Exploiting NVM in Large-scale Graph Analytics

Jasmina Malicevic[1] Subramanya Dulloor[2] Narayanan Sundaram[2] Nadathur Satish[2]
Jeff Jackson[2] Willy Zwaenepoel[1]

EPFL, Switzerland[1] Intel Labs[2]

jasmina.malicevic@epfl.ch
{subramanya.r.dulloor, narayanan.sundaram, nadathur.rajagopalan.satish, jeff.jackson}@intel.com
willy.zwaenepoel@epfl.ch

Abstract

Data center applications like graph analytics require servers with ever larger memory capacities. DRAM scaling, however, is not able to match the increasing demands for capacity. Emerging byte-addressable, non-volatile memory technologies (NVM) offer a more scalable alternative, with memory that is directly addressable to software, but at a higher latency and lower bandwidth.

Using an NVM hardware emulator, we study the suitability of NVM in meeting the memory demands of four state of the art graph analytics frameworks, namely Graphlab, Galois, X-Stream and Graphmat. We evaluate their performance with popular algorithms (Pagerank, BFS, Triangle Counting and Collaborative filtering) by allocating memory exclusive from DRAM (DRAM-only) or emulated NVM (NVM-only).

While all of these applications are sensitive to higher latency or lower bandwidth of NVM, resulting in performance degradation of up to $4\times$ with NVM-only (compared to DRAM-only), we show that the performance impact is somewhat mitigated in the frameworks that exploit CPU memory-level parallelism and hardware prefetchers.

Further, we demonstrate that, in a *hybrid* memory system with NVM and DRAM, intelligent placement of application data based on their relative importance may help offset the overheads of the NVM-only solution in a cost-effective manner (i.e., using only a small amount of DRAM). Specifically, we show that, depending on the algorithm, Graphmat can achieve close to DRAM-only performance (within $1.2\times$) by placing only 6.7% to 31.5% of its total memory footprint in DRAM.

Parameter	3D-DRAM	DDR-DRAM	NVM
Capacity per CPU	10s of GBs	100s of GBs	Terabytes
Read Latency	$\frac{1}{2}\times$ to $1\times$	$1\times$	$2\times$ to $4\times$
Write bandwidth	$4\times$	$1\times$	$\frac{1}{8}\times$ to $\frac{1}{4}\times$
Estimated cost	-	$5\times$	$1\times$
Endurance	10^{16}	10^{16}	10^{6} to 10^{8}

Table 1: Comparison of memory technologies [3, 5, 20, 27]. NVM technologies include PCM and RRAM [3, 27]. Cost is derived from the estimates for PCM based SSDs in [17].

1. Introduction

The availability of graph structured data has resulted in vast interest from the research community in how to mine this data. From a systems perspective, the irregular access pattern of many of the graph algorithms has led to DRAM becoming the chosen medium for data placement.

However, the capacity scaling of DRAM is not able to match the requirements for analytics over graphs with trillions of connections. In order to overcome this limitation of DRAM, existing systems either a) scale-out to multiple machines [15] or b) scale-up by storing and computing from secondary storage such as SSDs or magnetic disks [28], [18]. Each of these approaches is further limited by the network communication and the cost of additional machines or by the I/O capacity of attached devices.

This paper explores how replacing DRAM with emerging non-volatile memories (NVM)such as PCM or RRAM impacts state-of the art graph processing frameworks. We study whether NVM can eventually become a cheaper and more scalable alternative to DRAM reducing the degree of scale-out and the cost of graph analytics. Table 1 shows how these NVM technologies relate to DRAM. Increased capacity and lower cost come with higher latencies and lower bandwidth as well as limited write endurance. These characteristics can potentially degrade the performance of systems designed assuming DRAM as the underlying technology.

However, if carefully designed to mitigate the limitations of NVM, graph processing systems could leverage the un-

INFLOW '15, October 4–7, 2015, Monterey, CA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3945-2/15/10...$15. . . $15.00.
http://dx.doi.org/10.1145/2819001.2819005

derlying memory hierarchy in order to scale at a lower cost. The first step towards this goal is understanding how existing state of the art frameworks operate with NVM.
To that extent, we quantify the performance of four state-of-the-art frameworks on NVM using a hardware emulator described in Section 3.

Graph analytics algorithms and frameworks differ vastly in terms of access patterns, data structures and programming models. We chose a representative subset based on the work in [29]. Section 5 shows the impact that different bandwidth and latency points of NVM have on Galois[21], Graphlab[15], Graphmat[30] and X-Stream[28] running Pagerank, Bread-First Search (BFS), Triangle Counting and Collaborative Filtering. We did not find a publicly available implementation of Collaborative Filtering for Galois, hence we do not evaluate Galois for this algorithm.

NVM causes a degradation in performance for all test cases but the magnitude of degradation varies between algorithms and frameworks, ranging from 1.5× to 4×. In order to understand these differences, we perform a detailed characterisation of the frameworks using hardware performance counters. The analysis shows that, due to CPU memory-level parallelism and hardware prefetchers, the performance degradation is not necessarily proportional to the reduced bandwidth or the increased latency of NVM, but it is still substantial compared to the DRAM-only performance.

