step in Intel's usual cache coherence protocol [10], and are critical for high NVRAM write bandwidth [51]. Write bandwidth peaks with four threads, and is roughly the same for sequential and random access exceeding 256 B. Limited buffer space within the Optane DIMM decreases the media controller's ability to merge sequential 64 B writes into a single 256 B write, leading to write amplification and the observed drop in bandwidth [51].

In summary, with this system we can achieve just over to 30 GB/s read and 11 GB/s write to NVRAM.

TABLE I
SUMMARY OF GENERATED READS AND WRITES FOR 2LM. FIGURE 3 SHOWS DETAILS OF WHY THESE REQUESTS GENERATE THESE ACTIONS. THE DIRTY DATA OPTIMIZATION (DDO) ALLOWS THE IMC TO ELIDE THE TAG CHECK FOR SOME WRITES.

| | LLC Read | | | LLC Write | | | |
|---|---|---|---|---|---|---|---|
| | Hit | Miss | | Hit | Miss | | DDO |
| | | Clean | Dirty | | Clean | Dirty | |
| DRAM Read | 1 | 1 | 1 | 1 | 1 | 1 | |
| DRAM Write | | 1 | 1 | 1 | 2 | 2 | 1 |
| NVRAM Read | | 1 | 1 | | 1 | 1 | |
| NVRAM Write | | | 1 | | | 1 | |
| Amplification | **1** | **3** | **4** | **2** | **4** | **5** | **1** |

## IV. DRAM CACHE / 2LM MODE

Intel Cascade Lake chips support a 2LM mode, where the Optane DIMMs act as system memory and DRAM serves as a transparent, hardware managed, direct-mapped cache [26].

The access granularity of this cache is 64B, matching the cache line size of the underlying CPU. While not mentioned explicitly, Intel patents suggest that cache tags are stored along with ECC data [42]. ECC DRAM is implemented by adding an extra DRAM module to each DIMM. Thus, each 64B data transaction for each DIMM is accompanied by 8B (64 bits) of ECC. Of these 64 bits of ECC data, only 20 [5] are required to provide Single Error Correction/Double Error Detection redundancy, leaving ample room for tag metadata, including both physical address and cache line state. Our data is consistent with this approach.

In this section, we us microbenchmarks to try to deduce the performance implications of the Cascade Lake DRAM cache design. Our results are summarized in Table I and Figure 3.

### A. Methodology

To study the behavior of the 2LM DRAM cache, we used the same benchmarks discussed in Section III and the same methodology for measuring bandwidth. In this case, data gathered from the performance counters allows us to differentiate DRAM and NVRAM traffic. Furthermore, the tag related performance counters in each IMC allows us to correlate tag events with memory traffic. Each IMC only allows four events types to be recorded at a time. Since our benchmarks are long running and largely deterministic, we run them twice to obtain both bandwidth and tag events.

Table I summarizes the observed actions required for each type of access to the IMC. We define two types of requests to the IMC. An *LLC Read* is a request from the LLC for data from the DRAM cache or NVRAM. This request is generated on a load or store miss at the LLC. Stores can generate an LLC read as they may require a RFO. An *LLC Write* is a request from the LLC to write back dirty data to the DRAM cache. LLC write requests are generated either when a dirty line is evicted from the LLC or from a *nontemporal* store.

Furthermore, the hardware performance counters differentiate between three different types of cache accesses: *hit*, *clean miss*, and *dirty miss*. A *hit* implies that address accessed by an LLC request is present in DRAM. A *miss* means that an

address is *not* resident in DRAM and must be fetched from NVRAM. Since this cache is direct mapped, a miss implies that some other data is occupying the set corresponding to the requested address. A miss is *dirty* if this aliasing data has been modified since its original insertion and thus must be written back to NVRAM upon eviction.

To study read and write hits, we use the read-only and write-only benchmarks respectively on a 51 GiB array backed by 1 GiB hugepages to mitigate TLB overheads. Because the array is far larger than the 33 MB LLC cache, each CPU load generates an LLC read and each CPU *nontemporal* store generates an LLC write. This array is also small enough to fit in the DRAM cache without aliasing. Thus, all LLC reads/writes accesses will be cache hits.

Generating clean LLC read misses and dirty LLC write misses is also straightforward. We use a 420 GB array, which is over twice the size of the 192 GB DRAM cache per socket. Applying the read-only benchmark to this array for several iterations ensures a clean LLC read misses for each CPU load. Similarly, the write-only benchmark ensures that each *nontemporal* store generates a dirty LLC write miss.

Testing dirty LLC read misses and clean LLC write misses is more complicated. For dirty LLC read misses, we first prepare the 420 GB array from before by writing to it, making the entire DRAM cache is dirty. We then perform a single iteration of the read-only kernel. Thus, each CPU load early in the iteration generate LLC reads that will be a dirty miss in the cache. As the iteration progresses, however, a larger portion of these loads become clean misses as the dirty cache is replaced by clean data. Consequently, we determine cache behavior based on data collected early in the iteration. We use a similar procedure to prime and test clean LLC write misses.

