Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy

Jie Ren
jren6@ucmerced.edu
University of California, Merced

Jiaolin Luo
jluo38@ucmerced.edu
University of California, Merced

Ivy Peng
peng8@llnl.gov
Lawrence Livermore National Laboratory

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

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

ABSTRACT

Particle simulations of plasma are important for understanding plasma dynamics in space weather and fusion devices. However, production simulations that use billions and even trillions of computational particles require high memory capacity. In this work, we explore the latest persistent memory (PM) hardware to enable large-scale plasma simulations at unprecedented scales on a single machine. We use WarpX, an advanced plasma simulation code which is mission-critical and targets future exascale systems. We analyze the performance of WarpX on PM-based heterogeneous memory systems and propose to make the best use of memory hierarchy to avoid the impact of inferior performance of PM. We introduce a combination of static and dynamic data placement, and processor-cache prefetch mechanism for performance optimization. We develop a performance model to enable efficient data migration between PM and DRAM in the background, without reducing available bandwidth and parallelism to the application threads. We also build an analytical model to decide when to prefetch for the best use of caches. Our design achieves 66.4% performance improvement over the PM-only baseline and outperforms DRAM-cached, NUMA first-touch, and a state-of-the-art software solution by 38.8%, 45.1% and 83.3%, respectively.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

heterogeneous memory, memory management, plasma simulations

ACM Reference Format:
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 *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460356

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460356

1 INTRODUCTION

Plasma simulations are critical for understanding plasma dynamics in space weather and fusion devices [3, 34, 36]. The particle-in-cell (PIC) method is an important model that enables large-scale plasma simulations on high-performance computing (HPC) systems [4, 10, 33, 36]. The PIC method uses computational particles to simulate plasma particles, such as electrons and protons. High-fidelity PIC simulations often use billions and even trillions of particles, which require high memory capacity.

Persistent memory (PM), exemplified by the Intel Optane DC PM [13], provides a solution to meet the requirement of high memory capacity in HPC applications. For instance, the Intel Optane PM can provide up to six terabyte (TB) memory on a single machine. However, there is a performance gap between PM and DRAM [13, 25]. Read and write bandwidth of the Optane PM are only 38% and 16% that of DRAM, respectively. Hence, PM often comes with a small DRAM (tens of gigabytes) to boost performance. As a result, PM and DRAM form a heterogeneous memory (HM) system. How to place and migrate data between PM and DRAM to enjoy the speed of DRAM and capacity of PM remains active research [7, 11, 22, 26, 39, 40].

In this paper, we leverage the latest PM hardware to enable large-scale plasma simulations. We analyze the performance and develop a performance model for optimizing PIC codes on PM-DRAM systems. Our performance analysis and optimization use a state-of-the-art electromagnetic PIC code called WarpX [33]. Nonetheless, the optimization strategies derived from this work are generally applicable to other PIC-based simulation codes.

WarpX [33] is a mission-critical application designed for efficient executions on large-scale HPC systems and future Exascale machines. WarpX enables high-fidelity modeling of many complex processes, such as laser- and beam-driven plasma accelerators. As a PIC method, WarpX has high memory footprints for simulating particles moving in electromagnetic fields. The memory footprint scales up with the number of particles and field size. For example, the recent production run on 4,096 nodes on the Cori supercomputer simulates 62 billions of particles and consumes up to 8.9 TB

memory. Therefore, a large memory capacity is a key enabler for large-scale simulations in WarpX.

Our performance analysis identifies two challenges in optimizing WarpX on PM-based systems. First, WarpX has frequent read/write with a streaming-like access pattern, which intensifies memory accesses. Given the low bandwidth of PM compared to DRAM, this access pattern is unfavorable. Second, the WarpX code uses tens of millions of data objects and frequent memory (de)allocation. These data objects include long-lived data structures for particles, fields, and metadata, as well as short-lived buffers for communication and computation. Managing such a large number of data objects with diverse properties on DRAM and PM is complex.

We introduce a set of techniques to optimize the performance of WarpX on PM. Data objects are characterized and classified based on their lifetime and memory access patterns. This information guides their placement and migration on PM and DRAM at runtime. Ideally, frequently accessed data objects are placed into DRAM. However, due to the limited DRAM capacity and the large problem size in production runs, only some data objects or even partial data objects can fit into DRAM. To address this challenge, we make the best use of memory hierarchy and employ a combination of data migration and processor-cache prefetch mechanisms. In particular, we partition long-lived large data objects and migrate their partitions between PM and DRAM. We prefetch frequently reused data from PM to processor caches without using limited DRAM space.

To achieve efficient migration between PM and DRAM, we need to address two challenges. First, migrating data consumes memory bandwidth. However, the application also needs to access memory. Hence, data migration can compete with the application threads for memory bandwidth. Second, data migration uses helper threads in the background, other than the application threads, to avoid exposing data migration into the critical path. However, using helper threads reduces the availability of processor cores for the application threads. An optimal number of helper threads should expedite data migration without causing performance loss in the application threads. To address the above challenges, we develop a performance model to decide the optimal number of helper threads for data migration. Our model considers the constraints on memory bandwidth and core availability in realistic simulations. Based on the performance model, we use a lightweight runtime algorithm combined with runtime profiling and empirical observations to select and adapt the data migration between PM and DRAM for different input problems.

To enable efficient prefetch from PM to processor caches, we must decide when to prefetch, and the prefetch must be just early enough such that the data is in the caches right before computation without eliminating useful data or wasting cache space. We build an analytical model to decide when to prefetch based on an abstraction of memory accesses in WarpX.

We summarize the paper contributions as follows.

- We demonstrate and quantify the benefits of leveraging PM to enable large-scale plasma simulations in a mission-critical application called WarpX.
- We characterize the memory management, bandwidth consumption, and data object lifetime and access patterns in WarpX production simulations. We analyze the implication of the characterization for performance optimization on PM-based systems.
- We make the best use of memory hierarchy based on static and dynamic data placement strategies, and processor-cache prefetch guided by performance modeling.
- We improved the WarpX execution on Optane-only by 66.4% and outperformed DRAM-cached, the NUMA first-touch policy, and a state-of-the-art HM solution by 38.8%, 45.1% and 83.3%, respectively.

2 BACKGROUND

The WarpX particle-in-cell code. WarpX leverages MPI+OpenMP parallelism. It has two components, i.e., PICSAR [28] for particle-in-cell (PIC) routines at the innermost level and AMReX [44] for adaptive mesh refinement (AMR). A WarpX simulation may consist of multiple levels of resolution. Each level is an AMR level in the AMReX library and performs a PIC simulation at the resolution of that level.