As an attempt to bridge the gap between the performance on DRAM and NVM, we modify Graphmat to explore the opportunities for fine-grained data tiering in *hybrid* memory systems with DRAM and NVM. We show that by placing only a fraction of data in DRAM (6.7% to 31.5% of the total memory footprint), GraphMat achieves 2.1×-4× better performance than the corresponding NVM-only implementations, and within 1.02×-1.2× of the DRAM-only performance. This paper makes the following contributions:

- Characterization of a hardware emulator that accurately models various bandwidth and latency points expected for emerging NVM technologies.
- Detailed analysis of the impact of NVM on different graph analytic frameworks and algorithms, quantifying the overheads of NVM-only solutions (1.5× to 4×) compared to their DRAM-only counterparts.
- A study of the benefits of application-driven tiering with Graphmat, demonstrating that Graphmat can achieve close to DRAM-only performance (within 1.2×) by utilizing only a fraction of DRAM (as little as 6.7%) in a *hybrid* memory system with DRAM and NVM.

2. Background

As previously stated, graph algorithms suffer from irregular access patterns that may limit their performance even on DRAM. We provide a short discussion on the impact of

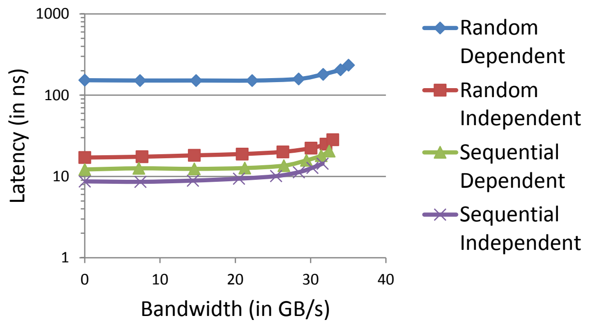


Figure 1: Memory read latency for various access patterns.

memory access patterns on the average memory access latency, and therefore performance of an application.

Depending on the actual implementation and the memory access pattern, mitigating factors to the high memory latency are modern processors' (e.g., *Intel*® *Xeon*®) extensive use of out-of-order execution and aggressive hardware prefetching [6]. These features can successfully reduce the average latency of memory reads, for certain access patterns, by reducing the number of cache misses and increasing memory-level parallelism (MLP) [10].

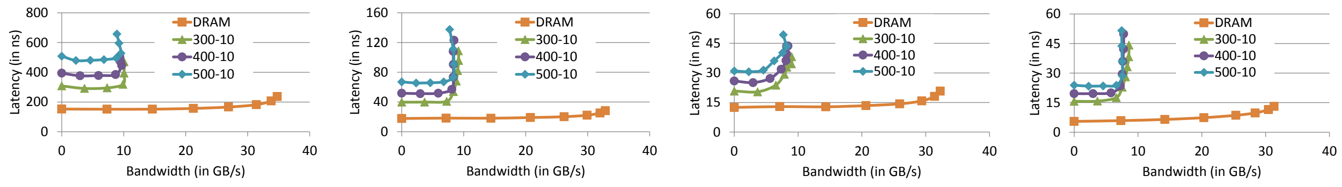
These effects are demonstrated in Figure 1, which shows the average latency of memory reads (for various access patterns) on an Intel Xeon E5-4620 system. In these experiments, one thread reads memory (in the specified pattern) and measures average latency while other threads consume memory bandwidth by accessing their private memory. For *dependent* accesses, the memory of the thread measuring latency is initialized for pointer chasing. The locations for *independent* accesses are generated on the fly without any dependencies. Dependent accesses can have only one memory load waiting for execution, and therefore does not benefit from out-of-order execution. In the case of independent accesses there can be many in-flight loads.

Random dependent represents the worst case scenario, with every load experiencing the entire memory latency. In comparison, *random independent* is an order of magnitude faster due only to MLP. Similarly, *sequential dependent* is an order of magnitude faster than random dependent, but entirely due to hardware prefetchers. *Sequential independent*, which benefits from both MLP and prefetchers, shows the best performance of all.

The key observation from this experiment is that the performance of memory-intensive applications depends heavily on the pattern of their memory accesses.

3. *Hybrid* Memory Emulator

The *hybrid memory emulation platform (HMEP)* enables the study of hybrid memory with real-world applications by implementing – (i) separate physical memory ranges for DRAM and emulated NVM, and (ii) fine-grained emulation of their relative latency and bandwidth characteristics. HMEP has been used in other research [8, 12, 13, 22, 23, 33],



a) Random Dependent b) Random Independent c) Sequential Dependent d) Sequential Independent

Figure 2: Read latency-bandwidth plots for several HMEP configurations and all access patterns

but it has not been described in the detail needed to explain the experimental results shown in this paper.

