Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory

Kai Wu
kwu42@ucmerced.edu
University of California, Merced

Yingchao Huang
yhuang46@ucmerced.edu
University of California, Merced

Dong Li
dli35@ucmerced.edu
University of California, Merced

ABSTRACT

Non-volatile memory (NVM) provides a scalable and power-efficient solution to replace DRAM as main memory. However, because of 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 best performance. In this paper, we introduce a lightweight runtime solution that automatically and transparently manage data placement on HMS without the requirement of hardware modifications and disruptive change to applications. Leveraging online profiling and performance models, the runtime 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

Non-volatile memory, runtime, data management

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 data sets in scientific simulations drive the fast development of NVM, because NVM can provide a scalable and power-efficient solution as main memory, alternative to DRAM. Such solution is based on the attractive characteristics of NVM, such as higher density and near-zero static power consumption.

However, comparing 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 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) [8, 10, 13, 19, 24, 25]. 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 data placement on HMS for HPC, we have several goals. First, we want to avoid disruptive changes to hardware. The existing hardware-based solutions to manage data placement on HMS [17, 18, 24, 26] may be difficult to be embraced by 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 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 one hand, we want data movement to be frequent, such that data placement is adaptive to variation of memory access patterns across execution phases. On the other hand, we want to minimize 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 is 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 data placement. This runtime 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 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, those 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 data sets 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 gap for HPC.
- We introduce a lightweight runtime system to manage 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). The performance difference between DRAM-only and HMS with Unimem is only 6.2% on average and 16% at most. We successfully narrow the performance gap and demonstrate better performance than a state-of-the-art software-based solution.

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 [8, 10, 13, 19, 25].
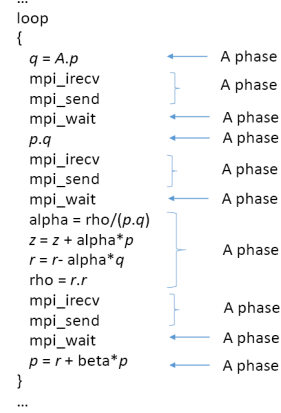

Figure 1: A conceptual description for an MPI-based program (CG) decomposed into phases. A is a 2D matrix, and q, p, z, and r are vectors.

2.1 Definitions and Basic Assumptions

We target on 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 on 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, Figure 1 depicts a typical iterative structure from CG (an NAS parallel benchmark [2]), which dominates the execution time of CG.

We claim a data object is *bandwidth sensitive*, if there is a big performance difference between placing it in NVM with less memory bandwidth and DRAM. We claim a data object is *latency sensitive*, if there is a big performance difference between placing it in NVM with longer memory access latency and DRAM.

2.2 Preliminary Performance Evaluation with NVM-Based Main Memory

NVM has relatively long access latency and low memory bandwidth. Table 1 shows NVM performance characteristics. The table is based on [21] that gathered a comprehensive survey of 340 non-volatile memory technology papers published between 2000 and 2014 in relevant conferences. Based on such performance characteristics, we perform preliminary performance study to quantify the impact of NVM on HPC application performance.

We use Quartz, a DRAM-based, lightweight performance emulator for NVM [23]. The existing work uses cycle-accurate simulation to study NVM performance [12, 25]. However, the long simulation time makes impossible simulate *HPC applications* with large data sets on multiple nodes. The performance of HPC workloads on

Table 1: NVM performance characteristics, and comparison between NVM techniques and DRAM [21]

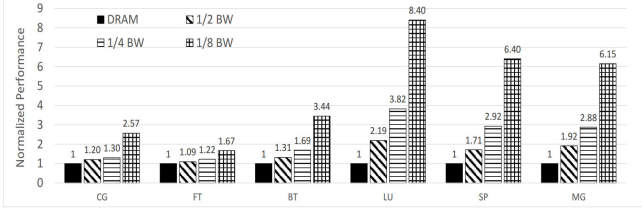|	Read time	Write time	Random read BW	Random write BW
DRAM	10 ns	10 ns	1,000 MB/s	900 MB/s
STT-RAM (ITRS'13)	60 ns	80 ns	800 MB/s	600 MB/s
PCRAM	20-200 ns	80-10,000 ns	200-800 MB/s	100-800 MB/s
ReRAM	10-1,000 ns	10-10,000 ns	20-100 MB/s	1-8 MB/s

