ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories

Yu Chen
Department of Computer Science
College of William & Mary
Williamsburg, VA, USA
ychen39@email.wm.edu

Ivy B. Peng
Lawrence Livermore National Laboratory
Livermore, CA, USA
peng8@llnl.gov

Zhen Peng
Department of Computer Science
College of William & Mary
Williamsburg, VA, USA
zpeng01@email.wm.edu

Xu Liu
Department of Computer Science
College of William & Mary
Williamsburg, VA, USA
xl10@cs.wm.edu

Bin Ren
Department of Computer Science
College of William & Mary
Williamsburg, VA, USA
bren@cs.wm.edu

Abstract

Active development in new memory devices, such as non-volatile memories and high-bandwidth memories, brings heterogeneous memory systems (HMS) as a promising solution for implementing large-scale memory systems with cost, area, and power limitations. Typical HMS consists of a small-capacity *high-performance* memory and a *large-capacity* low-performance memory. Data placement on such systems plays a critical role in performance optimization. Existing efforts have explored coarse-grained data placement in applications with dense data structures; however, a thorough study of applications that are based on graph data structures is still missing.

This work proposes ATMem—a runtime framework for *adaptive granularity* data placement optimization in graph applications. ATMem consists of a lightweight profiler, an analyzer using a novel *m-ary tree-based* strategy to identify sampled and estimated critical data chunks, and a high-bandwidth migration mechanism using a multi-stage multi-threaded approach. ATMem is evaluated in five applications on two HMS hardware, including the Intel Optane byte-addressable NVM and MCDRAM. Experimental results show that ATMem selects 5%-18% data to be placed on high-performance memory and achieves an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM, over the baseline that places all data on the large-capacity memory. On NVM-DRAM, ATMem achieves performance comparable to a full-DRAM system with as low as 9%-54% slowdown.

CCS Concepts • **General and reference** → **Performance**; • **Software and its engineering** → **Development frameworks and environments**.

Keywords Heterogeneous Memory Systems, Data Placement, Graph Application

ACM Reference Format:
Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377922

1 Introduction

Memory technologies are advancing fast, and new memory devices that feature high-performance, high-density, or low-power are emerging [13–15]. Recently, 3D-stacked memories, such as Hybrid Memory Cube (HMC) [5] and HBM [14], and byte-addressable non-volatile memories (NVM) have become commercially available. These new memory devices, together with the conventional DRAM technology, make heterogeneous memory system (HMS) a feasible solution for building large-scale systems under the limited area, power, and cost budget.

A typical HMS consists of a *high-performance memory* and a *large-capacity memory*, where the high-performance memory has a smaller capacity, and yields higher memory bandwidth and/or lower memory latency than the large-capacity memory. Consider two popular HMS examples. The 2nd Gen Intel® Xeon® Scalable processor supports up to 6 TB Optane byte-addressable NVM and up to 384 GB DDR4 DRAM on a single machine [25]. Another example is the Intel Knights Landing (KNL) processor with up to 192 GB

DRAM and 16 GB MCDRAM [31]. While DRAM is the high-performance memory compared to the Intel Optane NVM, providing three times bandwidth of NVM, it becomes the high-capacity memory on KNL, where MCDRAM provides nearly four times bandwidth of DRAM.

On HMS, data placement plays an essential role in performance optimization. There have been extensive efforts for tackling this challenge [6, 7, 10, 19, 23]. The general optimization strategy tries to place critical data onto the high-performance memory. Some specialized optimization also tries to utilize memory bandwidth on both memories concurrently, which is only feasible on architectures providing independent memory channels to each memory. State-of-the-art optimization techniques have explored coarse-grained solutions that place whole data structures onto the high-performance memory [6, 7, 10, 19, 23, 27, 33, 34]. However, these approaches are inefficient for graph applications—a more challenging class of applications that have massive data structures with skewed access patterns.

Graph applications play significant roles in a spectrum of fields ranging from bioinformatics, scientific computing, social network, to machine learning and data mining. The current solutions face two main challenges. First, whole data structure placement might move non-critical data regions with few reuses, e.g. the data associated with low-degree vertices in graph processing, to high-performance memory, resulting in a waste of scarce resource. For instance, applications running on servers need to share all resources, resulting in even smaller high-performance memory available to an application. Second, due to the data-driven behavior of graph applications, effective data placement largely depends on the feature of input data, i.e., the sparsity and the structure of a graph, and also the query at each run.

Relying on the application programmer to explicitly manage data placement is only feasible at a coarse granularity, i.e., changing the placement of a whole data structure. Even so, it is a tedious and error-prone process, especially for large-scale applications. Moreover, a statically managed placement may not be portable when applications are running on a different system. Thus, an optimal decision may include feedback from the application behavior on the underlying hardware into consideration. These challenges require a dynamic solution and fine-grained data placement scheme to address.

This paper proposes ATMem as a data placement optimization framework to tackle these challenges in graph applications. ATMem enables adaptive granularity in managing data placement on HMS with three novel designs. First, ATMem enables adaptive data chunk profiling for subsequent *partial* data structure placement. The ultimate goal is to achieve the maximum *performance gain per byte*, i.e., improving fast memory utilization by only placing critical data regions that yield the highest performance gains on it. This is especially meaningful for server machines with multiple applications competing for precious fast memory. Second, ATMem supports lightweight sampling-based profiling and more importantly, enhances the analysis of sampling results with a novel tree-based information patching procedure to promote prospective data chunks into the critical category. Third, ATMem supports high-bandwidth data migration between memories at the application level without changes to operating systems or hardware.

The main contributions of this work are as follows:

- It proposes a lightweight profiler that uses hardware sampling to identify access patterns to data chunks of adaptive granularity.
- It employs a local relative ranking strategy to select critical regions inside data structures based on sampling results.
- It employs a novel *m-ary tree-based* strategy to promote prospective data chunks into critical based on a global relative adaption.
- It designs a multi-stage multi-threaded migration strategy at the application level to enable high-bandwidth data migration and reduce TLB misses.
- We provide the implementation in a framework called ATMem and evaluate in five applications on two HMS hardware, including the state-of-the-art byte-addressable NVM.

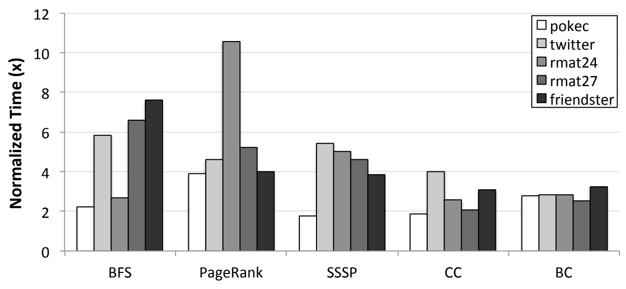
ATMem is evaluated in five graph applications on real NVM-DRAM and MCDRAM-DRAM hardware. Our evaluation results show that by selecting 5%-18% data onto high-performance memory, ATMem achieves an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM, compared to the baseline that places all data on the large-capacity memory. On NVM-DRAM, ATMem achieves performance comparable to a full-DRAM system with as low as 9%-54%. ATMem also enables 2.07×-5.32× faster data migration than the system service.