HMEP is based on a dual-socket Intel Xeon E5-4620 platform, with each processor containing eight 2.6 GHz cores. Hyperthreading is disabled. Each CPU supports four DDR3 channels and memory is interleaved across the CPUs.

Separate DRAM and NVM physical ranges: Using custom BIOS firmware, HMEP partitions the four memory channels of a CPU equally between DRAM and emulated NVM. The NVM region is available to software either as a separate NUMA node (managed by the OS) or as a reserved memory region (managed by PMFS) [13]. The total amount of memory in the system is 320 GB of which 256 GB is reserved for emulated NVM.

Read latency emulation: HMEP emulates read latency on the NVM physical range using special CPU microcode, which uses debug hooks in the CPU to implement a performance model for latency emulation. The model monitors a set of hardware counters over very small intervals, and for each interval estimates (and applies) the additional cycles that the core would have stalled if the underlying memory was slower than DRAM. A naive method of calculating *stall cycles* would be to count the number of actual memory accesses (i.e., last level cache misses) to NVM and multiply it by the desired extra latency. This method, however, is suited only for simple in-order processors and highly inaccurate for modern out-of-order CPUs (§2).

We implement a model based on the observation that the number of cycles that the core stalls waiting for the memory reads to complete is proportional to the actual memory latency. If Lp is the target latency to emulated NVM, then the additional (proportional) stalls that the model applies for the time interval is:

$\delta_{stall} = S \times \frac{L_p - L_d}{L_d}$, where S is the actual number of stall cycles due to accesses to the emulated NVM range, L_p is the desired NVM latency, and L_d is the actual latency to DRAM.

In calculating S, we are limited to the following available counters on our test processor:

- Core execution stall cycles due to second level cache (L2) misses (S_{L2}).
- Number of hits in LLC (H_{LLC}).
- Number of last level cache (LLC) misses to DRAM (M_{dram}) and NVM (M_{nvm}) ranges.

Using these counters, the model first computes the execution stalls due to LLC misses (S_{LLC}) as follows:

$S_{LLC} = S_{L2} - (H_{LLC} \times K)$, where K is the difference in latency of a LLC hit and a L2 hit.

Finally, the model computes S as:

$$S = S_{LLC} \times \frac{M_{nvm}}{M_{dram} + M_{nvm}}.$$

Validation: To validate the model, we emulate the latency of slower NUMA memories in multi-processor platforms and compare the performance of several application on emulated NVM vs. actual NUMA memory. Following this approach, we validated the latency emulation model for a large number of applications – including several microbenchmarks (e.g., various sort algorithms), benchmarks from SPEC CPU2006 and workloads in this paper. Performance with NVM (emulating remote memory latency) is always within 7% of the performance with actual remote memory.

Limitations: NVM device characteristics are very different from that of DRAM. For instance, reads and writes to a PCM device have to wait for the preceding writes to the same memory line to complete [25]. The HMEP latency emulation model emulates only the average latencies and not NVM's device-specific characteristics. This restriction is primarily due to the limited internal CPU resources available for NVM latency emulation.

CPU hardware prefetchers can drastically improve the performance of sequential and strided memory accesses. HMEP assumes that the prefetchers will continue to be at least as effective with NVM as they are today with slow remote memory (of comparable latency) on large NUMA platforms. This assumption is reasonable even if we ignore the fact that, if needed, CPU prefetchers could be assisted by some form of prefetching on the NVM modules as well.

Bandwidth emulation: NVM has lower sustained bandwidth than DRAM, particularly for writes (Table 1), though that could be improved using ample scale-out of NVM devices and buffers. [1] HMEP emulates read and write bandwidths by programming the memory controller to limit the maximum number of DDR transactions per μsec. This throttling feature can be programmed on a per-DIMM basis [4], and is applied only to the NVM range.

Limitations: The bandwidth throttling feature in the memory controller is a single knob that limits the rate of all DDR transactions. Therefore, HMEP is unable to vary the read and write bandwidths independently.

[1] Since writes to write-back caches are posted, NVM's slower writes result in lower bandwidth and not higher latency on every write.

Algorithm	Graph Type	Vertex Property	Edge Access Pattern	Message Size (Bytes/edge)	Vertices active	Input size (GB) - binary	Input size (GB) - text
Pagerank	Directed	Double	Sequential	8	All Iterations	12	18
BFS	Undirected	Int	Random	4	Some iterations	24	40
Collaborative Filtering	Bipartite	Double[ ]	Sequential	8K	All Iterations	24	34
Triangle Counting	Directed	Long	Sequential	0-10^6	Non-iterative	12	18