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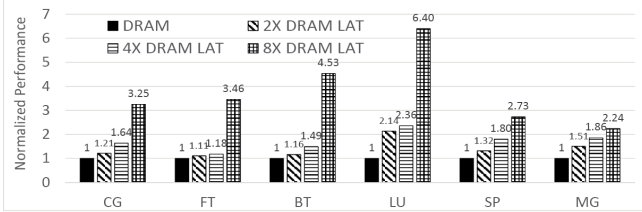

Figure 3: The benchmark performance (execution time) on NVM-based main memory (NVM-only) with various latency. The performance is normalized to that of DRAM-only systems.

NVM is always mysterious. Using Quartz, we can study 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 FT, we use CLASS C as input because of the long execution time with Class D. Figures 2 and 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 [12]. Furthermore, HPC application performance (execution time) is sensitive to different NVM technologies with various bandwidth and latency. With memory bandwidth reduced by only 1/2 or latency increased by only 2x in NVM, some benchmarks already show a big slowdown. For example, LU has 2.19x and 2.14x slowdown with NVM configured with 1/2 DRAM bandwidth (Figure 2) and 2x DRAM latency (Figure 3) respectively.

We further study whether data placement in HMS can bridge the performance gap between DRAM-based and NVM-based systems. We choose 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
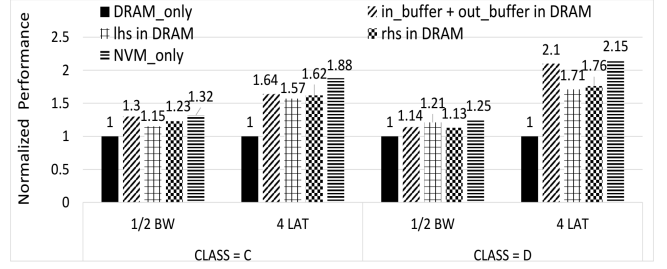

Figure 4: The impact of data placement on performance (execution time) of NVM-based main memory. The performance is normalized to DRAM-only systems. The legend entries "in_buffer+out_buffer", "lhs", and "rhs" are the data objects placed in DRAM in the DRAM+NVM system. The x axis shows the configuration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

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 Figure 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. Figure 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 Figure 4).

Observation 3: Different data objects manifest different sensitivity to limited NVM bandwidth and latency, shown in Figure 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 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 on directing data placement on HMS for HPC applications.

Unimem directs data placement for data objects (e.g., multidimensional 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 based. It decides and changes data placement for target data objects for each phase based on runtime profiling and lightweight performance models.

In particular, Unimem profiles memory references to target data objects with a few invocations of each phase. Then Unimem uses performance models to predict performance benefit and cost of data placement, and formulates the problem of deciding 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 iteration of the main computation loop of the application. At the end of the first iteration, we build performance models and make data placement decision. After the first iteration, 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 memory access information for each phase. This information is leveraged by the second and third steps to decide data placement for each phase.

We rely on hardware performance counters widely deployed in modern processors. In particular, we collect the number of last level cache miss event, 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 last level cache misses. Those memory addresses help us identify target data objects that have frequent memory accesses in main memory.

Note that the number of last level cache misses can reflect how intensive main memory 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, there are other events that cause main memory accesses, such as cache line eviction and prefetching operations. The current performance counters either do not support counting such event (cache line eviction) or do not have the sampling mode for such event (prefetching operation). Hence, we cannot include those events when counting main memory accesses. However, the last level cache miss accounts for a large part of main memory accesses. It can work as a reliable indicator to direct data placement, as shown in the evaluation section. The last level cache miss is also one of the most common events in modern processors, which makes our runtime highly portable across HPC platforms. 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. Those 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 (Equations 2 and 3) and data movement cost (Equation 4) 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 (Equation 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 Equation 1. This equation estimates main memory bandwidth consumption due to memory accesses to the data object (BW_{data_obj}).

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

The numerator of Equation 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 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 the equation 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 $\frac{\#samples\_with\_data\_accesses}{\#samples}$, 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 1000 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 $\frac{10^5}{10^7} \times 10 = 0.1s$.

