Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs

Kai Wu
kwu42@ucmerced.edu

Jie Ren
jren6@ucmerced.edu

Dong Li
dli35@ucmerced.edu

University of California, Merced

***Abstract*—Non-volatile memory (NVM) provides a scalable solution to replace DRAM as main memory. Because of relatively high latency and low bandwidth of NVM (comparing with DRAM), NVM often pairs with DRAM to build a heterogeneous main memory system (HMS). Deciding data placement on NVM-based HMS is critical to enable future NVM-based HPC. In this paper, we study task-parallel programs, and introduce a runtime system to address the data placement problem on NVM-based HMS. Leveraging semantics and execution mode of task-parallel programs, we efficiently characterize memory access patterns of tasks and reduce data movement overhead. We also introduce a performance model to predict performance for tasks with various data placements on HMS. Evaluating with a set of HPC benchmarks, we show that our runtime system achieves higher performance than a conventional HMS-oblivious runtime (24% improvement on average) and two state-of-the-art HMS-aware solutions (16% and 11% improvement on average, respectively).**

***Index Terms*—Non-volatile memory, runtime, task-parallel, data management**

I. Introduction

Non-volatile memory (NVM), such as phase change memory (PCM) and STT-RAM, is promising for future high-performance computing (HPC). Some NVM techniques have a higher density than DRAM. This indicates that NVM can have a larger capacity than DRAM with the same area size. NVM also offers much lower memory access latency and higher memory bandwidth (comparing with traditional hard drive and SSD). Such superior performance in combination with the non-volatility nature makes NVM a candidate to replace DRAM as main memory while providing data persistence.

NVM is often paired with a small portion of DRAM as main memory [1], [2], [3], [4], [5], [6], [7], because NVM techniques, although promising, still have relatively longer access latency (4x to 1000x longer [8]) and lower memory bandwidth (1/5x to 1/50x lower [8]) than DRAM. Without using DRAM with NVM, HPC applications can have large performance loss [1], [7]. On a heterogeneous memory system (HMS) built with NVM and DRAM, we must decide on data placement: given a data object, should it be placed in NVM or DRAM? DRAM has better performance but has limited space, and frequent data movement between NVM and DRAM may bring large runtime overhead. Existing solutions that require disruptive changes to hardware [9], [10], [5], [11] or software [1], [4], [2] can be difficult to be deployed in HPC. In addition, HPC is highly sensitive to performance. Any solution that causes large performance loss is not acceptable.

To enable future NVM-based HPC, we must evolve HPC runtime systems and programming models to accommodate unique features of emerging HMS (especially NVM-based HMS). In this paper, we focus on task-parallel programs and introduce a runtime system to address data placement on NVM-based HMS.

Task-based programming models for building task-parallel programs, such as OpenMP tasks, Cilk [12], and Legion [13], decompose a program into a set of tasks and distribute them between processing elements. Those programming models improve performance by exposing a higher level of concurrency than what is usually extracted by compiler and programmer. Task-based programming models and task-parallel programs have been widely explored in HPC.

Different from the existing performance optimization work for task-parallel programs, deciding on data placement on HMS for task-parallel programs is a new and challenging problem. *First*, the existing work for task-parallel programs [14], [15], [16] studies task movement (e.g., making a task close to data on a NUMA node), while on HMS, we study data movement. Moving data to DRAM can be beneficial for performance on HMS [7], [17], [1], [2], [11], [6] , because of a relatively big performance gap between DRAM and NVM. However, data movement is expensive. As a result, we want to move data that can bring the largest performance benefit among all data and avoid less beneficial data movement. Furthermore, a task can have its data distributed on both DRAM and NVM. Given many possible data distributions for each task and many tasks in a task-parallel program, it is non-trivial to make a decision on data placement.

Second, profiling the memory accesses of tasks to decide data placement is challenging. The existing work commonly uses online performance profiling [7], [18], [19], [20], [21] for HPC applications. Leveraging iterative structures in HPC applications, profiling an execution phase can often make good performance prediction for the future execution phases. This profiling method is based on an implicit assumption that the profiled phase and future execution phases access the same data. Hence, the profiling result in one phase can be used

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

to direct data placement for the same data for the future execution phases. However, this assumption does not hold for task-parallel programs: To enable task level and data level parallelism, different tasks in a task-parallel program often work on different data. No matter which task is profiled, the profiling result for one task is not usable to direct data placement for other tasks, because of the difference in memory addresses and access patterns between tasks. In essence, the execution model of task-parallel programs brings this unique profiling challenge.

In this paper, we introduce a runtime system, *Tahoe*, to enable efficient data management (i.e., data placement between NVM and DRAM) on NVM-based HMS. Leveraging the *semantics and execution mode* of task-parallel programs, Tahoe efficiently characterizes memory access patterns, decides data placement for many tasks, makes the best use of limited DRAM space, and reduces data movement overhead.

To address the challenge of profiling memory accesses for many tasks without causing expensive overhead, Tahoe chooses a few representative tasks to profile and decide the most accessed pages. Each representative task has similar memory access patterns to many other tasks. To make the memory access information generally applicable to other tasks with different memory pages (addresses), Tahoe leverages program semantics to transform the information from page level to data-object level, such that other tasks can decide their potentially most accessed pages using data object information.

To decide data placement, Tahoe is featured with a hybrid performance model to predict performance for various data placement cases. The hybrid performance model combines the power of both machine learning modeling and analytical modeling. Predicting the performance of various data placements must capture complicated (possibly non-linear) relationships between execution time and many performance events. A complicated analytical model is possible but would cause large runtime overhead and present challenges in model construction, even for simple data placement cases. We reveal that lightweight machine learning modeling is sufficient to make the prediction for simple data placement cases. However, lightweight machine learning modeling lacks flexibility, as making prediction for complicated data placement cases increases model parameters by 40%. Such a machine learning model is difficult to train and heavyweight for runtime. Analytical modeling does not have this problem because of its flexible parameter setting and formulation. Hence, to predict performance for a task with all of its data placed in one memory (simple data placement cases), we apply a machine learning model. To predict the performance for a task with its data distributed in both NVM and DRAM (complicated data placement cases), we apply a lightweight analytical model based on the machine learning modeling result. In essence, the machine learning model avoids most of modeling complexity, while the analytical model introduces modeling flexibility.

The primary contributions of this work are as follows:

We introduce a runtime system for task-parallel programs to manage data placement on NVM-based HMS;

We explore how to capture and characterize memory access information for many tasks;

We use a hybrid performance model to make data placement decisions with high prediction accuracy (the prediction error is less than 7%);

Evaluating with six benchmarks and one scientific application, we show that Tahoe achieves higher performance than a conventional HMS-oblivious runtime (24% improvement on average) and two state-of-the-art HMS-aware solutions (16% and 11% improvement on average, respectively).