PIC codes typically have the following characteristics. Field and particles are the main data structures, and particles consume the most memory footprint. The core PIC routines include four phases – `current deposition`, `field solver`, `field gather`, and `particle pusher`. In `current deposition`, all particles are iterated to deposit their charge and moments to the fields. In `field solver`, a linear system from the discretized Maxwell's equations is solved to compute electric and magnetic fields on the grid. During `field gather`, forces from the fields are calculated for each particle, which then in `particle pusher`, are used to update the location of particles. Both `current deposition` and `field gather` have mostly regular data access to the particles, exhibiting streaming-like read access in `current deposition` and read-write access in `field gather`.

Communication happens in `field solver` and `particle pusher`. Most communication in `field solver` is point-to-point (P2P) between neighbor processes for halo exchange. Both collective and P2P communications are used in `particle pusher` for communicating particles that move from one subdomain to another.

The Intel Optane DC PM. The Intel Optane DC Persistent Memory Module (PMM) is the first large-scale byte-addressable PM. The Intel Purley platform used in our study is equipped with Optane PM DIMMs and DRAM DIMMs. Each socket has six memory channels, and each is shared by a DRAM DIMM and a PMM DIMM. In total, there are 12 PM DIMMs and DRAM DIMMs, respectively, on two sockets. An Optane PM DIMM may have 128, 256, or 512 GB capacity, enabling up to 6 TB memory capacity on a single machine [13]. The latency to PM is measured as 174 ns for sequential reads and 304 ns for random reads, in contrast to 79 ns and 87 ns to DRAM [25]. The bandwidth to PM on one socket is 39 GB/s for read and 13 GB/s for write, while DRAM achieves 104 GB/s and 80 GB/s bandwidth on the same platform. There are two modes in PMM: *memory mode* and *app-direct mode*. In the memory mode, DRAM becomes a hardware-managed cache to PMM. Running the application on DRAM-cached PMM to use both DRAM and PMM requires no application modifications. In the app-direct mode, accesses to

Table 1: Compare the memory capacity and simulation scale on supercomputers

Supercomputer	Mem capacity per node	Largest problem (in terms of particles)
Sierra	320GB DRAM	10.6 trillions
Summit	608GB DRAM	18.9 trillions
Aurora	256GB DRAM (est.)	8.8 trillions
Taihu Light	32GB DRAM	1.1 trillions
Optane-based	1692GB (1.5TB PM + 192GB DRAM)	58.6 trillions

PM and DRAM can be explicitly controlled at the application level, either through a DAX-based file system [31] or exposing PM as separate NUMA nodes.

Enabling Large-Scale Simulations with PM. Using the Optane persistent memory, we can significantly increase the memory capacity per node to enable fine-grained and large-scale scientific simulations. An Optane-based machine has up to six TB memory [13], while a node in main-stream supercomputers has at most hundreds of GB (see Table 1). Given a fixed number of nodes, using the Optane PM allows us to perform scientific simulation previously unachievable due to limited memory capacity.

Table 1 presents an example case that performs a numerical simulation of a laser-driven plasma accelerator (i.e., the laser-wakefield accelerator) using WarpX [8]. This simulation uses a large number of particles in the time and space scales to gain knowledge on plasma structures towards a full-scale numerical study of the next generation laser-wakefield accelerator systems. Such numerical studies provide insights for compact high-energy colliders [16].

In this example, we assume the same simulation configuration as that in a production run on 4,942 nodes on the Cori supercomputer. Table 1 compares the largest simulation scale that can be supported on each supercomputer. The simulation scale is defined as the number of simulated particles – a larger number indicates a larger simulation scale. Clearly, *Memory capacity* is one main constraint on the simulation scale. The memory consumption of WarpX is calculated based on the estimation of the sizes of particles, fields, metadata, and temporal data objects.

Table 1 shows that an Optane-based supercomputer can enable larger-scale simulations than other supercomputers. Compared with Summit and Sierra (the top two supercomputers in the top500 list by April 2020) that use hundreds of Gigabytes of DRAM per node, the Optane-based supercomputer increases the simulation scale by 3.1x and 5.5x, respectively.

3 PERFORMANCE CHARACTERIZATION

We develop a heap profiler and a phase profiler to characterize the memory usage and bandwidth consumption in the application. The heap profiler tracks dynamic memory allocations and collects information on each allocation (data object). The phase profiler collects hardware events from performance counters and associates them to specific execution phases in the application.

Heap profiler interposes common memory management routines in C and C++, e.g., `malloc`, `calloc`, the operator `new` and its variants, `posix_memalign`, Linux-specific `aligned_alloc` and `valloc`. It collects the metadata of data objects, including size, time of allocation/deallocation, and lifetime (defined as the interval between allocation and deallocation). The timestamps of allocation and deallocation are used to map to specific execution phases. The tool also supports postmortem analysis of the profiling results.

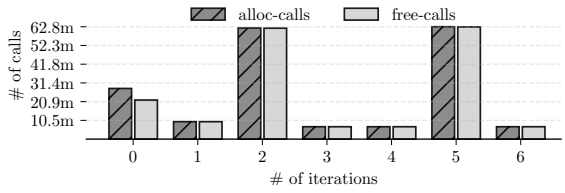

Figure 1: The number of memory allocation/deallocation across iterations.

Phase profiler use specific APIs to track execution phases. The user inserts the APIs into the WarpX code to mark execution phases. The API implementation includes two functionalities. First, it triggers a set of auxiliary external scripts to invoke the Linux performance profiling tool `perf` to collect information from hardware performance counters. Also, it invokes the Intel `PCM` [32] to collect memory bandwidth data.

3.1 Profiling Results

We use a representative laser-driven simulation configuration for profiling. The input problem uses 704×704×5664 cells and 8.4 billion particles (see Problem B in Table 5). The peak memory consumption exceeds 1.2 TB on DRAM-cached Optane (memory mode).

Memory allocation and deallocation analysis. We use the heap profiler to track memory allocation/deallocation in each iteration of the WarpX execution. Figure 1 presents the results for the first seven iterations. The profiling results show that millions of memory allocation and deallocation occur in each iteration. Across iterations, the number of memory allocation and deallocation varies. Such a massive amount of data objects, which are as resulted from frequent allocation and deallocation, imposes challenges in profiling at either data object level [12, 24, 39] or memory page level [2, 6, 14, 40, 42].