Table 2: An overview of the main algorithm characteristics

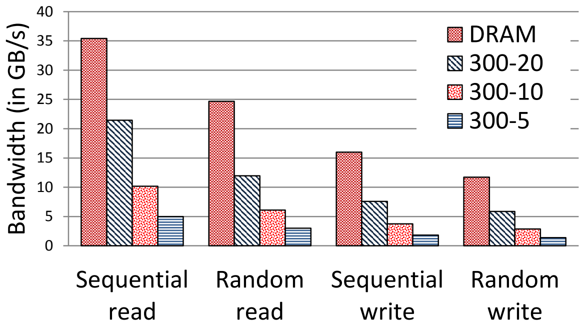


Figure 3: Bandwidth of HMEP configurations

Characterization: Figure 2 shows latency and bandwidth characteristics for memory reads to the NVM range in various HMEP configurations. These configurations are denoted by *x*-*y*, where *x* is the emulated NVM read latency (in ns) and *y* represents the peak bandwidth (in GB/s) to the NVM range. Access patterns are as described earlier (§2). Read latency to NVM depends heavily on the access pattern (as with DRAM) – sequential and independent reads are much faster than random and dependent reads. Figure 3 shows the measured sustained bandwidth to NVM for various HMEP configurations and access patterns. As expected, sequential accesses achieve higher bandwidth than random accesses, and read bandwidth is higher than write bandwidth.

To summarize, despite the stated limitations, HMEP adequately emulates the relative characteristics of DRAM and NVM in a hybrid memory system, and also the performance behavior of various memory access patterns.

4. Algorithm Characteristics

Achieving sequential and independent access in graph analytics is not always possible due to the irregular structure of the graphs and additional dependency on the programming model of the framework itself. To evaluate how NVM would impact graph analytics, we chose four algorithms representative of the different analytics disciplines as discussed in [29].

Pagerank is a widely used ranking algorithm for determining the popularity of webpages or user influence in a social network. It is an iterative algorithm that propagates the ranks of a node along its outgoing edges thereby changing the ranks of neighbouring nodes. The process is repeated until convergence or for a fixed number of iterations. Pagerank is a communication intensive algorithm with updates propagated along all edges in each iteration.

Bread-first search (BFS) is a traversal algorithm that starts at a given source node and produces a list of nodes ordered by the distance in terms of edges traversed to them from the source node. The algorithm continues until there are no nodes to be discovered. In each iteration only the nodes adjacent to a newly discovered node are processed, thereby reducing intra-node communication.

Triangle Counting is a technique used to discover cliques within a graph. A triangle exists when a node is connected to two other nodes that are mutually connected. Unlike BFS and Pagerank, Triangle Counting is a non-iterative algorithm that requires each node to count the intersections among the neighbours of its immediate neighbours. Depending on the actual graph structure, the size of messages exchanged between the nodes in this algorithm could be very large.

Collaborative Filtering is a widely used technique in machine learning for building recommender systems. The input is a bipartite graph of users and their ratings for a subset of items. The goal is to recommend items to a user based on her previous ratings. Depending on the framework, the (user,item) pairs are represented as a matrix or as a graph.

Table 2 provides an overview of algorithm characteristics along with the access pattern of each algorithm.

Algorithm Implementation The algorithms are commonly expressed as "vertex programs" where vertex state is propagated as a message along outgoing edges and updated based on messages along incoming edges. Table 3 shows the programming models of the frameworks evaluated.

Graphmat expresses computation as a vertex program but the operations are internally converted to sparse-vector matrix computations [30], resulting in a better compute time while maintaining a simple intuitive API.

Galois supports a slightly different computation model where each message activates a node which is thereafter put in a task-list. All items within the task-list are computed on in parallel by respective custom scheduling policies that account for locality and priorities [21].

Graphlab [15] follows the above described vertex-centric model whereas *X-Stream* [28] slightly modifies this model to optimise for access to secondary storage such that, instead of

Framework	Programming Model	Execution Scheduling
Graphmat	Vertex-program + SpMV backend	Synchronous
Graphlab	Vertex-program	Async/Sync
Galois	Task-based	Async/Sync
X-Stream	Edge-centric Vertex program	Bulk-synchronous

Table 3: Graph processing frameworks - characteristics

iterating through the vertex set, the program iterates through the edge-list sequentially.

5. Evaluation

5.1 Methodology

The frameworks are provided with synthetic graphs generated using the Graph500 RMAT generator[2]. The generator provides graphs that correspond to the structure of real-world graphs of interest and is widely used by the graph analytics community for system evaluation [29],[15],[28],[30].

For Pagerank and BFS, the input is a scale 26 graph with 2^{26} nodes and 2^{30} connections. Since BFS requires an undirected graph, we add reverse edges to the dataset. And, since the vertex state and intermediary data is larger for Triangle Counting, we use a smaller (scale 24) graph with 16M nodes with 268M edges. Finally, Collaborative Filtering requires a bipartite graph for which we generated a graph according to [29] with 8M nodes and an average of 256 connections per node. The input for all frameworks except for Graphlab is in binary format. The total size of the input depending on the format is shown in the last two columns of Table 2.

