Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness

Zhen Xie
zhen.xie@anl.gov
University of California, Merced
Argonne National Laboratory

Jie Liu
jliu279@ucmerced.edu
University of California, Merced

Jiajia Li
jiajia.li@ncsu.edu
North Carolina State University

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

Abstract

The emergence of heterogeneous memory (HM) provides a cost-effective and high-performance solution to memory-consuming HPC applications. Deciding the placement of data objects on HM is critical for high performance. We reveal a performance problem related to data placement on HM. The problem is manifested as load imbalance among tasks in task-parallel HPC applications. The root of the problem comes from being unaware of parallel-task semantics and an incorrect assumption that bringing frequently accessed pages to fast memory always leads to better performance. To address this problem, we introduce a load balance-aware page management system, named *Merchandiser*. Merchandiser introduces task semantics during memory profiling, rather than being application-agnostic. Using the limited task semantics, Merchandiser effectively sets up coordination among tasks on the usage of HM to finish *all* tasks fast instead of only considering any individual task. Merchandiser is highly automated to enable high usability. Evaluating with memory-consuming HPC applications, we show that Merchandiser reduces load imbalance and leads to an average of 17.1% and 15.4% (up to 26.0% and 23.2%) performance improvement, compared with a hardware-based solution and an industry-quality software-based solution.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577497

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Theory of computation → Parallel computing models**; • **Hardware → Non-volatile memory**.

Keywords: Data Placement, Heterogeneous Memory, Parallel Computing, Load Balance

ACM Reference Format:
Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. 2023. Merchandiser: Data Placement on Heterogeneous Memory for Task-Parallel HPC Applications with Load-Balance Awareness. In *The 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23), February 25-March 1, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3572848.3577497

1 Introduction

Many high-performance computing (HPC) applications are becoming memory-consuming. For example, the density matrix renormalization group (DMRG) [6, 77], a numerical algorithm to obtain the low-energy physics of quantum many-body systems, can consume 1.271 TB memory in a single machine when solving the Hubbard 2D model at the scale of 320×320 [21, 46]. To meet memory requirements of those applications, the big memory system is emerging. An example of such a system is the Amazon EC2 High Memory Instance built upon eight NUMA nodes and providing up to 12 TB memory [31]. The big memory system is often heterogeneous, which means multiple memory components with different latency and bandwidth form the main memory.

HM raises a data placement problem. Because of small capacity of fast memory and relatively worse performance of slow memory, memory pages have to be allocated and migrated between fast and slow memories, such that most of memory accesses can happen in fast memory for high performance. It has been shown that some HPC applications can suffer from up to 5.7× performance loss (compared with using a fast memory-only solution) with suboptimal data placement on HM [61, 63, 67, 84].

Many solutions [19, 25, 33, 34, 41, 54, 84, 86] to address the data placement problem on HM uses a profiling-guided optimization (PGO) approach. These solutions identify frequently accessed memory pages ("hot pages") by periodically sampling memory pages and tracking memory accesses to them. Hot pages are then migrated to fast memory for better performance. These solutions are application-agnostic, meaning that they do not need application knowledge or change applications. Success of these solutions is based on an implicit assumption that placing hot pages in fast memory always leads to better performance. However, we find that it is not true for many task-parallel HPC applications.

Task-parallel programs are common in HPC. A task-parallel program can be MPI-based, and each MPI process performs a task. It can be OpenMP-based, and each OpenMP thread performs a task. There is synchronization among tasks where tasks must reach the synchronization point before they move on to the rest of computation. Because of synchronization among tasks, finishing *all* tasks fast instead of finish individual tasks fast is a key for high performance.

The PGO on HM cannot work well for task-parallel applications. They lack a view of "finishing all tasks fast" for high performance. They migrate and place hot pages into fast memory, but do not consider which task accesses those memory pages. As a result, the existing efforts could introduce load imbalance: a task unnecessarily reaches the synchronization point earlier than the others and waits for other tasks to finish, because many pages of this task are resident in fast memory, leading to its shorter execution time.

To reveal the load imbalance problem on HM, we study five HPC applications on an Optane-based HM. This HM consists of 192GB DRAM and 1.5TB Optane [32]. We study two representative solutions: an industry-quality, software solution (Intel MemoryOptimizer [16]) and a hardware solution (Memory Mode of Optane). We have two observations (see Figure 5 in the evaluation section for details).

- Compared with running on homogeneous memory, running on HM increases performance difference among tasks: on average, the performance difference among tasks is increased by 17% and 16% (when MemoryOptimizer and Memory Mode are used respectively), which indicates more load imbalance after using MemoryOptimizer and Memory Mode on HM.
- Performance improvement is minimal after using MemoryOptimizer and Memory Mode. The performance improvement is only 4.32% and 3.71% respectively (compared with using Optane only), because the overall performance is hindered by the slowest task.

There are two fundamental reasons accounting for the above performance problem. *First*, the PGO solutions (such as MemoryOptimizer) are not aware of task parallelism. There is a lack of coordination among tasks to share the limited fast memory space. That space is allocated to tasks based on opportunistic detection of hot pages from tasks, not based on performance analysis on potential performance benefit of using fast memory for tasks. It may unfairly place too many pages from one task into fast memory, causing load imbalance. *Second*, the PGO solutions use random page sampling-based memory profiling. Random sampling is effective to avoid large overhead of profiling all memory pages in a big memory system. However, it may collect many memory accesses from one task, which leads to too many pages of that task migrating to fast memory, causing load imbalance.

We introduce a load balance-aware data placement system for HM, named *Merchandiser*, to address the problem. Merchandiser introduces task semantics during memory profiling. This means Merchandiser associates memory accesses with tasks during profiling, instead of being application-agnostic. Using limited task semantics, Merchandiser effectively sets up coordination among tasks on the usage of HM. Furthermore, Merchandiser uses historical, fine-grained profiling results of the task to guide data placement for the subsequent executions of the same task with new inputs.

However, to realize Merchandiser we face two challenges. *First*, the input problem to a task during program execution can vary, and the historical profiling results collected from one input cannot be directly used to predict performance for another input, because of the difference in the number of memory accesses. *Second*, how to partition the fast memory space among tasks is challenging. Unless all tasks have the same memory access patterns and data object sizes, evenly sharing fast memory among tasks cannot work. We must decide for each task with a new input, which objects should be placed in fast memory without priori knowledge on the number of memory accesses to the objects. We must also predict execution time of tasks after migration, such that the effectiveness of load balance can be quantified and estimated.

To address the first challenge on handling new input problems, Merchandiser classifies data objects in terms of their memory access patterns, based on which Merchandiser analytically derives the number of main memory accesses for a new input problem. The memory access patterns are mostly invariant across input problems for a given task in many HPC applications, providing a reliable indication on the number of memory accesses. We also recognize the difference in the impacts of memory access patterns, and estimate the number of memory accesses differently for different patterns.

Based on the estimated memory accesses, Merchandiser introduces performance modeling to predict execution time of the task when a certain portion of memory accesses happens in fast memory while the remaining memory accesses happen in slow memory. The novelty of our performance modeling lies in the modeling of performance correlation between different data placements of the task. In particular, the performance modeling take the performance of a data placement as input, and then predicts the performance of another data placement. The performance modeling sets up a correlation between the two above performances based on

task characteristics. The task characteristics are represented and quantified using a few performance events collected from only one execution of a specific data placement.

To address the second challenge on deciding which pages should be migrated to fast memory for parallel tasks, we introduce a greedy heuristic algorithm to decide how to allocate the fast memory space among tasks to maximize performance benefit of all tasks (not an individual task). The algorithm varies the portion of fast-memory accesses based on the performance modeling to find a load-balance solution.

In summary, we make the following contributions.

- We identify a new performance problem on HM.
- We introduce task semantics during profiling to enable accurate performance prediction on any data placement, and a greedy heuristic algorithm to guide the exploration of various data placements on HM.
- Merchandiser uses an automated workflow with high usability. It largely reduces load imbalance and leads to an average of 17.1% and 15.4% (up to 26.0% and 23.2%) performance improvement, compared with a hardware-based solution (Memory Mode in Optane) and an industry-quality software-based solution (MemoryOptimizer [16]) respectively on Optane-based HM.