Data object lifetime and size. We classify the distribution of lifetime and size of data objects. Table 2 reports the classification in the second iteration of the WarpX simulation. Other iterations exhibit similar distributions. A data object is alive after its allocation and before its deallocation. We categorize a data object as short-lived if its lifetime is within one iteration and long-lived otherwise. We observe that 92.7% of data objects are short-lived in the WarpX simulation. Furthermore, these short-lived data objects only account for less than 10% of the peak memory consumption of WarpX. This characterization motivates us to use a small DRAM space to host repeatedly allocated/freed short-lived data objects and avoid data movement between DRAM and PM. This static placement strategy is described in Section 4.1.

Execution time breakdown. We measure the time of major execution phases (Section 2). Each iteration of the main computation loop performs these major phases. Some "add-ons" execution (such as load redistribution and moving window) may also occur in some iterations, counted as `others`. Table 3 reports the breakdown of the execution time.

Overall, the `particle pusher` and `current deposition` phases account for about 84% of the total simulation time. `Particle pusher`

Table 2: The distribution of object size.

Bin (MiB)	Short-lived data object		Long-lived data object	
	Accumulated footprint	Peak footprint	Accumulated footprint	Peak footprint
(0,1)	897.7 GiB	10.4 GiB	840.3 GiB	840.3 GiB
[1,2)	34.3 GiB	10.8 GiB	1.9 GiB	1.9 GiB
[2,4)	262.5 GiB	66.0 GiB	285.5 GiB	285.5 GiB
[4,8)	144.0 MiB	16.0 MiB	543.0 MiB	543.0 MiB
[8,16)	96.0 MiB	16.0 MiB	14.0 MiB	14.0 MiB
[16,32)	192.0 MiB	32.0 MiB	28.0 MiB	28.0 MiB
[32,64)	384.0 MiB	64.0 MiB	0	0
[64,+∞)	768.0 MiB	1.6 GiB	0	0

Table 3: The breakdown of execution time.

	Particle pusher	Current deposition	Field solver	Field gather	Others
Ave. time	300.8s	132.0s	47.2s	25.2s	9.9s
Percentage	58%	26%	9%	4.9%	2.1%

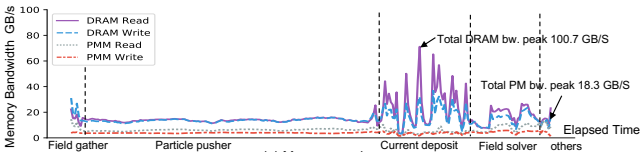

Figure 2: Memory bandwidth consumption in major phases.

reads the fields and updates the position of each particle. `Current deposition` reads each particle and updates the current densities on fields. These phases dominate the execution time and the read-/write accesses to the main memory. Therefore, we employ fine-grained dynamic data management to optimize their performance. We describe the dynamic strategy in Section 4.3.

Memory bandwidth analysis. We measure the memory bandwidth in major phases and report in Figure 2. We observe that the execution of WarpX is not bounded by DRAM/PM bandwidth in most of the execution time (e.g., `field gather` and `particle pusher`). When the memory bandwidth utilization is low, e.g., about 10%, prefetching data to DRAM would not constraint the bandwidth used by the application. Thus, performance improvement becomes feasible. However, since data prefetching consumes memory bandwidth, using it in bandwidth-intensive phases (e.g., `current deposit`) may cause performance loss in the application. The bandwidth analysis motivates us to develop a performance model to optimize data prefetching at runtime (Section 4.3).

4 PERFORMANCE OPTIMIZATION ON PM

We propose a runtime system, called WarpX-PM (Figure 3), to manage data placement on DRAM and PM automatically. WarpX-PM partitions DRAM into four spaces to store data objects with different functionality and access patterns in WarpX. The *metadata space* stores metadata updated infrequently but accessed frequently. The *temporary space* stores short-lived data objects frequently allocated and freed. Those short-lived data objects share and reuse the temporary space without causing data movement between DRAM and PM. The *migration space* acts as a software-managed DRAM cache

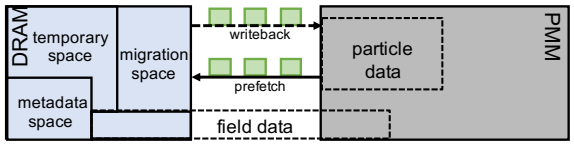

Figure 3: The overview of data management on Optane-based HM.

to prefetch particles from PM before they are used in computation. Finally, the *free space* stores the maximum possible field data.

We combine static and dynamic strategies, and processor-cache prefetch mechanism for data placement in the four spaces. Except for the migration space managed for dynamic data placement, the other three spaces are used for static data placement. We use performance modeling to guide the data copy between DRAM and PM without disturbing the WarpX performance. We also build an analytical model to decide when to prefetch for the best use of caches. Our designs are described in detail as follows.

4.1 Static Data Placement

WarpX-PM uses static placement to addresses the fundamental limitations in the memory mode. This memory mode uses DRAM as a direct-mapped hardware cache. Consequently, some performance-critical data objects are evicted from DRAM due to iterative accesses to large data objects, such as particles and fields. Examples of performance-critical data objects include metadata and temporary data, where metadata is used to compute the simulation domain iteratively, and temporary data is used to adjust the size of data objects during the computation. These performance-critical data objects are frequently referenced but only consume a small portion (less than 10%) of the total memory consumption. In the memory mode, these data objects are frequently moved between DRAM cache and PM, a typical manifestation of cache thrashing.

Static data placement takes effect on all execution phases. WarpX-PM pins the performance-critical data objects to DRAM to avoid moving them between DRAM and PM as in the memory mode. Depending on their lifetime, they can be categorized as long-lived and short-lived, and placed into the metadata space and temporary space, respectively. We describe the management of these two kinds of performance-critical data objects as follows.

Long-lived, performance-critical data objects are mostly metadata and are placed in the metadata space in DRAM directly. In WarpX, the whole simulation domain is decomposed into many *boxes* distributed over MPI ranks. Each box contains a fraction of fields and particles. Metadata is used to record the distribution of boxes in the simulation domain. For instance, *FArrayBox* is a part of box metadata for iterating particles in a box. Metadata are allocated before the main computation loop and only freed after the whole computation finishes – long-lived. During their life span, metadata are frequently accessed, and their size remains unchanged.

Short-lived, performance-critical data objects are typically allocated and freed within one iteration of the main computation loop. These data objects include communication buffers and the memory space used for resizing the data objects during the computation. WarpX-PM allocates these data objects on demand in the temporary

```
1  initial boxes, fields, particles
2  int k = prefetch_distance
3  ...
4  for(int step = 0; step < numsteps_max; ++step) {
5    amrex::ParallelFor(bx in boxes)
6    {
7      Particle &p = particles.get(bx)
8      compute(p)
9      Fields &f = fields.get(p)
10     warpx_pm_prefetch(fields.get(p+k))
11     compute(f)
12     ...
13   }
14 }
```

Figure 4: Applying prefetching techniques in WarpX.