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 Equation 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 Equation 3, which will be discussed next. If BW_{data_obj} of the data object is between $t_1\%$ and $t_1\%$, 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 benchmark [15], with maximum memory concurrency, and use Equation 1 and performance counters.

Calculation of data movement benefit. Equations 2 and 3 calculate performance benefits (after 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 application on NVM and on DRAM. If the data object is bandwidth-sensitive, then the application performance on a specific memory is modeled by $\frac{accssed\_data\_size}{mem\_bw}$ (*mem* is NVM or DRAM). *accessed_data_size* is $\#data\_access \times cacheline\_size$, the same as the one in Equation 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).

$$BFT_{data\_obj\_bw} = \\ (\frac{\#data\_access \times cacheline\_size}{NVM\_bw} - \\ \frac{\#data\_access \times cacheline\_size}{DRAM\_bw}) \times CF\_bw \quad (2)$$

$$BFT_{data\_obj\_lat} = \\ (\#data\_access \times NVM\_lat - \\ \#data\_access \times DRAM\_lat) \times CF\_lat \quad (3)$$

In the above two equations, we have constant factors CF_bw (see Equation 2) and CF_lat (see Equation 3). Such constant factors are used to improve 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 inability of counting cache eviction and prefetching operations and sampling nature of the approach. The constant factors CF_bw and CF_lat work as a simple but powerful approach to improve 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 measured performance, and such ratio is CF_bw. The predicted performance is calculated based on ($\#data\_access \times cacheline\_size/DRAM\_bw$), 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,
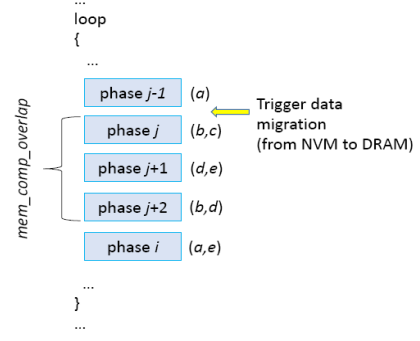

Figure 5: An example to show how to calculate $mem\_comp\_overlap$ for the data object a in the phase i. The yellow arrow is the point to trigger the migration of a from NVM to DRAM for the phase i, if a is not in DRAM. The letters in brackets represent target data objects referenced in the corresponding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

except that we use a latency-sensitive benchmark, the pointer-chasing benchmark [4] (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.

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 ($\frac{data\_size}{mem\_copy\_bw}$). To reduce the data movement cost, we want to overlap 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 Section 3.3. In summary, the data movement cost ($COST_{data\_obj}$) is modeled in Equation 4 with the overlapped cost (*mem_comp_overlap*) included.

$$COST_{data\_obj} = max(\frac{data\_size}{mem\_copy\_bw} - mem\_comp\_overlap, 0) \quad (4)$$

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*.

Figure 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 (the phase i). If a is not in DRAM, we can trigger data migration of a as early as the beginning of the phase j, because a is not referenced between j and i. We cannot trigger data migration of a at the beginning of the phase $j-1$, because a is referenced there. *mem_comp_overlap* is the application execution time between the phases j and i. The data movement time, $\frac{data\_size}{mem\_copy\_bw}$, 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.

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. Use Figure 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 data from DRAM to NVM when there is no enough space in DRAM and we need to switch 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 is 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:

$$w = BFT_{data\_obj} - COST_{data\_obj} - extra\_COST_{data\_obj} \quad (5)$$

$extra\_COST_{data\_obj}$ 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 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 [20].

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 $O((log(n))^2)$ [20], where n is the number of target data objects referenced in a phase.

The above approach can determine 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 Section 3.3 for implementation details) to proactively move data objects between NVM and DRAM based on the data placement decision for future phases.