2 Background

Heterogeneous memory (HM). We use Intel Optane Persistent Memory (PM) and DRAM as an example of HM in our study. With the emergence of other technologies (such as CXL [11] and high bandwidth memory), HM is becoming a trend. There is performance difference between PM and DRAM. In Optane PM 100 series, the memory latency of sequential and random read on PM is 2.08× and 3.77× longer than on DRAM; the peak memory bandwidth of read and write on PM is 3.87× and 4.74× lower than on DRAM [36, 64].

PM module can be configured as App Direct Mode or Memory Mode. With App Direct Mode, software explicitly controls the placement of memory pages on PM and DRAM. With Memory Mode, DRAM works as a direct-mapped, write-back cache to PM, and is managed by hardware. Merchandiser is a software solution, hence it uses App Direct Mode. Merchandiser performs better than Memory Mode.

Data placement on HM. The existing solutions [2, 15, 33, 34, 41, 60] manipulate page table entries (PTE) for memory profiling to detect hot pages. They repeatedly scan PTEs or intercept page protection faults to check if a specific bit in PTE is changed by hardware. If yes, a memory access is recorded and the bit is reset for future profiling. Using this method can accurately capture memory accesses. However it is slow (taking a couple of seconds to profile hundreds of GB memory), and cannot capture varying workload behaviors. To avoid long profiling time, it is natural to sample pages in the address space for profiling. However, in-discriminating page sampling of all tasks can lead to load imbalance.

(a) MPI-based App. (DMRG)

```
1 Partition Hamiltonian into blocks
2 Each MPI rank get a block
3 Block has its input data (H, PSI)
4 for sweep in sweeps:
5   S1: Construct problem
6   S2: Solve Davidson function
7   S3: Apply SVD to update (H, PSI)
8   Exchange boundary and sync.
```

(b) OpenMP-based App. (SpGEMM)

```
1 for (A*B) in an application:
2   Partition A into bins by rows
3   Each bin has its size and NNZ
4   #pragma omp parallel
5     {T1: Compute NNZ of C
6      Sync point 1
7      T2: Compute values of C
8      Sync point 2}
```

Figure 1. Two examples of task-parallel HPC applications. (NNZ = number of non-zero elements)

Task-parallel HPC applications. We study task-parallel HPC applications [22]. In such an application, multiple tasks run in parallel, and are commonly based on MPI or OpenMP. In an MPI-based application, each MPI process works on a task, and in an OpenMP-based application, each OpenMP thread in a parallel region works on a task There are synchronizations among tasks. Also, a task can be repeatedly executed, and each execution may use different inputs. We refer to each execution of a task as a *task instance*.

Figure 1.a gives an example of MPI-based task-parallel application, DMRG [6, 21, 77]. In DMRG, a Hamiltonian matrix is first partitioned into multiple blocks, each assigned to an MPI process (Lines 1-3). Then each MPI process runs a computation loop, running the DMRG algorithm iteratively using the assigned block (H) and matrix product states (PSI) as input (Lines 5-7). An iteration of the loop is regarded as a task instance. Hence the task in an MPI process is repeatedly executed. Task instances use the same H but different PSI as input. At the end of each iteration, there is a global synchronization among MPI processes.

Figure 1.b gives an example of OpenMP-based task-parallel application, SpGEMM ($C = A * B$) in Ginkgo [5]. In this example, a main loop runs many SpGEMM ($C = A * B$). In each iteration of the main loop, A is first partitioned into bins by rows, and then there is an OpenMP parallel region where each thread accesses B and a bin of A to produce a part of C. In an iteration of the loop, a thread works on a task instance, and in the next iteration this thread works on another but with different A and B. At the end of the OpenMP region, there is an implicit synchronization among threads.

We assume that for a given task, the algorithm or memory access patterns do not change across task instances. For example, in Figure 1.a, in an MPI process, no matter how PSI (the input data) is changed, the algorithm and memory access patterns in Lines 5-7 remain the same. However, if there is a change in the algorithm or memory access patterns across task instances, then those task instances should be classified into different tasks.

3 Overview

Merchandiser, sketched in Figure 2, uses performance modeling to guide data placement in HM. The performance modeling uses task information as input, which includes execution

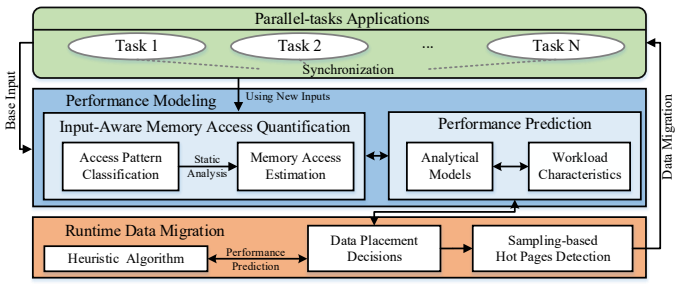

Figure 2. An overview of Merchandiser.

time of basic blocks in the task program and runtime performance events critical to decide the performance sensitivity of the task to data placement. The task information is collected in the first instance of the task using an input problem (called the *base input*), and used by the performance modeling to predict the performance of the same task for a new input under various data placement on HM. The performance modeling is integrated into a runtime system to decide if data migration can introduce load imbalance among tasks.

To accurately predict the execution time of a task with a new input, our performance modeling first estimates the number of memory accesses to data objects with the new input (see Section 4). Merchandiser performs analysis on memory access patterns at the data object level through static analysis; then the estimation is made according to data object sizes, the number of memory accesses collected from the base input, and memory access patterns.

The performance modeling predicts execution time of a task with a new input under various data placement on HM. The prediction uses the estimated number of memory accesses, workload characteristics, and prediction of execution time on homogeneous memories (i.e., PM or DRAM only) (see Section 5). The workload characteristics are selected from performance events based on quantification of each event's contribution to prediction accuracy. The prediction of execution time on homogeneous memories is established on static analysis on input-independent basic blocks and offline profiling.

Merchandiser has a runtime system using the performance modeling to decide if data migration should happen or not with load-balance awareness (see Section 6). Before task execution, the runtime first employs a heuristic algorithm to decide how many fast memory accesses should happen for each task based on the performance modeling. Then, utilizing memory profiling mechanisms in existing solutions, Merchandiser determines if the pages corresponding to each task should be migrated from slow memory to fast memory.

4 Input-Aware Memory Access Quantification

Estimating memory accesses to data objects for a new input problem is challenging. Without using extensive and costly memory profiling, how to capture the caching effect on main memory accesses and how to make the estimation without tightly coupling with architecture details to enable high usability are challenging.

Our solution is highlighted with two fundamental innovations: (1) using limited memory profiling with the base input to direct the estimation, which is useful to simplify the estimation method and improve usability; (2) distinguishing memory access patterns, which is useful to capture the caching effect and improve estimation quality.

Our estimation method has three steps: (1) using a user API to specify data objects for management on HM, (2) classifying memory access patterns of those objects in the task, and (3) estimating the memory access count.

User API. In a task-parallel HPC application, most memory accesses happen to several major data objects. For example, *H* and *PSI* in Figure 1.a (DMRG), and *A*, *B* and *C* in Figure 1.b (SpGEMM). Merchandiser is in charge of placement of data objects specified by the user through an API. We assume that the data object sizes are known right before task execution during runtime (e.g., Line 3 in Figure 1.a and Line 3 in Figure 1.b). This assumption is generally true in many HPC applications [4, 44, 57, 75, 79–81].

Merchandiser expects the user to use the following API to specify data objects for management:

```
void *LB_HM_config(void* objects, int* sizes)
```

where `*objects` points to a list of user-specified data objects to be managed for profiling and migration, and `*sizes` points to a list of their sizes (e.g., the length of PSI array for DMRG). The data object sizes can be variables, and their values are known right before task execution. The API is placed in the program right before task execution. In the example of DMRG and SpGEMM, the API is placed right before Line 5 and Line 4 in Figure 1.a and Figure 1.b respectively. Note that the user does not need any information on which data objects cause load imbalance when using the API. Any data object can be passed to the API.