As a starting point for our analysis we ran the different algorithm implementations on DRAM as our baseline case. Using the emulator described in Section 3, we analyse the behaviour of these implementations with increasing memory access latency and decreasing memory bandwidth. The latency is varied from 300ns to 500ns and the bandwidth is varied from 40GB/s (equal to DRAM) to 5GB/s. The DRAM latency on the system is 150ns. Since the absolute runtimes differ among frameworks due to different implementations, we plot the ratio between the runtime of the framework at a particular latency/bandwidth point compared to the corresponding runtime on DRAM.

5.2 Analysis of performance in DRAM

The peformance of the implementations depends on the programming model and data-structures of a particular framework, resulting in widely different runtimes as shown in Table 4.

The times reported are per iteration times for Pagerank and Collaborative filtering, whereas for BFS and Triangle counting we present the entire runtime, excluding the time taken to load the graph or for other setup.

	Pagerank	BFS	Triangle Counting	Collaborative Filtering
Graphmat	4.40	9.96	31.95	320.04
Graphlab	13.83	87.50	44.95	563.41
Galois	6.89	8.66	24.08	-
X-Stream	7.60	29.62	1058.00	79.68

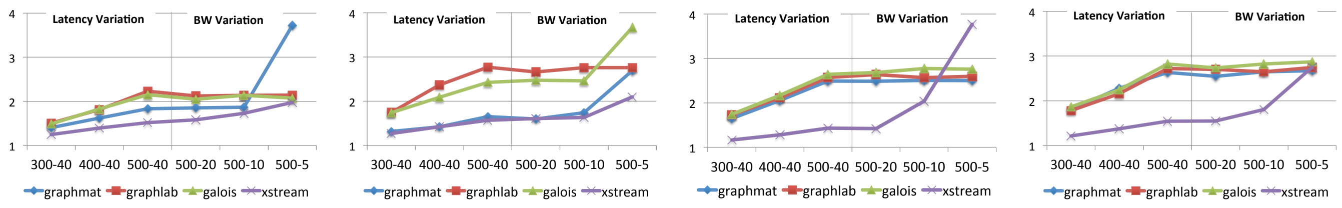
Table 4: Absolute runtimes in seconds. The differences between frameworks are explained in 5.2

Graphmat and Galois have similar performance but for different reasons. The SpMV backend of Graphmat allows for quick computations and better expressibility of the data, especially for Collaborative filtering where the average degree of a vertex is 256 and SpMV operations lead to a $2\times$ improvement over Graphlab. Unlike Graphmat and Graphlab which calculate the ratings for Collaborative Filtering using Stochastic Gradient Descent(SGD), X-Stream uses an optimized version of ALS[34]. The algorithm applies updates as they are generated computing the ratings faster than implementations of SGD.
Galois on the other hand supports asynchronous computation and its task-based programming model leads to quicker convergence. Asynchronous computation does not play a big role in algorithms such as Pagerank where we propagate updates along every edge in each iteration, but for traversal algorithms, where we pass an edge only once, it can significantly improve the time to converge. We can clearly observe this behaviour with X-Stream where all edges are streamed in every iteration, which for BFS leads to a large number of unnecessary reads.
Due to the need to support streaming, the implementation of Triangle counting in X-Stream is an approximate implementation [9], executed in a predefined number of iterations (100 in our case) which causes the significant difference in runtimes for this algorithm.
We note that Graphlab was designed as a distributed system and, therefore, some of their optimizations for distributed computation may have caused increased runtimes on a single node.

5.3 Analysis of performance in NVM

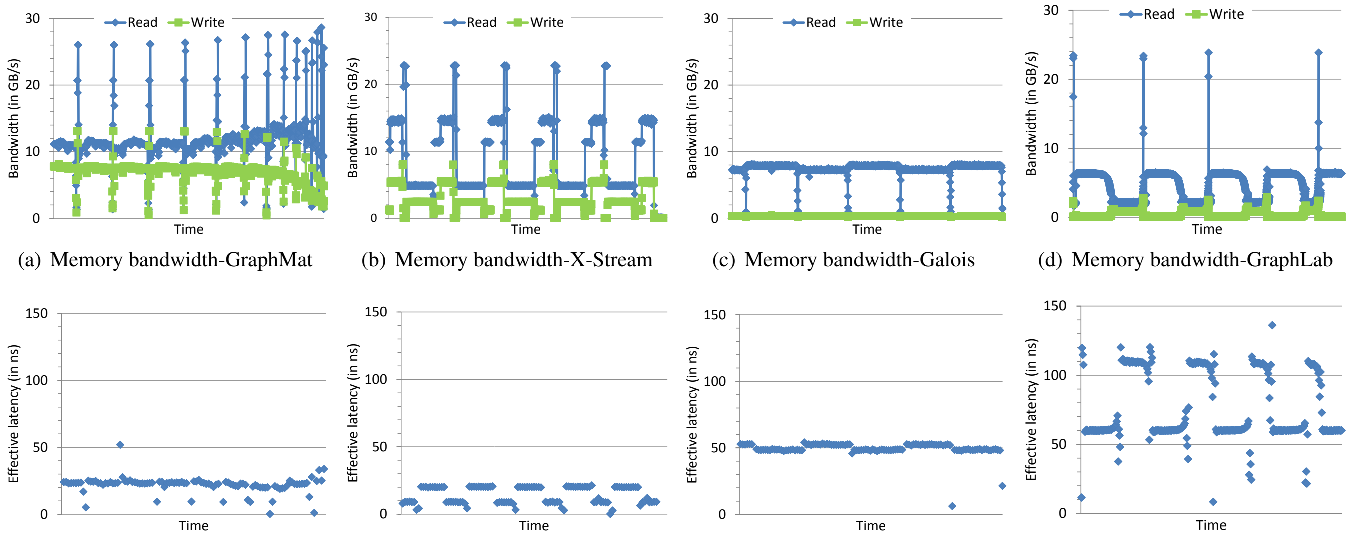
Figure 4 shows results for several NVM latency and bandwidth points. First we vary the latency of NVM from 300ns to 500ns, while the bandwidth is fixed at 40GB/s (same as DRAM). Then we vary the bandwidth from 40GB/s to 5GB/s, to highlight the impact of both increased latency and decreased bandwidth in a concise manner. While the performance degrades for all the frameworks, they are not equally sensitive to latency and bandwidth. Graphlab and Galois exhibit higher sensitivity to increased latency rather, while Graphmat and X-Stream are more sensitive to lower bandwidth.