II. Background

A. Task-parallel Programs

A task-parallel program is typically based on a task-based programming model. In such programming model, the programmer or compiler identifies tasks (code regions) that may run in parallel and annotates the memory footprint of task arguments (i.e., memory addresses of major data objects within tasks). The runtime system for a task-based programming model uses memory footprint information associated with tasks to identify task dependencies and build dependency graphs at runtime. Tasks without dependency can be immediately scheduled for execution on available processing elements; tasks with dependency stay in an internal data structure (e.g., a FIFO queue) within the runtime, waiting for their dependencies to be resolved. Hence, tasks can be executed out of order by the runtime scheduler without violating program correctness.

In this paper, we use two terms, *task type* and *task size*. We define them as follows. Tasks in a typical task parallel program can run the same code region or different code regions. If some tasks run the same code region with the same input data size, we claim those tasks have the same task type. Those tasks are different *instances* of the same task type. Task size is related to task execution time. A task with a small (or big) size has a short (or long) execution time.

In this paper, we consider the OmpSs programming model [22], which is a task-based programming model using syntax similar to the OpenMP task pragma. This programming model introduces a dependency clause that allows task arguments to be declared as *in* (for read-only arguments), out (for write-only arguments), and *inout* (for read and written arguments). Our runtime, Tahoe, is an extension of Nanos++ [23], a runtime system that supports OmpSs, OpenMP and Chapel task-based programming models.

Figure 1 gives an example code from a benchmark (heat) in the BSC application repository [24] to show task parallel programs. Lines 15-25 are a code region where a task construct (Lines 1-11) is enclosed in a parallel region (i.e., a three-level nested loop). All tasks running the code region have the same task type.

A task in a task-based programming model can be in different execution states. In Nanos++, a task can be in the following four states, and tasks in the state of *ready* are placed in a queue (*readyQueue*). Our runtime leverages the four states to make data migration. We review the four states as

follows. (1) *Initialized*: The task is created and dependencies are computed. (2) *Ready*: All input dependencies of the task are addressed. (3) *Active*: The task has been scheduled to a processing element, and will take a finite amount of time to execute. (4) *Completed*: The task terminates, and its state transformations are guaranteed to be globally visible. The task also frees its output dependencies to other tasks.

```
#pragma omp task n
    in(([realN]oldPanel)[1;BS][1;BS] ...) out (...)
void jacobi(long realN, long BS, n
        double newPanel[realN][realN], n
        double oldPanel[realN][realN]) f
  for (int i=1; i <= BS; i++) f
    for (int j=1; j <= BS; j++) f
      newPanel[i][j] = 0.25   (oldPanel[i  1][j] n
          + oldPanel[i+1][j] + oldPanel[i][j  1] n
          + oldPanel[i][j+1]);
g g g

void main()f
...
#pragma omp taskwait
for (int iters=0; iters<L; iters++) f
  int currentPanel = (iters + 1) % 2;
  int lastPanel = iters % 2;
  for (long i=BS; i <= N; i+=BS) f
    for (long j=BS; j <= N; j+=BS) f
      jacobi(realN, BS, n
            (m_t) &A[currentPanel][i  1][j  1], n
            (m_t) &A[lastPanel][i  1][j  1] );
g g g
#pragma omp taskwait
...
g
```

Fig. 1. Code snippet from a task parallel benchmark (heat).

B. Architecture for NVM-Based HMS

In this paper, we assume that NVM and DRAM are constructed as separate NUMA nodes within a machine. DRAM shares the same physical address space as NVM (but with different addresses). This assumption has been widely used in the existing work [17], [1], [2], [3], [6]. Because DRAM resides in a regular NUMA node, the DRAM space is manageable at the user level by the runtime, and data migration between NVM and DRAM can be implemented at the user level by using the existing system mechanism for NUMA (e.g., *move_pages*() and *mbind*). The virtual addresses of data objects after migration remain the same. Hence, such architecture can avoid disruptive changes to the operating system (OS) and application for data management on HMS. In this paper, we migrate data at the granularity of memory pages using *move_pages*() at the user level. In addition, we use the term "data migration" interchangeably with the term "page migration"; "memory page" and "memory address" in the rest of the paper refer to "virtual memory page" and "virtual memory address". NVM endurance is out of the scope of this paper and can be handled by memory controllers [25], [26]. Many related works focus on performance, not on NVM endurance [1], [4], [7], [17].

It is possible that NVM-based HMS uses an architecture different from the above. For such case, OS may be changed to accommodate data migration requests from the runtime. For such case, we assume that OS exposes an API that allows the runtime to migrate data between NVM and DRAM at the user level and guarantees the availability of data migration destination (NVM or DRAM).

III. Design

The design goal of Tahoe is to automatically manage data placement (or migrate data) on NVM and DRAM for tasks with minimum runtime overhead. Initially, all data objects (or memory pages) in all tasks are on NVM, but Tahoe moves data objects between NVM and DRAM before task execution to improve task performance. We describe the design of Tahoe in details in this section.

A. Overview

Tahoe is built with four basic components for data management, including task metadata and profiling, performance modeling, data migration, and DRAM space management. In addition, Tahoe has three optimization techniques for performance improvement. We explain the typical workflow of Tahoe to briefly introduce the four basic components. Figure 2 generally depicts the workflow.

Tahoe decides if a task is scheduled to immediately run, based on *task metadata*. Before a task (named as the "target task" in the rest of the discussion) to run, Tahoe determines which memory pages of the task should be migrated from NVM to DRAM. Tahoe makes such decision based on the information provided by the components of *DRAM space management* and *task profiling*.

The task profiling component collects task execution information and memory access information by running representative tasks. A representative task has the same task type as the target task. Using the representative task, we avoid the necessity of profiling every task. The memory access information is collected by performance counters in the sampling mode, such that we can attribute memory accesses to memory pages. The memory access information is compact to enable good performance.

To handle the cases where multiple target tasks are scheduled to immediately run and the DRAM space must be partitioned between those tasks, we introduce a hybrid *performance model* to predict what is task execution time when some memory pages of a task is on DRAM, while other pages of the task are on NVM. Using the performance model, the *data migration* component makes the best use of DRAM for performance improvement.

The DRAM space management component provides information on page residency on DRAM. This component also migrates memory pages from DRAM to NVM based on the recency of task execution. The DRAM space management component ensures that DRAM does not run out of space.

We describe the above components in details as follows.