2 Background and Motivation

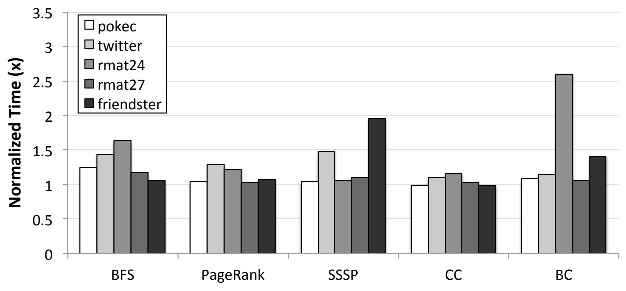
This section introduces the architecture of systems under consideration, the target workload, and the limitations of existing solutions. It also presents a preliminary study on real hardware (including the state-of-art Intel Optane byte-addressable NVM).

2.1 Heterogeneous Memory Systems

This work considers heterogeneous memory systems that place a small-capacity *high-performance memory* side-by-side to a low-performance *large-capacity memory*. The latest 2nd Gen Intel® Xeon® Scalable processor can work with up to 6 TB Optane byte-addressable NVM of 39 GB/s bandwidth and 384 GB DRAM of 104 GB/s bandwidth [25]. Another example is the Intel Knights Landing (KNL) processor that has 96 GB DRAM next to 16 GB MCDRAM of 400 GB/s bandwidth [31]. We conduct preliminary studies on real hardware and find that application performance on different



a Normalized execution time with data placement on the Intel Optane byte-addressable NVM as to that with data on DRAM.



b Normalized execution time with data placement on DRAM as to that with data preferably placed on MCDRAM.

Figure 1. Compare the performance of five applications on two HMS using different data placement for five datasets.

memories can have a much larger gap than that predicted by emulators [9, 26, 34].

Figure 1a and 1b report the slowdown when data are placed on the large-capacity memory compared to that on high-performance memory. The Intel Optane NVM has three times latency and 38% bandwidth of DRAM [25]. However, application performance could slow down by up to 10× (Figure 1a). MCDRAM has a limited capacity and system solutions like 'numactl -p' might choose less critical data onto high-performance memory, reducing performance improvement. These results highlight prospective benefits from fine-grained data placement that selects critical regions into high-performance memory.

Objective I: *our work identifies critical data chunks inside a data object and only migrates them onto high-performance memory for efficient memory utilization.*

2.2 Graph Applications

Graph applications, such as data analytic workload, often exhibit data-driven access pattern and have low data locality. Contemporary computer systems use highly optimized caches to keep frequently used data close to processing units. Such optimization, however, has shown its inadequacy for graph applications [24]. With low data reuse, the overhead of managing hardware cache might even hurt application performance. The opportunity in optimizing data placement stems from dense (hot) regions and sparse (cold) regions in data that drive accesses. Our work identifies these regions dynamically and manages them in the high-performance memory explicitly at the application level.

The application-level approach requires to address several challenges, specifically compared to system-level or architecture solutions. First, it lacks the full statistics of page accesses as the operating system have. Second, it cannot flexibly modify the hardware units on an existing platform. Thus, ATMem utilizes the widely available hardware counters to develop a sampling profiler for estimating dense regions in data. Unlike a system solution that usually operates at page-size granularity, ATMem adapts the size of data chunks, i.e., the basic unit of a data structure, to reduce metadata and migration overhead. Moreover, a sampling-based approach has to trade off overhead and accuracy. Even a high-frequency sampling approach cannot guarantee to capture all the information. ATMem proposes a *tree-based clustering* strategy to "patch up" information that is likely missed due to sampling.

Objective II: *our work uses low-overhead sampling profiling and addresses possible information loss in samples to estimate critical data regions.*

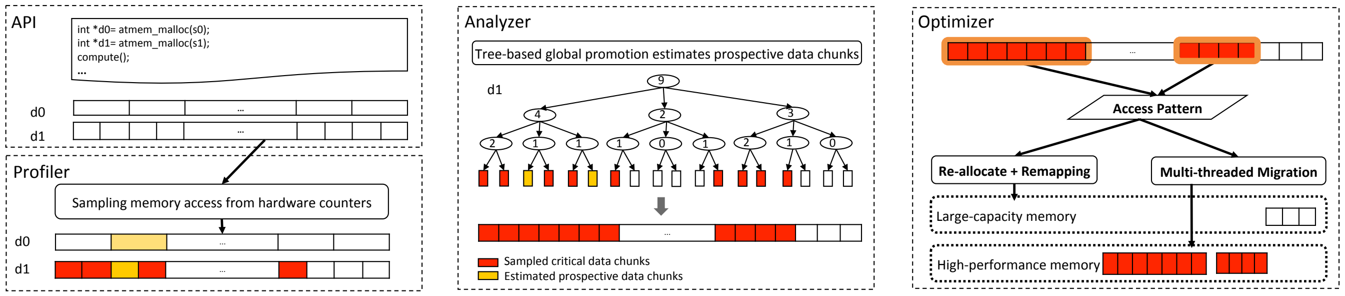
2.3 Migration Mechanism

On state-of-the-art heterogeneous memory systems, different memories, e.g., NVM and DRAM, are exposed to CPU as separate non-uniform memory access (NUMA) nodes [25]. In this way, traditional system services for NUMA control, i.e., *mbind*, can be used for migrating data from one memory to another. As pointed out by previous works [21, 35], the current system service is inefficient for heterogeneous memory systems. The migration mechanism is often single-threaded, which cannot exploit the high bandwidth supported by the hardware. Also, the data movement procedure is long and blocking, with substantial overhead spent for enforcing correctness for system-wide reliability. An application-level approach has the opportunity to bypass some of these overhead given sufficient application knowledge. Another side-effect from the standard service is the increased TLB misses after migration. While previous works have proposed solutions in the operating systems or hardware, we develop a multi-stage multi-threaded migration strategy to tackle these challenges at the application level on an existing platform.