When testing the behavior of the cache, we use *nontemporal* stores when writing. This ensures that the behavior shown by the IMC is purely the result of the incoming store and not an earlier RFO. For all benchmarks, we also compute an *effective* bandwidth as seen by the application. This is obtained using the size of the array and wall clock time for each benchmark.

While we only outlined several key benchmarks to test the different regimes of the DRAM cache, we also applied a whole range of microbenchmarks with different thread counts and access patterns to fully characterize the behavior of the cache and validate the results presented here.

### B. 2LM Observations

Table I summarizes our findings for the cache events and Figure 3 demonstrates a flow chart of IMC logic that models this behavior. We describe each of these columns in turn. To help with our discussion, we use the term **access amplification** [32] as the ratio of *memory* accesses (i.e., both DRAM and NVRAM) to demand accesses.

LLC read hits are simple. The IMC initiates a DRAM read, which fetches data along with thee tag in the ECC bits. A tag check is performed and since the tag matches, the data is immediately forwarded with no access amplification.

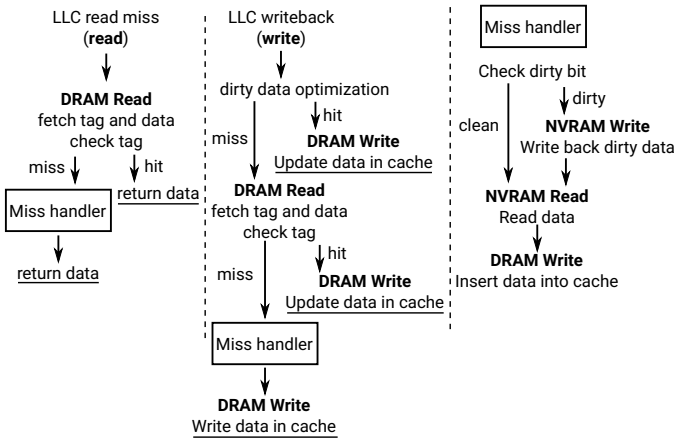



Fig. 3. Flowchart showing the operation of the DRAM for LLC read misses which occur on a processor load or store which misses in the LLC and LLC writebacks which occur when a dirty block is evicted from the LLC. The miss handler is the same for reads and writes and is factored out on the right. Underlines indicate where the actions end, and bold shows the hardware actions. A summary of total memory accesses is given in Table I.

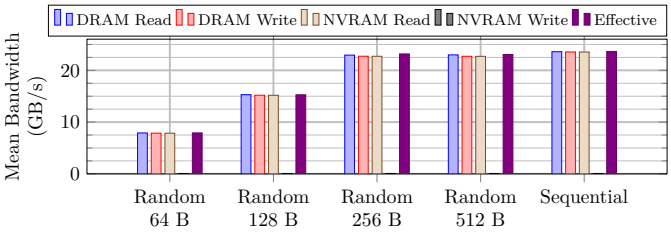
Figure 4a shows bandwidth for the read-only benchmark in the 100% clean miss scenario. Note a 3× access amplification for each miss. Essentially, the tag miss is serviced by a miss handler, which fetches the requested cache line from NVRAM, inserts into DRAM, and forwards to the CPU. Dirty read misses are handled much the same as clean read misses. The only change is that the cache line evicted from DRAM must be written back to NVRAM.

LLC write hits incur a 2× access amplification because the IMC must first emit a DRAM read to perform a tag check. Only upon verification of the tag can the line be safely written.

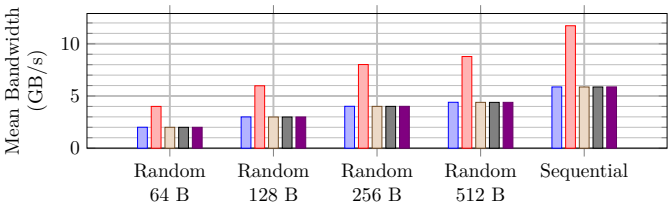
Next, we discuss dirty LLC write misses. Figure 4b shows collected bandwidth for the write-only benchmark where each *nontemporal* store is a dirty tag miss. Observe a 2× access amplification in DRAM writes alone. Upon receiving a completely dirty cache line store yielding a tag miss, we would expect the IMC to write the evicted line to NVRAM and directly insert the incoming line to DRAM. This would yield a total of 1 DRAM read (for the tag check), 1 NVRAM write, and 1 DRAM write. However, the data in Figure 4b suggests that this is not the case. Our best guess is that the memory controller *always* inserts on a miss (regardless of whether that miss was a read or write). The second DRAM write is thus the actual write of cache line to DRAM. Clean LLC write misses are similar dirty write misses without the NVRAM write back.

### C. Dirty Data Optimization

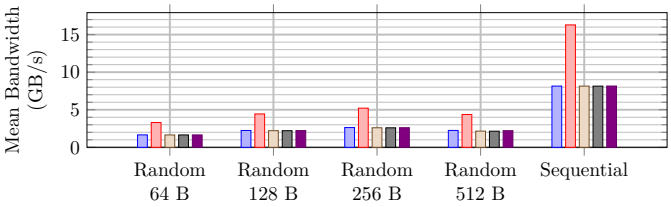
Finally, this brings us to the phenomenon that we call the Dirty Data Optimization (DDO). At times, the memory controller is able to elide the tag check (i.e. DRAM read) and instead directly forward LLC writes to DRAM. This can be seen in Figure 4c which shows the distribution of traffic for the read-modify-write benchmark in a 100% dirty LLC miss scenario using *standard* stores. The CPU load initiates a dirty LLC read miss (dirty from a previous write), accounting for




(a) *Read-only* benchmark, clean LLC read misses, 24 threads.