B. Task Metadata and Profiling

Our runtime leverages task metadata associated with each task to facilitate data migration. Also, data migration for each task is based on performance profiling on representative tasks. We describe those details in this section.

Task metadata. Task metadata is critical for data migration. In Nanos++, each task has metadata created during task creation. The metadata includes (1) task execution state and

TABLE I
THE NUMBER OF TASK TYPE FOR EVALUATION BENCHMARKS

FFT	BT	Strassen	CG	Heat	RandomAccess	SPECFEM3D
6	23	10	10	1	1	22

Fig. 2. The typical work ow of Tahoe

(2) data object information for task execution. The data object information includes data addresses (starting addresses) and data sizes for data objects referenced in the task. The data addresses and data sizes information are useful for Nanos++ to identify data dependency between tasks. Nanos++ also has a FIFO queue (i.e., readyQueue). This queue saves those tasks that already resolve data dependency and are ready to run.

Tahoe leverages the existing task execution state in Nanos++ to decide when to trigger data migration. A task with the execution state as initialized means that the task has the memory information ready, and Tahoe can use performance modeling (Section III-D) to decide which data should be migrated. A task with the execution state as ready is ready to migrate its data, but the data migration must nish before the runtime sets the task as active. A task with completed state is ready to release its data for migration by Tahoe.

Tahoe leverages the existing data object information in Nanos++ to determine which task should wait because of data migration of other tasks. Tahoe also calculates virtual page numbers by the aligned data addresses and data sizes. The virtual page numbers are needed to pro le page-level memory access information for tasks (see below).

Task pro ling. To decide which memory pages should be migrated for each task, we must collect task execution information and memory access information for memory pages. The task execution information of a task includes number of instructions, last level cache miss rate, and execution time when all data of the task are on NVM. Such task execution information is necessary for using our performance model (Section III-D). The memory access information includes the number of memory accesses to memory pages of the task.

To collect the above information, Tahoe pro les one instance (i.e., a representative task) of each task type, and then uses the pro ling information to direct data placement for the other instances of the same task type. This pro ling method is based on the observation that all instances of the same task type often perform the similar computation and have similar memory access patterns.

The task execution information can be easily measured with common performance counters in processors. To collect the memory access information, we use the common sampling mode in performance counters (e.g., Precise Event-based Sampling from Intel or Instruction-based Sampling from AMD). Such a sampling mode allows us to take a sample of a performance event (e.g., last level cache miss or rst-level cache hit) every n of such events. The sampling mode allows us to correlate the sample with a memory address whose associated memory reference causes the performance event. Using the memory address and task metadata (particularly data object addresses), we can know which memory page is accessed and which data object is accessed.

The number of last level cache miss can indicate the number of main memory accesses [7], [27]. Although other events, such as prefetching and cache coherence, can also cause main memory accesses, there is no common method to measure those events. To measure the number of main memory accesses, we use the approach in [27] by adding the number of hits in the rst-level cache to the number of last level cache misses as main memory accesses, because the rst-level cache loads include accesses to prefetched data. Using the above sampling mode, we can estimate the number of memory accesses to all memory pages of a task and decide the most accessed pages.

Pro ling overhead analysis. The runtime overhead is an important concern when attributing memory samples to memory pages. In our design, such runtime overhead is small, because of the following reasons. First, the number of representative tasks is typically small and each task type has many instances, which means we do not have many pages for pro ling. Studying all benchmarks (17 benchmarks) from the BSC application repository [24], we nd that the average number of task type per benchmark is 7 (23 at most). We list the number of task types for those benchmarks used in our evaluation in Table I. Each task type can have at least 30, and sometimes more than 1000 instances. Also, we observe that in many benchmarks, each representative task has a small memory footprint (less than a few megabytes) and the size of the memory footprint is independent of the input problem size of benchmarks. Having such task with a relatively small memory footprint is due to the nature of task-parallel HPC programs, which is to decompose computation into many ne-grained tasks and encourage task-parallelism.

Representation of memory access information. To reduce storage overhead of recording the number of memory accesses to each memory page of a representative task and quickly indicate the most accessed pages, we use the following method: we coalesce memory pages with continuous virtual addresses and with a similar number of memory accesses (less than 10% difference) into a memory group. The number of memory groups in a task is much less than the number of memory pages. The number of memory accesses for each page within a memory group is the average number of memory accesses of all pages within the group. The memory access information is represented as a list of items, each of which includes the

Fig. 3. Mapping memory access information from page level to data object level.

number of memory accesses and starting address for either a memory group or a memory page.

The memory access information is collected for the representative task and cannot be directly used by other tasks because different tasks can use different virtual addresses for their data objects. To solve this problem, we map the memory access information from page level to data object level. Leveraging data semantics, the memory access information at the data object level is generally applicable to any task with the same task type as the representative task.

We use Figure 3 to further explain the idea. In this figure, the task$_i$ has a memory page frequently accessed. Mapping the memory access information from page level to data object level, we know that this page is filled with elements of a data object$a[]$. Hence, those elements of $a[]$ are frequently accessed. The task$_j$ has the same task type as the task$_i$. Based on the profiling information at data object level in task$_i$, we reason that those elements of $a[]$ in task$_j$ will be frequently accessed and the corresponding memory page will also be frequently accessed.

Putting it all together. Tahoe maintains a hashmap, named as a *profiling database*. The profiling database uses the task type as the key and the profiling information (task execution information and memory access information) as the value. A task type is represented by a concatenation of the following items: (1) the address of the first instruction in the code region of the task type; and (2) the size of each data object listed in the dependency clauses of the task.

Tahoe picks up tasks from *readyQueue* one by one to decide data placement and run tasks. For each task, Tahoe queries the profiling database to decide if a task with the same task type has been executed before. If not, Tahoe will not make any data migration for the task. Instead, the task will be scheduled to run as usual with its data on NVM. The task is a representative task for any instance of the same task type. The profiling information is collected during the execution of the representative task and saved into the profiling database. If such a type of the task has been executed before, then the profiling information is loaded from the profiling database for deciding data migration and performance modeling.

Similarity of memory access patterns between tasks. Tahoe uses a single task as a representative task for a task type, based on the assumption that all tasks with the same task type have similar memory access patterns. However, we note a couple of cases, e.g., the benchmarks *Heat* and *RandomAccess* (see Table IV), that violate the assumption. Nevertheless, the profiling result from the representative task still provides better guidance for data placement than an HMS-oblivious runtime (see Figure 4 and 5). Also, profiling multiple representative tasks (instead of one) for a task type can make such guidance even more useful.

C. Data Migration

Whenever there is a processing element ready to run a task, a task at the front of readyQueue will be scheduled to immediately run. Right before the task runs, Tahoe decides which memory pages of the task should be migrated from NVM to DRAM.