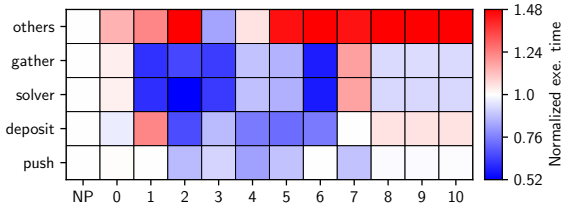

Figure 5: Performance of prefetch fields data at various prefetch distances. Performance is normalized to "NP" (no prefetch). Blue indicates performance improvement, and red indicates performance loss.

space, which is a pre-allocated memory space in DRAM. To ensure the pre-allocated temporary space is large enough for all temporary data objects throughout the computation loop, WarpX-PM uses the following algorithm.

WarpX-PM uses the first iteration of a simulation to measure the peak memory consumption of WarpX. Then, WarpX-PM deducts the sizes of particles, fields, metadata, and a fixed buffer per MPI rank for the migration space from the peak memory consumption. The resulted size is used to reserve the temporary space. This approach provides an estimation of the peak memory consumption of short-lived, performance-critical data objects. Across iterations, the peak memory consumption of these data objects may vary, mostly due to communication buffers. The variance is typically small (tens of MB). If the temporary space is exhausted, WarpX-PM increases the temporary space on demand to accommodate.

After the metadata, temporary, and migration spaces are allocated, the remaining space in DRAM is used as the free space, which is used to hold fields as much as possible. Fields are frequently accessed in all phases. WarpX-PM chooses fields instead of particles for static data placement because fields are not allocated in contiguous memory space. Hence, maintaining their location information and copying them between DRAM and PM incur high overhead. Besides, fields are smaller but more intensively accessed than particles, showing better data locality in processor caches. If fields cannot be completely placed in DRAM, we use a cache prefetch mechanism to fetch fields from PM to the last level cache to avoid PM's negative performance impact (see Section 4.2).

The static data placement completes after the first iteration. The memory allocation overhead is negligible because only three spaces need to be managed. Furthermore, using the pre-allocated temporary space reduces the overhead of frequent memory (de)allocation for short-lived data objects.

4.2 Cache Prefetch

Motivation. Fields are not dynamically migrated between PM and DRAM as particles, because fields are not contiguously allocated as particles, but spread across the memory address space, which leads to either highly inefficient page-level migration or costly engineering efforts for object-level migration. We do not prefetch particles because particles can be timely migrated to DRAM using the dynamic migration mechanism and prefetch data from DRAM to the cache leads to negligible performance improvement in WarpX (see Section 6).

Prefetch method. WarpX-PM triggers prefetch before computation by leveraging iterative structures in each execution phase. Figure 4 depicts the prefetch method in WarpX-PM. Each phase is characterized with a loop. Within each iteration of the loop, some field data are retrieved (Line 9) and then used for computation (Line 11). In the iteration i, WarpX-PM prefetches the field data for the next $(i + k)$th iteration, where k is a tunable parameter (named *prefetch distance*).

The prefetch effectiveness highly depends on k. If k is too small, prefetch cannot finish before the computation on fields happens; If k is too large, prefetched fields have a risk of being eliminated from the cache due to cache conflicts. Ideally, k should be just large enough such that when fields are prefetched into the cache, computation on those fields is about to start. To understand the impact of k, we use a range of k and measure the performance of each phase. Figure 5 shows the results normalized by the performance without prefetch in Problem F (see Table 5 for this problem). Depending on the computation time of each phase, the optimal k is different from one phase to another.

Performance modeling to guide prefetch. We propose to use performance modeling to decide k, and the performance modeling is based on our abstraction on memory accesses in WarpX. In essence, when a phase processes particles one by one, for each particle, memory accesses are characterized with a combination of accessing particle metadata, the field of the particle, and the particle itself. This combination provides a *basic unit* for particle-based computation. k quantifies the number of basic units whose computation time is just enough to allow the preftech of the field data for one unit to complete.

Based on the above discussion, we formulate the calculation of k in Equation 1, where the numerator is the time to prefetch the field data for one particle, and the denominator is the execution time to work on one basic unit (i.e., processing one particle). The numerator is further expanded in Equation 2, which models the prefetch time, including instruction issue and execution cost (α), and memory access latency. Since the fields could spread into DRAM and PM, DRAM/PM access latency in Equation 2 is weighted by the ratio of the fields in DRAM ($fields_{DRAM}$) to the fields in PM ($fields_{PM}$). This indicates that if the fields are completely placed in PM, then $fields_{DRAM}$ is 0. The DRAM/PM access latency is assumed to be constant in Equation 2, but weighted by $\beta 1$ and $\beta 2$ to model the

potential impact of internal buffers in memory devices (especially PM devices) [37].

$$k = \left\lceil \frac{prefetch\_time}{exe\_time\_on\_one\_basic\_unit} \right\rceil \quad (1)$$

$$\begin{aligned} prefetch\_time =& \alpha + mem\_access\_time \\ =& \alpha + \beta 1 \times \frac{fields_{DRAM}}{fields_{PM}} \times DRAM\_access\_latency + \\ & \beta 2 \times (1 - \frac{fields_{DRAM}}{fields_{PM}}) \times PM\_access\_latency \end{aligned} \quad (2)$$

To use the above equations, we use the following method. The denominator (the execution time) in Equation 1 is measured at runtime by each MPI process after the data placement is enforced based on Sections 4.1-4.3. α, $\beta 1$ and $\beta 2$ in Equation 2 are calculated using a microbenchmark offline. In particular, we develop a microbenchmark that spreads a collection of field data across DRAM and PM. We change the ratio of the field data in DRAM and PM ($\frac{fields_{DRAM}}{fields_{PM}}$), prefetch them one by one, and measure the latency. Given the latency measured with different ratios ($\frac{fields_{DRAM}}{fields_{PM}}$), we calculate α, $\beta 1$ and $\beta 2$ using linear regression. $DRAM\_access\_latency$ and $PM\_access\_latency$ in Equation 2 are measured using LMBench [21]. $\frac{fields_{DRAM}}{fields_{PM}}$ is determined after the static data placement at runtime.