(b) *Write-only* benchmark, dirty LLC write misses, 24 threads, *nontemporal* stores. Using 4 threads only increases the maximum write bandwidth by 1 GB/s.




(c) *Read-modify-write* benchmark, dirty LLC read miss followed by a later DDO LLC write, 4 threads, *standard* stores. Sequential achieves the highest NVRAM write bandwidth of any 2LM benchmark with neglibile difference between *nontemporal* and *standard* stores.

Fig. 4. Benchmark results on a large array exceeding the size of the DRAM cache. Because the array size exceeds DRAM, the miss rate in the DRAM cache is 100%. The "effective" bar illustrates performance as seen by the application, computed by wall clock time and data accessed.

one DRAM read (tag check) plus the traffic associated with a cache insert. Since *standard* stores are used, the subsequent CPU store will remain in the LLC for some time before being evicted and written to memory. Thus, there is low temporal locality between a cache line's load and its write back.

Due to this low locality, we would *expect* this delayed LLC write to require another tag check, resulting in a total of two DRAM reads per CPU load-store pair. However, this is not the case and it appears this second tag check is elided. While this could be explained by an inclusive cache, we found that this is not the case as it is possible to have small amounts ($< 8$ KiB) of aliasing data simultaneously within the CPU cache. Thus, we are not sure the exact mechanism driving this optimization.

### D. Discussion

We described our observation of 2LM's mechanics, but what does this mean for user applications? There are two points we want to make. First, contrast Figure 4, which shows the effective NVDIMM bandwidth in 2LM with a high miss rate, with Figures 2a and 2b, showing the maximum speed of NVRAM. The highest NVRAM read bandwidth in 2LM (Figure 4a) is 23 GB/s and the highest write bandwidth (Figure 4b) is 8 GB/s. This is 60% and 72% the demonstrated achievable bandwidth of our system's NVRAM. This is the ideal case with well formed traffic. We expect applications with a large memory footprint (exactly those that would benefit from the large memory pool provided by NVRAM) and a high DRAM cache miss rate to experience a severe bandwidth bottleneck. Second, cache misses are costly in terms of extra traffic generated, with LLC read and write misses generating up to $3\times$ and $5\times$ access amplification. This is costly both in terms of energy and lost bandwidth.

So far, we have demonstrated the potential for applications to experience bandwidth bottlenecks in 2LM. In the next two sections, we provide case studies demonstrating this effect on real applications.

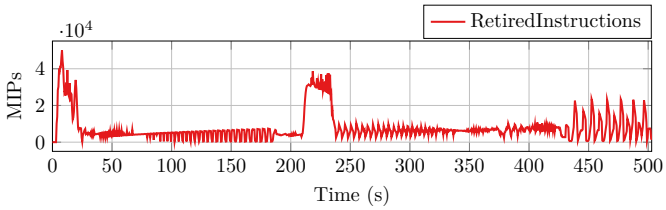
## V. Case Study 1: Convolutional Neural Networks

In this section, we will take a deep dive into some of pitfalls a bandwidth and compute heavy application can fall into when running under 2LM. Specifically, we consider the problem of training deep Convolutional Neural Networks (CNNs) whose working set size greatly exceeds the physical DRAM of a system, requiring the extra memory provided by NVRAM.

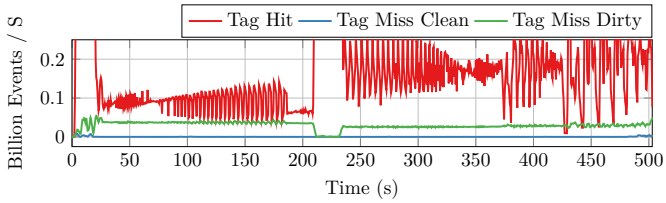
CNNs are typically expressed as a directed acyclic graph of computation primitives such as convolutions and matrix multipications, that are heavy on compute, and operations such as batch normalization and concatenation that are heavy on bandwidth requirements. At a high level, a single *iteration* of training consists of a *forward* pass, during which the network is evaluated (almost) normally on a batch of training data (some kernels like Batch Normalization have slightly different versions for training and inference [25]). The output of the forward pass is compared to an expected output to generate a loss value, which is used in the backward pass to compute the partial derivative of the loss with respect to each of the trainable parameters of the network. The parameters of the network are adjusted based on the these derivatives. An important aspect of the backpropagation algorithm is that many intermediate values computed during the forward pass must be preserved to compute the backward pass. Thus, the active memory footprint of the network during an iteration of training increases during the forward pass, then decreases during the backward pass. It takes many such iterations of training across different input samples to fully train a CNN.