To understand these results, we profiled the applications using hardware performance counters. Figure 5 shows the following key metrics from counter analysis of the Pagerank



a) Pagerank b) BFS c) Triangle Counting d) Collaborative Filtering

Figure 4: Performance variation on NVM. The X-axis shows HMEP configurations as **NVM latency(ns)-Bandwith(GB/s)**. The Y-axis shows the run time in NVM normalized to the run time in DRAM for a particular framework.



(a) Memory bandwidth-GraphMat (b) Memory bandwidth-X-Stream (c) Memory bandwidth-Galois (d) Memory bandwidth-GraphLab

(e) Effective latency-GraphMat (f) Effective latency-X-Stream (g) Effective latency-Galois (h) Effective latency-GraphLab

Figure 5: Bandwidth (in GB/s) and Effective memory latency (in ns) for Pagerank. The X axis represents time.

algorithm: (i) Read and write bandwidth, and (ii) Effective latency.

Effective latency approximates the average memory read latency in an application by measuring the core stalls due to pending reads per LLC miss. This metric, shown in nanoseconds, measures the effectiveness of MLP and hardware cache prefetchers. Higher effective latency means the workload is more sensitive to the higher latency of NVM. Similarly, applications with high bandwidth requirements are likely to perform worse with NVM.

Results in Figure 5 (shown in a timeline) correspond to the actual execution phase, excluding the loading and initialisation phases. GraphMat and X-Stream achieve significantly higher bandwidth than Galois and GraphLab, which explains the sensitivity of these frameworks to lower bandwidth. The observed effective latency for different frameworks confirms that the performance degradation at higher latencies is due to the stalls resulting from memory accesses in the framework. We attribute this result to the inability of certain frameworks (particularly Graphlab) to exploit the hardware prefetches and MLP. In fact, running these same experiments with prefetching disabled results in a performance drop of 12-25% for Graphmat and X-Stream, but has negligible impact on the performance of Galois and Graphlab.

X-Stream's sequential access pattern and GraphMat's efficient matrix representation of the data incur fewer random accesses than the indexing methods of Galois and Graphlab. The difference between Galois and Graphlab can be explained by the fact that Galois achieves better locality by placing the data as close to the execution threads as possible and then using a custom scheduler that efficiently schedules the active vertices for the next iteration.

The reduced bandwidth becomes a limiting factor for frameworks such as Graphmat and X-Stream, especially for communication intensive algorithms such as Pagerank where we observe a 30% drop performance when latency increases from 300ns to 500ns and a further 2× degradation when we reduce the bandwidth from 40 GB/s to 5 GB/s. In the case of Graphmat, even though the memory access latency is hidden well with prefetching, when the message size becomes large enough, such as for Triangle Counting and Collaborative Filtering, the framework becomes sensitive to higher latency instead of the lower bandwidth. The runtime

	Sparse Vectors	Vertex data	Matrix
Pagerank	1.53	1.06	18.84
BFS	1.02	1.56	35.71
Triangle Counting	0.63	2.64	7.10
Collaborative Filtering	3.89	1.28	31.38

Table 5: Size in GB of Graphmat datastructures and the initial input size

increases by over 50% at 500ns latency (compared to 300ns) but does not change much as we reduce the bandwidth. The increased message size in Triangle Counting causes X-Stream to become more sensitive to reduced bandwidth than in the case of Pagerank and BFS where, compared to the performance at 40 GB/s, we see the performance drop by 1.5× at 20 GB/s and 3.7× at 5 GB/s.