We must handle the following issues for data migration.

Deciding which memory pages to migrate. A task can reference many memory pages. Given the limited DRAM capacity, not all memory pages can be migrated. We must decide migrating which memory pages bring the largest performance benefit. We make such decision using two steps.

First, we decide how many memory pages can be migrated. Based on the DRAM space management (Section III-E), we can know which memory pages of the task are already in DRAM. Combining such information with the availability of DRAM space, we can decide how many memory pages can be migrated from NVM to DRAM. Second, based on the profiling information (Section III-B), we decide the most accessed memory pages to migrate and then update the DRAM information in the DRAM space management.

Data migration for multiple tasks. When there are multiple processing elements ready to co-run multiple tasks, we must partition the available DRAM space between the tasks to maximize performance benefit of data migration. We use a performance model to decide the partition.

Assume that we have K tasks to co-run. After deciding the DRAM space partition, a task$_i$ $(1 \le i \le K)$ has m_i pages on DRAM and its performance is $perf_i$. To maximize the system throughput to process tasks, we have the following formulation, where *size* is the available DRAM space and *Perf* is the execution time to finish all K tasks:

$$\sum_{i=1}^{K} m_i = size \quad (1)$$

$$Perf = \max_{1 \le i \le K} perf_i \quad (2)$$

To know $perf_i$, we use a performance model (Equation 4). To solve the above equations, we use dynamic programming. To avoid the overhead of dynamic programming, when co-run tasks have the same task type, we evenly partition available DRAM space between co-run tasks without using dynamic programming. This method is based on the observation that tasks with the same task type have similar memory access patterns in most cases.

Handling conflicting decisions on page migration. A memory page can be referenced by more than one task, and multiple tasks can make conflicting decisions on the placement of a page. For such a case, we always place the page on DRAM, because those tasks that decide to place the page on NVM do not lose performance when the page is actually placed on DRAM.

D. Performance Modeling

Performance modeling is used to decide the DRAM space partition between multiple tasks, when those tasks are ready to be run by multiple processing elements. To achieve the above modeling goal, our performance model aims to predict the performance for a task when a part of its memory pages is on DRAM and the other part is on NVM (i.e., $perf_i$ for task i in Equation 2).

Our performance model has two parts. The first part uses a machine learning model to predict the performance of a task with all of its memory pages on DRAM (we name such a case as *complete data placement*). The second part is based on the first one and predicts the performance when some (not all) of the task's memory pages are placed on DRAM (we name such case as *partial data placement*). The second part is an analytical model.

We have the following requirements for our performance modeling. (1) Application generality: the model must work for a large variety of applications; (2) Complexity: the model must be simple enough to have low runtime overhead; (3) Usability: the model must have low programmer involvement; (4) Hardware generality: the model must be easily extensible to different hardware platforms. We describe our performance model in details as follows.

1) Performance Modeling for Complete Data Placement. We introduce a performance model based on machine learning. We do not use analytical modeling because when capturing the sophisticated relationship between execution time and performance events, the analytical modeling tends to be complex (e.g., [28]). It can bring large runtime overhead and construction difficulty, violating the above requirements (2)-(4).

Given a task, the machine learning model uses the following information as input: (1) last level cache miss rate, and (2) IPC (instructions per cycle) when all memory pages of the task are on NVM. The model outputs (predicts) IPC for task execution when all memory pages of the task are on DRAM. The input of the model can be obtained from the profiling database. In particular, using the task type as a key, we can get the task execution information collected from a representative task from the database. Based on this information, we calculate the model input. With the model output (i.e., predicted IPC), we calculate the task execution time of complete data placement, using the number of instructions obtained from the profiling database.

We choose last level cache miss rate and IPC as the model input, because they are highly correlated with performance variation across different cases of data placement, and hence can serve as important performance indicators. In particular, the last level cache miss rate reflects how intensively main memory is accessed. The performance of an application with a high last level cache miss rate could be sensitive to the change of main memory bandwidth and latency. IPC can reflect main memory access intensity and overlapping between computation and memory access. The performance of an application with high IPC may not be sensitive to the change of main memory bandwidth and latency.

TABLE II
TRAINING TIME AND PREDICTION ACCURACY. NVM BANDWIDTH IS 1/X BANDWIDTH OF DRAM (X = 4; 8; AND 16).

	Multiple LR model			ANN model		
NVM bandwidth	1/4	1/8	1/16	1/4	1/8	1/16
Average training time per epoch (s)	25.3	23.5	22.4	32.4	31.7	33.8
Total training time (s)	207.2	191.4	195.0	254.9	249.6	262.3
Average prediction error	10.9%	26.4%	45.9%	3.6%	4.1%	5.1%
Prediction error variance	0.2	57.2	4.7 10^3	0.007	0.016	0.017

We explore two common supervised machine learning techniques to build our models and meet the modeling requirements: multiple linear regression analysis (LR) and artificial neural network (ANN).

Multiple LR analysis. Our regression model is as follows.

$$y = {}_1x_1 + {}_2x_2 + \quad (3)$$

where x_1, and x_2 are IPC and last level cache miss rate, respectively. y is the predicted IPC. $_1$, $_2$ and are modeling coefficients we learn through model training.

ANN. A typical ANN has a number of neurons. Each neuron receives inputs from other neurons or ANN input, and produces an output via activation functions. Neurons, connected with weights and organized as layers, constitute the network structure of ANN.

In our model, we use a three-layer, fully-connected ANN containing one input layer with ten input neurons, one hidden layer with five neurons, and one output layer with one output neuron. We use such simple ANN to avoid large runtime overhead when making online performance prediction. We use Rectified Linear Unit (ReLu) as the activation function in our ANN.

Model training and validation. We use seven task parallel benchmarks (see Table IV) from the BSC application repository [24] for model training and validation. In particular, we choose every six benchmarks of the seven task-parallel benchmarks to build two models (LR and ANN), and use the one remaining benchmark for validation (training and validation use different data sets). In total, we build seven LR models and seven ANN models for cross-validation. For each model, we have at least three million tasks from six benchmarks for training, and use at least 0.7 million of tasks from one remaining benchmark for validation.

The training data is collected in a machine described in Section IV. On this machine, we configure our NVM emulation with three different bandwidth (1/4, 1/8, and 1/16 of DRAM bandwidth). Hence, we have three NVM cases, and for each case, we collect the training data to train the two models (LR and ANN). The average training time of those models is summarized in Table II. Overall, the training time is short less than five minutes for all cases.