### A. Methodology

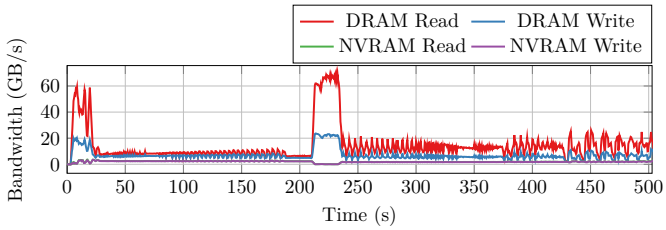
We implemented three popular large CNNs: Inception v4 [48], Resnet 200 [21], and DenseNet [23] using the *ngraph* compiler [11] on the NVRAM-based system described earlier. Intel's *ngraph* compiler is an optimizing compiler specifically targeting static deep neural networks that takes advantage of the Xeon CPU ISA. For these large networks, we scaled the training batch size until the overall footprint of these applications exceeded 650 GB, well beyond the capacity of the DRAM cache. All networks were run on a single NUMA node and assigned all 24 physical cores on that node with no hyper-threading. These networks were run for two warm



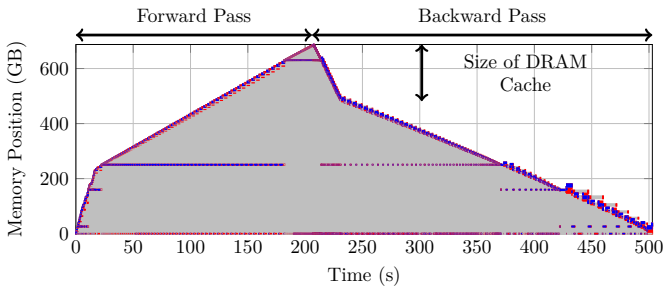

(a) System MIPS.




(b) DRAM cache statistics.




(c) Memory bandwidth through time. NVRAM read and write bandwidths are similar, thus the NVRAM read line is hidden behind the NVRAM write line.




(d) Live memory in the *ngraph* heap. Memory that is highlighted *gray* indicates memory that will be read before written (i.e., live memory). *Blue* indicates a write is happening. *Red* indicates a read is happening. *White* shows memory that will be written before read.

Fig. 5. Memory behavior of a single iteration of training for DenseNet 264 with a batchsize of 3072.

up iterations to trigger on-demand paging by the OS and to prepare the state of the DRAM cache.

During the execution of these networks, we sampled hardware performance counters for bandwidth and tag statistics. Furthermore, we modified the *ngraph* compiler in two ways. First, we added an option to emit high resolution timestamps when beginning the execution of each compute kernel, allowing us to correlate these events with performance counter data. Second, we exposed information regarding the memory assignment of intermediate tensors. This allows us to examine which regions of memory are being accessed throughout the network execution.

### B. Results

For the deep dive, we present the results for DenseNet [23], a CNN with a complicated dataflow pattern. In Figure 5 we break down the bottlenecks of a single iteration of training for DenseNet 264 with batchsize 3072. The baseline memory footprint for this application is around **688 GB**. Figure 5a demonstrates the system's retired instruction rate through time. Figure 5b shows the number of tag hits, dirty tag misses, and clean tag misses throughout the iteration.

Key observations to make are: (1) there very few clean tag misses, (2) there is a high percentage of dirty tag misses, both in the forward pass and the backward pass, and (3) there noticeable regions of high tag hits at the beginning of the forward and backward passes with a corresponding drop in dirty tag misses. Finally, Figure 5c breaks down the read and write bandwidths to DRAM and NVRAM. Regions of high dirty miss rate correspond to low bandwidth and instruction throughput. Reasonable system performance is only achieved when the hit rate is high.

So, a good question at this point is - *Why are so many dirty tag misses generated, and why are there regions of high cache hit rate?* Two related phenomena can explain this.

Figure 5d shows the memory usage of DenseNet through time for a single iteration of training. Before execution, the *ngraph* compiler allocates a single buffer for the *entire* network. The offset from the base of this buffer is shown on the vertical axis of Figure 5d. The change in memory state through time is shown using different colors. The color *white* indicates that the region of memory is **free** (semantically speaking). That is, it will always be written to before it is read by the program. A *blue* highlight indicates that a region of memory is being actively written to, *red* indicates a read, and *gray* indicates that the memory will be read from in the future.

For an iteration of training, first the *forward* pass of the model is computed (up to time around 220, annotated in Figure 5d). Throughout the forward pass, some of the generated intermediate tensors must be held in memory to facilitate computation of the backward pass. Thus, the amount of live memory (gray) accumulates through the forward pass. Once a preserved tensor is used on the backward pass, the region in memory where it was stored is free for further use (white). The *ngraph* compiler takes advantage of this newly freed area to allocate intermediate tensors required to compute the backward pass. This is the very subtle streak of blue on the right shoulder of Figure 5d.