Objective III: *our work improves data migration between memories at application level without changes in hardware or operating systems.*

3 Overview of ATMem

ATMem consists of three main components (as illustrated in Figure 2): a *profiler*, an *analyzer*, and an *optimizer*. **First**, ATMem profiler employs low-overhead hardware-counter based sampling to learn access patterns in an application.



```
int *d0= atmem_malloc(s0);
int *d1= atmem_malloc(s1);
compute();
...
```

API
d0
d1
Profiler
Sampling memory access from hardware counters
d0
d1
Analyzer
Tree-based global promotion estimates prospective data chunks
d1
Sampled critical data chunks
Estimated prospective data chunks
Optimizer
Access Pattern
Re-allocate + Remapping
Multi-threaded Migration
Large-capacity memory
High-performance memory

Figure 2. The overview of ATMem framework.

Second, based on the collected samples, ATMem analyzer identifies critical data chunks (of adaptive granularity) by a *global relative ranking* scheme. ATMem analyzer addresses one common challenge in sampling-based approaches, i.e., the loss in sampled information. The analyzer utilizes an *m-ary tree-based* strategy to "patch up" information to estimate potentially critical data chunks. **Finally**, ATMem optimizer performs a multi-stage multi-threaded migration to move both sampled and estimated critical data chunks onto high-performance memory. Section 4 introduces the design of these components in details.

4 Design of ATMem

This section describes the design of ATMem in identifying critical data regions of adaptive granularity, predicting prospective data regions, and migrating data regions at high bandwidth between memories.

4.1 Adaptive Data Chunks

The basic unit of data management in ATMem is called *data chunk*. Data chunk is an adaptive unit such that a data object (DO_i) is composed of N equal-sized data chunks ($DC_{ij}, j = 1..N$), while data chunks in different data objects can have different sizes. Figure 2 (Profiler panel) illustrates two chunk sizes in $d0$ and $d1$, respectively. ATMem adjusts the granularity of a data chunk DC_{ij} based on the size of the data object DO_i. This adaptive data chunk design has two main advantages. First, it breaks down a (potentially) large data object into finer-grained segments. By comparing the priority of data chunks inside a data object, ATMem can separate critical data chunks from non-critical data chunks. These critical data chunks correspond to dense (hot) regions of a data structure that has non-uniform access patterns, which is common in irregular applications. Second, ATMem can control the profiling overhead by managing the number of data chunks, i.e., coarsening the granularity of data chunks. Each collected sample in the profiling stage will be associated with a data chunk. Thus, changing the granularity of data chunks affects the metadata and profiling overhead directly.

4.2 Hybrid Local Selection

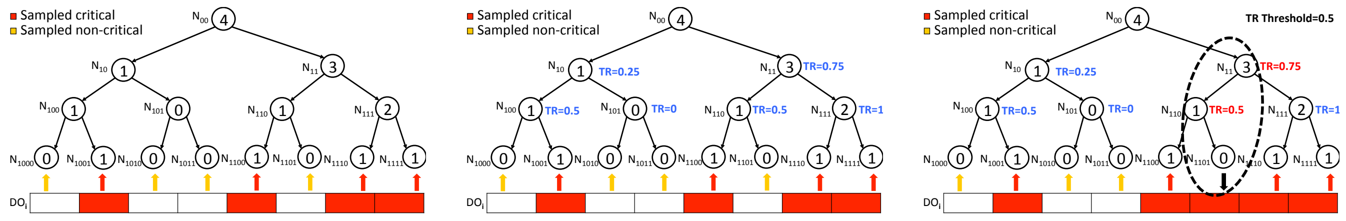
The first stage of ATMem analyzer employs a local relative ranking to select critical data chunks for each data object. These selected data chunks are called *sampled selection* to be distinguished from the estimated selection in a later stage. Equation 1 quantifies the metric of local priority (PR_{local}) for a data chunk DC_{ij} that represents the j-th data chunk in data object DO_i. ATMem uses the number of missed reads from the last-level cache LLC_{mr} as an indicator of priority and normalizes it to the size of a data chunk ($Size$). Note that the normalization is necessary for global relative ranking among different data objects in a later stage.

$$PR_{local}(DC_{ij}) = \frac{LLC_{mr}(DC_{ij})}{Size(DC_{ij})} \tag{1}$$

$$\theta(DO_i) = \max(P_n, \alpha \max PR, \min PR/Freq_{sample}) \tag{2}$$

$$CAT(DC_{ij}) = \begin{cases} 1, & \text{if } PR_{local}(DC_{ij}) \geq \theta \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

ATMem combines the conventional top-N selection with a derivative-based classification to select the threshold value of θ. In Equation 3, data chunks with priority score (PR) higher than the threshold θ has categorization (CAT) 1, i.e., critical. A top-N selection chooses a fixed ratio of data chunks that have the highest PR_{local} score in a data object, i.e., the n-th percentile P_n in Equation 2. However, a fixed selection is inefficient in two scenarios. First, a highly skewed access distribution has a high concentration in a small number of data chunks. In this case, the top $\frac{N}{2}$% data chunks have significantly higher priority than the next $\frac{N}{2}$% data chunks. Thus, selecting the second $\frac{N}{2}$% data chunks may not bring much improvement. On the contrary, in a relatively even distribution, the top N% data chunks may not have quantitatively significant difference compared to data chunks after them, i.e., more than N% data chunks should be selected. ATMem employs a derivative-based search, similar to a k-means clustering technique, to adjust the threshold value by quantifying the changes relative to the highest priority score ($\max PR$). Additionally, ATMem includes a theoretical minimum priority for a given data chunk size and adjusts for the sampling rate denoted as $Freq_{sampling}$.



a Construct a m-ary tree from the data chunks in DO_i.

b Calculate the tree ratio (TR) of each internal node bottom-up.

c Promote prospective data chunks from top-down.

Figure 3. Three stages in a tree-based global promotion: construction, bottom-up calculation of tree ratio (TR), and top-down promotion.

4.3 Tree-based Global Promotion

The second stage in ATMem analyzer considers the global view of all data objects and adapts the selection to reflect the relative importance of these data objects. ATMem constructs an m-ary tree for each data object to perform *estimated selection*. This procedure promotes data chunks that are not in sampled selection into *prospective critical*. ATMem adjusts the selection with two parameters, i.e., m and the threshold value of *tree ratio* (TR). This tree-based promotion helps "patch up" information that is likely missed due to sampling-based profiling. Similar approaches have shown effectiveness for prefetching data from CPU memory into GPU memory [11]. Additionally, the promotion can merge multiple discrete segments into a continuous one, which improves the efficiency of migrating data between memories. Figure 2 (the middle panel) illustrates an example of a ternary tree derived for data object $d1$. The leaf nodes in red represent critical data chunks from the sampled selection. In the remainder of this section, we introduce three main steps in the tree-based global promotion.