Performance modeling accuracy. Table II shows the prediction accuracy and reveals that the ANN model achieves high prediction accuracy (less than 6% prediction error on average)

for the three different NVM cases. LR, however, does not predict well (e.g., 45.9% prediction error on average, when the NVM bandwidth is con gured as 1/16 of DRAM bandwidth). Hence we use the ANN model in Tahoe.

Modeling complexity. Our ANN model is simple. The model training happens of ine, and to make a prediction at runtime, the model uses 76 oating point multiplications and 75 oating point additions. As a result, the modeling complexity is low.

In summary, our ANN model meets our modeling requirements: it has good application generality and is simple and usable. The model training time is also short.

However, the machine learning-based performance modeling is not suitable for making performance prediction for partial data placement (more complicated data placement cases) because we have to introduce at least two more input (one for DRAM and the other for NVM) to represent and distinguish cases with different numbers of memory pages on DRAM and NVM and possibly a couple more input to characterize memory access patterns to improve modeling accuracy. This increases model parameters by at least 40% (considering just two more input). Such a model is not only dif cult to train but also brings large runtime overhead, which violates the model requirements on complexity, usability, and generality. This problem, in essence, comes from the lack of exibility to build and use the machine learning model.

2) Performance Modeling for Partial Data Placement: We introduce an analytical model to make performance prediction for partial data placement. The model uses the prediction result of the complete data placement and uses simple formulation and parameters to capture the performance relationship between complete and partial data placement. The model avoids the problem of model training and concerns on runtime overhead in the machine learning model.

The analytical model is based on the following rationale. Assume that T_{c_NVM} and T_{c_DRAM} are the execution times with complete data placement on NVM and DRAM, respectively. We have performance difference T_c ($T_{c\_NVM} - T_{c\_DRAM}$), and the performance difference between partial data placement (T_p) and complete data placement on DRAM (T_{c_DRAM}) should be less than T_c ($T_{c\_NVM} - T_{c\_DRAM}$). In general, more NVM accesses in partial data placement result in a larger performance difference between partial data placement and complete data placement on DRAM. Such a performance difference should be related to the ratio of NVM accesses to total memory accesses (including both DRAM and NVM accesses).

$$T_p = (T_{c\_NVM} - T_{c\_DRAM}) \frac{p\_nvm\_acc}{tot\_mem\_acc} + T_{c\_DRAM} \quad (4)$$

Equation 4 shows the model based on the above rationale and predicts the performance for partial data placement. T_p (T_{c_NVM} in the model is measured and obtained from the pro ling database. T_{c_DRAM} is the predicted execution time with the ANN model. T_c ($T_{c\_NVM} - T_{c\_DRAM}$) is the performance difference for complete data placement, which is the largest performance difference we can have. The performance difference for partial data placement scales the largest performance difference by p_nvm_acc/tot_mem_acc, where p_nvm_acc is the number of NVM accesses in partial data placement and tot_mem_acc is total number of memory accesses in complete data placement. p_nvm_acc is the model input and can be leveraged to explore the performance of various data placement as in Equation 2. tot_mem_acc is measured and obtained from the pro ling database.

TABLE III
PERFORMANCE PREDICTION ERROR FOR PARTIAL DATA PLACEMENT

Benchmarks	FFT	BT	Strassen	CG	Heat	RA	SPECFEM3D
p_nvm _acc	5:7 10^7	1:9 10^8	7:7 10^6	4:3 10^7	5:2 10^7	1:0 10^8	7:4 10^7
tot _mem _acc	1:2 10^8	4:1 10^8	1:6 10^7	7:4 10^7	2:2 10^8	2:7 10^8	1:45 10^8
$\frac{p\_nvm\_acc}{tot\_mem\_acc}$	0.48	0.46	0.48	0.58	0.24	0.37	0.51
Prediction error	6.9%	3.6%	3.0%	1.5%	3.0%	3.0%	6.5%

To verify the modeling accuracy, we test the seven benchmarks listed in Table IV. We use a machine with two NUMA nodes to emulate NVM based on Quartz [29] emulator. Section IV has more details on our test platform. We do not set the limitation on DRAM size. Both DRAM and NVM can hold all memory pages of the benchmarks. We collect the execution times and the number of memory accesses on NVM and DRAM under three con gurations: (1) placing all memory pages on NVM, (2) memory is allocated using a round robin approach on both NVM and DRAM, and (3) placing all memory pages on DRAM. The model makes performance prediction for the second con guration, and uses the rst and third con gurations as model inputs. We compare the measured time and predicted time for the second con guration, and compute the prediction error shown in Table III. In summary, the prediction error is less than 7%, demonstrating the effectiveness of our model.

E. DRAM Space Management

DRAM space management has two functionalities: (1) recording which memory pages are in DRAM; (2) migrating memory pages from DRAM to NVM when DRAM runs out of space and there is a task pending to execute.

To implement the rst functionality, Tahoe represents DRAM pages as a list of memory regions. Each memory region is represented as the starting address and size of the region. Each memory region is a set of memory pages with continuous addresses. If a task wants to check which pages of the task are on DRAM, it sends the address ranges of its memory pages, and compares its address ranges with the address ranges in the list of memory regions.

All memory pages are initially allocated on NVM and no memory page is on DRAM. As memory pages are migrated from NVM to DRAM, DRAM can run out of space, and we must migrate some page from DRAM to NVM to accommodate new memory pages from the upcoming task executions. To decide which DRAM pages should be migrated to NVM, we could use an LRU policy and migrate those pages that are the least used. However, this would require the runtime to continuously track memory references to DRAM pages, which

is costly. To avoid large runtime overhead, we migrate those DRAM pages that are used by the least recently executed task. In particular, Tahoe maintains a FIFO queue with a length of ten to record DRAM memory footprints of the last ten executed tasks. If DRAM runs out of space, DRAM pages referenced by the task at the end of the queue are moved out of DRAM.

In other words, we migrate memory pages from DRAM to NVM based on the recency of task execution, not the recency of memory usage. This method has some limitation, however. A memory page used by the least recently executed task can still be referenced by recently executed tasks, and it is possible that the memory page will be accessed by the upcoming tasks too. To reduce this limitation, before migrating pages from DRAM to NVM, we quickly examine readyQueue to check if the most upcoming task is going to use the pages pending to migrate from DRAM to NVM. We do not migrate those DRAM pages that are going to be used by the most upcoming task.

F. Performance Optimization

We introduce several techniques to improve performance.

Using helper thread to reduce data migration cost. After migrating data for the task at the front of readyQueue, it is possible that DRAM still has space. For such case, Tahoe will proactively migrate data for the task after the front task in the queue. Such proactive data migration is implemented with a helper thread running in parallel with Tahoe, overlapping with task computation and minimizing data migration cost.

