Wu K, Li D. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory for high performance computing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(1): 90–109 Jan. 2021. DOI 10.1007/s11390-020-0942-z

Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory for High Performance Computing

Kai Wu and Dong Li*

Department of Electrical Engineering and Computer Science, University of California Merced, Merced 95343, U.S.A.

E-mail: {kwu42, dli35}@ucmerced.edu

Received August 25, 2020; accepted December 30, 2020.

Abstract Non-volatile memory (NVM) provides a scalable and power-efficient solution to replace dynamic random access memory (DRAM) as main memory. However, because of the relatively high latency and low bandwidth of NVM, NVM is often paired with DRAM to build a heterogeneous memory system (HMS). As a result, data objects of the application must be carefully placed to NVM and DRAM for the best performance. In this paper, we introduce a lightweight runtime solution that automatically and transparently manages data placement on HMS without the requirement of hardware modifications and disruptive change to applications. Leveraging online profiling and performance models, the runtime solution characterizes memory access patterns associated with data objects, and minimizes unnecessary data movement. Our runtime solution effectively bridges the performance gap between NVM and DRAM. We demonstrate that using NVM to replace the majority of DRAM can be a feasible solution for future HPC systems with the assistance of a software-based data management.

Keywords data management, non-volatile memory, runtime system

1 Introduction

Non-volatile memory (NVM), such as phase change memory (PCM) and resistive random-access memory (ReRAM), is a promising technique to build future high performance computing (HPC) systems. The popularity of many-core platforms in HPC and large datasets in scientific simulations drives the fast development of NVM, because NVM can provide a scalable and power-efficient solution as main memory, alternative to DRAM (dynamic random access memory). Such a solution is based on the attractive characteristics of NVM, such as a higher density and near-zero static power consumption.

However, compared with DRAM, NVM as main memory can be challenging. The promising NVM solutions (e.g., PCM and ReRAM), although providing larger capacity at the similar or lower cost than DRAM, can have a higher latency and lower bandwidth (see Table 1). Such NVM features can introduce a big performance gap between emerging NVM-based and traditional DRAM-based systems for HPC applications. Our initial performance evaluation with HPC workloads (Section 2) shows that there is 1.09x–8.4x slowdown on NVM-based systems, depending on bandwidth and latency features of NVM. Because of the limitation of NVM, NVM is often paired with a small fraction of DRAM to form a heterogeneous memory system (HMS) [1–7]. By selectively placing frequently-accessed data in the small amount of DRAM available in HMS, we are able to exploit the cost and scaling benefits of NVM while minimizing the limitation of NVM with DRAM.

To manage the data placement on HMS for HPC, we have several goals. First, we want to avoid disruptive changes to hardware. The existing hardware-based

Regular Paper

Special Section on Memory-Centric System Research for High-Performance Computing

A preliminary version of the paper was published in the proceedings of SC 2017.

This work was partially supported by the U.S. National Science Foundation under Grant Nos. CNS-1617967, CCF-1553645, and CCF1718194.

*Corresponding Author

Table 1. NVM Performance Characteristics and Comparison Between NVM Techniques and DRAM [13, 14]

	Read Time (ns)	Write Time (ns)	Random Read Bandwidth (MB/s)	Random Write Bandwidth (MB/s)
DRAM	10	10	10 000	9 000
STT-RAM (ITRS'13)	60	80	800	600
PCRAM	20–200	80–10 000	200–800	100–800
ReRAM	10–1 000	10–10 000	20–100	1–8
Optane PM	174–304	100–190	3 900	1 300

solutions to manage the data placement on HMS [5, 8–10] may be difficult to be embraced by the HPC data centers, because of the concerns on hardware cost. Second, we want to minimize changes to applications and system software. HPC legacy applications should be easily ported to NVM-based HMS with few programming efforts. Third, managing data placement should be as transparent as possible. We want to enable automatic data placement, and relieve users from managing data placement details.

In this paper, we introduce a software-based solution to decide and place data objects on NVM-based HMS. Using a software-based solution to meet the above goals must address the following research challenges.

First, how to capture and characterize memory access patterns associated with data objects? This question is important for making data placement decisions. As we show in Section 2, after we move some data object from NVM with less memory bandwidth to DRAM, there is a big performance improvement. However, we do not have such performance improvement after moving this data object from NVM with longer access latency to DRAM. We claim such data object is sensitive to memory bandwidth. Similarly, we find some data object which is only sensitive to memory latency, or sensitive to both bandwidth and latency. Characterizing data objects based on their sensitivity to bandwidth or latency is critical to model and predict the performance benefit of data placement.

Second, how to strike a balance between different requirements on the frequency of data movement (i.e., the implementation of data placement)? On the one hand, we want data movement to be frequent such that data placement is adaptive to the variation of memory access patterns across execution phases. On the other hand, we want to minimize the data movement to avoid performance loss.

Third, how to minimize the impact of data movement on application performance? Data movement is known to be expensive in terms of performance and energy cost. Hiding data movement cost and achieving high performance are a key to be successful in the HPC domain.

In this paper, we introduce a runtime system (named "Unimem") that automatically and transparently decides and implements the data placement. This runtime system meets the above goals and addresses the above three challenges. In particular, we employ online profiling based on performance counters to capture memory access patterns for execution phases, based on which we characterize the sensitivity of data objects in each phase to memory bandwidth and latency. This addresses the first challenge. We further introduce lightweight performance models, based on which we predict performance benefit and cost if moving data objects between NVM and DRAM. Given the performance benefit and cost of data movement, we formulate the problem of deciding optimal data placement as a knapsack problem. Based on the performance models and formulation, we avoid unnecessary data movement while maximizing the benefits of data movement. This addresses the second challenge.

To avoid the impact of data movement on application performance, we introduce a proactive data movement mechanism. Given an execution phase and a data movement plan for the phase, this mechanism uses a helper thread to trigger the data movement before the phase. The helper thread runs in parallel with the application, overlapping data movement with application execution. This proactive data movement mechanism takes data movement overhead off the critical path, which addresses the third challenge. To further improve performance, we introduce a series of techniques, including 1) optimizing initial data placement to reduce data movement cost at runtime, 2) exploring the tradeoff between phase local search and cross-phase global search for optimal data placement, and 3) decomposing large data objects to enable fine-grained data movement. Altogether, these techniques in combination with our performance models greatly narrow the performance gap between NVM and DRAM.

In summary, we make the following contributions.

• We study the performance of HPC workloads with

large datasets on multiple nodes with various NVM bandwidth and latency, which is unprecedented. Our study reveals a big performance gap between NVM-based and DRAM-based main memories. We demonstrate the feasibility of using a runtime-based solution to narrow such a gap for HPC.

• We introduce a lightweight message passing interface (MPI) runtime system to manage the data placement without hardware modifications and disruptive changes to applications and system software.

• We evaluate Unimem with six representative HPC workloads and one production code (Nek5000) using Intel Optane DC persistent memory and a DRAM-based simulator with various memory latency and bandwidth. Unimem significantly narrows down the performance gap between DRAM-only and NVM-only by 78.4%. It outperforms a software-based solution for HPC by up to 19%.

This work is built based on the conference paper by Wu *et al.*[11] The extension introduces a new profiling method (see in Subsection 3.1) that distinguishes read and write operations, and a new evaluation of Unimem on real NVM hardware.

2 Background

In HMS, we assume that DRAM shares the same physical address space as NVM (but with different addresses) and DRAM memory allocation can be managed at the user level. This assumption has been widely used in the existing work[1–7].

2.1 Definitions and Basic Assumptions

We target at the MPI programming model. For a parallel application based on MPI, we decompose the application into phases. A phase can be a computation phase delineated by MPI operations; a phase can also be an MPI communication phase doing collective operations, point-to-point communication operations, or synchronization. For a non-blocking communication (e.g., `MPI_Isend`), the MPI communication call is not a phase. Instead, it is merged into the immediately following phase. The communication completion operation (e.g., `MPI_Wait`) is a communication phase.

Furthermore, we target at parallel applications from the HPC domain with an iterative structure. In those applications, each program phase is executed many times. Such parallel applications are very common. As an example, Fig.1 depicts a typical iterative structure from CG (an NAS parallel benchmark[12]), which dominates the execution time of CG.

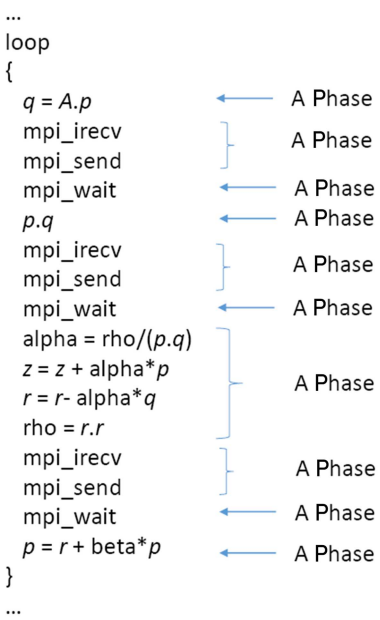

Fig.1. Conceptual description for an MPI-based program (CG) decomposed into phases. $\boldsymbol{A}$ is a 2D matrix, and $\boldsymbol{q}$, $\boldsymbol{p}$, $\boldsymbol{z}$, and $\boldsymbol{r}$ are vectors.

We claim a data object is bandwidth-sensitive, if there is a big performance difference between placing it on NVM with lower memory bandwidth and on DRAM with higher memory bandwidth. We claim a data object is latency-sensitive, if there is a big performance difference between placing it on NVM with longer memory access latency and on DRAM with shorter memory access latency.

2.2 Preliminary Performance Evaluation with NVM-Based Main Memory

NVM has relatively long access latency and low memory bandwidth. In addition, NVM has asymmetric read and write performance. Table 1 shows NVM performance characteristics. The table is based on [14] gathering a comprehensive survey of 340 non-volatile memory technology papers published between 2000 and 2014 in relevant conferences. The recent released Intel Optane DC Persistent Memory Module (PMM)① is the first mass-production of byte-addressable NVM. We also include its performance characteristics reported

①Intel, Inc. Myrinet Express (MX): A high-performance, low-level, message passing interface for Myrinet. http://www.myri.com/scs/MX/doc/MX.pdf, Dec. 2020.

in [13] in Table 1. Based on such performance characteristics, we perform the preliminary performance study to quantify the impact of NVM on the HPC application performance.