4.3 Dynamic Data Placement

The dynamic strategy takes effect at `particle pusher`, `current deposition`, and `field solver`. The accesses to particles mainly occur in these phases. The dynamic placement copies particles into DRAM in batches and only copies them back to PM if particles are updated in the computation. Particles consume at least 50% of memory consumption. For a large input problem, particles alone may unlikely fit in DRAM. However, directly accessing particles in PM in particle computation causes performance loss due to the low memory bandwidth. WarpX-PM uses software-managed *particle prefetching* to copy batches of particles into the migration space so that computation always accesses particles in DRAM.

`ParticleContainer` is the primary data object for particles. It contains an array of particle structures, each representing a particle and recording its position, velocities, ID, and the owner CPU. Thus, `ParticleContainer` occupies a contiguous space in physical memory. In each of the `particle pusher`, `current deposition`, and `field solver` phases, all particles in the `ParticleContainer` are iterated in a streaming-like access pattern at the granularity of `FArrayBox`. WarpX-PM leverages this characteristic to partition each phase into intervals based on the time of processing particles in `FArrayBox`. At an interval i, WarpX-PM copies a batch of particles needed for the next interval $i+1$ to DRAM. This data copy is expected to finish before the interval $i+1$. If particles are updated in the interval $i+1$, they are copied back to PM in the interval $i+2$. Given the streaming-like patterns to access particles, there is no data dependency between intervals.

To implement the particle prefetching strategy, two challenges must be addressed. First, WarpX-PM needs to decide the number of threads to copy particle batches. WarpX-PM uses helper threads instead of application threads to copy particles to avoid delaying the execution of application threads. Using a large number of helper threads accelerate data copy but reduces processor cores and memory bandwidth available for WaprX execution. Using a small number of helper threads increases the risk of exposing data copy into the critical path of WaprX execution if data copy cannot finish in time. Second, the decision of the number of helper threads must be adaptive and lightweight. Different input problems or MPI/OpenMP configurations may consume memory bandwidth differently and need different numbers of helper threads for the best performance.

Table 4: Notation for performance modeling

Source	Symbol	Description
Hardware parameters	$\mathbf{BW}^{DRAM\_to\_PM}()$	BW of copying data from DRAM to PM
	$\mathbf{BW}^{PM\_to\_DRAM}()$	BW of copying data from PM to DRAM
	$\mathbf{BW}_{max}$	Peak memory bandwidth
	$\mathbf{Thrd}_{max}$	Maximum number of hardware threads
App related parameters	$\mathbf{data}_{in}, \mathbf{data}_{out}$	Sizes of data copied in/out of DRAM
	$\mathbf{T}_{cp}$	Data copying time for an interval
	$\mathbf{T}_{comp}$	WarpX execution time of an interval
	$\mathbf{Thrd}_{cp}$	Number of threads to copy data
	$\mathbf{Thrd}_{comp}$	Number of threads for application
	$\mathbf{T}'_{comp}$	Optimal execution time of an interval

Performance Modeling. We introduce a performance model-based approach to decide the optimal number of helper threads for each phase. All intervals in the same phase use the same number of helper threads while different phases may use different numbers of helper threads. Table 4 summarizes the notations used in the performance model.

data copy time (T_{cp}) in an interval i includes the time to copy data needed by the interval $i+1$ from PM to DRAM (T^{in}_{cp}), and the time to copy data updated in the interval $i-1$ from DRAM to PM (T^{out}_{cp}).

$$\begin{aligned} T^{in}_{cp}(Thrd_{cp}) &= \frac{data_{in}}{BW^{PM\_to\_DRAM}(Thrd_{cp})} \\ T^{out}_{cp}(Thrd_{cp}) &= \frac{data_{out}}{BW^{DRAM\_to\_PM}(Thrd_{cp})}, \end{aligned} \quad (3)$$

where $data_{in}$ and $data_{out}$ are the size of data to be copied in and out of DRAM for an interval; $BW^{PM\_to\_DRAM}(Thrd_{cp})$ and $BW^{DRAM\_to\_PM}(Thrd_{cp})$ are the data copy bandwidth in and out of DRAM. These bandwidths are functions of the number of helper threads ($Thrd_{cp}$). Therefore, T_{cp} is also a function of $Thrd_{cp}$.

Equation 3 considers performance difference between copying data from DRAM to PM and from PM to DRAM. In our implementation, copying data in two directions happens in parallel. If memory bandwidth is not a bottleneck, we have

$$T_{cp} = max(T^{in}_{cp}, T^{out}_{cp}). \quad (4)$$

Overlap constraint. Copying data happens in parallel with WarpX execution. The data copy time should be no longer than the WarpX execution time, i.e.,

$$T_{cp}(Thrd_{cp}) \leq T_{comp}(Thrd_{comp}), \quad (5)$$

where T_{comp} is the execution time of an interval when the particles accessed by the interval are all in DRAM. T_{comp} is a function of the number of application threads ($Thrd_{comp}$) in an MPI rank.

Bandwidth constraint. The bandwidth consumption due to copying data should not reduce the bandwidth available for WarpX execution. Assume that without copying data, the bandwidth consumption of WarpX execution is BW_{comp}, including both read from and write to PM.

$$BW_{comp}(Thrd_{comp}) + BW^{PM\_to\_DRAM}(Thrd_{cp}) + BW^{DRAM\_to\_PM}(Thrd_{cp}) \leq BW_{max}/N \quad (6)$$

where BW_{max} is the peak memory bandwidth constrained by the hardware and N is the number of MPI ranks (We assume BW_{max} is evenly partitioned between MPI ranks). BW_{max} needs to satisfy the following equation to prevent performance loss.

$$BW_{max} = max(BW_{max}^{DRAM\_to\_PM}, BW_{max}^{PM\_to\_DRAM}) \quad (7)$$

$BW_{max}^{DRAM\_to\_PM}$ and $BW_{max}^{PM\_to\_DRAM}$ are the peak memory bandwidth supported by hardware from DRAM to PM and from PM to DRAM, respectively.

Thread constraint. The number of application threads and helper threads should be no larger than the maximum number of threads assigned to an MPI rank ($Thrd_{max}$), i.e.,

$$Thrd_{comp} + Thrd_{cp} \leq Thrd_{max}. \quad (8)$$

Optimization goal. Assume that T'_{comp} is the execution time for an interval, given $Thrd_{comp}$ and $Thrd_{cp}$ threads for WarpX execution and copying data, respectively. T'_{comp} is a function of $Thrd_{comp}$ and $Thrd_{cp}$. The goal of our performance modeling is to minimize T'_{comp} (Equation 9), subject to the constraints of overlap (Constraint 5), bandwidth (Constraint 6) and threads (Constraint 8), i.e.,

$$min(T'_{comp}(Thrd_{comp}, Thrd_{cp})). \quad (9)$$

BW_{max} and $Thrd_{max}$ are known from offline profiling; $BW^{DRAM\_to\_PM}()$ and $BW^{PM\_to\_DRAM}()$ are measured by a microbenchmark at various numbers of data copy threads; $data_{out}$ and $data_{in}$ are known from $FArrayBox$, whose value is set at the beginning of each iteration. Therefore, based on $data_{out}$ and $data_{in}$, we can calculate T_{cp} using Equation 4 given $Thrd_{cp}$.