Performance optimization for data migration. Calling the page migration function (i.e., move_pages()) involves flushing translation lookaside buffer (TLB). Migrating multiple pages of a task with one invocation of move_pages() often triggers TLB flush multiple times. TLB flushing is known for causing a large performance overhead [30], [31]. Hence, we combine multiple TLB flushes in one invocation of move_pages() into one TLB flush. Such a method reduces the number of TLB flushes, hence improve performance.

Note that an invocation of move_pages() only migrates pages for one task, not for multiple tasks, because a task cannot execute until the page migration function finishes. An invocation of move_pages() for multiple tasks delays the execution of multiple tasks and reduces system throughput.

Optimization of task scheduling. Tasks with the same parent usually perform the same computation and work on overlapped memory pages. Based on such observation, we slightly change scheduling orders of tasks in readyQueue, such that those tasks with the same parent are scheduled one after another. Such a task scheduling strategy maximizes DRAM page reuse before DRAM pages are evicted out of DRAM.

G. Discussions

NVM has asymmetric memory read and write latencies. However, we do not distinguish memory read and write, because using software techniques (e.g., using mprotect to make memory pages read-only and trigger a signal when write occurs) to collect read and write information for pages can be very costly. Most runtime designs for NVM rely on hardware mechanisms [32], [11], [33] to consider latency difference of read and write. The existing runtime solutions [1], [2], [3], [4], [7] do not consider such difference.

When profiling tasks and using performance models, we do not consider performance interferences between tasks. Those interferences can cause cache conflict misses and memory accesses. Due to the dynamic scheduling nature of task parallel programs, quantifying and predicting those performance interferences require runtime to infer possible task execution scenarios, which greatly increases runtime overhead and complicates runtime design. Hence, we do not consider performance interferences in our runtime.

IV. Evaluation

Experiment methodology. We use a 16-core machine with two eight-core Xeon E5-2630 processors and 32GB DDR4 (two NUMA memory nodes). We use this machine for model training and validation in Section III-D. We use Quartz [29] for NVM emulation. Quartz can emulate NVM with a range of latency and bandwidth, and offer high emulation accuracy. With Quartz, one NUMA node of the machine is used as NVM, while the other is as DRAM. We use six benchmarks from the BSC application repository [24] and one production code SPECFEM3D [34]. Appendix A has more details for benchmarks. For performance profiling, we use the sampling-based approach with sample rate as 1000. Such sampling rate offers high modeling accuracy with tolerable runtime overhead [7].

We use six systems for evaluation: HMS with Tahoe, unmanaged HMS with default Nanos++ (i.e., the HMS-oblivious runtime), DRAM-only (no NVM) with Nanos++, NVM-only (no DRAM) with Nanos++, HMS with X-mem [1], and HMS with Unimem [7]. With the unmanaged HMS, Linux allocates memory with no knowledge of the underlying memory types but is restricted by a limited DRAM size. X-mem and Unimem are two recent software-based solutions for data placement on HMS. X-mem uses offline profiling to characterize memory access patterns and make the decision on data placement. Unimem makes data placement decision at the granularity of execution phases delineated by MPI operations. Because five of our benchmarks do not have MPI, we delineate execution phases by task code regions for evaluating Unimem. Unless otherwise indicated, we use eight threads for evaluation and use the performance of the unmanaged HMS for performance normalization, and NVM is configured with 1/4 DRAM bandwidth. Unless otherwise indicated, we choose 128MB as DRAM size, which is the same as recent work [35], [7], [33], [36]. Such DRAM size is smaller than the total size of all data objects of the benchmarks, such that not all memory pages of the benchmarks are on DRAM. We list the ratio of the DRAM size to the total size of data objects of each benchmark in Table IV.

Basic performance tests. We first compare the performance (execution time) of the six systems. NVM has 1/4 DRAM bandwidth (Figure 4) or 4x DRAM latency (Figure 5).

Fig. 4. Performance (execution time) comparison between unmanaged HMS, NVM-only, X-mem, Unimem and Tahoe. The performance is normalized to that of unmanaged HMS. NVM has 1/4 DRAM bandwidth.

Fig. 5. Performance (execution time) comparison between Unmanaged HMS, NVM-only, X-mem, Unimem and Tahoe. The performance is normalized to that of unmanaged HMS. NVM has 4x DRAM latency.

Using the performance of the unmanaged HMS as the baseline, X-mem, Unimem and Tahoe reduce execution time by 5%, 11% and 21% on average respectively, when NVM has 1/4 DRAM bandwidth. When NVM has 4x DRAM latency, X-mem, Unimem and Tahoe reduce execution time by 10%, 14% and 26% on average, respectively. The unmanaged HMS does not know underlying memory types in HMS. Thus, it does not make good use of DRAM. Tahoe outperforms X-mem and Unimem by 16% and 11% on average, respectively. Tahoe performs better than X-mem, because X-mem uses of ine pro ling and uses the same data placement decision for all tasks. X-mem avoids frequent data movement, but lacks the exibility of data movement to maximize performance bene t of using DRAM. Unimem does not have the problem of X-mem, but it performs worse than Tahoe, because Unimem lacks a good capability to migrate large data objects from NVM to make best use of DRAM.

Detailed performance analysis. We quantify the contribution of our three optimization techniques to total performance improvement in Figure 6. The three techniques are (1) using helper thread for proactive data migration (labeled as “Using helper thread”), (2) performance optimization for data migration (labeled as “Optimized migration”), and (3) optimization of task scheduling (labeled as “Optimized scheduling”).

We perform our analysis with the following method. We rst remove the three techniques from Tahoe. The performance result of this case is labeled as “Preliminary Tahoe”. We then compute performance difference between the preliminary Tahoe and unmanaged case. Such performance difference is the performance contribution of the preliminary Tahoe. We then add the three techniques one by one. In particular, we apply (1), and then apply (2) to (1), and then apply (3) to (1)+(2). We measure performance variation for each case. Such performance variation is the performance contribution of each optimization technique. We normalize the performance contributions of all cases by the performance difference between the full-featured Tahoe and unmanaged case.

Fig. 6. Quantifying the performance contributions of the three optimization techniques.

Figure 6 shows the results. We notice using helper thread for proactive data migration particularly works well for CG and Strassen, because the two benchmarks have many tasks with small data sizes. Those tasks cannot make best use of DRAM, hence brings opportunities for proactive data migration. The technique of optimized migration makes big contributions to FFT, because FFT has a relatively large number of page migration requests shown in Figure 9. The technique of optimized scheduling makes limited contributions (comparing with other techniques), except in the benchmarks FFT, Strassen, and BT. Those benchmarks often use recursive task parallelism, thus have many tasks with the same parents, which provides opportunities for applying optimized scheduling.