Figure 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 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 the performance modeling and dynamic programming, we can decide whether the cross-phase global search or 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 performance benefit of the initial data placement on GPU with HMS [1, 24]. 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
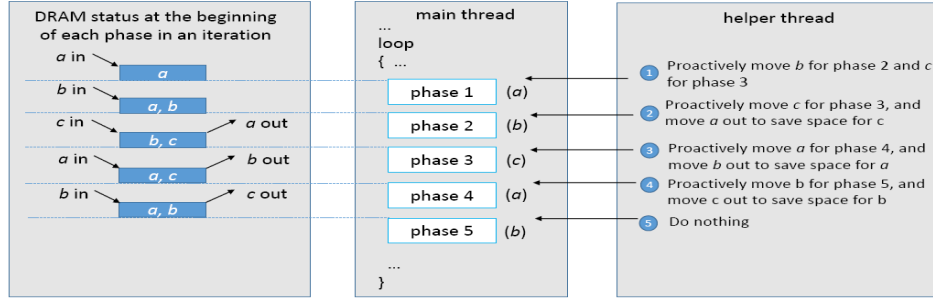
Figure 6: An example to show proactive data migration with a helper thread. The letters in the figure represent data objects. The letters in brackets (e.g., (a) and (b)) represent target data objects that are determined to be placed in DRAM for the corresponding phases. DRAM can hold two data objects at most.

the DRAM space limitation). To calculate the number of memory reference for each target data object, we employ compiler analysis and represent the number of memory reference as a symbolic formula with unknown application information, similar to [7]. 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 iteration is determined by a convergence test at run time).

Our method determines initial data placement simply based on the number of memory reference 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, initial 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 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 overweight the cost of data chunk movement. This method exposes new opportunities to manage data and improve 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

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

some regular memory references into new ones based on chunk-based partitioning (assuming the input problem size and 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 online adaptation of data placement on HMS. To leverage the library, the programmer needs to insert a couple of APIs into the application. Such 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 list those APIs and their functionality.

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
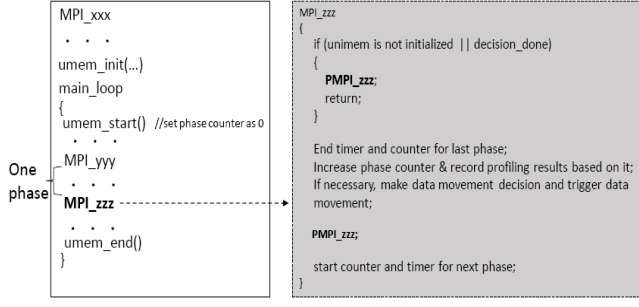
Figure 7: Transparently identifying phases based on PMPI.

Figure 8: The general workflow for Unimem.

can transparently identify execution phases and control profiling without programmer intervention. Figure 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 allows Unimem to collect pointers pointing to target data objects. Collecting those pointers are 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 alias 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 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 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 [3] and the lock-free allocator [16].

As discussed in Section 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 movem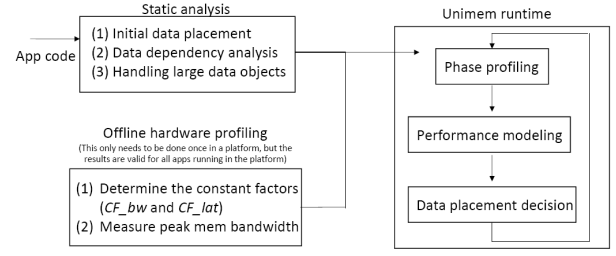ent requests into the queue; the helper thread checks the queue, performs data movement, and removes the data movement request off the queue once the data movement is done. At the beginning of each phase, the runtime of the main thread will check the queue status to determine if all proactive data movement for the current phase is 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 is lightweight, because we avoid frequent data movement in our design.

As discussed in Section 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 [11] 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 [6]. 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. Figure 8 depicts the general workflow.

4 EVALUATION METHODOLOGY

In our evaluation, we use Quartz emulator [23]. Quartz enables an efficient emulation of a range of NVM latency and bandwidth characteristics. Quartz has low overhead and good accuracy (with emulation errors 0.2% - 9%) [23]. 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 approaches [14, 22]. 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 NUMA architecture to emulate NVM. In particular, we carefully manage data placement at the user level, such that, given

Table 3: Target data objects in NPB benchmarks and Nek5000

Benchmark	Target data objects	% of total app mem footprint
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%

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 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.

We have two test platforms for performance evaluation. One test platform (named "Platform A") is a small cluster. Each node of it has two eight-core Intel Xeon E5-2630 processors (2.4 GHz) and 32GB DDR4. We use this platform for tests in all figures except Figure 12. We deploy Quartz on such platform. The other test platform is the Edison supercomputer at Lawrence Berkeley National Lab (LBNL). We use this platform for tests in Figure 12. Each Edison node has two 12-core Intel Ivy Bridge processor (2.4 GHz) with 64GB DDR3. As discussed before, we perform strong scaling tests and leverage NUMA architecture to emulate NVM on this system.