Classification of memory access patterns. We perform object-level memory access pattern analysis on data objects specified by the user, and classify memory accesses into four patterns. The patterns are depicted with the following code (as the body of a loop), where *i* is a loop induction variable:

- **Stream**: A[i] = B[i] + C[i]
- **Strided**: A[i*stride] = B[i*stride]
- **Stencil**: A[i] = A[i−1] + A[i+1]
- **Random**: A[i] = B[C[i]]

The stream pattern is manifested as stepping through any array in a loop where the index is determined by a loop induction. This pattern also includes the delta pattern (e.g., A[i] = A[i] + d), reduction (e.g., x = x + A[i]), and transpose (e.g., A[i][j] = B[j][i]). The strided pattern is a more general case of the stream pattern, where the stride is a constant known from application knowledge. The stencil pattern involves accessing an array sequentially with a dependency between iterations of the loop, such as 7-point stencil used in Jacobi

Table 1. Access patterns detected in five applications

Applications	SpGEMM	WarpX	BFS	DMRG	NWChem-TC
Access Patterns	Stream Random	Strided Stencil	Stream Random	Stream Strided	Stream Random

and Gauss-Seidel kernels [38]. The random pattern includes pointer chase, gather (such as B in A[i] = B[C[i]]) and scatter (such as A in A[B[i]] = C[i]) using indirect addressing.

We use Spindle [83] to identify these patterns. Spindle is an LLVM-based static analysis tool. It identifies memory access patterns at the data object level by extracting structural information relevant to memory access instructions. In our evaluation (see Tables 1 and 2), these patterns exist in major data objects accounting for at least 98% of memory consumption of the applications.

Estimation of memory access count. Given a task, we measure the number of memory accesses at the data object level during the first execution of the task (using the base input). Then, we estimate the number of memory accesses for subsequent executions of the same task with new inputs, based on the measurement of memory accesses.

To profile (measure) with the base input, we use the following method. Merchandiser profiles memory accesses in DRAM and PM using different methods to avoid large profiling overhead. In PM, Merchandiser uses the profiling method in MemoryOptimizer to identify hot pages, because that profiling method constrains the number of memory pages for profiling to make the profiling overhead small. In DRAM, Merchandiser uses the profiling method in Thermostat [3]. This profiling method chooses one 4 KB page out of each 2 MB page to profile, and scales the number of memory accesses in the 4 KB page to represent the number of memory accesses to the 2 M page. This profiling method is more accurate and can be used to identify cold pages to eliminate out of DRAM. It causes less than 1% decrease in memory access performance when profiling tens of GB of DRAM [3], but causes large profiling overhead for profiling hundreds of GB or TB scales, which prevents it to be applied to PM with large capacity. Both of the profiling methods manipulate PTEs to detect memory accesses, as discussed in Section 2.

To estimate the memory access count for a new input, we use the following method. We assume that the measured number of memory accesses to a data object is $prof\_mem\_acc$ and the data object size is S_{base}. We also assume that the number of memory accesses to be estimated for the new input is $esti\_mem\_acc$ and the data object size is S_{new}. We have Equation 1.

$$esti\_mem\_acc = \frac{S_{new}}{S_{base} \times \alpha} \times prof\_mem\_acc \quad (1)$$

The term $\frac{S_{new}}{S_{base}}$ captures the change of data object size from the base input to the new input. $esti\_mem\_acc$ is in proportion to that change (i.e., the ratio of the new size to the base size). Furthermore, this proportion should factor in the fact that the memory access pattern may have input-dependent behavior and hit a variable number of cache lines, causing distinct memory access counts. This fact is captured by α, a parameter aiming to quantify the memory-access differences across inputs by considering the caching effect.

Calculation of α is challenging. We rely on the classification of memory access patterns and runtime refinement to make it possible. For the ***stream*** and ***strided*** patterns, α is calculated by considering stride length and data type. For example, assuming that the cache line size is 64 bytes and the data type is integer (4 bytes), and assuming that S_{new} and S_{base} are 192 bytes and 128 bytes respectively, then for the streaming pattern, S_{new} will cause 3 memory accesses, and S_{base} will cause 2 memory accesses (i.e., $proc\_mem\_acc = 2$). Hence, $alpha = 1$. For the stream and strided patterns, if S_{new} or S_{base} is not divisible by the cache line size, it is rounded to a slightly larger, divisible size. We enumerate various stride lengths and data types, and then calculate corresponding α offline. These values of α are used in Equation 1 at runtime once S_{new} and S_{base} are known.

For the ***stencil*** pattern, α is calculated according to whether the pattern is input-independent or not. The popular input-independent stencils, such as 5/7/9-point stencils, update elements of data objects solely based upon loop induction variables. For these stencils, α is measured offline. In particular, we run a microbenchmark practicing the stencil pattern on a data object in a loop, and then measure how many main memory accesses are caused by the stencil code using performance counters. We also count how many memory accesses happen at the program level. Then α is the ratio of the program-level measurement to the counter-based measurement. If the stencil is input-dependent, which means the stencil is changed across inputs, then we set α to 1 and rely on a refinement process to improve α during task executions with new inputs. The ***random*** pattern takes the same refinement approach due to its input-dependent feature.

Runtime refinement of α is an iterative process over task instances based on Equation 1 (α is initialized as 1). Given a data object with input-dependent stencil or random pattern, the number of memory accesses to the data object is measured by using performance counters in the sampling mode (e.g., Precise Event-Based Sampling from Intel or Instruction-based Sampling from AMD). This mode allows us to associate memory accesses with specific memory addresses through which we connect with the data object. This performance counter-based measurement happens whenever the task instance is executed. With the measured memory access counts, α is continuously updated along with the execution of task instances, and used to estimate the number of memory accesses for the next task instance.

Handling unknown patterns. Although the four patterns widely exist in HPC applications, if a data object in a task has an unknown memory access pattern, that access pattern is treated as random, and α relies on the refinement process to improve estimation accuracy.

5 Performance Modeling

Modeling application performance under various data placement on HM is non-trivial, because memory accesses are distributed to DRAM and PM, and modeling the impact of such a distribution depends on both workload characteristics and memory performance.

To address the modeling challenge, our performance model introduces two fundamental innovations: (1) bound the performance prediction by the best and worst performances. The two performances are collected on DRAM only and PM only respectively, and implicitly capture the impact of memory architecture on performance (e.g., memory level parallelism); (2) build upon our estimation of memory counts (Section 4) to scale the two performance bounds based on workload characterization. Such performance scaling driven by workload characterization simplifies our efforts to model memory access patterns but significantly improves usability.

Our performance modeling takes the following information as input: (1) the total number of memory accesses with the new input ($esti\_mem\_acc$). This number is an accumulation of estimated numbers of memory accesses across all data objects. (2) The predicted execution time of the task with the new input on DRAM only ($T_{new\_dram\_only}$) and PM only ($T_{new\_pm\_only}$) (see Section 5.2). The performance modeling predicts the execution time of the task on the new input with some pages migrated to DRAM. We use $dram\_acc$ to represent the number of DRAM accesses when running the new input, and the predicted execution time for the new input is T_{new_hybrid}.

Our performance modeling is based on the following rationale: (1) T_{new_hybrid} should be bounded between $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$; (2) for an individual task, more DRAM accesses lead to better performance. Equation 2 predicts T_{new_hybrid} and reflects the above rationale.

$$T_{new\_hybrid} = T_{new\_pm\_only} \times (1 - r_{dram\_acc}) \times f(PMCs, r_{dram\_acc}) + T_{new\_dram\_only} \times r_{dram\_acc} \quad (2)$$

In Equation 2, *PMCs* are Performance Monitor Counters; $r_{dram\_acc} = \frac{dram\_acc}{esti\_mem\_acc}$. The term $(1 - r_{dram\_acc})$ reflects the rationale (2) and models the correlation between $T_{new\_pm\_only}$ and T_{new_hybrid}. When all memory accesses happen in DRAM (i.e., $dram\_acc = esti\_mem\_acc$) or PM (i.e., $dram\_acc = 0$), the performance becomes $T_{new\_dram\_only}$ or $T_{new\_pm\_only}$.