However, from the perspective of the 2LM cache, the fact that writes are occurring to a region of memory on the backward pass makes memory is dirty with respect to the DRAM cache. Hence, even when this region of memory is semantically free from the program's perspective, the cache must still generate a dirty write back upon eviction. *Because the DRAM cache is unaware of the meaningful lifetime of memory, it generates a large amount of unnecessary traffic.*

Finally, the regions of high DRAM cache hit rate occur at the beginning of the forward and backward pass because the area of active memory folds back on itself. Recent data is in

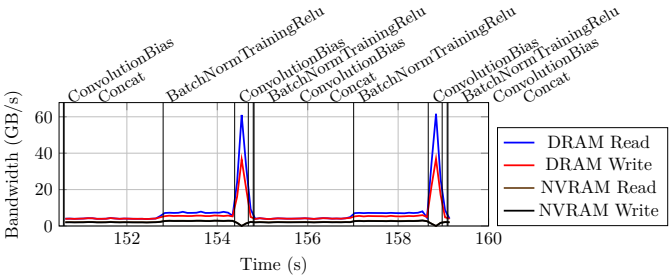



Fig. 6. Snapshot of periodic bandwidth behavior during the forward pass of training DenseNet 264 in 2LM. Vertical bars mark the start of kernel execution. Very short running kernels have been excluded for clarity.

the cache, so all accesses are cache hits. This continues until the entire cache has been read, at which point further accesses are cache misses.

### C. Problematic Kernels

To wrap up this section, we will explain the relatively high frequency periodic behavior that is noticeable in the Tag Hit line of Figure 5b. DenseNet is composed of a linear chain of "dense blocks" where each dense block consists of a sequence of Concat, BatchNorm, Conv, BatchNorm, and Conv operators. Figure 6 shows a high resolution snapshot of the bandwidth for two such dense blocks during the forward pass of DenseNet. The point where kernels begin execution is annotated on the graph. The main performance bottlenecks apparent in Figure 6 are Concat and BatchNorm. These are both memory-bound kernels with little data reuse and are more affected by the low bandwidth associated with a high dirty tag miss rate. The second BatchNorm within each dense block operates on much smaller intermediate tensors, and is thus less impactful on overall performance. Similar problematic kernels exist on the backwards pass as well, including BatchNorm-Backprop and the back-propagation kernels for the filter/bias inputs of $3x3$ convolutions.

### D. Discussion

In summary, the overall performance of CNN training in 2LM mode in NVRAM-based systems is affected by two factors: (1) low effective bandwidth with a high miss rate and (2) a significant amount of unnecessary dirty writebacks. From the microbenchmarks, the first of these is not too surprising. However, the second exposes exposes a performance pathology *not* demonstrated by the microbenchmarks, made worse by the relatively low write bandwidth of NVRAM. Next, we will look at a different class of algorithms that suffer similarly.

## VI. Case Study 2: Graph Processing

In this section, we perform a preliminary study on applications known for having diverse performance characteristics and irregular memory access patterns. To accomplish this, we evaluate a variety of graph processing algorithms on large real world graph inputs using Galois [34], a high performance shared memory graph analytics framework.

### A. Background

Large graph processing has garnered substantial research interest across a variety of use cases, including the identification of social media influencers and decision makers, or finding fraudulent actors within a business network. These real world large systems require frameworks process representative graphs with tens of billions of nodes and trillions of edges, incurring a high memory footprint that is expensive to accommodate in DRAM. Depending on the topology of the input graph and the processing algorithm being used, the memory access pattern can vary wildly. This presents a challenge when optimizing such workloads for systems with limited main memory.

To address this issues, several efforts [14], [18] have explored leveraging NVRAM for graph analytics on a single machine. However, such works focused on performing an analysis and comparison of different graph processing frameworks and system settings to optimize the use of Optane for graph workloads. Here, we evaluate the bandwidth characteristics of such irregular workloads in 2LM.

### B. Methodology

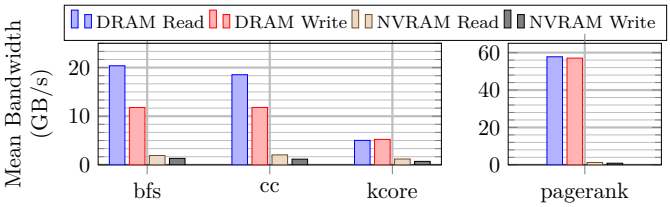
Graph kernel experiments were run on the shared memory graph analytics framework Galois. Specifically, our evaluations consisted of 4 benchmarks from the lonestar suite: breadth-first search (bfs) [8], connected components (cc) [44], [46], k-core decomposition (kcore) [12], and pagerank-push (pr) [37]. These kernels were chosen based on their diverse execution characteristics [1]. Our workloads were run with the settings by Gill et al. [18]. For *bfs*, the source node was the maximum out-degree node. The tolerance of pr was set to $10^{-6}$ and we used the $k = 100$ for kcore. Each kernel ran until convergence, except for pr which ran for 100 rounds.

We used two realistic unweighted massive input graphs: wdc12 [36], the largest publicly available graph, and kron30 [30], a randomized scale free graph generated using a graph500 based kronecker generator [19]. Each were chosen to highlight the differences between when a graph fit and did not fit in the DRAM cache. While these graphs have different structures, we can still draw conclusions from kernels' relative performance on these graphs. Both were processed using the provided graph-converter in Galois and resulted in binaries of size 507 GB and 73 GB respectively.