We use six benchmarks from NAS parallel benchmark (NPB) suite 3.3.1, and one production scientific code Nek5000 [9]. For Nek5000, we use eddy input problem with a 256×256 mesh. The target data objects of those benchmarks are listed in Table 3. 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 and 6.1.0 on Edison) and use default compiler options for building benchmarks. We use the sampling-based approach to collect performance events on the two platforms. The sampling interval is chosen as 1000 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, performance in this section is normalized to that of the DRAM-only system.

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
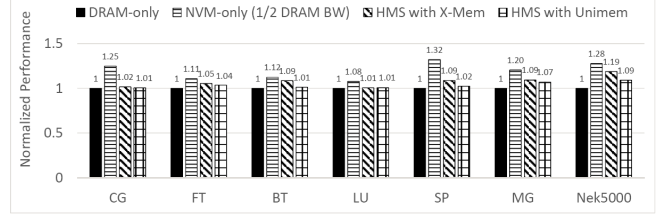

Figure 9: The performance (execution time) comparison between DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 1/2 DRAM bandwidth.
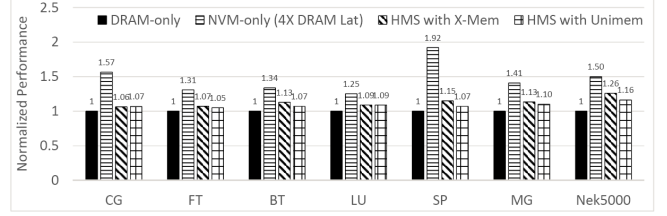

Figure 10: The performance (execution time) comparison between DRAM-only, NVM-only, the existing work (X-Mem), and HMS with Unimem. NVM has 4x DRAM latency.
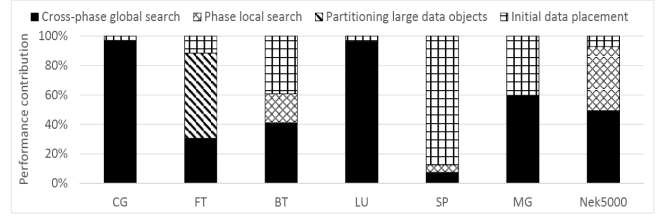

Figure 11: Quantifying the contributions of our four major techniques to performance improvement.

use CLASS C as input problem for NPB benchmarks. NVM and DRAM sizes are 16GB and 256MB respectively. Figures 9 and 10 show the results. NVM is configured with 1/2 DRAM bandwidth (Figure 9) or 4x DRAM latency (Figure 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 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 [8] (a recent software-based solution for data placement in HMS). The results are shown in Figures 9 and 10. X-Mem uses PIN-based *offline profiling* to characterize memory access patterns and make the decision on data

Table 4: Data migration details for HMS with Unimem (NVM has 1/2 DRAM bandwidth).

Benchmark	Times of Migration	Migrated data size (MB)	Pure runtime cost	% overlap
CG	3	132	0.5%	66.7%
FT	4	201	1.5%	75%
BT	24	720	1%	87.5%
LU	3	187	1%	60%
SP	9	348	1.5%	66.7%
MG	1	17	2%	100%
Nek5000(eddy)	102	1101	3%	70.6%

Figure 12: CG strong scaling tests on Edison (LBNL).

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 (1), and then apply (2) to (1), and then apply (3) to (1)+(2), and then apply (4) to (1)+(2)+(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. Figure 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 performance on individual phases, because it uses the same data placement decision on all phases. Using phase local search can complement cross-phase global search. For BT and SP, using 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 technique).

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 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 4 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 main thread. "Pure runtime cost" does not include data movement cost and benefit. "% overlap" in the table shows the
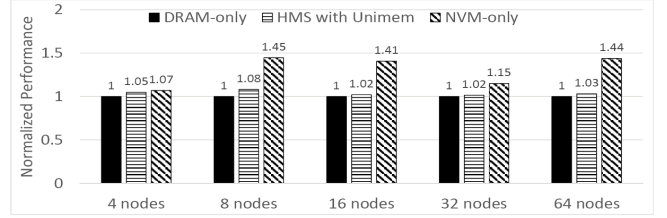
percentage of data movement cost that is successfully overlapped with the computation.