We use Quartz, a DRAM-based, lightweight performance emulator for NVM [15]. The existing work uses the cycle-accurate simulation to study the NVM performance [7, 16]. However, the long simulation time makes it impossible to simulate HPC applications with large datasets on multiple nodes. The performance of HPC workloads on NVM is always mysterious. Using Quartz, we can study the performance (execution time) of HPC workloads with much shorter time. We deploy our tests on four nodes in platform A (the configurations of those nodes and platform A are summarized in Section 5). We change the emulated NVM bandwidth and latency, and run a set of NAS parallel benchmarks. We use class D as input and run 16 MPI processes (4 MPI processes per node). For the benchmark Fourier transform (FT), we use class C as input because of the long execution time with class D. Fig.2 and Fig.3 show the emulation results.

Observation 1. We find a big performance gap between DRAM-only and NVM-only systems. This observation is contrary to an existing conclusion (i.e., no big gap) for HPC workloads based on a single node simulation [16]. Furthermore, HPC application performance (execution time) is sensitive to different NVM technologies with various bandwidth and latency. With the memory bandwidth reduced by only 1/2 or the latency increased by only 2x in NVM, some benchmarks have already shown a big slowdown. For example, LU has 2.19x and 2.14x slowdown with NVM configured with 1/2 DRAM bandwidth (Fig.2) and 2x DRAM latency (Fig.3) respectively.

We further study whether the data placement in

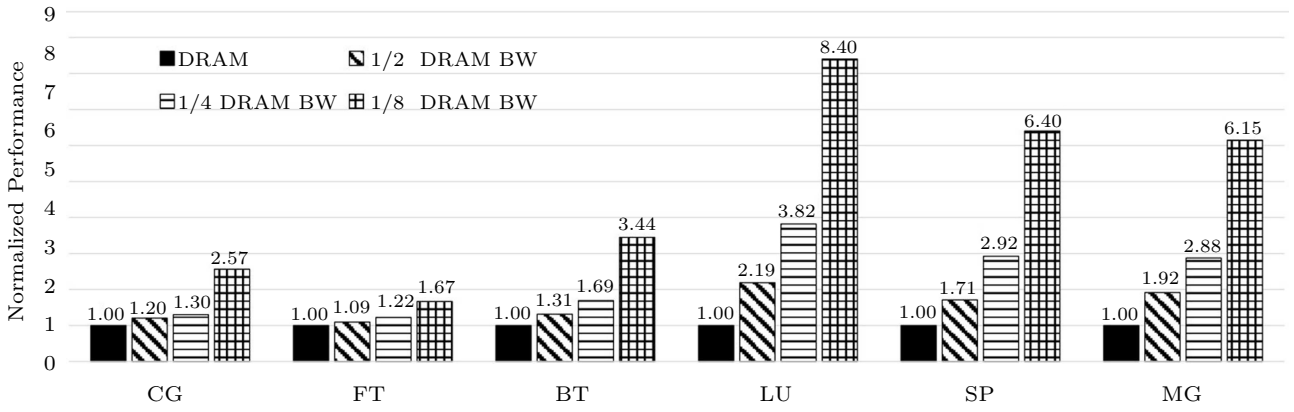

Fig. 2. Benchmark performance (execution time) on NVM-based main memory (NVM-only) with various bandwidth (BW). The performance is normalized to that of DRAM-only systems.

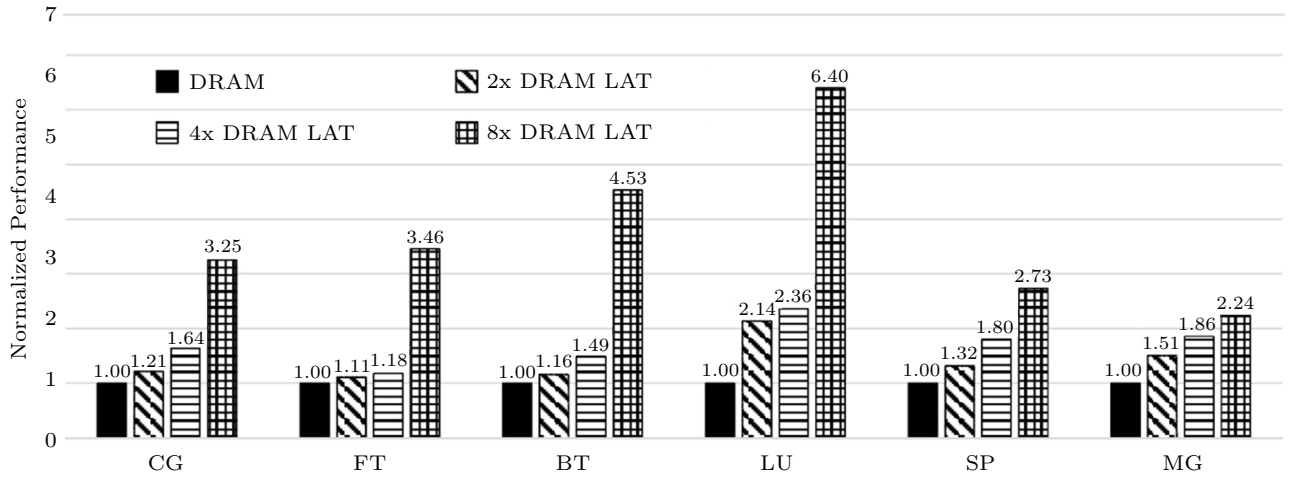

Fig.3. Benchmark performance (execution time) on NVM-based main memory (NVM-only) with various latency (LAT). The performance is normalized to that of DRAM-only systems.

HMS can bridge the performance gap between DRAM-based and NVM-based systems. We choose the SP benchmark and focus on four critical data objects of SP (the arrays *lhs*, *rhs*, *in_buffer* and *out_buffer*). We use two configurations for NVM, one with 1/2 DRAM bandwidth and the other with 4x DRAM latency. For each data object with an NVM configuration (either 1/2 DRAM bandwidth or 4x DRAM latency), we do three tests. In the first test, we use a DRAM-only system. In the second test, we use a DRAM+NVM system. For this test, a target data object is placed in DRAM (see the legend entries in Fig.4), while the rest of data objects are placed in NVM. In the third test, we use an NVM-only system. In each test, we use four nodes with one MPI task per node, and use class C and class D as input. Fig.4 shows the results. The results are normalized to the performance of DRAM-only.

Observation 2. A good data placement can effectively bridge the performance gap. For example, with the data object *lhs* placed in DRAM, we bridge the performance gap between DRAM and NVM (using the configuration of 4x DRAM latency and class C) by 31% (see Fig.4).

Observation 3. Different data objects manifest different sensitivity to limited NVM bandwidth and latency, shown in Fig.4. For example, for the data objects *in_buffer* and *out_buffer* (class D), there is no big performance difference (2.1 vs 2.15) between placing them in DRAM and placing them in NVM configured with 4x DRAM latency. However, there is a big performance difference (1.14 vs 1.25) between placing them in DRAM and placing them in NVM configured with 1/2 DRAM bandwidth (class D). This indicates that the two data objects are sensitive to memory bandwidth but not memory latency. *lhs* (class D) tells us a different story: it is sensitive to latency (1.71 vs 2.15), but not bandwidth (1.21 vs 1.25). Also, *rhs* is sensitive to both latency and bandwidth.

Different data objects have different memory access patterns which manifest a different sensitivity to bandwidth and latency. A data object with a memory access pattern of bad data locality and massive, concurrent memory accesses (e.g., streaming pattern) is sensitive to memory bandwidth, while a data object with a memory access pattern of bad data locality and dependent memory accesses (e.g., pointer-chasing) is sensitive to memory latency.

Our preliminary performance study highlights the importance of capturing memory access patterns of data objects. It also shows us that it is possible to bridge the performance gap between NVM and DRAM by appropriately directing data placement on HMS.

3 Design and Implementation

Motivated by the preliminary performance study, we introduce a runtime system (named "Unimem") targeting at directing data placement on HMS for HPC applications.

Unimem directs the data placement for data objects (e.g., multi-dimensional arrays). The data objects must be allocated using certain Unimem APIs by the programmer. We call those data objects, the target data objects, in the rest of the paper. Unimem is phase-

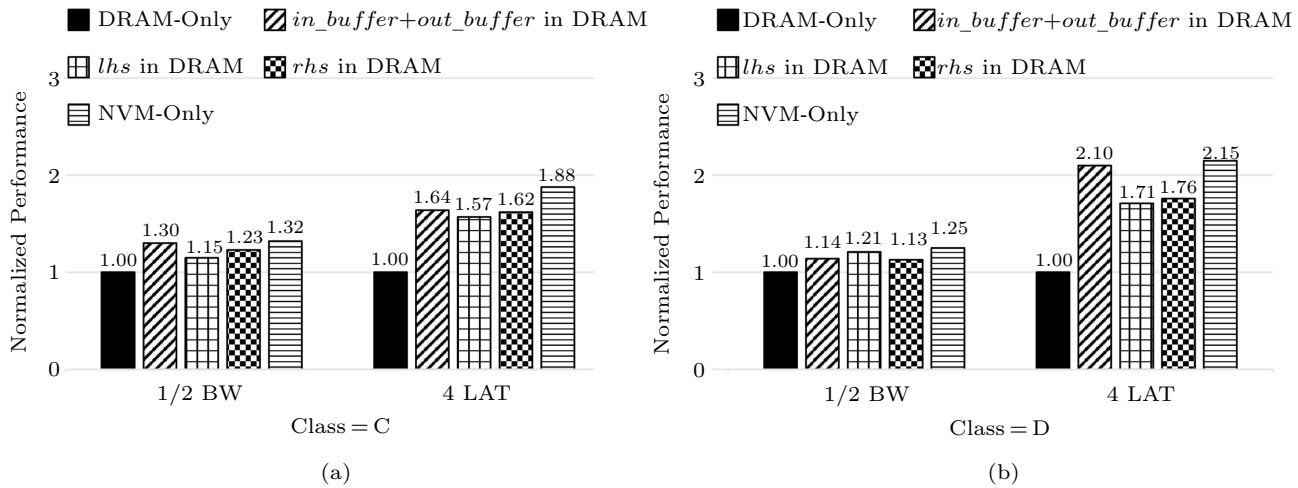

Fig. 4. Impact of data placement on performance (execution time) of NVM-based main memory. The performance is normalized to DRAM-only systems. The legend entries "in_buffer+out_buffer", "*lhs*", and "*rhs*" are the data objects placed in DRAM in the DRAM+NVM system. The x axis shows the configuration of NVM (4x DRAM latency or 1/2 DRAM bandwidth).

based. It decides and changes the data placement for target data objects for each phase based on runtime profiling and lightweight performance models.

In particular, Unimem profiles the memory references to target data objects with a few invocations of each phase. Then Unimem uses performance models to predict the performance benefit and cost of data placement, and formulates the problem of deciding the optimal data placement as a knapsack problem. The results of the performance models and formulation direct data placement for each phase in the rest of the application execution. We describe the design and implementation details in this section.