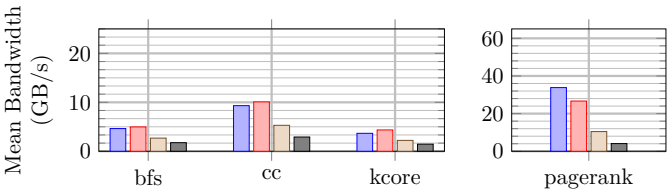
In 2LM, all benchmarks were run on two NUMA nodes and assigned all 96 threads. Since two sockets are used, the size of the DRAM cache is effectively doubled to 384 GB with 6 TB of NVRAM. The total NUMA interleaving and 2 MiB hugepages were used with no page migration to maximize performance [18].

To find the baseline data movement required by the algorithms, we configured the NVRAM regions on each socket as extra NUMA nodes. This is facilitated through the `daxctl`[4] tool with the machine in 1LM. Since Galois uses a NUMA preferred policy, the threads on each socket will initially

[4]https://docs.pmem.io/ndctl-user-guide/daxctl-man-pages

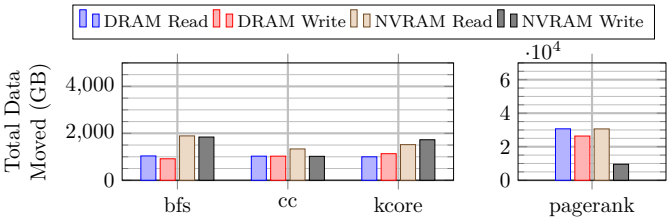



(a) Performance of graph kernels on *kron30* which fits in DRAM

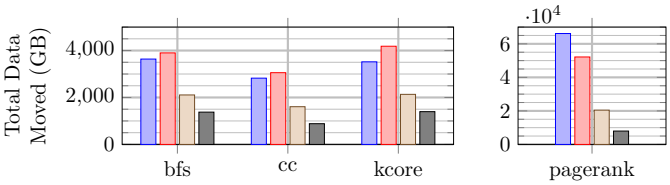



(b) Performance of graph kernels on *wdc12* which exceeds DRAM capacity

Fig. 7. Graph kernel performance in 2LM run on 96 threads. When the input graph does not fit in the DRAM cache, bandwidth significantly drops.




(a) NVRAM as extra NUMA nodes.




(b) NVRAM as system memory with a DRAM cache.

Fig. 8. Total amount of data moved during the execution of a graph kernel when the input graph does not fit in the DRAM cache.

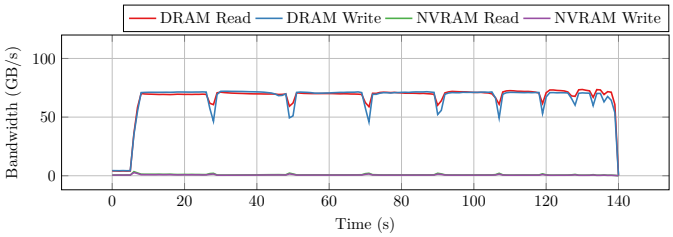
allocate memory on that socket's DRAM. When DRAM is exhausted, further allocations are serviced by NVRAM. By summing the traffic to DRAM and NVRAM, we can establish the baseline memory traffic required by each application.

As with our previous experiments, measurements on bandwidth and tag statistics were gathered using hardware performance counters.

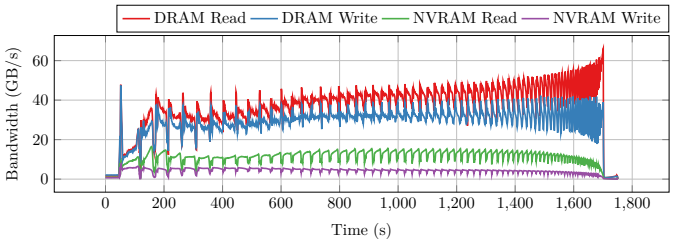
### C. Results

Figure 7 compares the observed bandwidth when running the graph kernels on *kron30* and *wdc12*. When processing *kron30*, the kernels have a working set that largely fits within the DRAM cache while the working set when processing *wdc12* greatly exceeds the DRAM cache. When the working set does not fit in the DRAM cache, there is a significant decrease in DRAM bandwidth during an algorithm's execution.

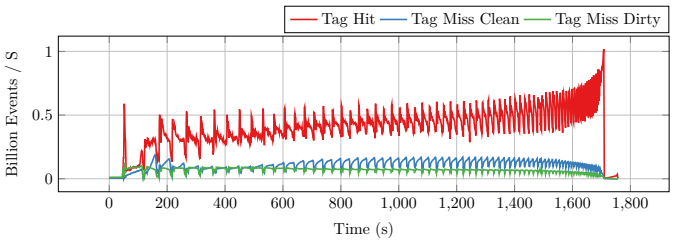
Figure 8 shows the total amount of data moved in the NUMA and 2LM configurations for NVRAM. Since page




(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*.

Fig. 9. Traces for the *pagerank-push* algorithm. Figure 9a demonstrates behavior when the graph largely fits within the DRAM cache. Conversely, Figures 9b and 9c shows behavior when the working set greatly exceeds the DRAM cache.

migration was disabled, Figure 8a shows the true demand accesses of the workload. Comparing this with Figure 8b we see significant access amplification.