Note that the term $(1 - r_{dram\_acc})$ alone is not sufficient to capture the correlation between T_{new_hybrid} and $T_{new\_pm\_only}$. When more memory accesses happen in DRAM, the improved performance does not have a simple linear relationship to the number of DRAM accesses. When more memory accesses happen in DRAM, the instruction pipelining is able to run faster, which impacts both memory parallelism and instruction scheduling order. To verify the above conclusion, we run tensor contraction sequences of NWChem (referred as NWChem-TC), an ab initio computational chemistry software package, with a representative input problem listed in Table 2. We measure the performance of all the five execution phases of NWChem-TC and change the ratio of DRAM accesses to the total number of memory accesses. Figure 3 shows the performance normalized to that of using PM only. When a half of the total memory accesses is moved from PM to DRAM, the execution time of Writeback and Input Process (two execution phases in NWChem-TC) is reduced by 47.5% and 26.2% respectively.

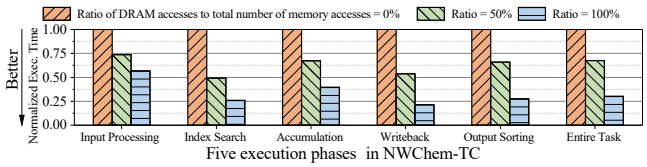

Figure 3. Performance variance when we change the ratio of DRAM accesses to the total number of memory accesses.

To model the correlation between T_{new_hybrid} and $T_{new\_pm\_only}$ better, we introduce a correlation function $f(\cdot)$ in Equation 2. We discuss how $f(\cdot)$ is built in Section 5.1, and how to estimate execution time on homogeneous memory (i.e., $T_{new\_dram\_only}$ and $T_{new\_pm\_only}$) in Section 5.2.

5.1 Construction of Correlation Function

We build the correlation function based on the principle that the correlation function should include workload characteristics that indicate how sensitive the application is to data placement on HM. The correlation function takes workload characteristics and the ratio of DRAM accesses to the total number of main memory accesses (r_{dram_acc}) as input.

The correlation function in Merchandiser is a statistical model. We do not use analytical modeling because of the following reasons. First, analytical modeling has difficulty to capture the overlap between memory accesses and computation. Such an overlap impacts the performance sensitivity of the application to memory latency and bandwidth, and hence should be modeled. Although existing efforts use analytical modeling to model the overlap [24, 28, 30, 78], they are built upon detailed architecture information (e.g., data distribution between memory banks) and strong supports from compilers (e.g., quantifying instruction level parallelism), which limits their feasibility. Second, the complexity of analytical modeling can cause large runtime overhead.

We study statistical models listed in Table 3 in Section 7. We use all performance events collectable from performance counters as the workload characteristics. These events are used as the model input (attributes). Then we train the models with calculated target values of $f(\cdot)$. We choose the Gradient Boosted Regressor (GBR) as the final correlation function, because it leads to the highest modeling accuracy among the statistical models we studied. In the following, we discuss how we train the models and generate training data.

Training data generation. Our training data includes thousands of training samples. Each training sample is a pair consisting of task characteristics (*PMCs* in Equation 2) and a specific r_{dram_acc}, and a calculated value of $f(\cdot)$.

To generate a training sample, the target value of $f(\cdot)$ is calculated according to Equation 2 by running a code sample. Particularly, we run a code sample on PM only and DRAM only with the same random input, and measure performance (used as $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$ in Equation 2). We change the allocation of data objects on DRAM to generate 10 different data placements. Each data placement is applied to the code sample with the same input, and we measure r_{dram_acc} and T_{new_hybrid}. Substituting $T_{new\_pm\_only}$, $T_{new\_dram\_only}$, T_{new_hybrid}, and r_{dram_acc} by the measured values in Equation 2, we get the value of $f(\cdot)$.

To generate code samples, we use CERE [10]. CERE is an LLVM-based compiler tool that can automatically extract code regions (a.k.a., loops) out of a program. We use the NAS parallel benchmarks [7] and SPEC 2006 FP benchmarks [26] to extract 281 code regions as code samples.

Each training sample must include *PMCs* as representative of workload characteristics. Collecting *PMCs* and generating the training sample use the same code, but different inputs. The reason we use different inputs is that in the performance model (Equation 2), the workload characteristics is collected using the base input, but the predicted performance is for a new input different from the base input. We name the input used for collecting *PMCs*, *seed input*.

Selection of workload characteristics. When selecting statistical models, we use all collectable hardware events as the workload characteristics, which ensure that the model selection is not impacted by the selection of workload characteristics. Once a model is selected as the correlation function, we reduce those hardware events as the workload characteristics, because of the following reasons. First, some of those events are conflicting and cannot be collected as the same time. This means we have to run the tasks multiple times to collect all of those events, which limits the usability of our performance modeling. Second, using all of events to construct the model can cause larger runtime overhead and demand more training data for high modeling accuracy.

We use the following method to select hardware events. We first train the model using all events (or features), and then removes a hardware event which is the least important to the model accuracy. We quantify the importance of hardware events using a metric, the Gini importance [52], because of its strong differentiability. After removing a hardware event, we re-train the model and then remove the least important feature again. We continue the process until the model accuracy after removing the least important features is worse than the second best model.

We choose 8 events to represent workload characteristics: `LLC_MPKI`, `IPC`, `PRF_Miss`, `MEM_WCY`, `L2_LD_Miss`, `BR_MSP`, `VEC_INS`, and `L3_LD_Miss` (listed in a decreasing order of importance). `LLC_MPKI`, `IPC`, and `PRF_Miss` are the most important events. `LLC_MPKI` represents the last level cache misses per kilo instructions, which indicates how often the task fetches data from memory; `IPC` is the average number of instructions executed per clock cycle, which indicates whether the task is compute or memory bound; `PRF_Miss` is the ratio of data prefetches that cause misses to total number of data prefetches, indicating whether data prefetching is successful and whether memory access patterns are highly irregular. In general, the three events are highly discriminatory and can indicate the sensitivity of application performance to data placement, hence can be used to build a robust model.

5.2 Performance Prediction on Homogeneous Memory

Many efforts predict execution time of an application with various inputs [29, 47, 55, 56, 65, 73, 76]. They are based on the assumption that there is no change of workload characteristics (e.g., memory access patterns and control flow) across inputs. We use the same assumption, and use the work [55] to predict $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$.

We use the method in [55] to identify input-independent basic blocks and measure their execution times offline on PM and DRAM. Beyond the work [55], at runtime, Merchandiser counts how many times each basic block in a task is executed using the base input. Then, Merchandiser computes the similarity between the base input and new input, based on the sizes of input data objects. In particular, given an input including one or multiple data objects, we build a vector and each element of the vector represents the size of an input data object. We quantify the similarity between the base input and new input by calculating the cosine similarity [17] of the two vectors. We use the value of cosine similarity to scale the number of times the basic block is executed using the base input. The scaling result is an estimation of the number of times the basic block is executed using the new input. Our method does not cause large runtime overhead, because it only needs to calculate cosine similarity to predict $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$ for a new (unseen) input problem.

5.3 Putting All Together

In summary, the following workflow happens automatically for the user.

Offline model construction and code analysis.

1. **Offline construction of the scaling function $f(\cdot)$.** This includes generating training dataset using code samples and collecting features of the code samples (i.e., workload characteristics) using some seed inputs. *The construction of $f(\cdot)$ happens only once.*
2. **Preparation for online prediction of $T_{new\_dram\_only}$ and $T_{new\_pm\_only}$.** This includes measuring the execution time of basic blocks on DRAM and PM. *For an application, this step happens only once.*

3. **Offline application code analysis.** This is based on compiler analysis, including getting input-independent basic blocks, getting memory access patterns for online estimation of total number of memory accesses (see Section 4). *For an application, the offline code analysis happens only once.*
4. **Offline calculation of α for input-independent stencil or random access patterns.**

Online profiling and performance prediction.