3.1 Design

Unimem includes three steps in its workflow: phase profiling, performance modeling, and data placement decision and enforcement. The phase profiling happens in the first two iterations of the main computation loop of the application. At the end of the first two iterations, we build performance models and make data placement decision. After the first two iterations, we enforce the data placement decision for each phase. We describe the three steps in details as follows.

3.1.1 Phase Profiling

This step collects the memory access information for each phase. This information is leveraged by the second and the third steps to decide the data placement for each phase.

We rely on hardware performance counters widely deployed in modern processors. In particular, we collect the number of store and load events, and then map the event information to data objects. Leveraging the common sampling mode in performance counters (e.g., precise event-based sampling from Intel or instruction-based sampling from AMD), we collect memory addresses whose associated memory references cause memory stores or loads. These memory addresses help us identify target data objects that have frequent memory accesses in main memory.

Note that the number of loads and stores can reflect how intensive data accesses happen within a fixed sampling interval. It works as an indication for which target data objects potentially suffer from the performance limitation of NVM. However, load and store events do not filter out cache effects. The current performance counter supports counting the last-level cache misses in sampling mode, which can eliminate the impact of the cache. However, that event does not support distinguishing read and write. NVM has asymmetric performance on read and write. In addition, we find that not distinguishing between reads and writes can significantly affect the effectiveness of data placement decisions and lose application performance. Hence, we cannot include that event. Instead, we use the load and store events, which account for a large part of the main memory access, and also allow Unimem to distinguish reads and writes to better estimate the performance benefit of data placement. According to the experimental results in the evaluation section, the load and store events can work as a reliable indicator to direct data placement. To compensate for the potential inaccuracy caused by the limitation of performance counters, we introduce constant factors in the performance models in step 2.

3.1.2 Performance Modeling

Given the memory access information collected for each phase, we select those target data objects that have memory accesses recorded by performance counters. These data objects are potential candidates to move from NVM to DRAM. To decide which target data objects should be moved, we introduce lightweight performance models.

General Description. The performance models estimate performance benefit ((2) and (3)) and data movement cost ((6)) between NVM and DRAM. We trigger data movement only when the benefit outweighs the cost. To calculate the performance benefit, we must decide if the data object is bandwidth-sensitive or latency-sensitive ((1)). This is necessary to model the performance difference between bandwidth-sensitive and latency-sensitive workloads.

Bandwidth Sensitivity vs Latency Sensitivity. To decide if a target data object in a phase is bandwidth-sensitive or latency-sensitive, we use (1). This equation estimates main memory bandwidth consumption due to memory accesses to the data object (BW_{data_obj}).

$$BW_{data\_obj} = \#data\_access \times cacheline\_size / \left(\frac{\#samples\_with\_data\_accesses}{\#samples} \times phase\_execution\_time \right). \quad (1)$$

The numerator of (1) is the accessed data size. $\#data\_access$ in the numerator is the number of memory accesses to the data object in main memory.

$\#data\_access$ is the sum of the number of stores and loads collected in step 1 (phase profiling) with performance counters. For a target data object in a phase, the accessed total data size is calculated as $(\#data\_access \times cacheline\_size)$.

The denominator of (1) is the fraction of the execution time that has memory accesses to the target data object in main memory. This fraction of the execution time is calculated based on $\#samples\_with\_data\_accesses/\#samples$, which is the ratio between the number of samples that collect non-zero accesses to the target data object and the total number of samples.

For example, suppose that the phase execution time is 10 seconds, the hardware counter sampling rate is 1 000 cycles, and the CPU frequency is 1 GHz. Then we will have 10^7 samples in total during the phase execution. Assuming that 10^5 samples of all samples have memory accesses to the data object, then the fraction of the execution time that accesses the data object is $10^5/10^7 \times 10 = 0.1$ s.

Given a data object in a phase, if its BW_{data_obj} reaches $t_1\%$ of the peak NVM bandwidth BW_{peak} ($t_1 = 80$ in our evaluation), then this data object is most likely to be bandwidth-sensitive. The performance benefit after moving the data object from NVM to DRAM (i.e., $BFT_{data\_obj\_bw}$) is dominated by the memory bandwidth effect, and can be calculated based on (2), which will be discussed next. If BW_{data_obj} of the data object is less than $t_2\%$ of BW_{peak} ($t_2 = 10$ in our evaluation), then this data object is most likely to be highly latency-sensitive. The performance benefit of moving the data object from NVM to DRAM (i.e., $BFT_{data\_obj\_lat}$) is dominated by the memory latency effect, and can be calculated based on (3), which will be discussed next. If BW_{data_obj} of the data object is between $t_1\%$ and $t_2\%$, then the data object is likely to be sensitive to either bandwidth or latency. The performance benefit after data movement from NVM to DRAM is estimated by $\max(BFT_{data\_obj\_bw}, BFT_{data\_obj\_lat})$. To measure BW_{peak}, we run a highly memory bandwidth intensive benchmark, the STREAM benchmark②, with maximum memory concurrency, and use (1) and performance counters.

Calculation of Data Movement Benefit. (2) and (3) calculate performance benefits (after the data movement from NVM to DRAM) for bandwidth-sensitive and latency-sensitive data objects, respectively. The two equations are simply based on an estimation of the performance difference between running the applications on NVM and on DRAM. If the data object is bandwidth-sensitive, then the application performance on a specific memory is modeled by $accssed\_data\_size/mem\_bw$ (mem is NVM or DRAM). $accessed\_data\_size$ is $\#data\_access \times cacheline\_size$, the same as the one in (1). If the data object is latency-sensitive, then the application performance on a specific memory is modeled by $\#data\_access \times mem\_lat$ (mem is NVM or DRAM).

$$\begin{aligned} & BFT_{data\_obj\_bw} \\ = & \left(\frac{\#data\_access \times cacheline\_size}{NVM\_bw} - \right. \\ & \left. \frac{\#data\_access \times cacheline\_size}{DRAM\_bw}\right) \times CF\_bw, \quad (2) \\ & BFT_{data\_obj\_lat} \\ = & (\#data\_access \times NVM\_lat - \\ & \#data\_access \times DRAM\_lat) \times CF\_lat. \quad (3) \end{aligned}$$

In (2) and (3), we have constant factors CF_bw (see (2)) and CF_lat (see (3)). Such constant factors are used to improve the modeling accuracy. To meet high performance requirement of our runtime, the performance models are rather lightweight, and only capture the critical impacts of memory bandwidth or memory latency. However the models ignore some important performance factors (e.g., overlapping between memory accesses, and overlapping between memory accesses and computation). Also, the limitation of the sampling-based approach to count performance events can underestimate the number of memory accesses due to the sampling nature of the approach. The constant factors CF_bw and CF_lat work as a simple but powerful approach to improve the modeling accuracy without increasing modeling complexity and runtime overhead.

The basic idea of the two factors is to measure performance ratios between measured performance and predicted performance for representative workloads, and then use the ratios to improve online modeling accuracy for other workloads.

In particular, we run the bandwidth-sensitive benchmark STREAM to obtain CF_bw offline. We calculate the performance ratio between the predicted performance and the measured performance, and such a ratio is CF_bw. The predicted performance is calculated based on $(\#data\_access \times cacheline\_size/DRAM\_bw)$,

②McCalpin JD. STREAM: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream, March 2017.

where $\#data\_access$ is collected with performance counters using the sampling-based approach. Hence, CF_bw accounts for the potential performance difference between our sampling-based modeling and real performance. The constant factor CF_lat is obtained in the similar way, except that we use a latency-sensitive benchmark, the pointer-chasing benchmark[③] (using a single thread and no concurrent memory accesses). Also, to calculate the predicted performance, we use $(\#dataaccess \times DRAM\_lat)$. Given a hardware platform, CF_bw and CF_lat need to be calculated only once.

Accounting for Performance Difference Between Read and Write. NVM is featured with asymmetric read and write performance. The performance difference between read and write operations can be as large as 50x in terms of latency and 8x in terms of bandwidth (see PCRAM in Table 1). We account for the performance difference in performance modeling. In particular, we extend (2) and (3) by counting $\#load$ and $\#store$ separately (see (4) and (5)). The numbers of $\#load$ and $\#store$ are measured in step 1 (phase profiling).

$$\begin{aligned} & BFT_{data\_obj\_bw} \\ = & \left(\frac{\#load \times cacheline\_size}{NVM\_bw_{read}} + \right. \\ & \frac{\#store \times cacheline\_size}{NVM\_bw_{write}} - \\ & \left. \frac{(\#load + \#store) \times cacheline\_size}{DRAM\_bw} \right) \times CF\_bw, \quad (4) \\ & BFT_{data\_obj\_lat} \\ = & (\#load \times NVM\_lat_{read} + \\ & \#store \times NVM\_lat_{write} - \\ & (\#load + \#store) \times DRAM\_lat) \times CF\_lat. \quad (5) \end{aligned}$$

Calculation of Data Movement Cost. Data placement comes with data movement cost. The data movement cost can be simply calculated based on data size and memory copy bandwidth between NVM and DRAM, which is $(data\_size/mem\_copy\_bw)$. To reduce the data movement cost, we want to overlap the data movement with application execution. This is possible with a helper thread that runs in parallel with the application to implement an asynchronous data movement. We discuss this in details in Subsection 3.3. In summary, the data movement cost $(COST_{data\_obj})$ is modeled in (6) with the overlapped cost $(mem\_comp\_overlap)$ included.

$$\begin{aligned} & COST_{data\_obj} \\ = & \max \left(\frac{data\_size}{mem\_copy\_bw} - mem\_comp\_overlap, 0 \right). \quad (6) \end{aligned}$$

We describe how to calculate $mem\_comp\_overlap$ as follows. To minimize the data movement cost, we want to overlap data movement with application execution as much as possible. Meanwhile, we must respect data dependency and ensure execution correctness. This means during data movement, the migrated data object must not be read or written by the application. Given the above requirement on respecting data dependency and minimizing the data movement cost, we can estimate $mem\_comp\_overlap$.

Fig.5 explains how to calculate $mem\_comp\_overlap$ with an example. This example shows how to calculate $mem\_comp\_overlap$ for a data object a in a specific phase (phase i). If a is not in DRAM, we can trigger the data migration of a as early as the beginning of phase j, because a is not referenced between j and i. We cannot trigger the data migration of a at the beginning of phase $j-1$, because a is referenced there. $mem\_comp\_overlap$ is the application execution time between phases j and i. The data movement time, $data\_size/mem\_copy\_bw$, can be smaller than $mem\_comp\_overlap$. In this case, the data movement is completely overlapped with application execution, and the data movement cost $COST_{data\_obj}$ is 0.