Figure 9 shows the workload characteristics of the pagerank-push algorithm for both *kron30* and *wdc12*. Figure 9a 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 9b demonstrates the bandwidth of pagerank-push when its working set *does not* fit in the DRAM cache. Not 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 9c show the presence of both clean and dirty tag misses as well as the correlation between hit rate and DRAM bandwidth.

### D. Discussion

As with CNN training, large scale graph processing is a workload with a high DRAM cache miss rate. This is made worse since traditional graph algorithm implementations involve mutating the in-memory representation of the graph [18]. In 2LM, this mutation will mark the corresponding memory as dirty. Thus, not only is the miss rate high, but many of

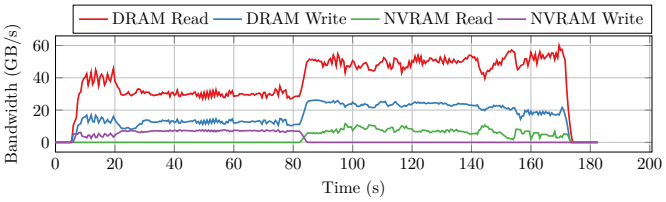



Fig. 10. Memory bandwidth under AutoTM. Samples are averaged over a 2.5 second sliding window to filter high frequency components.

these misses require NVRAM write backs, which we have demonstrated to be inefficient. As a result, it is not surprising that 2LM behaves poorly for these particular implementations.

## VII. Discussion and Mitigation Strategies

In this paper, we demonstrated that the DRAM cache as currently implemented in Intel's Cascade Lake systems performs poorly for applications with a high miss rate. We showed that a DRAM cache miss can cause 3–5× more memory accesses than the original demand requests. Further, we showed that this causes performance degradation in two bandwidth-limited workloads: CNN training and graph analytics which are important use cases for NVRAM since they have extremely large memory footprints. Furthermore, we show that certain data reuse semantics at the program level can cause severe degradation.

For instance, in the deep neural network training workload, a significant amount of the data movement from the DRAM cache to NVRAM is *useless* as this data was only meant to be used temporarily by the program and will be overwritten before it is read again. This dirty temporary data dominates the DRAM cache leading to more misses than necessary and limiting performance to the smaller NVRAM write bandwidth.

### A. Software-managed multi-level memory

So what can be done about this? In this section, we look at an example of software-managed memory for each of the case studies presented previously: CNNs training and graph analytics. We show that through software-managed memory, we can obtain better performance than using the hardware-managed cache in 2LM mode for these miss heavy bandwidth-bound workloads.

Software management relies on decoupling the DRAM and NVRAM memory pools. So far, this paper focused on the 2LM (or "memory mode") of the NVRAM systems, these systems can also be configured in "app-direct mode" or 1LM where the programmer has full control over the data location and movement. NVRAM is simply mapped into a program's address space.

*1) CNN Training:* Hildebrand et al. showed that for static compute graphs such as static CNNs, where there is no data dependent behavior and the structure of the network and sizes of intermediate tensors are fully known ahead of time, that software data movement can provide a significant performance boost over hardware management [22]. This work, AutoTM, does so by using an integer linear programming and a profile guided optimizer. AutoTM understands the execution time of kernels with input and output tensors in various combinations of DRAM and NVRAM and can manage these locations and data movement to minimize execution time under a set DRAM budget. With this knowledge, AutoTM achieves a $1.88\times$, $2.24\times$, and $3.10\times$ speedup over 2LM for Inception v4, ResNet 200, and DenseNet 264 respectively [22].

First, AutoTM is aware of the difference between semantically *live* data versus *dead* data and thus elide the unnecessary dirty write-backs on the backward. This can be seen in Figure 10, which shows the trace of bandwidth through out a single iteration of training for the large DenseNet model under AutoTM. Contrast this with Figure 5c. AutoTM only generates NVRAM writes during the forward pass (where it is storing intermediate activations for use on the backward pass). Similarly, AutoTM only generates NVRAM reads during the backward pass. Table II compares the total amount of data moved for these workloads in 2LM and under AutoTM. AutoTM generates similar amounts of DRAM traffic, but only 50% to 60% of the NVRAM traffic.

The average read and write bandwidth that AutoTM achieves is to NVRAM is also significantly higher than that achieved during 2LM. This is because AutoTM is designed to read and write to NVRAM in the patterns discussed in Section III for achieving high bandwidth. However, the average bandwidth in Figure 10 does not tell the whole story. Under AutoTM, tensors are usually moved between DRAM and NVRAM (and vice versa) synchronously between compute kernel execution. Therefore, during kernel execution, there is no data movement. Thus, we are seeing the bandwidth averaged over times of data movement and times of no data movement, implying the active bandwidth is much higher.

*2) Graph Analytics:* As pointed out in Section VI, graph algorithm implementations in Galois and other graph frameworks often mutate graph data structure. With NVRAM, this is an issue due its low write bandwidth (which is further exacerbated by 2LM's write amplification). To tackle this issue, the authors of Sage [18] designed that software specifically with NVRAM in mind. Their key approach is to (as much as possible) use NVRAM for read only data.