4.3.1 Tree Construction

ATMem analyzer uses the classification results in the first stage to construct an m-ary tree for each data object. A data chunk has value of either 1 (critical) or 0 (non-critical) from Equation 3. These data chunks correspond to the leaf nodes of the tree. Each leaf node has the value from its corresponding data chunk. Figure 3a illustrates an example tree constructed from the eight data chunks in DO_i. Each critical data chunk (in red) becomes a leaf node of value 1. From bottom-up, ATMem creates internal nodes of the tree. Each internal node carries value as the sum of its children nodes.

Tree ratio (TR) of an internal node is defined as the ratio between its value and the number of its descendant leaf nodes. In Figure 3b, node N_{11} has four leaf nodes and its TR is calculated as 3/4. Tree ratio is a metric that quantifies the likelihood of critical data chunks in a *range* of memory space. Here, the range of memory space depends on the level of a node. For instance, the root node N_0 covers the entire address space of DO_i, while the node N_{100} only covers the first quarter address space. ATMem adjusts m to control the range of memory space represented by an internal node as well as the sensitivity to a threshold value of tree ratio. For instance, a quad-tree can have more threshold values of tree ratio than a binary tree.

4.3.2 Global Adaptive TR Threshold

ATMem uses the tree ratio as an indicator for whether the sampled non-critical data chunks (yellow arrows in Figure 3) could still be important but not captured in the sampling. Also, a small gap in a large continuous address space can be "patched up" to improve data migration because launching multiple migrations would have higher overhead than a single migration. A naive design would use a fixed threshold value for tree ratio ($\theta(TR_i)$) such that if an internal node has TR value higher than the threshold value, all its non-critical children will be promoted to critical.

$$W(DO_i) = \frac{\sum_{j=1}^{N} PR_{local}(DC_{ij}) \cdot CAT(DC_{ij})}{\sum_{j=1}^{N} CAT(DC_{ij})} \tag{4}$$

$$\theta(TR_i)' = \epsilon + \frac{\theta(TR_i) \cdot (\max W - W(DO_i))}{\| \min W - \max W \|} \tag{5}$$

ATMem adjusts the threshold value for each data object based on its global relative ranking. The adapted threshold value ($\theta(TR_i)'$ in Equation 5) mitigates influence from different sampling frequencies, applications, data sets, and platforms. ATMem calculates the averaged priority of a data object in Equation 4 denoted as its weight (W). Weight quantifies the significance of selected data chunks, where a data structure of fewer critical data chunks with high priority has a higher weight than a data structure of more critical data chunks with low priority. From Equation 5, a large weight value would decrease the tree ratio threshold, causing the top-down promotion procedure (to be introduced next) to promote more non-critical data chunks. ATMem calculates the weight space as the gap between the minimum and maximum weight of all data structures. It also includes ϵ as a theoretical minimum threshold value that depends on the value of m. For instance, an octree would have $\epsilon = 0.125$ as a meaningful lower bound.

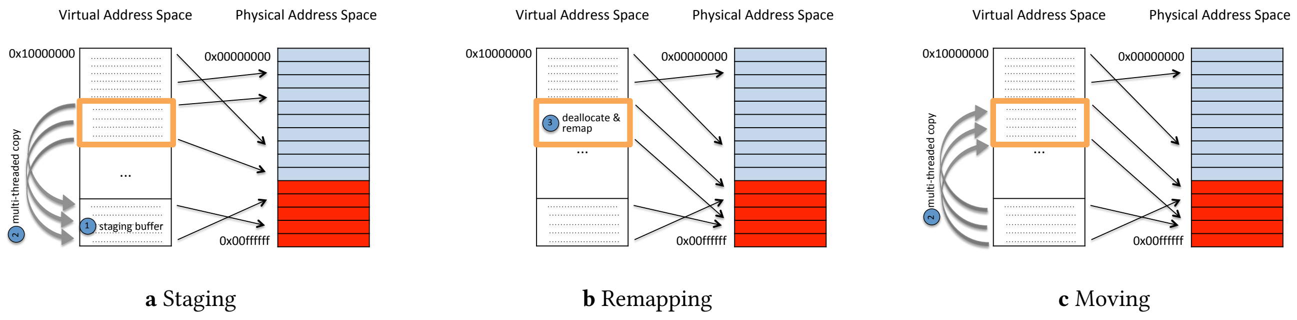


Figure 4. Fast migration at application level using staging buffers and multi-threaded data copy.

4.3.3 Top-down Promotion

ATMem uses the threshold value of tree ratio from Equation 5 to promote prospective data chunks into *estimated selection*. Each data object may be assigned a different threshold value after the global adaptive ranking. ATMem starts traversing from the root node and performs a breadth-first search to find an internal node with tree ratio higher than the threshold value. Starting from that node, ATMem tries to "patch up" data chunks on its descendant leaf nodes. For example, in Figure 3c, the data object has a threshold value of 0.5, and the node N_{11} has a tree ratio of 0.75. Next, ATMem identifies those N_{11}'s children nodes whose tree ratio are lower than the threshold value, i.e., the node N_{110}. ATMem promotes the right children node with zero tree ratio to be estimated critical. This top-down promotion procedure results in a single continuous region in the data object DO_i to be placed on high-performance memory.

4.4 Data Placement Optimization

ATMem uses the decision from the analyzer to optimize data placement at the application level. In particular, ATMem changes the physical memory of the selected data chunks to high-performance memory without changing the virtual memory address of the data object. This partial migration of a data object minimizes the modifications to the application source code.

Figure 4 illustrates three main steps of the multi-stage multi-threaded data migration approach. In this example, a data object has a virtual memory address from `0x10000000` to the low end of the yellow box. The system has two types of physical memories, as indicated in blue and red in the physical address space. Each segment in the address space represents a physical page. ATMem analyzer has determined that the data chunks in the yellow box, denoted as a *source region*, should be placed on high-performance memory.

In the first step, ATMem uses multiple threads to copy the value in the source region to a staging buffer concurrently. The staging buffer is physically located on the target physical memory, as indicated by the mapping (black arrows Figure 4a) to red pages in physical space. After that, ATMem remaps the virtual address of the source region to point to (empty) physical pages on the target memory (Figure 4b). Note that no data movement occurs in this stage and the virtual address space of the data object remains intact. Finally, in Figure 4c, ATMem uses multiple threads to copy the stored value from the staging buffer to the yellow region. The whole procedure has data moved twice, i.e., one between two memories and one within the same memory. ATMem adjusts the concurrency for data copy to exploit memory bandwidth supported by the hardware.

5 Implementation and Optimization

We implement ATMem as a runtime library for general heterogeneous memory systems. ATMem uses precise address sampling for profiling and provides a set of API for registering data structures and initiating data migration.