6. Tiering

Our NVM-only analysis shows that, due to CPU's prefetch and MLP capabilities, the performance degradation of graph analytic applications with NVM is mitigated to an extent. However, depending on the implementation, the impact of NVM latencies and bandwidth can be further mitigated using only a modicum of DRAM in a hybrid memory system, and by intelligently tiering the data between DRAM and NVM.

In this setting, the system would place only the most performance-sensitive data(e.g., frequently accessed random data or critical write-only data) in high-performance DRAM and leverage the capacity (and cost) of NVM for all other data. Ideally, the data structures chosen for placement in DRAM would have to be small enough (compared to the overall graph size) for tiering to be effective from the cost perspective. To evaluate the potential of intelligent data placement in graph analytic platforms, we implemented a simplistic version of data tiering in Graphmat. The choice of Graphmat for this experiment is due to the fact that it is easy to identify the critical data structures in the Graphmat implementation. The SpMV backend of Graphmat defines three important data structures: sparse vectors (SV), vertex associated data (VxD) and (sparse) matrices (MTX) allocated to represent the data in memory. In this model, the 'messages' sent from one node to another are translated into sparse vectors. This vector is then applied to a vector containing the vertex data and the graph is represented as a matrix. The size of each of the data structures (per algorithm) is shown in Table 5. The size of SV ranges from 2.7% to 10% of the total memory footprint, while the vertex data ranges from 3.2% to 25.6%. Other data (e.g., for book-keeping) is negligible in size for all algorithms.

For the tiering experiments, we use HMEP in NUMA mode – i.e., software can access DRAM and NVM as separate memory nodes and use the NUMA API (e.g., libNUMA in Linux) to control the allocations from DRAM/NVM. We perform experiments where we allocate only SV or only SV+VxD in DRAM, while the rest of the application memory is allocated in NVM. Table 6 shows these tiering results with the two baselines – NVM-only and DRAM-only. We assume NVM latency of 500ns and bandwidth of 5GB/s in this case.

	PR	BFS	Triangle Counting	Collab. Filtering
NVM-only	17.58	26.78	79.87	854.51
SV in DRAM	5.49	13.62	72.71	628.31
SV+VD in DRAM	4.94	12.54	34.74	328.85
DRAM-only	4.40	9.96	31.95	320.04

Table 6: Static tiering of data between DRAM and NVM. The table shows runtimes in seconds for various tiering options.

In these experiments, Graphmat's NVM-only performance is 2.5×-4× worse than its corresponding DRAM-only performance. By placing the sparse vectors alone in DRAM, Graphmat's performance improves to within 1.97× of DRAM-only for all algorithms other than Triangle Counting. The gains are particularly impressive for Pagerank and BFS (1.25× and 1.32×, respectively). Placing vertex data vectors (along sparse vectors) results in even better performance — 1.03×-1.2× of DRAM-only — but at a higher cost in terms of the amount of DRAM required (6.7% to 31.5% of the total memory footprint). For the previously stated reasons, vertex data vectors in Triangle Counting are very large in size (25.6% of the total size) and latency-sensitive, and therefore result in the worst case scenario w.r.t. the amount of DRAM needed relative to NVM (31.5%).

To summarize, our initial experiments with tiering suggest that it has the potential to enable graph analytic frameworks to achieve close to DRAM-only performance, while requiring that only a fraction of the application memory footprint be present in DRAM in a hybrid memory system. As the next step, we plan to build more generalized analytics systems based on this observation.

7. Related Work

Qureshi et al. [27] discuss the use of NVM as main memory and evaluate several main memory organizations with DRAM and PCM, including *NVM-only* and *multi-level memory*. Their evaluation is based on simulation of a simple in-order processor model and memory that models only higher latency of PCM and not lower bandwidth. Further, their evaluation is limited to simple medium-sized application kernels. Our goal is to quantify the performance of NVM on modern CPUs with out-of-order execution and prefetch capabilities (§2), and with large-scale applications that are both latency-sensitive and bandwidth-intensive.

Lim et al. [19] study the use of slow memory in the context of shared, network-based (disaggregated) memory and conclude that a fast paging-based approach performs better than directly accessing slow memory. While their choice of target applications is key to their findings, their work also relies on a simple processor model and does not account for CPU's MLP and prefetch features (unlike our work).

Ferdman et al. [14] conduct a thorough study of many scale-out workloads using hardware performance counters and conclude that these workloads are unable to exploit the CPU's MLP, leading to poor power efficiency. While similar in the use of counters, our work is different from theirs in several ways – (i) since our goal is to study the use of NVM, our workloads are all large in-memory applications, (ii) depending on the implementation, our workloads are able to achieve high MLP, and (iii) we conclude that, for future heterogeneous memory architectures with NVM, it is imperative (and hugely beneficial) for the application's performance to exploit MLP and hardware prefetching when accessing NVM, even if it requires re-designing these applications.

Qureshi et al. [26] study the impact of *MLP* on the effective cost of LLC misses in an application, and categorize those misses as costly isolated/dependent misses and cheaper parallel/independent misses. Their proposal to expose this information to cache replacement algorithms to reduce the number of isolated misses is even more relevant to the NVM architectures in this paper, owing to NVM's higher latencies.