When running algorithms that require tracking state (such a nodes visited for bfs), an auxiliary DRAM-based data structure is used. This data structure is greatly compressed and supplements the read-only NVRAM-based adjacency list. Mutation is only performed on the auxiliary data structure, and hence write traffic is only generated to DRAM. To optimize for multiple sockets, Sage takes advantage of NVRAM's capacity to keep a full copy of the graph on both CPU sockets. With these techniques, they were able to design algorithms $1.87\times$ faster on average than GBBS and $1.94\times$ faster on average than Galois in 2LM [18].

This is another example demonstrating the clever software management can over come the bandwidth limitations of NVRAM. Conversely, these same limitations are exacerbated by access amplification caused by the DRAM cache.

TABLE II
COMPARISON OF DATA MOVED AND EXECUTION TIME FOR THREE COMMON CNNS RUNNING IN 2LM AND UNDER AUTOTM. ALL DRAM AND NVRAM VALUES ARE IN GB.

| | **2LM** | | | | | **AutoTM** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | DRAM Read | DRAM Write | NVRAM Read | NVRAM Write | Runtime (s) | DRAM Read | DRAM Write | NVRAM Read | NVRAM Write | Runtime (s) |
| Inception v4 | 8338 | 4254 | 1019 | 919 | 572 | 8103 | 3459 | 543 | 473 | 304 |
| Resnet 200 | 8565 | 3914 | 950 | 903 | 514 | 8565 | 3316 | 652 | 467 | 229 |
| DenseNet 264 | 7418 | 3559 | 1027 | 969 | 524 | 7419 | 2947 | 639 | 510 | 169 |

### *B. Limitations of software approaches and future directions*

Even though the software approaches discussed above provide some mitigation to the problems of hardware-managed DRAM caches, these approaches have limitations. These approaches use the CPU cores to move data via loads and non-temporal stores. The DMA copy engines in current systems are designed for I/O data movement and not high bandwidth movement between different memory technologies. These DMA devices' programming models and performance characteristics do not fit the requirements of this data movement. Additionally, because these approaches use CPUs for data movement it is difficult to transfer data asynchronously.

Looking forward, future research should concentrate on providing hardware-software co-design for data movement between NVRAM and DRAM. If software, with its high level knowledge of data access patterns, could work *with* the hardware, then we could realize the benefits of hardware acceleration without the limitations presented above.

## ACKNOWLEDGMENTS

This work is supported in part by the Intel Corporation and by the National Science Foundation under Grant No. CNS-1850566.

We would also like to thank our anonymous reviewers and members of the Davis Computer Architecture Research Group (DArchR) for their valuable feedback.

## REFERENCES

[1] Scott Beamer. *Understanding and improving graph algorithm performance*. PhD thesis, UC Berkeley, 2016.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 648–658. IEEE, 2016.

[5] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[6] C. Chou, A. Jaleel, and M. K. Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 198–210, 2015.

[7] C. C. Chou, A. Jaleel, and M. K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12, 2014.

[8] Thomas Cormen, Charles Leiserson, Ronald Rivest, and editors Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[9] Intel Corporation. Optane dc persistent memory brief.

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, August 2016.

[11] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[12] N. S. Dasari, R. Desh, and M. Zubair. Park: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 9–16, 2014.

[13] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proceedings of the VLDB Endowment*, 13(9), 2020.

[14] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *Proc. VLDB Endow.*, 13(9):1598–1613, May 2020.

[15] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[16] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. *Proceedings of Machine Learning and Systems*, 1:40–52, 2019.

[17] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proceedings of the VLDB Endowment*, 13(8):1304–1318, 2020.

[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(8):1304–1318, April 2020.

[19] GitHub. Graph500. https://github.com/graph500/graph500, 2019.

[20] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[22] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 875–890, New York, NY, USA, 2020. Association for Computing Machinery.

[23] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[24] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[26] 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. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[27] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–37, 2014.

[28] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. *SIGARCH Comput. Archit. News*, 41(3):404–415, June 2013.

[29] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 211–222, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, March 2010.

[31] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–464, 2011.

[32] Jason Lowe-Power. *On Heterogeneous Compute and Memory Systems*. PhD thesis, University of Wisconsin, Madison, 2017.

[33] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[34] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweigth infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.

[36] Alexander Outman. Web data commons - hyperlink graphs. Technical report, 2017.

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[38] Wen Pan, Tao Xie, and Xiaojia Song. Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 921–931. IEEE, 2019.

[39] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 288–303, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[41] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246, 2012.

[42] Raj K. Ramanujan, Rajat Agarwal, and Glenn J. Hinton. Apparatus and method for implementing a multi-level memory hierarchy having different operating modes, February 2 2017.

[43] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. Large-scale in-memory analytics on intel® optane™ dc persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Linda G. Shapiro. Connected component labeling and adjacency graph construction. In T. Yung Kong and Azriel Rosenfeld, editors, *Topological Algorithms for Digital Image Processing*, volume 19 of *Machine Intelligence and Pattern Recognition*, pages 1 – 30. North-Holland, 1996.

[45] Yishu Shen and Zhaonian Zou. Efficient subgraph matching on non-volatile memory. In *International Conference on Web Information Systems Engineering*, pages 457–471. Springer, 2017.

[46] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[47] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[48] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[49] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[50] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jisen Zhano. Characterizing and modeling non-volatile memory systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[51] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.