5.1 Hardware Supported Sampling

ATMem profiler uses hardware counters for low-overhead profiling. In particular, ATMem relies on the precise address sampling capability supported by hardware to collect the memory addresses of data accesses and correlates them to data chunks. Currently, ATMem is implemented on performance monitoring units (PMU) with processor event-based sampling (PEBS) on Intel processors [12]. ATMem can be easily extended to other processors with similar features, e.g., AMD processor [8].

ATMem automatically adjusts the sampling frequency of PMUs at runtime. Before enabling PMUs, ATMem combines the size and number of all data chunks and the number of application threads to adjust an empirical sampling rate on a given platform. This adaption avoids unnecessarily high sampling frequency while also ensures efficient information collection.

5.2 API

ATMem provides a minimal set of API (Listing 1) for easy adoption in existing applications. The main purpose of the interface is to inform ATMem runtime about the data objects

Listing 1. API for registering, profiling and optimization.

```
void *atmem_malloc (size_t);
void atmem_free (void *);
void atmem_profiling_start();
void atmem_profiling_stop();
void atmem_optimize();
```

Table 1. Experiment Platform Specifications.

NVM-DRAM Testbed	
Model	Intel® Xeon® Platinum 8260L
Processor	2nd Gen Intel® Xeon® Scalable processor
Core	2.4 GHz, 3.9 GHz Turbo frequency
Cache	32 KB d-cache and 32 KB i-cache, 1 MB private L2, 35.75 MB shared L3
Memory	96 GB DDR4 DRAM and 768 GB Optane DC NVDIMM per socket
MCDRAM-DRAM Testbed	
Model	Intel® Xeon® Phi 7200
Processor	2nd Gen Intel® Xeon Phi™ processor
Core	1.1 GHz, 1.7 GHz Turbo frequency
Cache	32 KB d-cache and 32 KB i-cache, 512 KB private L2
Memory	16 GB MCDRAM and 96 GB DDR4 DRAM

so that ATMem can link the collected memory address to data objects. Upon registration using `atmem_malloc()`, ATMem runtime internally determines the granularity of the data chunks for that data structure. Code transformation that converts memory allocation routines from `malloc`-like functions to `atmem_alloc()` is also feasible. However, programmers could use application knowledge to improve this decision.

ATMem profiler monitors PMU on cores and aggregated data for analysis for multi-threaded applications. One possible optimization is to monitor only a subset of PMUs, which is beyond the scope of this work. Currently, ATMem requires programmers to indicate when to start migrating data, i.e., `atmem_optimize()`. Future works on compiler optimization could automatically insert this function based on static analysis.

6 Experimental Setup

Our evaluation is performed on two real hardware testbeds. The first testbed is the 2nd Gen Intel® Xeon® Scalable processor platform that features the Intel Optane byte-addressable NVM and DDR4 DRAM. Table 1 summarizes the configurations. The Optane NVM is configured in App Direct mode and exposed to CPUs as a NUMA node. DRAM and NVM on the same socket share six memory channels that operate 2400 GT/s, i.e., a theoretical peak bandwidth of 115 GB/s. Our experiment uses 48 hardware threads and memories on one socket to eliminate the reported NUMA issues [25]. The second testbed is the Inter Knights Landing Xeon Phi processor (KNL) [31] that features 256 hardware threads, 16 GB high-bandwidth 3D MCDRAM and 96 GB DDR4 DRAM. MCDRAM is configured in flat mode.

Table 2. Characteristics of graph inputs.

Graphs	Number of Vertices	Number of Edges
pokec	1.6 M	30.6 M
rmat24	16.8 M	268.4 M
twitter	41.7 M	1.5 B
rmat27	134.2 M	2.1 B
friendster	68.3 M	2.1 B

We use SIMD implementation of five irregular applications [28], i.e., breadth-first search (`BFS`), single-source shortest path (`SSSP`), PageRank (`PR`), betweenness centrality (`BC`), and connected components (`CC`) for evaluation. `BFS`, `SSSP`, `PR`, `BC`, and `CC` use a mixture of five graphs, including `pokec`, `rMat24`, `twitter` (`twt`), `rMat27`, and `Friendster` (`friend`). Table 2 shows these graphs' vertices count and edges count, e.g., `Friendster` network dataset [1] contains 68.3M vertices and 2.1B edges.

Applications were compiled by the Intel C++ Compiler (`icc` 19.0.2.187) with `-O3` option and AVX512 flag. For each test, ATMem turns on hardware profiling in the first iteration and migrates data before the second iteration starts. The evaluation uses the benchmark run time from the second iteration as the optimized execution time. On both platforms, data to memory binding is controlled by 'numactl' in libnuma [17]. The experiments are repeated ten times and the average time is reported.

7 Evaluation

In this section, we first present the overall performance of ATMem in five graph applications using five data sets on two testbeds. Next, we perform a sensitivity test to evaluate the effectiveness of tree ratio in selecting data chunks. Finally, we compare the multi-stage multi-threaded migration strategy with the standard system service.

Table 3. ATMem performance on NVM-DRAM testbed compared to an all-DRAM ideal case

Slowdown	BFS	SSSP	PR	BC	CC
Min.	25%	26%	24%	9%	54%
Max.	2.4×	2.0×	3.0×	1.8×	2.0×

7.1 Overall Performance

This section evaluates the performance improvement from ATMem adaptive data placement. In particular, we compare the performance of applications on each testbed with the two reference performance. On the NVM-DRAM system, the baseline places data on the Intel Optane NVM (blue bars in Figure 5) while the ideal reference places all data in the DDR4 DRAM (green bars in Figure 5). On the MCDRAM-DRAM system, the baseline places all data in the DDR4 DRAM. We cannot have an ideal reference where all data is placed in

MCDRAM due to its limited capacity and the large data sets in use. Thus, we use the MCDRAM preferred NUMA policy provided by libnuma, i.e., 'numactl -p MCDRAM' (denoted as MCDRAM-p) as the ideal reference.

On the NVM-DRAM testbed, ATMem can effectively bridge the performance gap between NVM and DRAM with a low requirement on DRAM capacity. Figure 5 presents the execution time of applications on this NVM-DRAM testbed. This result shows that data placement by ATMem significantly reduces the execution time compared to the all-NVM baseline, reaching 1.25×-8.4× improvements—this is calculated from the first bar (blue) and second bar (red) in Figure 5. In Table 3, we compare ATMem with the ideal case where all data is placed in DRAM. ATMem can either achieve comparable performance or reduce the performance gap in Figure 1a significantly. For instance, `SSSP` with `Friendster` dataset using ATMem data placement is only 26% slower than placing all data in DRAM. Note that ATMem solution only places 12% data in DRAM (as shown in Figure 7). Figure 7 also reports the ratio of data that is placed by ATMem on high-performance memory for other applications. This ratio is calculated by the data placed on the high-performance memory (DRAM in this case) over the total data size.