1. **Online collection of task information using the base input.** This includes collecting workload characteristics of the task using the 8 performance events and counting how many times basic blocks are executed using the base input.
2. **Online performance prediction for the task with a new input.** This happens right before task execution, and includes: (a) estimating the number of memory accesses to data objects; (b) predicting $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$. The runtime system uses the performance modeling to predict task performance with the new inputs.

User feasibility. Merchandiser takes user feasibility into consideration. All steps are automated. Offline steps 1 and 4 are constructed only once and can be used for any application. Offline steps 2-3 happen only once for a given application. Online steps 1-2 are application- and input-dependent, but are automated. The user only needs to insert the API into the application without changing application code.

Extensibility. Merchandiser can be easily extended to other HM systems. Three steps are needed: (1) the training data is collected to reflect the performance sensitivity of the application to different memories; (2) the scaling function is re-constructed (13 minutes in this work) and the most critical performance events are selected; (3) measure the performance of basic blocks in new memory systems.

Limitation Merchandiser requires the application source code, because it expects the user to insert an API and compile the code with Spindle for static analysis to identify memory access patterns. Requiring the application source code is a limitation of Merchandiser. When the source code is not available, we can use a dynamic binary instrumentation tool (e.g., [37, 53]) to insert the API, intercept memory allocation, and generate instruction traces. Then, we use a tool (e.g., [58, 59]) to identify memory access patterns of the traces.

6 Load Balance-Aware Data Migration

Data migration is based on the performance modeling with the awareness of load balance.

Usage of performance modeling. Using the performance modeling, we decide how many memory accesses in each task should be on DRAM to enable load balance and high performance. The decision process can be formulated as a knapsack problem, if we regard the DRAM capacity as the limit of knapsack weight, each page as an item in the knapsack, the performance benefit after placing the item (a page) on DRAM as the item value, and the page size as the item weight. Since the knapsack problem is NP-hard, deciding how many memory accesses in each task should be on DRAM is NP-hard. We introduce a greedy heuristic algorithm to address the problem.

The basic idea of the algorithm is as follows. Tasks take many rounds to gradually increase DRAM accesses (by page migration) and improve performance until the DRAM space is exhausted. In each round, the task with the longest execution time increases its DRAM accesses until it is shorter than the second longest task.

We depict the algorithm in detail in Algorithm 1. The algorithm takes as input the information needed for the performance modeling, and outputs the number of DRAM accesses for each task. The algorithm tracks DRAM allocation to each task and initializes it with 0 (Line 6). After initialization (Lines 7 and 8), the algorithm iteratively finds the longest task (Line 10) and the second longest task (Line 11), and improves the performance of the longest task by increasing its DRAM accesses (Lines 14). For the longest task, the algorithm iteratively increases its DRAM accesses (Lines 13-16). In each iteration, the number of DRAM accesses is increased by 5% (Line 14), and then the algorithm uses the performance modeling to predict the performance (Line 15). The increase of DRAM accesses stops when the predicted performance is no longer than the execution time of the second longest task (Line 16). The increase of DRAM accesses of a task is implemented by migrating its pages to DRAM. The algorithm assumes that the memory accesses are evenly distributed to memory pages of the task, and when the DRAM accesses are increased by 5%, the number of DRAM pages is also increased by 5% (Lines 18). Based on the above assumption, the algorithm tracks DRAM allocation to make sure the total number of memory pages migrated to DRAM does not violate the DRAM capacity (Line 19).

Page migration. We extend the page migration strategy in the existing solution (particularly MemoryOptimizer) with the awareness of load balance. The existing solution uses sampling-based memory profiling, identifies the most accessed PM pages in a time interval, and then migrates them to DRAM. Merchandiser extends the existing solution by checking whether the tasks that access the to-be-migrated pages have reached the goal of DRAM accesses decided by Algorithm 1, before page migration happens. If yes, then the corresponding pages will not be migrated.

DRAM space management. When page migration from PM to DRAM is about to happen but DRAM does not space, the least frequently accessed pages in DRAM are migrated to PM. Merchandiser determines these pages based on the profiling method discussed in Section 4.

Algorithm 1 Runtime algorithm for Data Migration

Input: Execution time of each task using PM-only configuration $\{D\_1, \ldots, D\_n\}$
Input: Measured hardware events of each task using PM-only configuration $\{PCs\_1, \ldots, PCs\_n\}$
Input: Total accesses of each task $\{Total\_Acc\_1, \ldots, Total\_Acc\_n\}$
Input: Total DRAM capacity DC.
Input: Performance model $Model$.
DRAM allocation used for each task $\{DC\_1, \ldots, DC\_n\} \leftarrow \{0, \ldots, 0\}$
DRAM accesses required $\{DRAM\_Acc\_1, \ldots, DRAM\_Acc\_n\} \leftarrow \{0, \ldots, 0\}$
New exec. time $\{D'\_1, \ldots, D'\_n\} \leftarrow \{D\_1, \ldots, D\_n\}$
Repeat:
  Task $i \leftarrow$ Tasks with the longest execution time found from $\{D'\_1, \ldots, D'\_n\}$
  $Second\_T \leftarrow$ The second longest execution time from $\{D'\_1, \ldots, D'\_n\}$
  Temporary accesses $DRAM\_Acc \leftarrow DRAM\_Acc\_i$
  Repeat:
    $DRAM\_Acc \leftarrow DRAM\_Acc + 5\% * Total\_Acc\_i$
    Predict improvements $D'\_i \leftarrow Model(D\_i, PCs\_i, DRAM\_Acc)$
  Until: $D'\_i \leq Second\_T$
  $DRAM\_Acc\_i \leftarrow DRAM\_Acc$
  $DC\_i \leftarrow MAP\_TO\_PAGES(DRAM\_Acc\_i)$
Until: $\sum_{i=1,\ldots,n} DC\_i$ less than DC
Return: $\{DRAM\_Acc\_1, \ldots, DRAM\_Acc\_n\}$

7 Evaluation

Platform and Libraries We evaluate Merchandiser on a two-socket server with two Intel Xeon Gold 6252N 24-core processors running Linux 5.17.0. Each socket has 12 DIMM slots: six for 16-GB DDR4 DRAM modules, and six for 128-GB Optane PMM. In total, the system has 192 GB DRAM and 1.5 TB PM. We use Memkind [9] to manage the page placement and migration on HM.

Applications and inputs. We use five task-parallel HPC applications listed in Table 2. SpGEMM [5] and BFS [12] are derived from high-performance math libraries. WarpX [44, 80] is a production code for plasma simulation. DMRG [6, 77] comes from Itensor [21] and simulates quantum many-body systems. NWChem-TC [79] is the tensor contraction component in NWChem [40].

Implementation and Comparison. The modifications to the applications are small (less than 10 lines in each applications). We compare Merchandiser with four solutions:

- A hardware-based solution: Memory Mode.
- A software-based solution: Intel MemOptimizer [16].
- Sparta [50] (the only application-specific solution for sparse tensors or matrices on HM) for SpGEMM.
- WarpX-PM [68] for WarpX.

Table 2. Applications and their inputs. LOC = Lines of code.

Application	LOC	Problem and Input Size	Memory Consumption	Configuration
SpGEMM (General Sparse Matrix-Matrix Multiplication)	$2.21e^3$	$A * A^T$ using matrix GAP-kron with 4.22E+9 nonzero elements	429.3 GB	MPI processes: 1 OpenMP threads: 12
WarpX (ECP-WarpX)	$6.78e^4$	Beam–plasma simulation with the scale of 1024*1024*2048	1.056 TB	MPI processes: 1 OpenMP threads: 24
BFS (Breadth-first search)	$1.95e^3$	com-Orkut with 3.07E+6 vertices and 1.17E+8 edges	731.9 GB	MPI processes: 1 OpenMP threads: 12
DMRG (density-matrix renormalization group)	$8.79e^4$	Hubbard 2D model with Nx = 320 and Ny = 320	1.271 TB	MPI processes: 6 OpenMP threads: 2
NWChem-TC (Tensor Contraction)	$7.36e^5$	Cytosine tensor with dims of 400*400*58*58	308.1 GB	MPI processes: 1 OpenMP threads: 24

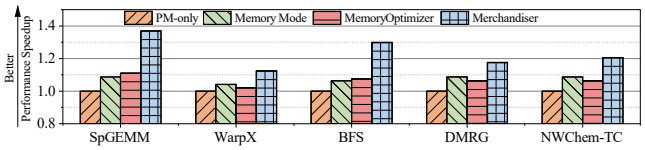

Figure 4. Performance of Memory Mode, MemoryOptimizer, and Merchandiser, compared to the PM-only execution.