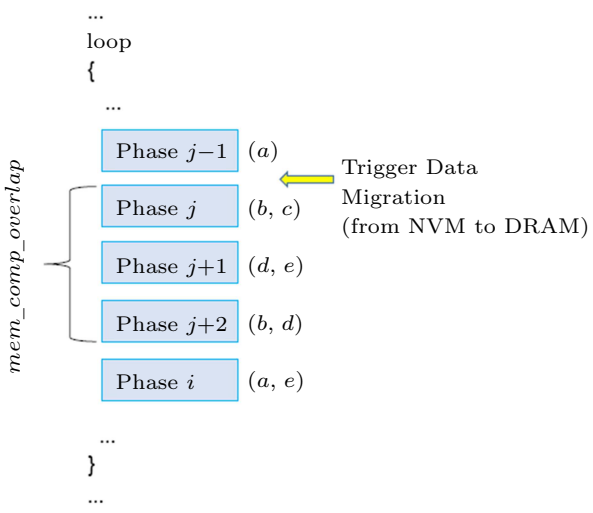

Fig.5. Example to show how to calculate $mem\_comp\_overlap$ for the data object a in phase i. The yellow arrow is the point to trigger the migration of a from NVM to DRAM for phase i, if a is not in DRAM. The letters in brackets represent target data objects referenced in the corresponding phases.

③Besard T. Pointer-chasing memory benchmark. https://github.com/maleadt/pChase, March 2017.

Our estimation on $COST_{data\_obj}$ could be an overestimation (a conservative estimation). In particular, when a data object is to be migrated from NVM to DRAM for a phase, it is possible that the data object is already in DRAM. We use Fig.5 as an example again. Since the phase $j-1$ references a, it is possible that a is already in DRAM before the point to trigger the data migration. Also, $COST_{data\_obj}$ does not include the cost of moving the data from DRAM to NVM when there is no enough space in DRAM and we need to switch the data. Such overestimation and ignorance of data movement from DRAM to NVM are due to the fact that the data movement cost for each phase is isolatedly calculated during the modeling time. Hence, what data objects are in DRAM and whether there is enough space in DRAM are uncertain during the modeling time. We will solve the above problems in the next step (step 3).

3.1.3 Data Placement Decision and Enforcement

Based on the above formulation for the benefit and cost of data movement, we determine data placement for all phases one by one. In particular, to determine data placement for a specific phase, we define a weight w for each target data object referenced in this phase:

$$w = BFT_{data\_obj} - COST_{data\_obj} - extra\_COST_{data\_obj}. \quad (7)$$

$extra\_COST_{data\_obj}$ accounts for the data movement cost, when there is no enough space in DRAM to move the target data object from NVM to DRAM and we have to move the data from DRAM to NVM to save space. To calculate $extra\_COST_{data\_obj}$, we must decide which data object in DRAM must be moved. We make such decision based on the sizes of data objects in DRAM. In particular, we move data objects from DRAM to NVM whose total size is just big enough to allow the target data object to move from NVM to DRAM. Note that since we determine data placements for all phases one by one, when we decide the data placement for a specific phase, we have made the data placement decisions for previous phases. Hence, we have a clear knowledge on which data objects are in DRAM and whether the target data object is already in DRAM.

Besides the weight w, each data object has a data size. Given the DRAM size limitation, our data placement problem is to maximize total weights of data objects in DRAM while satisfying the DRAM size constraint. This is a 0-1 knapsack problem [17].

The knapsack problem can typically be solved by dynamic programming in pseudo-polynomial time. If each data object has a distinct value per unit of weight ($data\_size/w$), the empirical complexity is $O((\log(n))^2)$ [17], where n is the number of target data objects referenced in a phase.

The above approach can determine the data placement for individual phases. We name this approach as "phase local search". Determining data placement at the granularity of individual phases can lead to the optimal data placement for each phase, but result in frequent data movements, some of which may not be able to be completely overlapped by application execution. Alternatively, determining data placement at the granularity of all phases (named "cross-phase global search") has less data movement than phase local search, because all phases are in fact treated as a combined single phase: once the optimal data placement is determined within the combination of all phases, there is no data movement within the combination. However, the optimal data placement for the combination of all phases does not necessarily result in the best performance for each individual phase.

Based on the above discussion, we use dynamic programming to determine the data placement using both phase local search and cross-phase global search, and then choose the best data placement of the two searches.

After we make the data placement decision at the end of the first iteration, we enforce data placement since the second iteration. At the beginning of each phase, the runtime asks a helper thread (see Subsection 3.3 for implementation details) to proactively move data objects between NVM and DRAM based on the data placement decision for future phases.

Fig.6 gives an example for how to enforce data placement with a helper thread after determining data placement. In this example, there are three target data objects (a, b, and c) and five phases. The data placement decision for each phase is represented with letters in brackets (e.g., (a) for the phase 1). We assume DRAM can hold two data objects at most. The data movement enforced by the helper thread respects data dependence across phases and the availability of DRAM space. Such an example is a case of phase local search, where each phase makes its own decision for data placement. There are eight data movements in total. With a cross-phase global search, only two data objects will be moved to DRAM for all phases. The cross-phase global search results in only two data movements. Based on

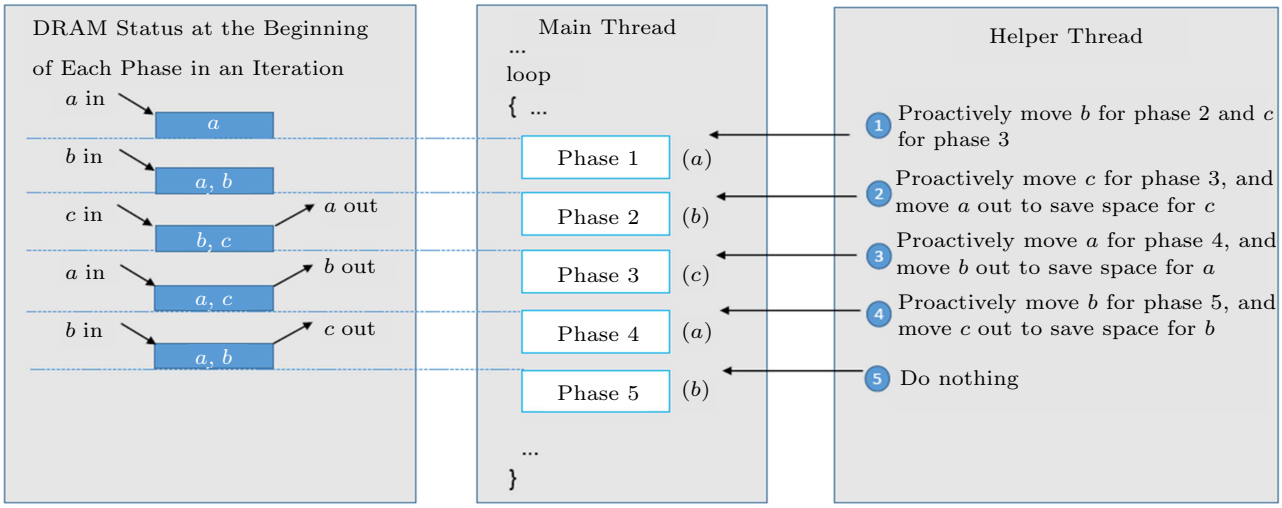

Fig.6. Example to show proactive data migration with a helper thread. The letters in the figure represent data objects. The letters in brackets (e.g., (a) and (b)) represent target data objects that are determined to be placed in DRAM for the corresponding phases. DRAM can hold two data objects at most.

the performance modeling and dynamic programming, we can decide whether the cross-phase global search or the phase local search is better.

3.2 Optimization

To improve runtime performance, we introduce a couple of optimization techniques as follows.

Handling Workload Variation Across Iterations. In many scientific applications, the computation and memory access patterns remain stable across iterations. This means once the data placement decision is made at the end of the first iteration, we can reuse the same decision in the rest of iterations. However, some scientific applications have workload variation across iterations. We must adjust data placement decision correspondingly.

To accommodate workload variation across iterations, Unimem monitors the performance of each phase after data movement. If there is obvious performance variation (larger than 10%), then the runtime will activate phase profiling again and adjust the data placement decision.

Initial Data Placement. By default, all data objects are initially placed in NVM and moved between DRAM and NVM by Unimem at runtime. However, data movement can be expensive, especially for large data objects, even though we use the proactive data movement to overlap data movement with application execution. To reduce the data movement cost, we selectively place some data objects in DRAM at the beginning of the application, instead of placing all data objects in NVM. The existing work has demonstrated the performance benefit of the initial data placement on GPU with HMS [5, 18]. Our initial data placement technique on NVM-based HMS is consistent with those existing efforts.

For initial data placement, we place in DRAM those target data objects with the largest amount of memory references (subject to the DRAM space limitation). To calculate the number of memory references for each target data object, we employ compiler analysis and represent the number of memory references as a symbolic formula with unknown application information, similar to [19]. Such information includes the number of iterations and coefficients of array access. This information is typically available before the main computation loop and before memory allocation for target data objects. Hence it is possible to decide and implement initial data placement before main computation loop for many HPC applications. However, we cannot determine initial data placement for those data objects that do not have the information available before the main computation loop (e.g., the number of iterations is determined by a convergence test at run time).

Our method determines initial data placement simply based on the number of memory references and ignores caching effects. The ignorance of caching effects can impact the effectiveness of initial data placement. In particular, some data objects with intensive memory references may have good reference locality and do not cause a lot of main memory accesses. However, our practice shows that in all cases of our evaluation, ini-

tial data placement based on compiler analysis makes the data placement decision consistent with the runtime data placement decision using the cross-phase global search. Using compiler analysis can work as a practical and effective solution to direct initial data placement, because the target data objects with a large amount of memory references tend to frequently access main memory.

Handling Large Data Objects. We move the data between DRAM and NVM at the granularity of data object. This means a data object larger than the DRAM space cannot be migrated. This problem is common to any software-based data management on HMS.

A method to address the above problem is to partition the large data object into multiple chunks with each chunk smaller than the DRAM size. At runtime, we can profile memory access for each chunk instead of the whole data object, and move data chunk if the benefit overweights the cost of data chunk movement. This method exposes new opportunities to manage the data and improve the performance.

However, this solution is not always feasible, because it can involve a lot of programming efforts to refactor the application such that memory references to the large data object are based on chunk-based partitioning. A compiler tool can be helpful to transform some regular memory references into new ones based on chunk-based partitioning (assuming the input problem size and the number of loop iterations are known). However, this kind of automatic code transformation can be impotent for high-dimensional arrays with the notorious memory alias problem and irregular memory access patterns. In Unimem, we employ a conservative approach which only partitions those one-dimensional arrays with regular memory references.