We build $T_{comp}()$ based on online profiling and empirical observation. In particular, we use an interval in the second iteration of the main computation loop to measure the execution time online, and use $Thrd_{max}$ as $Thrd_{comp}$ during the execution of the interval. This measurement is done after static data placement and after loading the required particles by the interval into DRAM. Furthermore, we empirically observe that the execution of WarpX using various input problems is not bounded by memory bandwidth on Optane (see Section 3.1); Using $Thrd_{max}$ as $Thrd_{comp}$ gives the best performance. Using $Thrd_{max} - 1$ and $Thrd_{max} - 2$ as $Thrd_{comp}$ give less than 10% performance loss, while using the number of threads smaller than $Thrd_{max} - 2$ for $Thrd_{comp}$ causes more than 20% loss. Hence, we use the measured online execution time as the result of $T_{comp}(Thrd_{comp})$, when $Thrd_{comp} \in [Thrd_{max} - 2, Thrd_{max}]$. We do not consider other cases of $Thrd_{comp}$ to avoid performance loss of WarpX execution. Note that this approach gives us a high requirement on data copy overhead because of Constraint 5.

We employ a similar approach to build $BW_{comp}()$. We measure memory bandwidth in an interval in the second iteration of simulation and using $Thrd_{max}$ as $Thrd_{comp}$. This memory bandwidth is used for $Thrd_{comp} \in [Thrd_{max} - 2, Thrd_{max}]$. We do not consider other cases to avoid performance loss.

We use the following approach to find the optimal $Thrd_{comp}$ and $Thrd_{cp}$ to minimize $T'_{comp}()$. We use Constraints 6 and 8 to select the numbers of helper threads to meet the bandwidth constraint. Then, among the selected numbers, we use Constraint 5 to find those that meet the overlap constraint. Finally, we choose the smallest number as the optimal number of helper thread from those selected numbers. Given the constraints, the WarpX execution time is minimized, $T'_{comp} = T_{comp}(Thrd_{max})$.

Our modeling approach is lightweight, because we avoid exhaustive search of all combinations of $Thrd_{comp}$ and $Thrd_{cp}$ by eliminating those that can obviously cause performance loss. The overhead to find the optimal is almost zero.

5 IMPLEMENTATION DETAILS

WarpX-PM is implemented as a patch to WarpX and AMRex. Running WarpX with WarpX-PM on Optane (or other HM) requires no efforts from the user. We release WarpX-PM in [1]. The statistics of modifications given by git diff is 15 files changed, 1031 insertions(+), 12 deletions(-).

WarpX-PM uses pthread to implement helper threads for each MPI rank. For static data placement, data objects that needed to be placed in DRAM are allocated into DRAM NUMA nodes using *numa_alloc_local()*. For dynamic data placement, each MPI process pre-allocates a 500MB temporary space in DRAM to copy particles between DRAM and PM. We use 500MB because the dynamic data placement handles particles batch by batch, and the batch size is determined by `FArraybox`. The size of all particles in one `FArraybox` is bounded by 500MB. All MPI processes evenly partition DRAM initially. To accommodate the size variance of short-lived data objects across interations, WarpX-PM increases the temporary space by reserving extra 100MB DRAM space for each rank.

Avoiding NUMA effects is important for high performance on an Optane-based machine with multiple sockets, each equipped with both DRAM and PM [25]. We observe that allocating data in remote DRAM and PM nodes (i.e., DRAM and PM on the remote socket) leads to up to 2x performance loss for large input problems in WarpX. To address this NUMA effect, in WarpX-PM, once an MPI rank is pinned to a processor, those DRAM spaces for static and dynamic data placements are allocated from local DRAM NUMA nodes. Also, all data objects of the MPI process are allocated from local PM nodes.

WarpX-PM implements the model-guided cache prefetching in *warpx_pm_prefetch()* based on *_mm_prefetch* intrinsic. This operation prefetches cache lines in the field data to the processor's last-level cache by pointer chasing the metadata related to field data. WarpX-PM uses high-performance data copying to implement data placement based on AVX-512 streaming load/store intrinsics and multi-threading. Alternatively, we could use a page migration

Table 5: Input problems used in evaluation

ID	Type	# of cells	# of particles	Peak consumption
A	Laser-driven	(512, 512, 4096)	1.1B	228.5 GiB
B	Laser-driven	(704, 704, 5664)	8.4B	1.2 TiB
C	beam-driven	(512, 512, 4096)	2.1B	306 GiB
D	beam-driven	(864, 864, 7200)	10.7B	960 GiB
E	Uniform-plasma	(384, 384, 3104)	3.7B	525 GiB
F	Uniform-plasma	(512, 512, 4096)	8.6B	1.2 TiB
G	Laser-driven	(256, 256, 2048)	134.2M	19.2 GiB

* The names of particle species of A, E, F and G are set to *electrons*; the names of B are set to *electrons, ions* and *beam*; the names of C and D are set to *driver, plasma_e, plasma_p, beam* and *driverback*. The blocking factor is 32.

Table 6: Platform Specifications

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

mechanism such as *move_pages()* and *mmap()* to implement data migration between DRAM and PM instead of data copying. However, these data migration mechanisms work at the page level, requiring setting up a mapping between data objects and pages, which is difficult to implement at the user level. Furthermore, these mechanisms can cause frequent TLB misses because of page remapping, which leads to performance loss [42]. Note that our data copying mechanism in WarpX does not impact program correctness because our implementation has no pointer alias – the pointers pointing to the old data is updated after data copying.

6 EVALUATION

6.1 Experimental Setup

Table 6 summarizes the hardware features of our testbed. When the Optane DC PMM is in app-direct mode and exposed as NUMA nodes, we use `numactl` [18] to control data placement on PMM and DRAM. The platform runs Fedora 29 (Linux 5.1.0). We use the Intel Processor Counter Monitor (PCM) tool [32] to access hardware counters to collect core activities and off-core events.