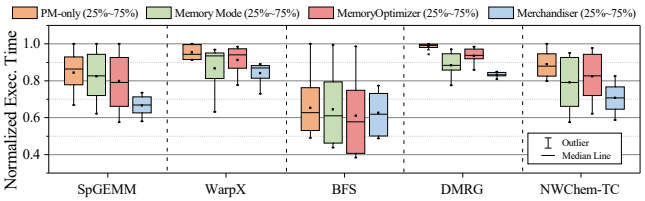

Figure 5. Task execution time and their variance.

The last two solutions use application knowledge on memory access patterns and lifetime of data objects to guide data placement, hence they are application-specific.

7.1 Overall Performance

Figure 4 shows overall performance normalized to PM-only. Merchandiser introduces 23.6%, 17.1%, and 15.4% performance improvement on average (up to 37.8%, 26.0%, and 23.2%) over PM-only, Memory Mode, and MemoryOptimizer respectively. We have the following 4 observations: (1) By introducing task semantics, Merchandiser provides larger performance improvement than task-agnostic page management solutions (Memory Mode and MemoryOptimizer). (2) Compared to Memory Mode, Merchandiser brings larger benefits to SpGEMM, BFS, and NWChem-TC, because they involve sparse matrix/graph/tensor computation and random access patterns, which have bad locality in the hardware-managed cache. (3) Compared to MemoryOptimizer, Merchandiser brings larger benefits to WarpX and DMRG. These applications have regular memory access patterns. Our performance modeling works better to guide data placement for such applications. (4) For applications with large load imbalance, such as BFS and NWChem-TC, the number of pages being migrated among tasks can vary by up to 21.4×.

Compared with the two application-specific solutions (Sparta and WarpX-PM), Merchandiser achieves 17.3% improvement and 4.6% degradation over Sparta and WarpX-PM respectively. The reason for better performance on SpGEMM is that Sparta ignores the load balancing caused by multiple matrix multiplications. WarpX-PM relies on manual analysis of the lifetime of data objects using program semantics, which provides better guidance on data placement. Although Merchandiser performs worse than the manual approach WarpX-PM, the two performances are very close.

7.2 Performance Analysis and Overhead

Quantifying load balance. We use the average coefficient of variation (A.C.V) [1] of execution time across tasks, which is a statistical metric to quantify variability: a smaller value

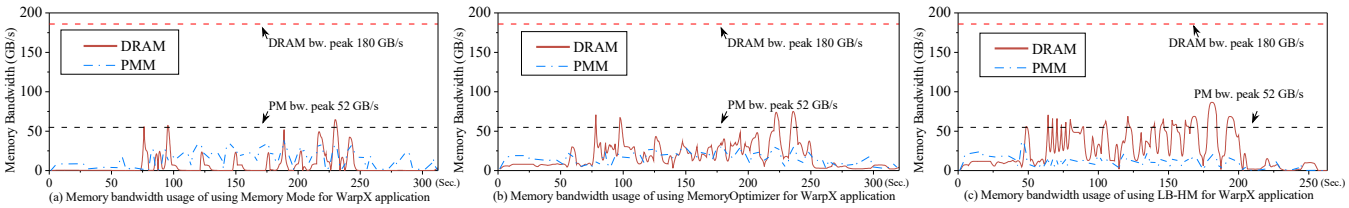

Figure 6. Memory bandwidth consumption during the execution of WarpX.

Table 3. Statistical models, parameters, and accuracy.

Model	Parameter	R^2 (Accuracy)
DTR (Decision Tree Regressor)	criterion=gini, max_depth=10	78.1%
SVR (Support Vector Regressor)	kernel='rbf', degree=5	83.6%
KNR (K-Neighbors Regressor)	n_neighbors=8	72.9%
RFR (Random Forest Regressor)	n_estimators=20, max_depth=10	89.2%
GBR (Gradient Boosted Regressor)	base_estimator='DTR'	94.1%
ANN (MLP Regressor)	alpha=$1e^{-6}$, hidden_layer=(200, 20)	93.2%

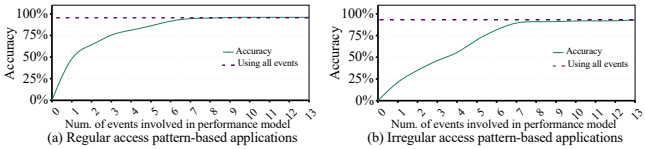

Figure 7. Accuracy of the correlation function using different amounts of performance events as input.

of A.C.V means less difference in execution time among tasks, thus the task executions are more balanced.

Figure 5 shows the results based on the boxplot. In the figure, the box displays the interquartile range (25%-75%) with the median, and the whiskers represents the outliers. Wider box and longer whiskers indicate larger performance variance and worse load balance among tasks. Compared with Memory Mode and MemoryOptimizer, Merchandiser reduces A.C.V by 51.6% and 42.7% on average respectively.

Load imbalance could be caused by the applications themselves, such as the different distributions of non-zero elements of each matrix in SpGEMM, the uneven graph partitioning approach in BFS, and the inequable tensors with different memory access patterns in NWChen-TC. On the other hands, WarpX and DMRG do not have such load imbalance caused by themselves. The performance of PM only in Figure 5 shows the load imbalance from the applications themselves. We notice that using Merchandiser, A.C.V is reduced by 39.1% and 21.4% for SpGEMM and BFS, compared with using PM-only. This indicates that Merchandiser can even remove load imbalance in applications themselves.

Study DRAM utilization. We use Intel Performance Counter Monitor [35] to measure memory bandwidth. Figure 6 shows runtime bandwidth in WarpX. Compared with Memory Mode, Merchandiser increases average DRAM bandwidth usage from 5.98 GB/s to 24.31 GB/s, indicating the usage of fast memory is improved; Meanwhile, the average PM bandwidth usage is reduced from 13.74 GB/s to 9.97 GB/s, indicating the effectiveness of page migration in Merchandiser. Furthermore, MemoryOptimizer and Merchandiser perform similarly in terms of memory bandwidth usage. But Merchandiser outperforms MemoryOptimizer because of the reduction of load imbalance (shown in Figure 4.)

Runtime overhead of Merchandiser. Multiple components contributes to the runtime overhead - (1) online refinement of α, (2) online collection of task information using the base input, and (3) online performance prediction for the task with new inputs. (1) and (2) only need to use performance counters, which is lightweight (less than 0.1% of performance loss in our evaluation). (3) uses a lightweight performance modeling (Equations 1-2), which takes 0.031 ms in our system. Compared with long execution time of tasks (i.e., at least a few seconds), this overhead is small.

7.3 Performance Modeling Analysis

Construction of correlation function. We use Python scikit-learn package to train six statistical models (listed in Table 3) to select the correlation function. 70% of data generated from Section 5.1 is used for training, and the remaining 30% for testing. We use the average of the squares of the residuals, R-squared (R^2), as the metric to measure how far the prediction deviates from the measurement. R^2 ranges from 0.0 to 1.0, where 1.0 means the prediction is exactly the same as the measurement. Table 3 lists model parameters and accuracy using all the performance events. The Gradient Boosted Regressor (GBR) has the highest modeling accuracy.

Values of α. Different tasks have distinct values of α. The average values of α are: 1.9, 4.3, 2.4, 5.7, and 2.6 for SpGEMM, WarpX, BFS, DMRG, and NWChem-TC, respectively.