In our evaluation with representative numerical kernels, we find that partitioning large data objects is often not helpful, because making the data placement decision based on chunks leads to much more frequent data movements, most of which are difficult to be overlapped with application execution and hence exposed to the critical path, but we do have a benchmark (FT) benefit from partitioning large data objects.

3.3 Implementation

We have implemented Unimem as a runtime library to perform the online adaptation of data placement on HMS. To leverage the library, the programmer needs to insert a couple of APIs into the application. Such a change to the application is very limited, and is used to initialize the library and identify the main computation loop and target data objects. In all applications we evaluated, the modification to the applications is less than 20 lines of code. Table 2 lists those APIs and their functionality.

Table 2. APIs for Using Unimem Runtime

API Name	Functionality
`unimem_init`	Initialization for hardware counters, timers and global variables
`unimem_start`	Identify the beginning of the main computation loop
`unimem_end`	Identify the end of the main computation loop
`unimem_malloc`	Identify and allocate target data objects
`unimem_free`	Free memory allocation for target data objects

The runtime library decides data placement at the granularity of execution phase. As discussed before, a phase is delineated by MPI operations. To automatically form phases, we employ the MPI standard profiling interface (PMPI). `PMPI_` function behaves in the same way as `MPI_` function, but PMPI allows one to write functions that have the behavior of the standard function plus any other behavior one would like to add. Based on PMPI, we can transparently identify execution phases and control profiling without programmer intervention. Fig.7 depicts the general idea. In particular, we implement an MPI wrapper based on PMPI. The wrapper encapsulates the functionality of enabling and disabling profiling and uses a global counter to identify phases.

To identify target data objects, the programmer must use `unimem_malloc` to allocate them before the main computation loop. This API is similar to the memory allocation API in the existing NVM management library, such as `pmalloc` in Intel PMDK library[④], which explicitly specifies the type of the memory allocated for the data objects. This API also allows Unimem to collect pointers pointing to target data objects. Collecting these pointers is necessary to implement data movement without asking the programmer to change the application after data movement. In particular, after data movement for a target data object, the runtime changes the data object pointer and makes it point to the new memory space of the data object without disturbing execution correctness. If there is a memory alias to the data object but such an alias

④Intel. Persistent Memory Development Kit. https://pmem.io, Dec. 2020.

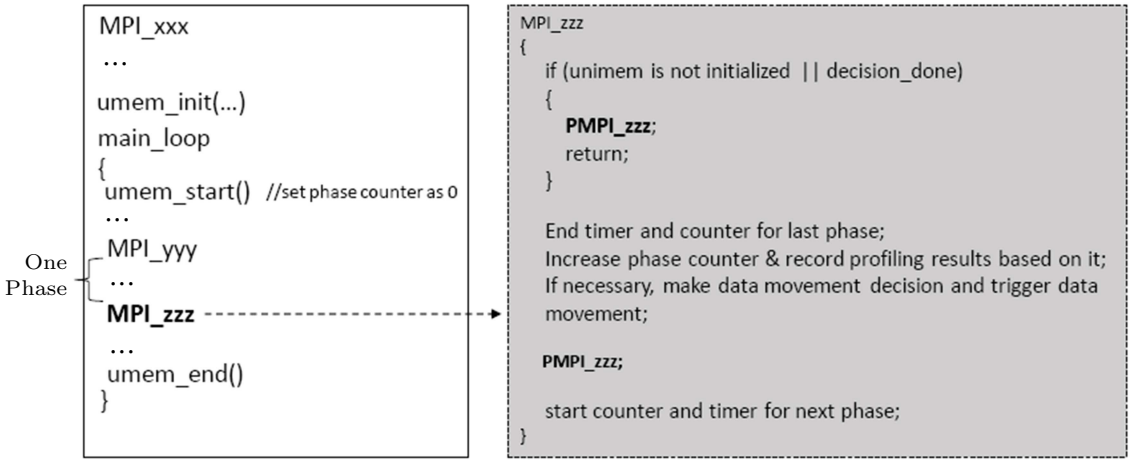

Fig.7. Transparently identifying phases based on PMPI.

is created within the main computation loop, then the memory alias can still work correctly, because it is updated in each iteration and will point to the new memory space of the data object after data movement. If the memory alias to the data object is created before the main computation loop, then such memory alias information must be explicitly sent to the runtime by the programmer using `unimem_malloc` such that the memory alias can be updated and points to the correct memory space after data movement.

The DRAM space is limited in HMS. To manage the DRAM space, we avoid making any change to the operating system (OS), and introduce a user-level service. Each node runs an instance of such a service. The service coordinates the DRAM allocation from multiple MPI processes on the same node. In particular, the service responds to any DRAM allocation request from the runtime, and bounds the memory allocation within the DRAM space allowance. Our current implementation for such service is based on a simple memory allocator without the consideration of memory allocation efficiency and fragmentation, because we expect that data movement should not be frequent, and data allocation for data movement should not be frequent for performance reason. However, an advanced implementation could be based on an existing memory allocator, such as HOARD [20] and the lock-free allocator [21].

As discussed in Subsection 3.1 (see step 2), we use a helper thread to proactively trigger data movement such that data movement is overlapped with application execution. The helper thread is invoked in `unimem_init`. In the main computation loop, the helper thread and the main thread interact through a shared FIFO queue. The main thread puts data movement requests into the queue; the helper thread checks the queue, performs data movement, and removes the data movement request off the queue once the data movement is done. At the beginning of each phase, the runtime of the main thread will check the queue status to determine if all proactive data movements for the current phase are done. Hence, the queue works as a synchronization mechanism between the helper thread and the main thread. Note that checking the queue status and putting data movement requests into the queue are lightweight, because we avoid frequent data movement in our design.

As discussed in Subsection 3.1 (see step 2), to ensure execution correctness, the runtime must respect data dependency across phases when moving data objects with the helper thread. The data dependency check is implemented by static analysis. We introduce an LLVM [22] pass to analyze data references to target data objects between MPI calls. To handle those unresolved control flows during the static analysis, we embed data dependency analysis result for each branch, and delay data dependency analysis until runtime. The compiler-based data dependency analysis can be conservative due to the challenge of pointer analysis [23]. There is also a large body of research related to the approximation of pointer analysis to improve compiler-based data dependency analysis. However, to simplify our implementation, we currently use a directive-based approach that allows the programmer to use directives to explicitly inform the runtime of data dependency for target data objects across phases. This approach is inspired by task dependency clauses in OpenMP, and works as a practical solution to address complicated data dependency analysis. Fig.8 depicts the general workflow.

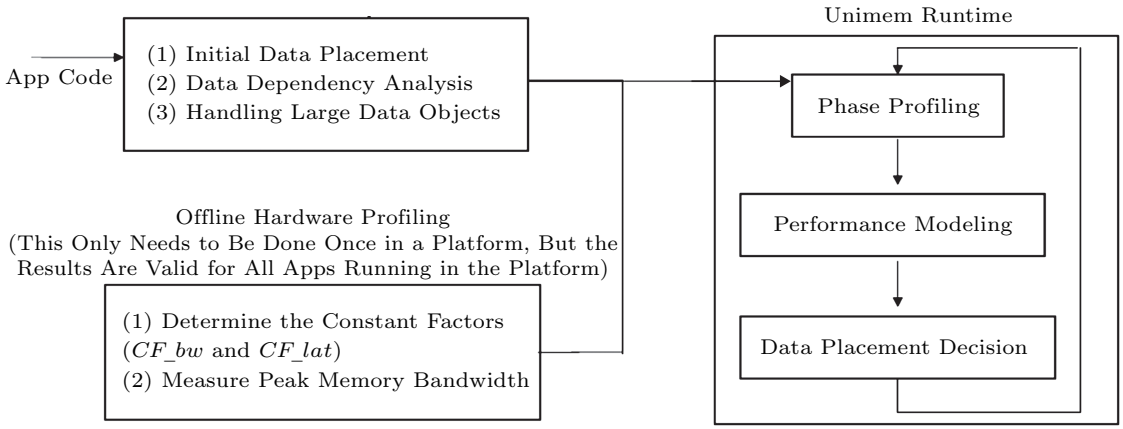

Fig.8. General workflow for Unimem.

4 Evaluation Methodology

In our evaluation, we use Quartz emulator [15]. Quartz enables an efficient emulation of a range of NVM latency and bandwidth characteristics. Quartz has low overhead and good accuracy (with emulation errors of 0.2%–9%) [15]. We do not use cycle-accurate architecture simulators because of their slow simulation which cannot scale to large workloads. Furthermore, Quartz allows us to consider cache eviction effects, memory-level parallelism, and system-wise memory traffic, which is not available in other state-of-the-art, software-based emulation approaches⑤ [24]. However, due to the limitation of Quartz, we can only emulate either bandwidth limitation or latency limitation, but cannot emulate both of them.

Using Quartz requires the user to have privilege access to the test system. We do not have such privilege access on the test platform for our strong scaling tests. Hence, instead of using Quartz, we leverage the NUMA architecture to emulate NVM. In particular, we carefully manage data placement at the user level such that, given an MPI task, a remote NUMA memory node works as NVM while the NUMA node local to the MPI task works as DRAM. The latency and bandwidth difference between the remote and the local NUMA memory nodes emulates that between NVM and DRAM. On our test platform for strong scaling tests, the emulated NVM has 60% of DRAM bandwidth and 1.89x of DRAM latency.

In addition to the above emulated NVM platform, we use a real NVM platform based on Intel Optane PMM for evaluation. This Optane-based machine has two sockets, each with two integrated memory controllers (iMCs) and six memory channels. In total, the system has 192 GB DRAM and 1.5 TB PMM on two sockets. There are two operating modes for PMM, Memory Mode and App-direct Mode. In Memory Mode, DRAM works as a hardware-managed, direct-mapped, write-back cache to PMM. In this mode, the program cannot control data placement on DRAM and PMM. On the contrary, App-direct Mode allows the runtime system to explicitly control memory accesses to DRAM and PMM. In App-direct Mode, PMM on each socket can be exposed as a non-uniform memory access (NUMA) node to CPUs. Standard NUMA management mechanisms such as *numactl* can be used to control data placement in this mode. We use PMM in App-direct Mode exposed as NUMA nodes; we use *numactl* to control data placement on DRAM and PMM.

In general, we have three test platforms for performance evaluation. The first test platform (named "platform A") is a small cluster. Each node of this platform has two 8-core Intel Xeon E5-2630 processors (2.4 GHz) and 32 GB DDR4. We use this platform for all tests except the scalability study and the performance study on Optane PMM. We deploy Quartz on this platform. The second test platform is the Edison supercomputer at Lawrence Berkeley National Lab (LBNL). We use this platform for the scalability tests. Each Edison node has two 12-core Intel Ivy Bridge processors (2.4 GHz) with 64 GB DDR3. As discussed before, we perform strong scaling tests and leverage NUMA architectures to emulate NVM on this system. The third test platform is an Optane PMM-based machine. Table 3 summarizes hardware features of the Optane platform.