Table 5 summarizes input problems for evaluation. They come from various plasma accelerator simulations with a wide range of memory consumption (up to 1.2TB). We use WarpX 20.04, OpenMPI 4.0.2 and GCC 7.5.0. For all problems except Problem G (a relatively small input), we run 10 iterations and report average execution time per iteration. There is less than 1% difference in average execution time if we use more than 10 iterations. For Problem G, we run 30 iterations to report average execution time, because average execution time becomes stable only after 20 iterations.

6.2 Evaluation Results

Overall performance. We compare WarpX-PM with Optane-only (i.e., no DRAM) and two common strategies (i.e., NUMA first-touch and memory mode) to use Optane-based systems. We evaluate Problems A-F in Table 5. All these problems have peak memory consumption larger than DRAM (192 GB). For Problem G, all data objects can be placed in DRAM, i.e., little performance difference between NUMA first-touch, memory mode, and WarpX-PM.

Figure 6 reveals that WarpX-PM performs the best in all cases. On average, WarpX-PM outperforms memory mode, Optane-only, and NUMA first-touch by 38.8%, 66.4%, and 45.2%, respectively. We notice that NUMA first-touch performs worse than memory mode and WarpX-PM. NUMA first-touch decides data placement based on when data allocation happens, instead of memory access patterns, which leads to sub-optimal data placement if a performance-critical data object is allocated at a later stage of execution. For example, particles are allocated before fields in WarpX, and NUMA first-touch places particles in DRAM, which forces fields to go to PM because of limited DRAM capacity. However, fields is more frequently accessed throughout simulation — placing it into PM leads to substantial performance loss. WarpX-PM avoids this problem because it prioritizes the placement of fields over particles on DRAM.

WarpX-PM outperforms memory mode because it avoids DRAM-cache thrashing for small and short-lived data objects. DRAM-cache thrashing happens due to accesses to the large data object – particles. Without application knowledge, the DRAM cache may evict small and short-lived data objects to make space for particles.

We find that WarpX execution has large performance variance in memory mode (up to 30.5%) for Problem A. This problem has peak memory consumption slightly larger than DRAM. Such performance variance in memory mode has been confirmed by Intel, and imposes a big challenge on controlling performance variability in HPC applications. WarpX-PM avoids this performance variance because of its static placement of critical data objects.

Performance breakdown. We quantify the performance improvement from static data placement, cache prefetch, and dynamic data placement in Figure 7. For each problem, we report the per-iteration time in the memory mode as the baseline (M). For comparison, we run WarpX-PM with only static data placement (S), with static data placement and cache prefetch (P), and with all the three techniques (D). WarpX-PM with the three proposed techniques achieves the best performance in all problems. Different input problems exhibit different sensitivity to these techniques. For instance, cache prefetch achieved over 12%-16% improvement in Problem B and C while problem A and E are more sensitive to dynamic data placement.

For large input problems (e.g., Problems B and D with 1.2TB peak memory consumption), static data placement outperforms memory mode by 29% on average. Memory mode cannot work well for the large input problems because metadata and temporary data are not efficiently cached in DRAM. On average, static data placement effectively speedup `field gather` and `others` by 10% and 7% in all input problems, compared to memory mode. `Field gather` and `others` involve large metadata and access to temporary data objects. Static data placement effectively prevents data migration in these two phases and thus avoids the migration overhead.

Dynamic data placement improves the performance of `particle pusher`, `current deposition` and `field solver` by 11%, 17%, and 12%, compared to static data placement. By proactively migrating particles from PM to DRAM, dynamic data placement outperforms memory mode and static data placement by 34% and 41%.

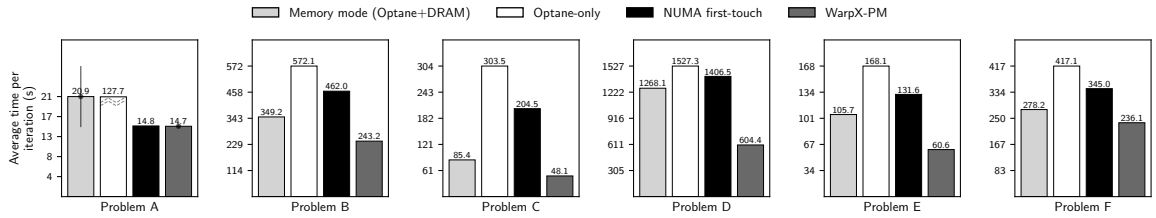

Figure 6: Performance comparison between Memory mode, Optane-only, NUMA first-touch and WarpX-PM.

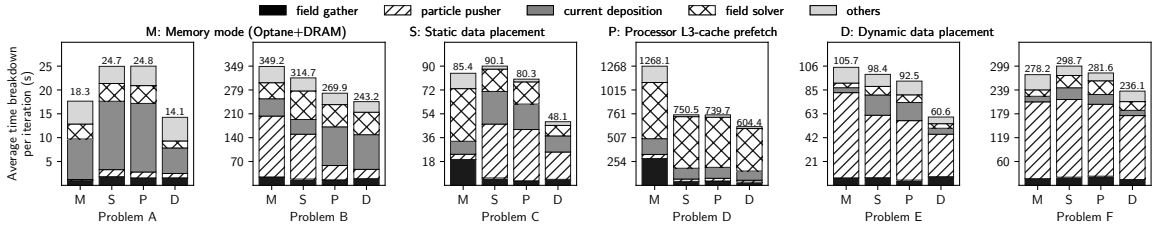

Figure 7: Performance breakdown of main phases of execution to compare the static data placement, cache prefetch, and dynamic placement with memory mode.

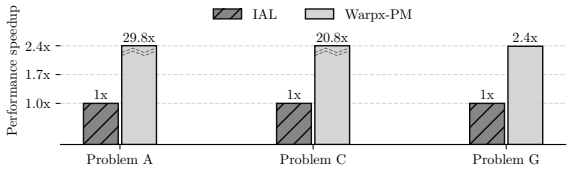

Figure 8: Performance comparison between IAL (a state-of-the-art page migration solution for HM) and WarpX-PM.

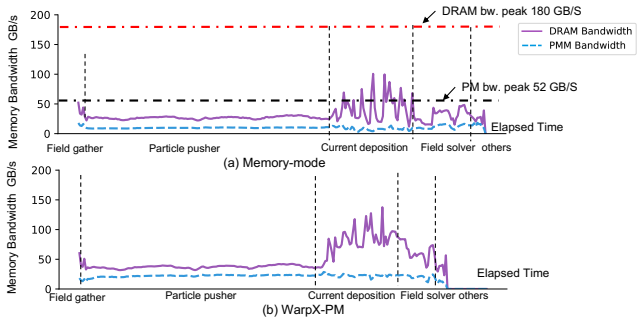

Figure 9: Memory bandwidth consumption in one iteration.