On the MCDRAM-DRAM testbed, a similar trend is preserved between the ATMem data placement and the baseline all-DRAM case, i.e., ATMem data placement can significantly outperform the baseline, achieving 1.1×-3× execution time speedup with only placing a small portion of data (3.8%-18.2%) on high-performance MCDRAM. An interesting result comes from the superior performance of ATMem compared to the ideal reference. In Figure 6, ATMem placement significantly reduces the execution time for large datasets like `Friendster` and `rMat27`, compared to MCDRAM-p option. For `Friendster`, ATMem reaches up to 2.79× improvement in `BFS` with only 15% data in MCDRAM. Figure 8 reports the ratio of data that is placed by ATMem on high-performance MCDRAM for all applications.

7.2 Impact of Data Ratio

ATMem uses the tree ratio threshold value to tradeoff the data size on the high-performance (but small) memory with performance improvement. An optimal tradeoff would reach a data ratio beyond which performance improvement is not proportional to the increased data size on high-performance memory. We evaluate the effectiveness of the ATMem tree ratio by manually sweeping ϵ values in Equation 5. Consequently, ATMem would place different data ratios on high-performance memory. Figure 9 and 10 report the performance sensitivity to data ratio on two testbeds using the `BFS` benchmark.

ATMem consistently reaches the optimal tradeoff between performance and data ratio in all tested data sets. In Figure 9 and 10, there exist optimal regions in each dataset, where most data points are gathered. On the left of this region, increasing data size could still bring significant performance improvement. Beyond this region, however, the performance only shows minimal changes, even when the data ratio is substantially increased. For instance, `Twitter` dataset on NVM-DRAM testbed (Figure 9d) stabilizes at approximately 15% data ratio. Further migrating data to DRAM only gains negligibly. Overall, the results indicate that ATMem can effectively detect the dense regions in graph applications to achieve near ideal performance using small memory capacity.

MCDRAM-DRAM testbed has limited capacity on the high-bandwidth memory, unable to accommodate all data in `rMat27`, `Twitter`, and `Friendster` in `BFS`. Thus, the maximum data ratio in Figure 10 is less than one. We notice that placing data size near the memory capacity, i.e., 16 GB, could sometimes lower performance. Instead, the ATMem identifies optimal regions much smaller than the capacity, avoiding this peformance degradation. The experimental results also indicate that the efficient detection and placement of dense regions in graph applications is essential for performance optimization on heterogeneous memory systems.

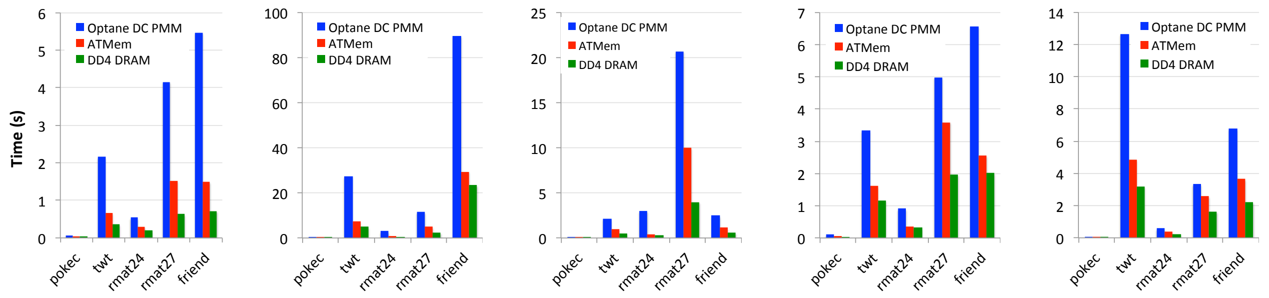
7.3 Data Migration

We evaluate the effectiveness of the multi-stage multi-threaded migration in ATMem by comparing it to the system service `mbind`. In this experiment, each benchmark has two versions of implementation using the two mechanisms, respectively. Table 4 reports the number of TLB misses after migration and the time spent in migration using `mbind` implementation, as normalized to that using our ATMem approach.

The results demonstrate that our ATMem approach dramatically reduces the number of TLB misses after data migration on both testbeds. The improvement in TLB misses on the NVM-DRAM testbed is considerably higher than that on MCDRAM-DRAM testbed. However, the data migration time on MCDRAM-DRAM shows higher speedup than that on the NVM-DRAM testbed. On NVM-DRAM testbed, ATMem reduces the data migration by 1.3×-2.7× (average 2.07×), as compared to `mbind`. On MCDRAM-DRAM testbed, ATMem manages to achieve 3.0×-8.2× (average 5.32×) improvement. The different bandwidth of source memories on the two testbeds likely causes this difference. Migration from NVM to DRAM is bottlenecked at the read bandwidth of the Intel Optane NVM, while data migration from DRAM to MCDRAM can exploit the bandwidth of DRAM. The Intel Optane NVM read bandwidth is reported to be 39 GB/s [25] while DRAM on KNL platform can reach 90 GB/s bandwidth [31].

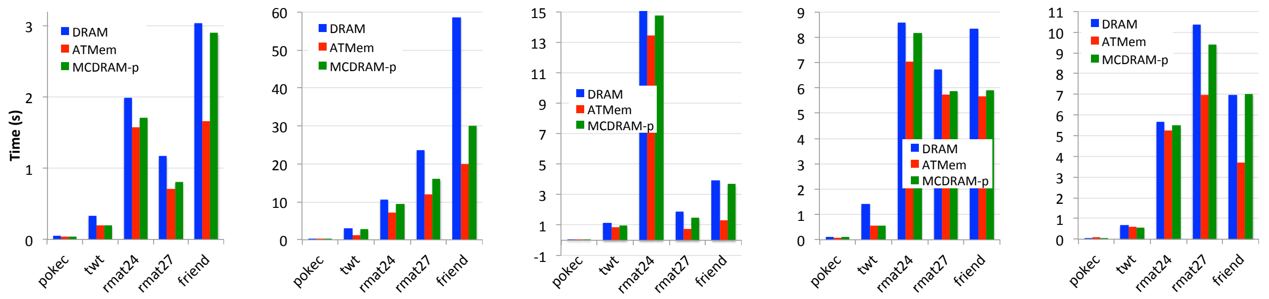
7.4 Overhead Analysis

The overhead of ATMem comes from two sources: profiling and data movement. With the help of the hardware PMU, ATMem incurs minor overhead during profiling, i.e., less