⑤Macko P. PCMSIM: A simple PCM block device simulator for Linux. https://code.google.com/p/pcmsim, Dec. 2020.

Table 3. Platform Specifications

Hardware	Specification
Processor	2nd Gen Intel® Xeon™ Scalable processor
Cores	2.4 GHz (3.9 GHz Turbo frequency) × 24 cores (48 HT) × 2 sockets
L1-icache	Private, 32 KB, 8-way set associative, write-back
L1-dcache	Private, 32 KB, 8-way set associative, write-back
L2-cache	Private, 1 MB, 16-way set associative, write-back
L3-Cache	Shared, 35.75 MB, 11-way set associative, non-inclusive write-back
DRAM	Six 16-GB DDR4 DIMMs × 2 sockets (192 GB in total)
PM	Six 128-GB Optane DC NVDIMMs × 2 sockets (1.5 TB in total)
Interconnect	Intel® UPI at 10.4 GT/s, 10.4 GT/s, and 9.6 GT/s

We use six benchmarks from NAS parallel benchmark (NPB) suite 3.3.1, and one production scientific code Nek5000[6]. For Nek5000, we use eddy input problem with a 256 × 256 mesh. The target data objects of those benchmarks are listed in Table 4. Those data objects are the most critical data objects accounting for more than 95% of memory footprint except CG and Nek5000. For CG, there are three large data objects (*aelt*, *acol*, and *arow*) only used for problem initialization. They are not treated as target data objects. For Nek5000, we use main simulation variables and geometry arrays in Nek5000 core. Those are the most important data objects for Nek5000 simulation. We use GNU compiler (4.4.7 on Platform A, 6.1.0 on Edison and 8.6.0 on Optane platform) and use default compiler options for building benchmarks. We use the sampling-based approach to collect performance events on the three platforms. The sampling interval is chosen as 1 000 CPU cycles such that the sampling overhead is ignorable while the sampling is not sparse to lose modeling accuracy.

5 Evaluation

The goal of our evaluation is multiple-folding. First, we want to test if our runtime can effectively direct data placement to narrow the performance gap between NVM and DRAM. Second, we want to test if our runtime is lightweight enough. Third, we want to test the performance of our runtime in various system configurations, including different DRAM sizes and different system scales. Unless indicated otherwise, the performance in this section is normalized to that of the DRAM-only system.

Table 4. Target Data Objects in NPB Benchmarks and Nek5000

Benchmark	Target Data Object	Percentage of Total Application Memory (%)
CG	$colidx$, a, w, z, p, q, r, $rowst$, x	42
FT	u, $u0$, $u1$, $u2$, $twiddle$	99
BT	rhs, $forcing$, u, us, vs, ws, qs, rho_i, $square$, out_buffer, in_buffer, $fjac$, $njac$, $lhsa$, $lhsb$, $lhsc$	99
LU	u, rsd, $frct$, $flux$, a, b, c, d, buf, $buf1$	99
SP	u, us, vs, ws, qs, rho_i, $square$, rhs, $forcing$, out_buffer, in_buffer, lhs	98
MG	$buff$, u, v, r	99
Nek5000 (eddy)	Geometry arrays and main simulation variables (48 data objects in total)	35

Basic Performance Tests. We compare the performance (execution time) of DRAM-only, NVM-only, and HMS with Unimem. We use four nodes in platform A with one MPI task per node. We use class C as the input problem for NPB benchmarks. NVM and DRAM sizes are 16 GB and 256 MB respectively. Figs.9 and 10 show the results. NVM is configured with 1/2 DRAM bandwidth (Fig.9) or 4x DRAM latency (Fig.10).

We first notice that there is a big performance gap between NVM-only and DRAM-only cases. On average, the gap is 18% for NVM with 1/2 DRAM bandwidth and 47% for NVM with 4x DRAM latency. However, Unimem greatly narrows the gap and makes performance very close to DRAM-only cases: the average performance difference between DRAM-only and HMS is only 3% for NVM with 1/2 DRAM bandwidth and 7% for NVM with 4x DRAM latency, and the performance difference is no bigger than 10% in all cases. This demonstrates that Unimem successfully directs the data placement for those performance-critical data objects. This also demonstrates that Unimem is very lightweight after we optimize runtime performance and hide data movement cost.

We compare Unimem and X-Mem [1] (a recent software-based solution for data placement in HMS).

[6] Fischer P, Lottes J. Nek5000. http://nek5000.mcs.anl.gov, Dec. 2020.

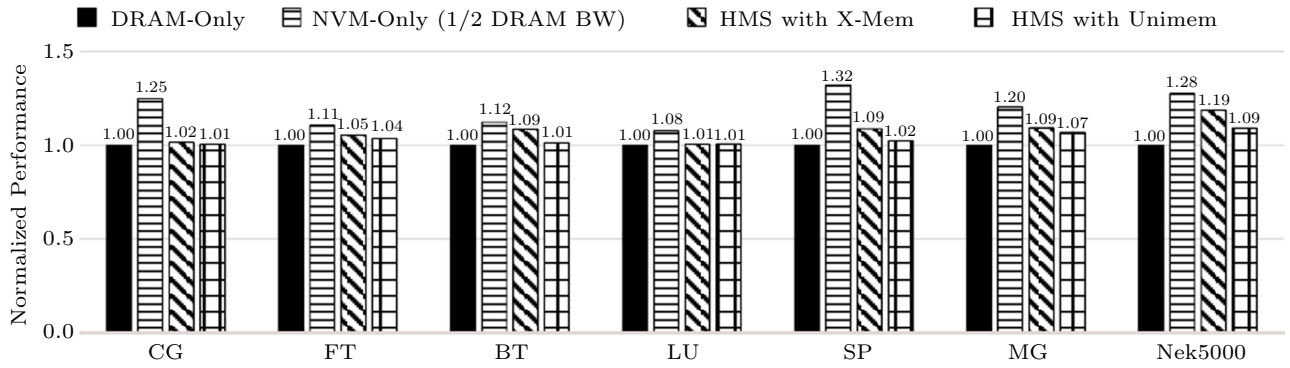

Fig.9. Normalized performance (execution time) comparison among DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 1/2 DRAM bandwidth.

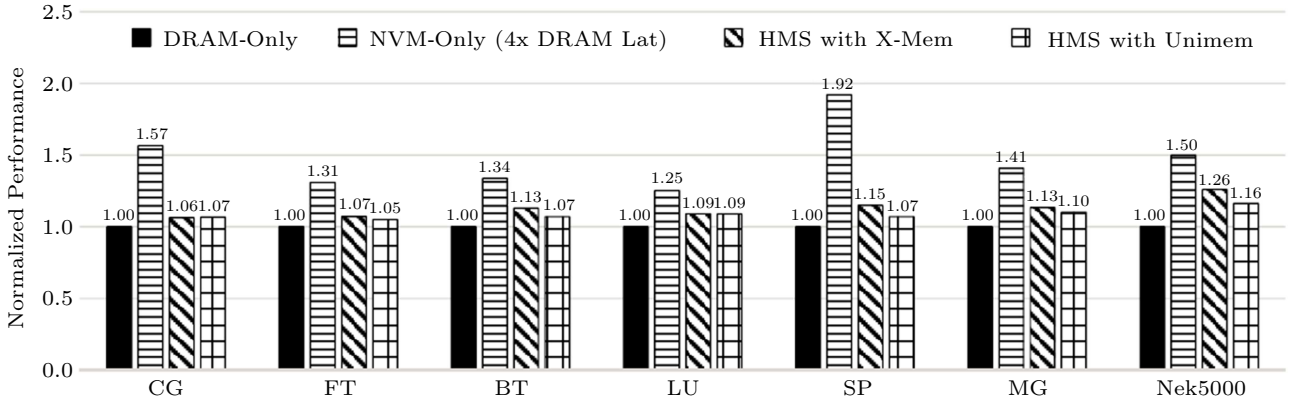

Fig.10. Normalized performance (execution time) comparison among DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 4x DRAM latency.

The results are shown in Figs.9 and 10. X-Mem uses PIN-based offline profiling to characterize memory access patterns and make the decision on data placement. They do not consider data movement cost and assume a homogeneous memory access pattern within a data object. The results show that Unimem performs similarly to X-Mem, but performs 10% better than X-Mem for Nek5000. Nek5000 is a production code with various memory access patterns across phases. Unimem adapts to those variations, hence performing better. Also, Unimem does not need any offline profiling for applications.

Detailed Performance Analysis. Based on the results of basic performance tests, we further quantify the contributions of our runtime techniques to performance improvement on HMS. This quantification study is important to investigate how effective our techniques are and when they can be effective. We study four major techniques: 1) cross-phase global search, 2) phase local search, 3) partitioning large data objects, and 4) initial data placement.

We apply the four techniques one by one. In particular, we apply technique 1, and then apply technique 2 to technique 1, and then apply technique 3 to technique 1 + technique 2, and then apply technique 4 to technique 1 + technique 2 + technique 3. We measure the performance variation after applying each technique to quantify the contribution of each technique to performance. We use the same system configurations as basic performance tests with NVM configured with 1/2 DRAM bandwidth. Fig.11 shows the results.

We notice that cross-phase global search can be very effective. In fact, in benchmarks CG and LU, more than 90% of the contribution comes from this technique. However, cross-phase global search could lose some opportunities to improve the performance on individual phases, because it uses the same data placement decision on all phases. Using the phase local search can complement the cross-phase global search. For BT and SP, using the phase local search we improve performance by 19% and 5% respectively.

Initial data placement is very useful. In fact, it takes effect on all benchmarks. For SP, it is the most effective approach (87% contribution comes from this tech-

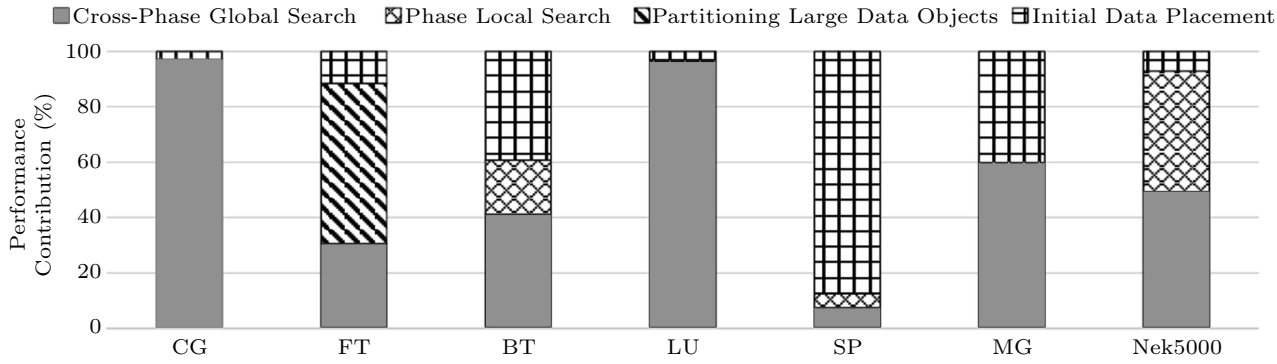

Fig.11. Quantifying the contributions of our four major techniques to performance improvement.

nique).