Selection of workload characteristics. We evaluate the impact of selecting performance events on the scaling function ($f(.)$). We use different performance events as the input to $f(.)$. Using Evaluation 2, measured T_{new_hybrid}, $T_{new\_pm\_only}$ and $T_{new\_dram\_only}$, we calculate the value of $f(.)$ as the golden output. We compare the function output and golden output to quantify the accuracy of $f(.)$.

Figure 7 shows the average accuracy of $f(.)$ when using different numbers of performance events as the input to $f(.)$. The figure shows that using the top 8 events, the model accuracy is 93.7% and 93.2% for regular- (i.e., WarpX and DMRG) and irregular- applications (i.e., SpGEMM, BFS, and NWChem-TC) respectively, which is close to the accuracy of using all events (94.8% and 94.1%). Hence we choose top 8 events (listed in Section 5.1) as $f(.)$'s input.

Evaluating performance modeling accuracy. Table 4 shows the average prediction accuracy over all task instances in each application. Overall, the accuracy is at least 71.3%. For comparison, we evaluate another performance model [8].

Table 4. Accuracy of the whole performance modeling.

Application	Profiling-based regression	Performance model
SpGEMM	37.4%	74.2%
WarpX	75.1%	87.4%
BFS	38.6%	71.3%
DMRG	83.9%	89.2%
NWChem-TC	62.5%	83.0%

This model uses the data-object-size difference between the base and new inputs to scale the performance of the base input to predict the performance of the new input. Our performance modeling outperforms by 12.3%-36.8%.

8 Related Work

Data management on HM has attracted many research efforts. They can be classified into two classes: application-agnostic solutions and application-specific solutions. Application-agnostic solutions [3, 27, 39, 49, 70, 71, 89, 89, 91–93] use system-level profiling to track page hotness and migrate pages accordingly without using application knowledge. Those solutions do not modify applications but miss performance improvement opportunities embedded in application knowledge. Application-specific solutions [13, 20, 23, 43, 45, 48, 51, 67, 69, 72, 82, 85, 87, 94] rely on algorithm knowledge and program semantics to decide when to trigger data migration and migrate which data. Those solutions make the best use of fast memory and can outperform application-agnostic solutions. Different from existing efforts, Merchandiser uses task semantics.

Load balance in HPC parallel applications. There are existing efforts for load balance for OpenMP applications [18, 19, 62, 88, 90, 94]. They use loop scheduling algorithms to dynamically assign iterations to different threads. Chameleon [42] presents the first conceptual generalization of load balancing to arbitrary MPI applications. In [14, 62, 66, 74], a data-driven approach was introduced to balance loads on I/O servers for HPC applications. However, no work considers the load balance problem caused by data placement in HM as us.

9 Conclusions

This paper finds a new performance problem on HM: load imbalance in task-parallel HPC applications because of inappropriate placement of data objects in HM. This problem is due to the ignorance of the association between memory accesses and specific tasks and an incorrect assumption that bringing hot pages to fast memory always leads to better performance. We address this problem by introducing task semantics during memory profiling and migration. Also using performance modeling to guide data migration, we outperform the existing software- and hardware-based solutions by 17.1% and 15.4% respectively.

Acknowledgement. This work was partially supported by U.S. National Science Foundation (CCF-2217086 and OAC-2104116), and the Chameleon Cloud. We thank all the reviewers for their constructive comments.

10 Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We conduct all experiments on a two-socket server with two Intel Xeon Gold 6252N 24-core processors running Linux 5.17.0. Each socket has 12 DIMM slots, six for 16-GB DDR4 DRAM modules, and six for 128-GB Optane PMM. In total, the system has 192 GB DRAM and 1.5 TB PM. We configure the Optane PMM to App Direct Mode for maximum control.

We evaluate this work Merchandiser on this heterogeneous memory system for five common task-parallel HPC applications. SpGEMM and BFS are derived from high-performance scalable math libraries. They are the basis for solving sparse systems of equations. WarpX is a production code for plasma simulation. DMRG comes from Itensor and simulates the low-energy physics of quantum many-body systems. NWChem-TC is the tensor contraction component in NWChem to model complex chemical and materials processes.

10.1 ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are maintained in a public repository or are available under an OSI-approved license.

List of URLs and/or DOIs where software artifacts are available: https://doi.org/10.5281/zenodo.7453555

Hardware Artifact Availability: There is no author-created hardware artifact;

Data Artifact Availability: There is no author-created data artifact;

10.2 BASELINE EXPERIMENTAL SETUP

Relevant hardware details: Intel Optane 256GB persistent memory module;

Operating systems and versions: Ubuntu 20.04 running Linux kernel 5.17.0-206-generic;

Compilers and versions: GCC v10.3.0;

Applications: SpGEMM, BFS, WarpX, DMRG, and NWChem-TC;

Libraries: LLVM-based tool (Spindle), ML-based algorithm (GBR model), PAPI v5.6.0, ipmctl - Intel Persistent Memory Control;

Key algorithms: Dynamic programming and greedy heuristic algorithm;

Baseline implementation and version: Intel MemOptimizer (commit: 3832466) https://github.com/intel/memory-optimizer

References

[1] Hervé Abdi. 2010. Coefficient of variation. *Encyclopedia of research design* 1 (2010), 169–171.

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[3] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644.

[4] Francis Alexander, Ann Almgren, John Bell, Amitava Bhattacharjee, Jacqueline Chen, Phil Colella, David Daniel, Jack DeSlippe, Lori Diachin, Erik Draeger, et al. 2020. Exascale applications: skin in the game. *Philosophical Transactions of the Royal Society A* 378, 2166 (2020), 20190056.

[5] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S Quintana-Ortí. 2020. Ginkgo: A modern linear operator algebra framework for high performance computing. *arXiv preprint arXiv:2006.16852* (2020).

[6] Alberto Baiardi. 2021. Electron Dynamics with the Time-Dependent Density Matrix Renormalization Group. *Journal of Chemical Theory and Computation* (2021).

[7] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[8] Bradley J Barnes, Barry Rountree, David K Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd annual international conference on Supercomputing*. 368–377.

[9] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[10] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 1 (2015), 1–24.

[11] C.Consortium. [n.d.]. ComputeExpressLink. https://www.computeexpresslink.org

[12] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.

[13] Yu Chen, Ivy B Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. Atmem: adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 293–304.

[14] Minh Thanh Chung, Josef Weidendorfer, Philipp Samfass, Karl Fuerlinger, and Dieter Kranzlmüller. 2020. Scheduling across Multiple Applications using Task-Based Programming Models. In *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*. IEEE, 1–8.

[15] J. Corbe. [n.d.]. AutoNUMA: the Other Approach to NUMA Scheduling. http://lwn.net/Articles/488709.

[16] Intel Corporation. 2021. MemoryOptimizer – hot page accounting and migration daemon. https://github.com/intel/memory-optimizer.

[17] Najim Dehak, Reda Dehak, James R Glass, Douglas A Reynolds, Patrick Kenny, et al. 2010. Cosine similarity scoring without score normalization techniques.. In *Odyssey*. 15.

[18] Bang Di, Daokun Hu, Zhen Xie, Jianhua Sun, Hao Chen, Jinkui Ren, and Dong Li. 2021. TLB-pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–23.

[19] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[20] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.

[21] Matthew Fishman, Steven R White, and E Miles Stoudenmire. 2020. The ITensor software library for tensor network calculations. *arXiv preprint arXiv:2007.14822* (2020).

[22] Marta Garcia-Gasulla, Guillaume Houzeaux, Roger Ferrer, Antoni Artigues, Victor López, Jesús Labarta, and Mariano Vázquez. 2019. MPI+X: task-based parallelisation and dynamic load balance of finite element assembly. *International Journal of Computational Fluid Dynamics* 33, 3 (2019), 115–136.

[23] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2019. Single machine graph analytics on massive datasets using intel optane DC persistent memory. *arXiv preprint arXiv:1904.07162* (2019).