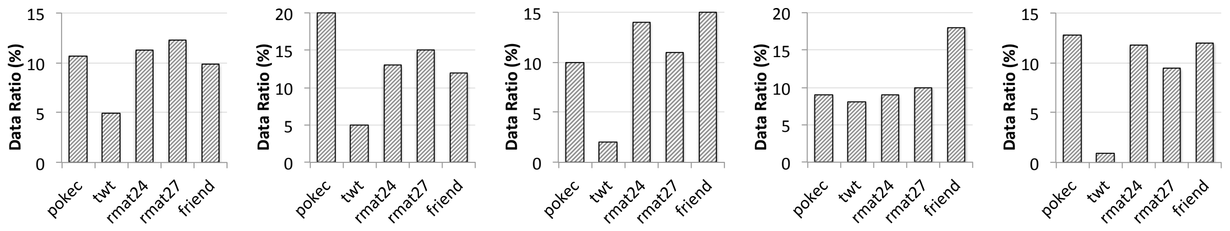
a BFS **b** SSSP **c** PR **d** BC **e** CC

Figure 5. Execution time on NVM-DRAM testbed: NVM-only, NVM-DRAM with ATMem, and DRAM-only.



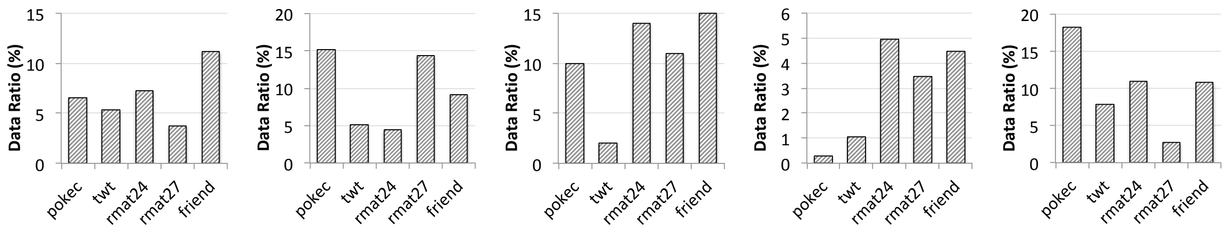
a BFS **b** SSSP **c** PR **d** BC **e** CC

Figure 6. Execution time on MCDRAM-DRAM testbed: DRAM-only, DRAM-MCDRAM with ATMem, and MCDRAM-p.



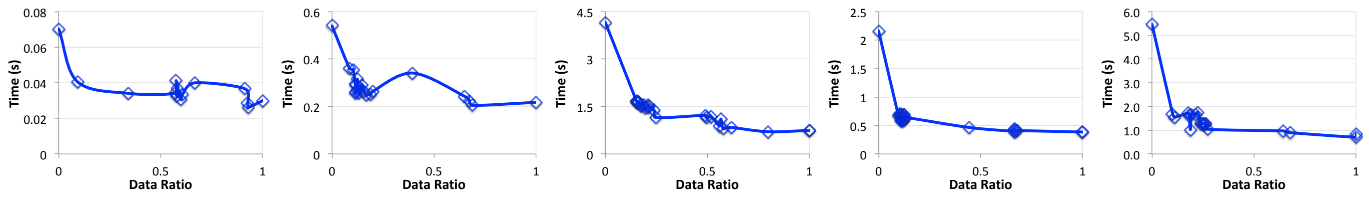
a BFS **b** SSSP **c** PR **d** BC **e** CC

Figure 7. Data ratio on NVM-DRAM testbed: Data ratio is calculated by DRAM data size over total size.



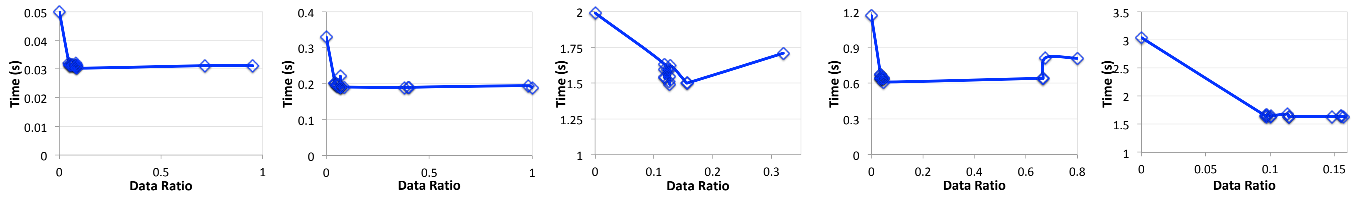
a BFS **b** SSSP **c** PR **d** BC **e** CC

Figure 8. Data ratio on MCDRAM-DRAM testbed: Data ratio is calculated by MCDRAM data size over total size.



a Pokec **b** rMat24 **c** rMat27 **d** Twitter **e** Friendster

Figure 9. Data ratio impact on performance on NVM-DRAM testbed for `BFS`: x-axis is the data ratio placed in DRAM.



a Pokec **b** rMat24 **c** rMat27 **d** Twitter **e** Friendster

Figure 10. Data ratio impact on performance on MCDRAM-DRAM testbed for `BFS`: x-axis is the data ratio placed in MCDRAM.

Table 4. Reduction in TLB Misses and migration time by the multi-stages multi-threaded approach compared to `mbind` in `PR`.

Dataset	**NVM-DRAM**		**MCDRAM-DRAM**	
	TLB misses	Time	TLB misses	Time
pokec	2.09×	1.32×	2.00×	8.26×
rmat24	73.62×	2.71×	2.53×	4.42×
rmat27	15.77×	2.66×	1.17×	5.71×
twitter	1.16×	1.94×	1.64×	3.08×
friendster	12.26×	1.72×	1.25×	5.16×
Avg.	20.98×	2.07×	1.72×	5.32×

than 10% of the first iteration. The data movement overhead depends on the amount of data selected for migration.

The number of iterations required to amortize ATMem's overhead is decided by the kernel and the input data. In our experiments, most benchmarks can get enough benefits to compensate the overhead caused by ATMem within a few iterations. For example, data movement operation incurs 37% overhead for *SSSP* with *Friendster* dataset on HBM for the first (single) iteration. Since ATMem brings over 50% speedup for a single iteration in *SSSP*, the overhead is amortized after only one more iteration.

8 Related Work

Heterogeneous memory systems have been extensively studied recently. Before real hardware became available, many prior efforts used emulators and simulators [9, 18, 20, 22, 29, 32, 34]. Constraint by the gap between emulation and real commodity hardware, some previous findings may need to be revisited. For this consideration, ATMem is evaluated on two real hardware. The following works are related to ATMem.