Partitioning large data objects does not take effect except FT, because it introduces very frequent data movement which loses performance. In FT, this technique contributes to 58% performance improvement, while the other three techniques make 42% contribution by manipulating small data objects. In general, by this study, we learn the importance of combining all techniques to maximize the performance improvement for various HPC workloads.

To further study the effectiveness of Unimem, we collect some detailed data migration information for HMS with Unimem (NVM has 1/2 DRAM bandwidth). Table 5 shows the results. "Pure runtime cost" in the table accounts for the overhead of collecting hardware counters, modeling costs, and synchronization cost between the helper thread and the main thread. "Pure runtime cost" does not include data movement cost and benefit. "%overlap" in the table shows the percentage of data movement cost that is successfully overlapped with the computation.

In Table 5, we notice that Unimem has very small runtime overhead (less than 3% in all cases). Directed by Unimem, the data migration can happen very often (e.g., 102 times in Nek5000 and 24 times in BT), and the migrated data size can be very large (e.g., 1.1 GB in Nek5000 and 720 MB in BT). However, even with the frequent data migration, Unimem successfully overlaps data migration with computation (70.6% in Nek5000 and 87.5% in BT). Also, the performance benefit of data migration outweighs those non-overlapped data migration, and narrows down the performance gap between NVM and DRAM to 9% at most (see Fig.9).

Scalability Study. To study how Unimem performs in larger system scales, we do strong scaling tests on Edison at LBNL. For each test, we use one MPI task per node and use class D as input problem. We use 256 MB for DRAM and 32 GB for NVM. Fig.12 shows the results for CG. The performance (execution time) in the figure is normalized to the performance of DRAM-only.

As we change the system scale, the sizes of data objects change. The numbers of main memory accesses also change because of caching effects: such changes in main memory accesses impact the sensitivity of data object to memory bandwidth and latency. Because of the above changes, the runtime system must be adaptive enough to make a good decision on data placement. In general, Unimem does a good job for all cases: the performance difference between DRAM-only and HMS with Uimem is no bigger than 7%.

Sensitivity Study. We use various configurations of the DRAM size in HMS and test if our runtime can perform well. As DRAM size changes, we will have diffe-

Table 5. Data Migration Details for HMS with Unimem

Benchmark	Times of Migration	Migrated Data Size (MB)	Pure Runtime Cost (%)	%Overlap (%)
CG	3	132	0.50	66.7
FT	4	201	1.50	75.0
BT	24	720	1.00	87.5
LU	3	187	1.00	60.0
SP	9	348	1.50	66.7
MG	1	17	2.00	100.0
Nek5000 (eddy)	102	1 101	3.00	70.6

Note: NVM has 1/2 DRAM bandwidth.

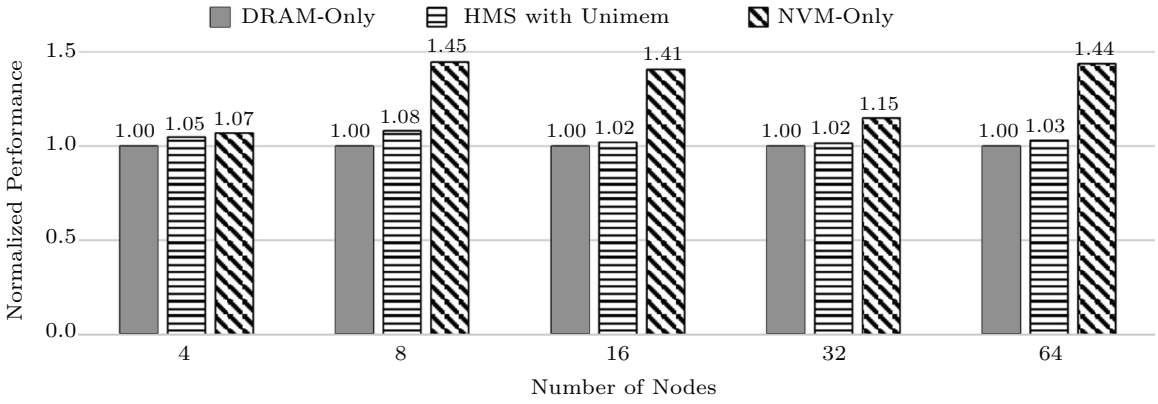

Fig.12. CG strong scaling tests on Edison (LBNL). It shows normalized performance (execution time) comparison among DRAM-only, HMS with Unimem, and NVM-only.

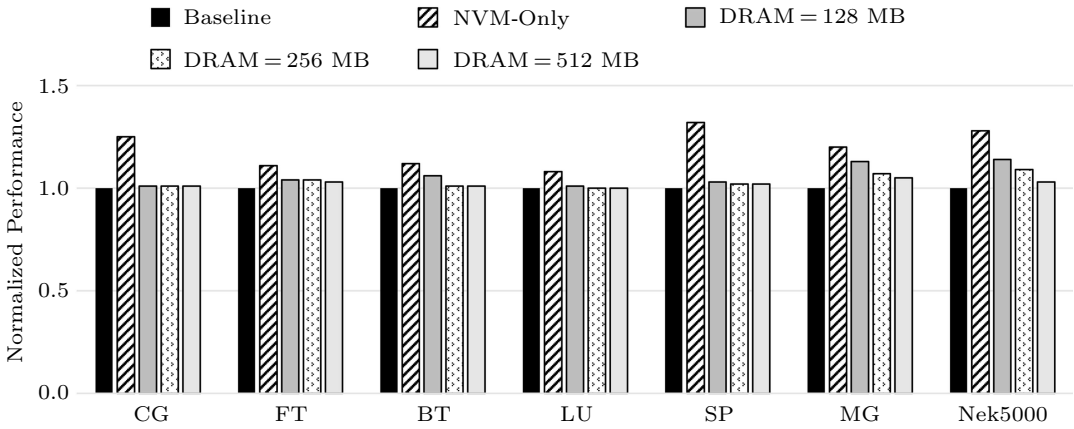

Fig.13. Unimem performance sensitivity of the DRAM size in HMS.

rent opportunities to place data objects. The change of the DRAM size will impact the frequency of data movement and impact whether we should decompose large data objects to improve performance. Fig.13 shows the results as we use 128 MB, 256 MB and 512 MB DRAM. In all tests, we use 16 GB NVM configured with 1/2 DRAM bandwidth and class C as the input problem. We use platform A and four nodes (1 MPI task per node) to do the tests. In Fig.13, the performance (execution time) is normalized to that of DRAM-only.

In general, Unimem performs well in all cases except one case: the performance difference between DRAM-only and HMS with Unimem is no bigger than 7% in all cases except MG with 128 MB DRAM. For MG with 128 MB DRAM, we have 13% performance difference between DRAM-only and HMS with Unimem. After careful examination, we find that DRAM is not well utilized, because large data objects cannot be placed in such small DRAM. We also cannot partition large data objects in MG by using our compiler tool because of widely employment of memory alias in the benchmark. But even so, our runtime still narrows performance gap between NVM-only and DRAM-only by 35%.

Performance Test on Optane PMM. In each test, we use one MPI task per core and use class C as an input problem. Since peak memory consumption of all benchmarks is smaller than 192 GB (the DRAM capacity in our platform), we limit the DRAM size to 256 MB in our evaluation to avoid placing all data objects in DRAM. Fig.14 shows the results. The performance (execution time) in the figure is normalized by that of DRAM-only. We show the performance of Unimem with and without distinguishing read and write in performance modeling, labelled as "w. drw" and "w.o drw" respectively in the figure.

Overall, the performance of the NVM-only system is significantly worse than that of DRAM-only system (12.7x performance loss on average). Unimem significantly reduces the performance gap. In particular, Unimem outperforms NVM-only system by 6.9x on average and up to 8.7x. Unimem (i.e, w. drw) outperforms X-Mem by 9% on average (19% at most). The above

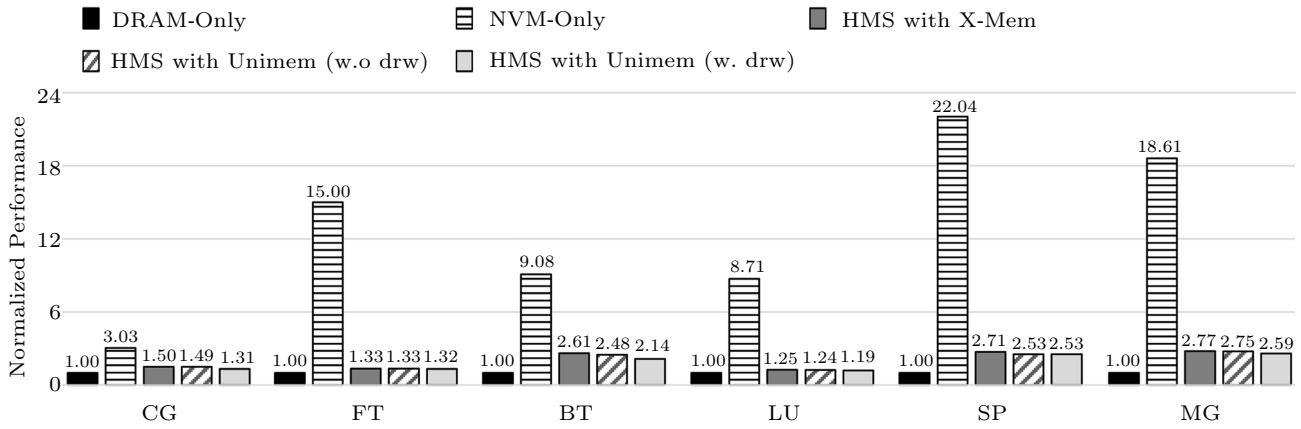

Fig.14. Unimem performance test on Optane PMM.

performance improvement highlights the effectiveness of Unimem. In addition, distinguishing read and write in performance modeling improves the performance by 12% over no distinction.