Performance sensitivity analysis. We change NVM bandwidth and latency, number of threads, number of nodes and DRAM size to study how Tahoe responses with the various system con gurations. In this section, we present the results for changing the number of threads, but leave the other results in Appendix B.

When changing the number of threads, our machine can only offer 8 threads at most because of Quartz emulation. To enable better performance study, we use the Edison supercomputer at Lawrence Berkeley National Lab (LBNL). Each node of Edison has two 12-core Intel Ivy Bridge processors (2.4 GHz) with 64GB DDR3 (two NUMA nodes). On this platform, we leverage its NUMA architecture to emulate NVM instead of using Quartz, because Quartz requires the user to have privilege access to the test system, and we do not have such access on Edison. On the Edison nodes, threads run on one processor using the processor's local attached NUMA node as DRAM and the remote NUMA node as NVM. The latency and bandwidth difference between the remote and local NUMA nodes emulates the difference between NVM and DRAM. The emulated NVM has 60% of DRAM bandwidth and 1.89x of DRAM latency. Because of such NVM emulation, the Edison node can offer up to 12 threads. Figure 7 shows the results when we change the number of threads (from 1 to 12 threads) on an Edison node. We only report average performance of all benchmarks, because of limited paper space. Tahoe performs well consistently in all cases. In particularly, FFT (not shown in Figure 7) has only 2% performance variance when we change the number

Fig. 7. Tahoe performance (execution time) sensitivity to the number of threads on a single Edison node. Performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

Fig. 8. Memory access breakdowns. The number of memory accesses is normalized by that of the unmanaged cases.

Fig. 9. Comparing different systems in terms of number of page migrations per second.

of threads. RandomAccess (not shown in Figure 7) has the largest performance difference (only 6%).

Memory utilization analysis. Figure 8 shows the number of main memory accesses for DRAM and NVM, normalized by the numbers with the unmanaged cases. Tahoe has larger numbers of DRAM memory accesses than other systems, and hence effectively utilizes DRAM space. This result is aligned with Figures 4 and 5, where Tahoe performs consistently better than other systems.

Figure 9 shows number of page migrations per sec. The unmanaged and NVM-only do not have page migration. X-mem does not have either, because it is not a runtime solution. The page migration is more frequent in Tahoe than in Unimem, because Tahoe and Unimem work on different data granularities (page vs. data object). The ner-grained data migration as in Tahoe triggers more frequent data migration and makes the best use of DRAM, which transforms to better performance.

V. Related Work

Data management on HMS.Software-based solutions are summarized as follows. Du et. al [4] develop an of ine pro ling tool to analyze memory accesses to guide data placement. Lin et. al [3] introduce an OS service for asynchronous memory movement on HMS. Dulloor et. al [1] introduce a data placement runtime based on classi cation of memory access patterns. Giardino et. al [2] rely on OS and application co-scheduling data placement. Wu et.al [7] introduce MPI runtime for data placement. Yu et. al [17] propose three bandwidth-aware memory placement policies. Perarnau et.al [37] study data migration performance with user-space memory copy and Linux kernel-based memory migration. They demonstrate the importance of choosing a good ratio of worker threads to migration threads for performance.

Different from the prior efforts, our work does not require of ine pro ling as in [1], [4] nor programmer involvement to identify memory access patterns as in [2]. Our work also supports data migration for large data objects which is not fully supported in [7]. Furthermore, our work does not require the modi cation of OS, which is different from [3], [17]. We do not use user-space memory copy as in [37], because that may involve extensive application modi cation to use new data addresses after data copy.

Hardware-based solutions are summarized as follows. Yoon et al. [11] dynamically determine data placement based on row buffer locality. Wang et al. [5] use static analysis and memory controller to determine replacement on GPU. Wu et al. [6] use numerical algorithms and hardware modi cation to decide data replacement. Agarwal et al. [38] introduce a bandwidth-aware data placement on GPU. The major drawback of those solutions is hardware modi cations. Some work, such as [9], [10], [5], [11], ignores application semantics and triggers data movement based on temporal memory access patterns, which could cause unnecessary data movement. Our work avoids hardware modi cation and leverage application semantics.

Performance optimization for task parallel programs. Papaefstathiou et al. [39] modify hardware to prefetch task data and guide the replacement decision in caches. Ni et al. [40] uses a runtime based on Charm++ to prefetch data into fast memory. This work, however, cannot decide optimal data placement for multiple ready tasks. Pan and Pai [41] introduce a runtime to instruct hardware to prioritize data blocks with future reuse. This work needs application and hardware modi cations. Li et al. [16] adopt machine learning to estimate scheduling performance for task parallel programs. However, they cannot predict performance for various data placement cases. Our work is different from the existing efforts, and we are the rst one to study performance optimization for task parallel programs on NVM-based HMS.

VI. Conclusions

Using runtime of a programming model to direct data placement on HMS is promising. In this paper, we introduce a runtime system for task parallel programs. It leverages task metadata and representative tasks to collect memory access information and make data migration decisions. It uses a hybrid performance model to decide optimal data placement for multiple tasks. Our runtime system effectively uses DRAM space for performance improvement.

Acknowledgement. This work is partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194). We thank anonymous reviewers for their valuable feedback.

REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime Data Managementon Non-volatile Memory-based Heterogeneous Main Memory," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2017.

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

[9] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009.

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

[11] 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.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), 1995.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2012.

[14] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective," in Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism (IWOMP), 2009.

[15] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP Task Scheduling Strategies for Multicore NUMA Systems," Int. J. High Perform. Comput. Appl., vol. 26, no. 2, pp. 110–124, May 2012.

[16] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee, "Machine learning based online performance prediction for runtime parallelization and task scheduling," in 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009.

[17] S. Yu, S. Park, and W. Baek, "Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems," in Proceedings of the International Conference on Supercomputing (ICS), 2017.

[18] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid MPI/OpenMP Power-aware Computing," in International Symposium on Parallel Distributed Processing (IPDPS), 2010.

[19] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores," in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008.

[20] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," in Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, 2005.

[21] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Exploiting Barriers to Optimize Power Consumption of CMPs," in International Parallel and Distributed Processing Symposium (IPDPS), 2005.

[22] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures," Parallel Processing Letters, vol. 21, no. 02, pp. 173–193, 2011.

[23] Barcelona Supercomputing Center, "Nanos ++," https://pm.bsc.es/nanox.

[24] ——, "BSC application repository," https://pm.bsc.es/projects/bar/wiki/Applications.