From Table 4, 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., 1.1GB in Nek5000 and 720MB 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 Figure 9).

Scalability study. To study how Unimem performs in larger system scales. We did 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 256MB for DRAM and 32GB for NVM. Figure 12 shows the results for CG. Performance (execution time) in the figures 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 DRAM size in HMS and test if our runtime can perform well. As DRAM size changes, we will have different opportunities to place data objects. The change of DRAM size will impact the frequency of data movement and impact whether we should decompose large data objects to improve performance. Figure 13 shows the results as we use 128MB, 256MB and 512MB DRAM. In all tests, we use 16GB NVM configured with 1/2 DRAM bandwidth and CLASS C as input problem. We use Platform A and four nodes (1 MPI task per node) to do the tests. In the figure, 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 128MB DRAM. For MG with 128MB 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%.
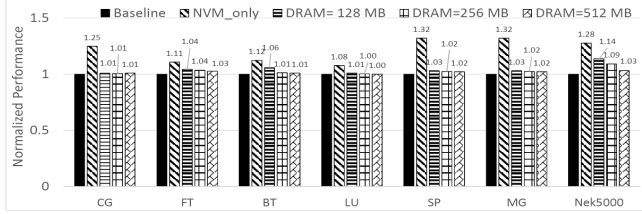

Figure 13: Unimem performance sensitivity to DRAM size in HMS.

6 RELATED WORK

Software-managed HMS has been studied in prior work. Dulloor et. al [8] introduce a data placement runtime based on *offline profiling* of application memory access patterns. Their work targets on enterprise workloads. To decide data placement, they classify memory access patterns into streaming, pointer chasing, and random. Giardino et. al [10] rely on OS and application co-scheduling data placement. In particular, they build 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 [13] introduce a protected OS service for asynchronous memory movement on HMS. Du et. al [19] develop a PIN-based *offline profiling* tool to collect memory traces and provide guidance for placing data on HMS.

Different from the prior efforts, our work requires neither offline profiling as in [8, 19] nor programmer involvement to identify memory access patterns as in [10]. Furthermore, our work does not require the modification of OS, which is different from [13]. Our work aims for legacy HPC applications and systems.

Some studies introduce hardware-based data placement solutions for the NVM-based HMS. Bivens et al. [5] and Qureshi et al. [17, 18] use DRAM as a set-associative cache logically placed between processor and NVM. NVM is accessed when DRAM buffer eviction or buffer miss happens. Yoon et al. [26] place data based on row buffer locality in memory devices. Wang et al. [24] rely on static analysis and advanced memory controller to monitor memory access patterns to determine data placement on GPU. Wu et al. [25] leverage the knowledge of numerical algorithms to direct data placement. They introduce hardware modifications to support massive data migration and performance optimization. Agarwal et al. [1] introduce 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 [17, 18, 24, 26], 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 quantify the performance gap between NVM-based and DRAM-based systems, and demonstrate 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.

REFERENCES

[1] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[2] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. Nas parallel benchmark results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-2630-5.

[3] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[4] T. Besard. Pointer-chasing memory benchmark. https://github.com/maleadt/pChase.

[5] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao. Architectural Design for Next Generation Heterogeneous Memory Systems. In *International Memory Workshop*, 2010.

[6] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003.

[7] C. Ding and K. Kennedy. Bandwidth-based performance tuning and prediction. In *International Conference on Parallel Computing and Distributed Systems*, 1999.

[8] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[9] P. Fischer and J. Lottes. *nek5000 Web page.* http://nek5000.mcs.anl.gov, Web page, 2008.

[10] M. Giardino, K. Doshi, and B. Ferri. Soft2LM: Application Guided Heterogeneous Memory Management. In *International Conference on Networking, Architecture, and Storage (NAS)*, 2016.

[11] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization.* PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2002.

[12] D. Li, J. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications. In *International Parallel and Distributed Processing Symposium*, 2012.

[13] F. X. Lin and X. Liu. memif: Towards Programming Heterogeneous Memory Asynchronously. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[14] P. Macko. PCMSIM: A Simple PCM Block Device Simulator for Linux. https://code.google.com/p/pcmsim.

[15] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream.