6 Related Work

Software-managed HMS has been studied in prior work. Dulloor *et al.* [1] introduced a data placement runtime based on offline profiling of application memory access patterns. Their work targets at enterprise workloads. To decide data placement, they classified memory access patterns into streaming, pointer chasing, and random. Giardino *et al.* [2] relied on OS and application co-scheduling data placement. In particular, they built APIs that allow programmers to describe their memory usage characteristics to OS, through which OS receives and implements responsive page placement and data migration. Lin *et al.* [3] introduced a protected OS service for asynchronous memory movement on HMS. Du *et al.* [4] developed a PIN-based offline profiling tool to collect memory traces and provide guidance for placing data on HMS. ProfDP [25] is a lightweight profiler that employs differential data object level analysis to provide intuitive guidance for data placement on HMS.

Different from the prior efforts, our work requires neither offline profiling as in [1, 4, 25] nor programmer involvement to identify memory access patterns as in [2]. Furthermore, our work does not require the modification of OS, which is different from [3]. Some existing efforts [26–30] use the hardware performance counter-based approach (e.g., Precise Event-Based Sampling from Intel or Instruction-based Sampling from AMD) to identify locality, scalability and NUMA bottlenecks. In contrast, Unimem focuses on managing data placement on HMS. ATMem [31] employs a sampling-based profiler to select performance-critical data to guide data migration for graph applications. Our work aims for legacy HPC applications and systems. Moreover, ATMem does not distinguish between read and write events of data objects. NVMs such as Intel Optane have asymmetric read and write characteristics. Ignoring asymmetric read and write can lead to inaccurate modeling of performance benefits, which affects the effectiveness of data placement and loses application performance. We show the importance of distinguishing between reads and writes in the performance model in Fig.14.

Some studies introduce hardware-based data placement solutions for the NVM-based HMS. Bivens *et al.* [32] and Qureshi *et al.* [8, 9] used DRAM as a set-associative cache logically placed between the processor and NVM. NVM is accessed when DRAM buffer eviction or buffer miss happens. Yoon *et al.* [10] placed data based on row buffer locality in memory devices. Wang *et al.* [7] relied on static analysis and advanced memory controller to monitor memory access patterns to determine data placement on GPU. Wu *et al.* [7] leveraged the knowledge of numerical algorithms to direct data placement. They introduced hardware modifications to support massive data migration and performance optimization. Agarwal *et al.* [18] introduced a bandwidth-aware data placement on GPU, driven by compiler extracted insights and explicit hints from programmers.

A key limitation of the above hardware-based approaches is that they heavily rely on modified hardware to monitor memory access patterns and migrate data. Some work, such as [5, 8–10], ignores application semantics and triggers data movement based on temporal memory access patterns, which could cause unnecessary data movement. Our work avoids any hardware

modification, and explores global optimization on data placement.

7 Conclusions

The limitation of NVM imposes a question on whether NVM is a feasible solution for HPC workloads. In this paper, we quantified the performance gap between NVM-based and DRAM-based systems, and demonstrated that using a carefully designed runtime, it is possible to significantly reduce the performance gap. We hope that our work can lay the foundation to embrace NVM for future HPC.

Acknowledgement(s) We thank Intel and the anonymous reviewers for their constructive comments.

References

[1] Dulloor S R, Roy A, Zhao Z G *et al.* Data tiering in heterogeneous memory systems. In *Proc. the 11th European Conference on Computer Systems*, April 2016, Article No. 15. DOI: 10.1145/2901318.2901344.

[2] Giardino M, Doshi K, Ferri B. Soft2LM: Application guided heterogeneous memory management. In *Proc. the 2016 International Conference on Networking, Architecture, and Storage*, Aug. 2016. DOI: 10.1109/NAS.2016.7549421.

[3] Lin F X, Liu X. *memif*: Towards programming heterogeneous memory asynchronously. In *Proc. the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2016, pp.369-383. DOI: 10.1145/2980024.2872401.

[4] Shen D, Liu X, Lin F X. Characterizing emerging heterogeneous memory. In *Proc. the 2016 ACM SIGPLAN International Symposium on Memory Management*, June 2016, pp.13-23. DOI: 10.1145/2926697.2926702.

[5] Wang B, Wu B, Li D, Shen X, Yu W, Jiao Y, Vetter J S. Exploring hybrid memory for GPU energy efficiency through software-hardware co-design. In *Proc. the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2013, pp.93-102. DOI: 10.1109/PACT.2013.6618807.

[6] Wu K, Ren J, Li D. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 31. DOI: 10.1109/SC.2018.00034.

[7] Wu P, Li D, Chen Z, Vetter J, Mittal S. Algorithm-directed data placement in explicitly managed no-volatile memory. In *Proc. the 25th ACM Symposium on High-Performance Parallel and Distributed Computing*, May 2016, pp.141-152. DOI: 10.1145/2907294.2907321.

[8] Qureshi M K, Franchescini M, Srinivasan V, Lastras L, Abali B, Karidis J. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proc. the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009, pp.14-23. DOI: 10.1145/1669112.1669117.

[9] Qureshi M K, Srinivasan V, Rivers J A. Scalable high-performance main memory system using phase-change memory technology. In *Proc. the 36th International Symposium on Computer Architecture*, June 2009, pp.24-33. DOI: 10.1145/1555754.1555760.

[10] Yoon H, Meza J, Ausavarungnirun R, Harding R, Mutlu O. Row buffer locality aware caching policies for hybrid memories. In *Proc. the 30th IEEE International Conference on Computer Design*, Sept. 30–Oct. 3, 2012, pp.337-344. DOI: 10.1109/ICCD.2012.6378661.

[11] Wu K, Huang Y, Li D. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 58. DOI: 10.1145/3126908.3126923.

[12] Bailey D H, Barszcz E, Dagum L, Simon H D. Nas parallel benchmark results. In *Proc. the 1992 ACM/IEEE Conference on Supercomputing*, Nov. 1992, pp.386-393. DOI: 10.1109/SUPERC.1992.236665.

[13] Izraelevitz J, Yang J, Zhang L *et al.* Basic performance measurements of the Intel Optane DC persistent memory module. arXiv: 1903.05714, 2019. https://arxiv.org/pdf/1903.05714v3.pdf, October 2020.

[14] Suzuki K, Swanson S. The non-volatile memory technology database (NVMDB). Technical Report, Department of Computer Science & Engineering, University of California, 2015. http://cseweb.ucsd.edu/~swanson/papers/TR2015-NVMDB.pdf, Oct. 2020.

[15] Volos H, Magalhaes G, Cherkasova L, Li J. Quartz: A lightweight performance emulator for persistent memory software. In *Proc. the 16th Annual Middleware Conference*, November 2015, pp.37-49. DOI: 10.1145/2814576.2814806.

[16] Li D, Vetter J, Marin G, McCurdy C, Cira C, Liu Z, Yu W. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Proc. the 26th International Parallel and Distributed Processing Symposium*, May 2012, pp.945-956. DOI: 10.1109/IPDPS.2012.89.

[17] Silvano M, Toth P. Knapsack Problems: Algorithms and Computer Implementations (1st edition). John Wiley & Sons, 1990.

[18] Agarwal N, Nellans D, Stephenson M, O'Connor M, Keckler S W. Page placement strategies for GPUs within heterogeneous memory systems. In *Proc. the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2015, pp.607-618. DOI: 10.1145/2775054.2694381.

[19] Ding C, Kennedy K. Bandwidth-based performance tuning and prediction. In *Proc. the 1990 IASTED International Conference on Parallel Computing and Distributed Systems*, November 1999.

[20] Berger E D, McKinley K S, Blumofe R D, Wilson P R. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000, pp.117-128. DOI: 10.1145/378993.379232.

[21] Michael M M. Scalable lock-free dynamic memory allocation. In *Proc. the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004, pp.35-46. DOI: 10.1145/996893.996848.

[22] Lattner C. LLVM: An infrastructure for multi-stage optimization [Ph.D. Thesis]. Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2002.

[23] Chakaravarthy V T. New results on the computability and complexity of points-to analysis. In *Proc. the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2003, pp.115-125. DOI: 10.1145/640128.604142.

[24] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory. In *Proc. the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2011, pp.91-104. DOI: 10.1145/2248487.1950379.

[25] Wen S, Cherkasova L, Lin F X, Liu X. ProfDP: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proc. the 2018 International Conference on Supercomputing*, June 2018, pp.263-273. DOI: 10.1145/3205289.3205320.

[26] Lachaize R, Lepers B, Quéma V. MemProf: A memory profiler for NUMA multicore systems. In *Proc. the 2012 USENIX Annual Technical Conference*, June 2012, pp.53-64.

[27] Liu X, Mellor-Crummey J. A data-centric profiler for parallel programs. In *Proc. the International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2013, Article No. 28. DOI: 10.1145/2503210.2503297.

[28] Liu X, Wu B. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, Article No. 47. DOI: 10.1145/2807591.2807648.

[29] Liu X, Mellor-Crummey J. Pinpointing data locality problems using data-centric analysis. In *Proc. the 9th International Symposium on Code Generation and Optimization*, April 2011, pp.171-180. DOI: 10.1109/CGO.2011.5764685.

[30] McCurdy C, Vetter J. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. the 2010 IEEE International Symposium on Performance Analysis of Systems Software*, March 2010, pp.87-96. DOI: 10.1109/ISPASS.2010.5452060.

[31] Chen Y, Peng I B, Peng Z, Liu X, Ren B. ATMem: Adaptive data placement in graph applications on heterogeneous memories. In *Proc. the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, February 2020, pp.293-304. DOI: 10.1145/3368826.3377922.

[32] Bivens A, Dube P, Franceschini M, Karidis J, Lastras L, Tsao M. Architectural design for next generation heterogeneous memory systems. In *Proc. the 2010 International Memory Workshop*, May 2010. DOI: 10.1109/IMW.2010.5488395.

Kai Wu is a Ph.D. candidate at University of California Merced (UC Merced), Merced. Before coming to UC Merced, he earned his Master's degree in computer science and engineering from Michigan State University, East Lansing, in 2016. His research areas are computer system and high performance computing (HPC) with a focus on hardware heterogeneity. He designs high performance computer systems with memory heterogeneity. His recent work focuses on designing system support for persistent memory-based big memory platforms. He has published in the top-tier system/HPC conferences and journals, including FAST, HPCA, SC, PACT, ICPP, CLUSTER, etc.

Dong Li is an associate professor in the Department of Electrical Engineering and Computer Science, University of California Merced, Merced. He is the director of Parallel Architecture, System, and Algorithm Lab (PASA). Previously (in 2011–2014), he was a research scientist at the Oak Ridge National Laboratory (ORNL). Before that, he earned his Ph.D. degree in computer science from Virginia Polytechnic Institute and State University, Virginia. Dong received a CAREER Award from U.S. National Science Foundation in 2016, a Berkeley Lab University Faculty Fellowship (2016), and an ORNL/CSMD Distinguished Contributor Award (2013). His paper in SC 2014 was nominated as the Best Student Paper.