NVM in the *hybrid* architecture has been explored in several contexts. Prior work has examined the use of NVM for both capacity and persistence, with emphasis on the necessary system software and libraries to provide applications with efficient access to NVM [11, 13, 16, 31, 32]. Lessons learned from our analysis are applicable to all of them.

Researchers have previously explored the use of data classification and intelligent data placement in hybrid memory systems, particularly in the context of HPC applications [7, 24]. We apply this well-studied concept to large scale graph analytics applications and present our initial results that demonstrate the benefits of tiering with Graphmat.

8. Conclusions and future directions

Emerging NVM technologies are likely to bridge the gap between DRAM and block devices in terms of capacity and cost but they come with increased latencies and reduced bandwidth [1]. Our study shows that, despite optimized software implementations, NVM-only performance of these frameworks is 1.5×-4× worse than that of their DRAM-only counterparts, due to either higher latency or lower bandwidth of NVM.

Our subsequent experiments with data tiering suggest that, with optimal data placement in a well-suited implementation (such as Graphmat), it is possible to achieve close-to-DRAM (1.02× to 1.2×) performance in hybrid memory system with only a fraction of the costly DRAM (6.7% to 31.5% in Graphmat's case).

We believe that this conclusion can be generalised to other big-data applications that employ indices or cache a small portion of the data in order to achieve good performance. In addition to analyzing the impact of NVM on other applications, we are exploring system software to automate the classification and optimal placement of data in hybrid memory systems with any number of different physical memories.

Finally, though the density of NVM compared to DRAM enables processing more data on a single machine than previously possible, we do not expect it to eliminate the need for scaling out. More likely NVM will reduce the degree of scale-out, resulting in interesting implications to the complexity of the overall system, particularly of the networking subsystem. We plan to explore these aspects in future.

References

[1] http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[2] Introducing the Graph 500 - Cray User Group. https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf, 2010.

[3] Crossbar Resistive Memory: The Future Technology for NAND Flash. http://www.crossbar-inc.com/assets/img/media/Crossbar-RRAM-Technology-Whitepaper-080413.pdf, 2013.

[4] Intel Xeon Processor E5 v2 Product Family (Vol 2). http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-2.pdf, 2013.

[5] Intel Xeon Phi (Knights Landing) Architectural Overview. http://www8.hp.com/hpnext/posts/discover-day-two-future-now-machine-hp#.U9MZNPldWSo, 2014.

[6] Intel64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[7] Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., and Keckler, S. W. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15.

[8] Arulraj, J., Pavlo, A., and Dulloor, S. R. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15.

[9] Becchetti, L., Boldi, P., Castillo, C., and Gionis, A. Efficient algorithms for large-scale local triangle counting. *ACM Trans. Knowl. Discov. Data 4*, 3 (Oct. 2010), 13:1–13:28.

[10] Chou, Y., Fahs, B., and Abraham, S. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (2004), ISCA '04.

[11] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the*

Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (2011), ASPLOS XVI.

[12] DeBrabant, J., Arulraj, J., Pavlo, A., Stonebraker, M., Zdonik, S., and Dulloor, S. A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS@VLDB* (2014).

[13] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14.

[14] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII.

[15] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12.

[16] Kannan, S., Gavrilovska, A., and Schwan, K. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014).

[17] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14.

[18] Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12.

[19] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09.

[20] Loh, G. H. 3D-Stacked Memory Architectures for Multicore Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08.

[21] Nguyen, D., Lenharth, A., and Pingali, K. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[22] Oukid, I., Booss, D., Lehner, W., Bumbulis, P., and Willhalm, T. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (2014), DaMoN '14.

[23] Oukid, I., Lehner, W., Thomas, K., Willhalm, T., and Bumbulis, P. Instant Recovery for Main-Memory Databases. In *Proceedings of the Seventh Biennial Conference on Innovative Data Systems Research* (2015), CIDR '15.

[24] Pavlovic, M., Puzovic, N., and Adrian, R. Data placement in hpc architectures with heterogeneous off-chip memory. In *Proceedings of the 31st IEEE International Conference on Computer Design* (2013), ICCD '13.

[25] Qureshi, M. K., Franceschini, M. M., Jagmohan, A., and Lastras, L. A. PreSET: Improving Performance of Phase Change Memories by Exploiting Asymmetry in Write Times. *SIGARCH Comput. Archit. News 40*, 3 (June 2012).

[26] Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (2006), ISCA '06.

[27] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09.

[28] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[29] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14.

[30] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Vadlamudi, S. G., Das, D., and Dubey, P. GraphMat: High performance graph analytics made productive. VLDB '15.

[31] Venkataraman, S., Tolia, N., Ranganathan, P., and Campbell, R. H. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (2011), FAST'11.

[32] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[33] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15.

[34] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R. Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the International conference on Algorithmic Aspects in Information and Management* (2008), Springer-Verlag, pp. 337–348.