[16] M. M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[17] M. K. Qureshi, M. Franchescini, V. Srinivasan, L. Lastras, B. Abali, and J. Karidis. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *MICRO*, 2009.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.

[19] D. Shen, X. Liu, and F. X. Lin. Characterizing Emerging Heterogeneous Memory. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2016.

[20] M. Silvano and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[21] K. Suzuki and S. Swanson. The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, 2015. http://nvmdb.ucsd.edu.

[22] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Architectural Support for Programming Languages and Operating*

Systems (ASPLOS), 2011.

[23] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Annual Middleware Conference (Middleware)*, 2015.

[24] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[25] P. Wu, D. Li, Z. Chen, J. Vetter, and S. Mittal. Algorithm-Directed Data Placement in Explicitly Managed No-Volatile Memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[26] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *International Conference on Computer Design (ICCD)*, 2012.

A. Artifact Description: Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory

A.1 Abstract

Our SC'17 submission introduces a lightweight runtime solution, *Unimem*, that automatically and transparently manage data placement on a heterogeneous non-volatile memory (NVM) and DRAM system. In this appendix, we describe some details, including how to emulate NVM based on an NVM emulator, how to emulate NVM based on NUMA architecture at Edison supercomputer at Lawrence Berkeley National Lab, how to use sampling based performance counters, and the general workflow of our solution.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:** the paper describes in details the algorithms used by Unimem, including when to trigger data placement and how to place data to maximize performance benefits.
- **Program:** a runtime library for MPI programs
- **Compilation:** GCC 4.4.7, Open MPI 2.0, GNU Make and ROSE compilation framework 0.9.8.31 (for data dependency analysis)
- **Transformations:** insert a few APIs into the application and link it with our library to enable runtime data management.
- **Data set:** NPB3.3.1 and Nek5000
- **Run-time environment:** Quartz emulator (an NVM emulator) needs privilege access; Our Unimem runtime library can run on Linux without privilege access.
- **Hardware:** Platform A: Each node of this platform has two eight-core Intel Xeon E5-2630 processors (2.4 GHz) and 32GB DDR4. Paltform B (Edison at Lawrence Berkeley National Lab): each node has two 12-core Intel Ivy Bridge processor (2.4 GHz) with 64GB DDR3.
- **Run-time state:** The system is idle and only running our tests
- **Output:** benchmark execution time
- **Experiment workflow:** See below
- **Experiment customization:** Increase the input problem size of eddy for Nek5000
- **Publicly available?:** No

A.2.2 Hardware Dependencies

The Unimem runtime library runs on x86 platforms and has been successfully tested on Intel SandyBridge, IvyBridge, and Haswell processors.

A.2.3 Software Dependencies

- The Quartz emulator (NVM emulator) that requires cmake 2.8, libconfig and libconfig-devel, numactl-devel, uthash-devel, and kernel-devel.
  https://github.com/HewlettPackard/quartz

- Intel Precise Event Based Sampling (PEBS) that requires libpfm4 and libnuma.
  http://perfmon2.sourceforge.net/docs_v4.html
- ROSE compiler infrastructure (including its dependencies)
  http://rosecompiler.org/ROSE_HTML_Reference/installation.html

A.3 Quartz (NVM Emulator)

We use the Quartz emulator to emulate different latency and bandwidth of NVM. To use Quartz, one must specify whether a latency model or a bandwidth model should be employed. Also, one must set concrete memory latency or bandwidth numbers in *nvmemul.ini* file. The following is the configuration we use to emulate 1/8 DRAM bandwidth and 4x DRAM latency:

- latency:
  {
  enable = true;   //Turn on the latency model
  inject_delay = true;
  read = 344 ;
  write = 344 ;
  max_epoch_duration_us = 10000 ;
  min_epoch_duration_us = 10000 ;
  calibration = false;
  };
- bandwidth:
  {
  enable = false;   //Turn on the bandwidth model
  model = "/tmp/bandwidth_model";
  read = 2000;
  write =2000;
  };

  One can only use either latency model or bandwidth model, but not both. Also, "read" and "write" should be given the same latency or bandwidth value.

A.4 NUMA Control on Edison to Emulate NVM

We cannot use Quartz on Edison, because Quartz needs privileged accesses to the system to install a driver. To emulate NVM on Edison, we leverage two NUMA nodes in each Edison node, and control data placements on the two NUMA nodes. When we run an MPI task on a processor of an Edison node, the processor's local NUMA node works as DRAM, while the other NUMA node works as NVM.