Data placement. [33] employs a data-centric analysis and a differential analysis to profile and associate each data structure with varied latency and bandwidth configurations. [9] classifies the memory access pattern of each memory region into three classes using an Intel Pin tool. It aims to maximize the overall data placement benefit with a greedy algorithm. [30] provides guidance for data placement by collecting memory traces with Intel Pin-based offline profiling tool. Above works employ offline profiling. More recently, [34] proposes Tahoe, a run-time PMU based data placement tool. Tahoe also uses LLC miss as the main metric to identify the data placement benefit of each data structure. It targets the whole data structure placement. These works do not target graph applications and thus, not considering adaptive granularity or dynamics as ATMem.

Data movement. [21] introduces a new OS-service for asynchronous memory movement with hardware acceleration. Similar efforts in [2, 3, 16, 35] also intend to optimize memory movement at either operating system or architecture level. As system-level services, their solutions need to ensure performance reliability at the cost of extra overhead. While they target future operating system or architecture, ATMem leverages application knowledge by using an application-level mechanism to improve migration on the existing systems.

Data placement on GPUs. Modern GPUs also feature heterogeneous memories, i.e., high-bandwidth GDDR or HBM on the device and DRAM on the host. [30] introduces a Pin based offline profiling tool. [4] proposes a memory specification language to guide data placement on GPUs. Extending our approach on GPUs requires special consideration in CPU-GPU links and data coherence and GPU execution model.

9 Discussion

This section discusses some current limitations and possible generalization in the future work.

Limitations. First, ATMem currently focuses on the performance aspect. Our future work will extend the heuristic in data management to guarantee data consistency (particularly for NVM) when on demand. Second, some HMS architecture could support aggregated bandwidth from memories. For instance, KNL has independent memory channels to MCDRAM and DRAM respectively. In contrast, the Intel Optane NVM is sharing memory channels with DRAM. ATMem will continue enhance placement decisions to utilize both memory bandwidth when supported by the architecture. Third, ATMem migrates data during the iterations of graph execution. Using advanced compiler analysis to automatically insert ATMem API between iterations could overlap the data movement.

Generalization. Although ATMem is specifically presented as an HMS management framework for graph applications, it also works well for other irregular applications or even regular ones because its profiling and data migration mechanisms are generally applicable to any applications. We also evaluated ATMem on sparse matrix computations, such as SpMV, and it achieved similar results as the graph applications. Data accesses in regular applications are more uniformly distributed so that adjusting data chunks to equal size of the whole data structure results in the same data placement as exiting coarse-grained solutions.

10 Conclusion

Active development in new memory devices brings heterogeneous memory systems as a solution to address the scaling challenge in DRAM. Efficient data placement in graph applications on heterogeneous memory systems needs to leverage the advantage of each memory while avoiding their limitations. This work proposes ATMem, an adaptive-grained runtime framework that consists of a lightweight profiler based on hardware sampling, a novel analyzer using *m-ary tree-based* heuristics to classify and predict data regions, and a high-bandwidth migration mechanism at the application level.

ATMem is evaluated on two real heterogeneous memory systems including the latest Intel Optane NVM, with five graph applications. The experimental results show that ATMem can achieves an average of 1.7×-3.4× speedup on NVM-DRAM and 1.2×-2.0× speedup on MCDRAM-DRAM over the baseline by selecting merely 5%-18% data onto high-performance memory. ATMem also helps to bridge the gap between NVM and DRAM on the NVM-DRAM machine, achieving a comparable performance to the case that places all data in the high-performance DRAM. ATMem data migration also outperforms the system service with 2.07×-5.32× speedup.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable and thorough comments. This work was partially supported by a Google Faculty Research Award and a Google gift. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by the Exascale Computing Project (17-SC-20-SC), LLNL-CONF-795320.

References

[1] Friendster network dataset – KONECT, April 2017.

[2] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[3] Santiago Bock, Bruce R Childers, Rami Melhem, and Daniel Mossé. Concurrent migration of multiple pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 420–423. IEEE, 2016.

[4] Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 88–100. IEEE Computer Society, 2014.

[5] HMC Consortium. Hybrid Memory Cube Specification 2.1. http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf, 2015. [Online; accessed 22-May-2018].

[6] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S Nikolopoulos, Bronis R De Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 250–259. ACM, 2008.

[7] Thaleia Dimitra Doudali and Ada Gavrilovska. Mnemo: Boosting memory cost efficiency in hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 412–421. IEEE, 2019.

[8] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007.

[9] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

[10] Vincent W Freeh, Nandini Kappiah, David K Lowenthal, and Tyler K Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. *Journal of Parallel and Distributed Computing*, 68(9):1175–1185, 2008.

[11] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu

unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 224–235, New York, NY, USA, 2019. ACM.

[12] Intel. Intel® 64 and ia-32 architectures software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, May 2019.

[13] JEDEC JESD229-2. Wide I/O 2 (WideIO2), 2014. [Online; accessed 22-May-2018].

[14] JEDEC JESD235A. High bandwidth memory (HBM) DRAM. *JEDEC Solid State Technology Association*, Nov 2015.

[15] JEDEC JESD250. Graphics double data rate 6 (GDDR6) SGRAM standard. *JEDEC Solid State Technology Association*, Jul 2017.

[16] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos—os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534. IEEE, 2017.

[17] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.

[18] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[19] Dong Li, Bronis R de Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[20] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. Enforcing crash consistency of evolving network analytics in non-volatile main memory systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 124–137. IEEE, 2019.

[21] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. *ACM SIGARCH Computer Architecture News*, 44(2):369–383, 2016.

[22] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[23] M Ben Olson, Tong Zhou, Michael R Jantz, Kshitij A Doshi, M Graham Lopez, and Oscar Hernandez. Membrain: Automated application guidance for hybrid memory systems. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE, 2018.

[24] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[25] 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*. ACM, 2019.

[26] Ivy B Peng and Jeffrey S Vetter. Siena: exploring the design space of heterogeneous memory systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 427–440. IEEE, 2018.

[27] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. RTHMS: A Tool for Data Placement on Hybrid Memory System. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, pages 82–91. ACM, 2017.

[28] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graph-Phi: Efficient Parallel Graph Processing on Emerging Throughput-oriented Architectures. In *2018 International Conference on Parallel Architecture and Compilation (PACT)*. ACM, 2018.

[29] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3), 2009.

[30] Du Shen, Xu Liu, and Felix Xiaozhu Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 13–23. ACM, 2016.

[31] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[32] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–362. ACM, 2019.

[33] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 263–273. ACM, 2018.

[34] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 31. IEEE Press, 2018.

[35] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 331–345, New York, NY, USA, 2019. ACM.