[24] Nagendra Gulur, Mahesh Mehendale, Raman Manikantan, and Ramaswamy Govindarajan. 2014. ANATOMY: An Analytical Model of Memory System Performance. In *International Conference on Measurement and Modeling of Computer Systems*.

[25] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. 2018. Reliability-aware data placement for heterogeneous memory architecture. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 583–595.

[26] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[27] Takahiro Hirofuchi and Ryousei Takano. 2016. RAMinate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 112–125.

[28] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[29] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting execution time of computer programs using sparse polynomial regression. *Advances in neural information processing systems* 23 (2010), 883–891.

[30] Yingchao Huang and Dong Li. 2017. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *IEEE International Conference on Cluster Computing*.

[31] Amazon Inc. 2018. Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfectfor-sap-hana/.

[32] Intel. [n.d.]. *Intel Optane™ Persistent Memory 200 Series Brief*. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html

[33] Intel. 2019. Intel Memory Optimizer. https://github.com/intel/memory-optimizer.

[34] Intel. 2021. Intel Memory Tiering. https://lwn.net/Articles/802544/.

[35] Intel. 2021. Processor Counter Monitor (PCM). https://github.com/opcm/pcm.

[36] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[37] Tomislav Janjusic, Christos Kartsaklis, and Wang Dali. 2014. Scalability analysis of gleipnir: A memory tracing and profiling tool, on titan. *Cray User Group* (2014).

[38] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. 2005. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 workshop on Memory system performance*. 36–43.

[39] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 521–534.

[40] Ricky A. Kendall, Edoardo Aprà, David E. Bernholdt, Eric J. Bylaska, Michel Dupuis, George I. Fann, Robert J. Harrison, Jialin Ju, Jeffrey A. Nichols, Jarek Nieplocha, T. P. Straatsma, Theresa L. Windus, and Adrian T. Wong. 2000. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Physics Communications* 128, 1-2 (June 2000), 260–283. https://doi.org/10.1016/S0010-4655(00)00065-5

[41] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[42] Jannis Klinkenberg, Philipp Samfass, Michael Bader, Christian Terboven, and Matthias S Müller. 2020. Chameleon: reactive load balancing for hybrid MPI+ OpenMP task-parallel applications. *J. Parallel and Distrib. Comput.* 138 (2020), 55–64.

[43] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 335–349.

[44] Lawrence Berkeley National Laboratory. 2021. WarpX. https://github.com/ECP-WarpX/WarpX.

[45] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[46] Ryan Levy, Edgar Solomonik, and Bryan K Clark. 2020. Distributed-memory DMRG via sparse and dense parallel tensor contractions. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[47] Yan Ling, Fang Liu, Yue Qiu, and Jiajie Zhao. 2016. Prediction of total execution time for MapReduce applications. In *2016 Sixth International Conference on Information Science and Technology (ICIST)*. IEEE, 341–345.

[48] Jiawen Liu, Dong Li, and Jiajia Li. 2021. Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory. In *International Conference on Supercomputing (ICS)*.

[49] Jie Liu, Jiawen Liu, Zhen Xie, and Dong Li. 2020. FLAME: A Self-Adaptive Auto-labeling System for Heterogeneous Mobile Processors. *arXiv preprint arXiv:2003.01762* (2020).

[50] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 318–333.

[51] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory. In *Principles and Practice of Parallel Programming*.

[52] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. 2013. Understanding variable importances in forests of randomized trees. *Advances in neural information processing systems* 26 (2013).

[53] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally A Mckee, Bronis R De Supinski, and Andy Yoo. 2007. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 2 (2007), 12–es.

[54] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 126–136.

[55] Thierry Monteil. 2013. Coupling profile and historical methods to predict execution time of parallel applications. *Parallel and Cloud Computing* 2, 3 (2013), pp–81.

[56] Farrukh Nadeem and Thomas Fahringer. 2009. Using templates to predict execution time of scientific workflow applications in the grid. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 316–323.

[57] Sai Narasimhamurthy, Nikita Danilov, Sining Wu, Ganesan Umanesan, Stefano Markidis, Sergio Rivas-Gomez, Ivy Bo Peng, Erwin Laure, Dirk Pleiter, and Shaun De Witt. 2019. Sage: percipient storage for exascale data centric computing. *Parallel computing* 83 (2019), 22–33.

[58] S Arash Ostadzadeh, Roel J Meeuws, Carlo Galuzzi, and Koen Bertels. 2010. Quad–a memory access pattern analyser. In *International Symposium on Applied Reconfigurable Computing*. Springer, 269–281.

[59] Eunjung Park, Christos Kartsaklis, Tomislav Janjusic, and John Cavazos. 2014. Trace-driven memory access pattern recognition in computational kernels. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*. 25–32.

[60] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality.

[61] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*.

[62] Arnab K Paul, Arpit Goyal, Feiyi Wang, Sarp Oral, Ali R Butt, Michael J Brim, and Sangeetha B Srinivasa. 2017. I/o load balancing for big data hpc applications. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 233–242.

[63] Ivy Peng, Kai Wu, Jie Ren, Maya Gokhale, and Dong Li. 2020. Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems. In *IEEE International Parallel and Distributed Processing Symposium*.

[64] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM. https://doi.org/10.1145/3357526.3357568

[65] Thanh-Phuong Pham, Juan J Durillo, and Thomas Fahringer. 2017. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing* 8, 1 (2017), 256–268.

[66] Eric Raut, Jie Meng, Mauricio Araya-Polo, and Barbara Chapman. 2020. Evaluating Performance of OpenMP Tasks in a Seismic Stencil Application. In *International Workshop on OpenMP*. Springer, 67–81.

[67] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy. In *International Conference on Supercomputing (ICS)*.

[68] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*. 203–214.

[69] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2020. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[70] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840* (2021).

[71] Jie Ren, Kai Wu, and Dong Li. 2020. Exploring non-volatility of non-volatile memory for high performance computing under failures. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 237–247.

[72] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[73] Seyed Masoud Sadjadi, Shu Shimizu, Javier Figueroa, Raju Rangaswami, Javier Delgado, Hector Duran, and Xabriel J Collazo-Mojica. 2008. A modeling approach for estimating execution time of long-running scientific applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.

[74] Philipp Samfass, Tobias Weinzierl, Dominic E Charrier, and Michael Bader. 2020. Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement. *Concurrency and Computation: Practice and Experience* 32, 24 (2020), e5916.

[75] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. 2015. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro* 35, 4 (2015), 26–36.

[76] Sarah Shah, Yasaman Amannejad, Diwakar Krishnamurthy, and Mea Wang. 2019. Quick Execution Time Predictions for Spark Applications. In *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE, 1–9.

[77] Samantha Sherman and Tamara G Kolda. 2020. Estimating higher-order moments using symmetric tensor decomposition. *SIAM J. Matrix Anal. Appl.* 41, 3 (2020), 1369–1387.

[78] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard W. Vuduc. 2012. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming*.

[79] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489. https://doi.org/10.1016/j.cpc.2010.04.018

[80] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D.P. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, and et al. 2018. Warp-X: A new exascale computing platform for beam–plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 909 (Nov 2018), 476–479. https://doi.org/10.1016/j.nima.2018.01.035

[81] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Bradford M Beckmann, William C Brantley, Joseph L Greathouse, Wei Huang, Arun Karunanithi, et al. 2017. Design and Analysis of an APU for Exascale Computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 85–96.

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

[83] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. 2018. Spindle: informed memory access monitoring. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 561–574.

[84] K. Wu, Y. Huang, and D. Li. 2017. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[85] Kai Wu, Yingchao Huang, and Dong Li. 2017. 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*. 1–14.

[86] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

[87] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 401–413.

[88] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 426–440.

[89] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*. 215–226.

[90] Zhen Xie, Jie Liu, Sam Ma, Jiajia Li, and Dong Li. 2022. LB-HM: load balance-aware data placement on heterogeneous memory for task-parallel HPC applications. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 435–436.

[91] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 94–105.

[92] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2021. A pattern-based spgemm library for multi-core and many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 159–175.

[93] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[94] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.