There are two ways to direct data placement on a specific NUMA node.

- Use numactl command. For example,
  numactl –membind=1 –cpunodebind=0 ./app

With the above command line, *app* will be pinned to a CPU attached to the NUMA node 0, and the memory will be allocated from the NUMA node 1. This method can easily enable the emulation of an NVM-only system.

- The other method is to use the libnuma library (https://linux.die.net/man/3/numa). The libnuma library offers a simple programming interface to control data placements on multiple NUMA nodes. In our paper, we adopt this method to enable fine-grained control over data placement.

We use the following two APIs to allocate data on a specific NUMA node. The first API allocates memory on a specific NUMA node; the second one runs the current task and its children on a specific NUMA node.

```
numa_alloc_onnode(...)
numa_run_on_node(...)
```

A.5 PMPI Wrapper

To automatically form MPI-delineated execution phases, we build a PMPI-based library. To build the library, we use a PMPI wrapper generator from LLNL to generate a PMPI wrapper including customized timing and performance counter measurement (see Figure 7 in the SC'17 paper). Source code and instruction for the wrapper generator are available from the LLNL Github website. (https://github.com/LLNL/wrap)

A.6 Pointer Switch in Fortran Program to Implement Data Placement

Since we dynamically change data placement between DRAM and NVM, we use a pointer switch-based approach to switch pointers pointing to the data object in either DRAM or NVM. Our benchmarks are based on Fortran, but we use a c-based function to switch pointers. The following is an example showing how we switch pointers at runtime without impacting program correctness

```fortran
!Fortran
program test
integer val1,val2   !val1 in DRAM, and val2 in NVM
pointer(ptr_val1, val1)   !bind ptr_val1 to val1
pointer(ptr_val2, val2)   !bind ptr_val2 to val2
call unimem_pointer_switch(ptr_val1, ptr_val2)
end program test


/* pointer-switch based on c */
void unimem_pointer_switch_(void **a, void **b){
      void *tmp = void *b;
   *b = a;
   *a = tmp;
}
```

Figure 1. c-based implementation for pointer switch in Fortran program.

A.7 Sampling-based Performance Counter Measurement

Precise event-based sampling (PEBS) is an advanced sampling feature for Intel processors. With this feature, the processor directly records hardware event samples, including memory addresses, into a designated memory region. We use this library (https://github.com/numap-library/numap) to enable online recording of last level cache miss events based on PEBS.

A.8 Benchmarks

We use six benchmarks (CG,FT,BT,LU,SP,MG) from NAS parallel benchmark (NPB) suite 3.3.1, and one production scientific code Nek5000(eddy).
Compilation Method:

- NPB bechmark:
  make cg CLASS=C NPROCS=16
  Note, "cg" is the bechmark name, "C" is the input problem size, and "16" is the number of MPI tasks.

- Nek5000 (https://nek5000.mcs.anl.gov):
  To enable a large scale simulation, we generate a larger problem size for an input problem, eddy. The following is how we generate the problem.
  (1) Edit $SIZE$ and *eddy_uv.box* and change the mesh size to 256x256
  (2) cd NekExamples/eddy && makenek clean
  (3) genbox
  (4) eddy_uv.box
  (5) genmap
  (6) eddy_uv
  (7) makenek eddy_uv && nekmpi eddy_uv 16

A.9 Experiment Workflow

- Install and load the Quartz emulator kernel module. Choose either latency or bandwidth model. Configure latency and bandwidth parameters.

- Change the application by adding `unimem_init` in front of the main computation loop

- Change the application by adding `unimem_start` and `unimem_end` to enclose the main computation loop.

- Choose target data objects and use `unimem_malloc` to allocate them. Free data objects by `unimem_free` after the main computation loop.

- Use the two OpenMP-like directives to mark the data dependency in the application (see SC'17 paper). Run the ROSE compiler to generate the data dependency result.

- Recompile the application and link it with our PMPI wrapper and Unimem library.

- Use Linux shell command to execute test benchmarks. e.g: mpirun -np 16 -host node1,node2,node3,node4 cg.C.16

- Observe results and measure performance.