[25] M. Poremba and Y. Xie, "NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories," in 2012 IEEE Computer Society Annual Symposium on VLSI, 2012.

[26] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating System Support for NVM+DRAM Hybrid Main Memory," in Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS), 2009.

[27] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

[28] Y. Huang and D. Li, "Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems," in 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017.

[29] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A Lightweight Performance Emulator for Persistent Memory Software," in Proceedings of the 16th Annual Middleware Conference (Middleware), 2015.

[30] M. Sparsh, "A Survey of Techniques for Architecting TLBs," Concurrency and Computation: Practice and Experience, vol. 29, no. 10, p. e4061, 2016.

[31] N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking," in 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017.

[32] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting Program Semantics to Place Data in Hybrid Memory," in Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015.

[33] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in International Conference on Supercomputing (ICS), 2011.

[34] D. Komatitsch and J. Tromp, "Introduction to the Spectral Element Method for Three-dimensional Seismic Wave Propagation," Geophysical Journal International, vol. 139, no. 3, pp. 806–822, Dec 1999.

[35] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," IEEE Micro, vol. 31, no. 1, pp. 99–108, Jan 2011.

[36] S. Bock, B. R. Childers, R. Melhem, and D. Moss, "Concurrent Migration of Multiple Pages in Software-managed Hybrid Main Memory," in 2016 IEEE 34th International Conference on Computer Design (ICCD), 2016.

[37] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman, "Exploring Data Migration for Future Deep-Memory Many-Core Systems," in 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016.

[38] 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.

[39] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, "Prefetching and Cache Management Using Task Lifetimes," in Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS), 2013.

[40] X. Ni, N. Jain, K. Chandrasekar, and L. V. Kale, “Runtime Techniques for Programming with Fast and Slow Memory,” in 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017.
[41] A. Pan and V. S. Pai, “Runtime-driven Shared Last-level Cache Management for Task-parallel Programs,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2015.

APPENDIX A
BENCHMARK INFORMATION

In the evaluation, we use six task parallel benchmarks from the BSC application repository [24] and one production code SPECFEM3D [34]. Table IV summarizes their input parameters and the ratio of the DRAM size (128 MB) to the total size of data objects of each benchmark.

TABLE IV
BENCHMARKS FOR EVALUATION. SIZE RATIO IS THE RATIO OF DRAM SIZE TO THE TOTAL SIZE OF ALL DATA OBJECTS.

Benchmark	Input Parameter	Size ratio
FFT	4096 4096 double matrix	1:15
BT-MZ (BT)	CLASS=C NPROCS=1	1:10
Strassen	4096 4096 double matrix	1:5
CG	4096 4096 double matrix	1:6
Heat (Jacobi)	4096 4096 double matrix	1:10
RandomAccess (RA)	1024MB memory with 1000 tasks	1:9
SPECFEM3D (SF3D)	NEX_XI=128 NEX_ETA=128	1:11

APPENDIX B
ADDITIONAL STUDY FOR PERFORMANCE SENSITIVITY

Except for the experiments presented in the evaluation section (Section IV), we perform other sensitivity study. In particular, we change NVM bandwidth and latency, number of nodes and DRAM size to study how Tahoe responses with the various system configurations. Except Figure 12, we report average performance of all benchmarks in this section, because of limited paper space.

Figure 10 shows the results when NVM has 1/4, 1/8 and 1/16 DRAM bandwidth. Tahoe brings larger performance gains (from 21% to 29%) as NVM bandwidth decreases from 1/4 to 1/16 DRAM bandwidth. This result is especially pronounced in RandomAccess (not shown in Figure 10): The performance gain increases from 32% to 86% as the NVM bandwidth decreases from 1/4 to 1/8 DRAM bandwidth.

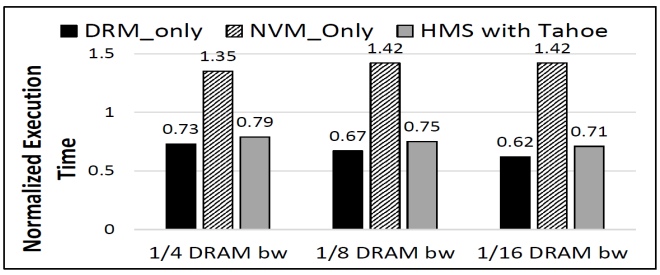

Fig. 10. Tahoe performance (execution time) sensitivity to NVM bandwidth. The performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

The performance results are slightly different when we increase NVM latency from 4x to 16x DRAM latency (Figure 11). Tahoe has only 4% performance variance when NVM latency increases. The biggest improvement (from 28% to 37%) happens in CG (not shown in Figure 11).

Figure 12 shows the results when we use different number of nodes (up to 64 nodes). We perform strong scaling tests. We only use BT and SPECFEM3D, because other benchmarks do not have MPI support. For each test, we use one MPI process per node, and each MPI process uses either 4 or 8 threads. For BT, we use CLASS D as input problem; For SPECFEM3D, we use NEX_XI = 256 and NEX_ETA = 128. As the system scale becomes larger, the performance gain of Tahoe decreases from 10% to 3% and from 16% to 4% for BT and SPECFEM3D, respectively (comparing with the unmanaged case), because the memory footprint size per node becomes smaller and more data objects can be placed into DRAM by the unmanaged case. Tahoe performs well in all cases no matter how large the memory footprint size is.

Figure 13 shows the results when we change the DRAM size. Overall, Tahoe brings performance benefit in all cases (comparing to the unmanaged case), but as the DRAM size becomes bigger, the benefit decreases from 21% to 15%, because a larger DRAM provides better opportunities to place data on DRAM for the unmanaged case.

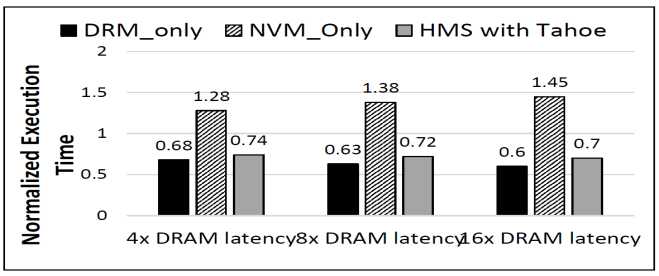

Fig. 11. Tahoe performance (execution time) sensitivity to NVM latency. The performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.

Fig. 12. Tahoe performance (execution time) sensitivity to the number of nodes on Edison. Performance is normalized to that of unmanaged HMS.

Fig. 13. Tahoe performance (execution time) sensitivity to DRAM size. Performance is average performance of all benchmarks. Performance is normalized to that of unmanaged HMS.