Comparison with state-of-the-art. We compare WarpX-PM with a state-of-the-art page migration system for HM, named *improved active list* (IAL) [42]. This system improves an existing page replacement mechanism in the Linux kernel (i.e., an FIFO-based active list). Among the 7 input problems listed in Table 5, we can only run three of them successfully with IAL. Running other problems with IAL has either 10x lower performance than WarpX-PM or segmentation faults.

Figure 8 reveals that WarpX-PM outperforms IAL by 83.3% on average and up to 96.6%. There are three main reasons for the inferior performance of IAL. First, IAL is a reactive approach – it takes effects only after it collects enough information on memory accesses. This indicates that it cannot efficiently prefetch data objects into DRAM to reduce data movement cost. Second, IAL periodically samples memory page accesses to identify page hotness. Finding hot pages from a large amount memory pages (tens of millions) incurs significant overhead. Third, IAL heavily relies on helper threads to enable parallel page migration for high performance. However, IAL does not consider the impact of using helper threads on the WarpX execution. Using an excessive number of helper threads decreases computation capability available for WarpX and consumes large memory bandwidth, which negatively impact the WarpX performance.

Memory bandwidth analysis. Figure 9 depicts read/write bandwidth for memory mode and WarpX-PM. We use input Problem F, because its peak memory consumption is the largest and pressures the memory bandwidth. Compared with memory mode, WarpX-PM consumes higher DRAM bandwidth, indicating that fast memory accesses happen more often in WarpX-PM to make best use of DRAM. More specifically, for execution phases that only involve static data migration (i.e., `field gather` and `others`), PM bandwidth consumption is lower than memory mode, indicating the effectiveness of static data placement. For execution phases that involve dynamic data migration (`current deposition`, `field solver` and `particle pusher`), WarpX-PM has higher PM bandwidth than memory mode. This is because dynamic data placement prefetches data objects before they are accessed, but data prefetch overhead is hidden by overlapping with the computation.

NUMA effects. Optane-based systems have multiple sockets, each with DRAM and PM DIMMs. Efficient data placement is not only about using DRAM or PM but also about avoiding memory accesses to a remote socket. We compare memory mode, NUMA

Table 7: Quantifying the memory traffic between NUMA nodes.

Problem ID	Remote DRAM traffic (GB)		
	Memory mode	First-touch	WarpX-PM
A	0.92	1.01	0.01
B	4.08	3.75	0.02
C	3.69	4.76	0.02
D	18.09	23.90	0.05
E	1.01	0.90	0.01
F	2.22	2.28	0.01

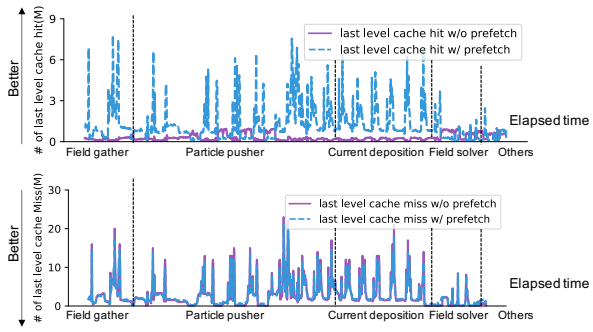

Figure 10: Last-level cache hits (top) and misses (bottom) in one iteration in problem F.

first-touch with WarpX-PM to quantify NUMA effect by tracking memory traffic between two sockets. We use six input problems whose peak memory consumption is larger than DRAM to allow us to evaluate the NUMA effect fully. Table 7 shows the results.

The results show that WarpX-PM has the lowest inter-socket traffic (close to zero). Memory mode is not NUMA-aware and cannot cache accesses to remote PM in a local DRAM [9]. The NUMA first touch policy is NUMA-aware, but data may be distributed to remote DRAM when the local DRAM is exhausted. WarpX-PM avoids these problems by explicitly placing data in local buffers (Section 5).

Effectiveness of cache prefetch. Figure 11 compares the model-guided selections of prefetch distance with selections based on exhaustive search. The exhaustive search-based selection represents the best possible performance, though infeasible at runtime due to its high cost. Among 30 cases (i.e., five phases in six input problems), our model agrees with the exhaustive search in 20 cases. In the remaining 10 cases, the performance difference between our model and exhaustive search is less than 5%. In all cases, our model leads to performance improvement (i.e., 9%-20%).

We evaluate the impact of the model-guided cache prefetch on traffic to the processor's last-level cache. Figure 10 reports the number of last-level cache hits and misses for Problem F. The results show that WarpX-PM significantly increases the number of last-level cache hits with spikes orders higher than that without prefetch. The increased cache hits only occurs in field gather, particle pusher, and current deposition, where particles are accessed in a streaming-like pattern, but not in field solver, where particles are not accessed. Meanwhile, our model-guided prefetch causes no increases in the number of cache misses.

Effects of performance modeling. We evaluate the effectiveness of the performance model in determining the number of helper threads. The performance variance due to different numbers of helper threads is the same across phases. Hence, we use the same number of helper threads for all phases for evaluation. We manually sweep the number of helper threads and compare their performance with the automatically adapted performance in WarpX-PM. Figure 12 shows the results. For Problem A, B, C, and E, the optimal number of helper threads is two. For Problem D and F, the optimal number of helper thread becomes one. WarpX-PM achieves similar performance as the optimal one in all problems. We note an over 30% performance loss when more than two helper threads are used. As the number of helper threads increases, the available processor cores for the computation in WarpX simulation decreases, which prolongs the total execution time.

WarpX-PM at scale. Because of hardware limitation, we cannot evaluate WarpX-PM on multiple Optane-based machines. However, WarpX-PM focuses on intra-node data movement optimization, and significantly improves performance without impacting communication patterns and application algorithms. We expect that WarpX-PM can be applied on larger scales without decreasing application scalability.

7 RELATED WORK

HPC workloads Many works have explored PM-based HM for HPC [7, 19, 20, 22, 26, 29, 39–41]. Nguyen et. al [22] introduce a multi-version octree on PM to enable adaptive mesh simulation on PM. Unimem [39] uses performance modeling to decide data placement for MPI-based HPC applications. Siena [26] explores rich organizations and configurations of HM architecture for HPC applications to determine optimal system designs. Tahoe [40] combines a machine learning model and an analytical model to predict application performance across multiple memory components for task-parallel programs. NVStream [7] uses non-temporal store and delta compression to reduce overhead for maintaining crash consistency and reduce I/O traffic for HPC workloads. These works use emulated PM to demonstrate their functionality. Recent works are characterizing HPC applications on Optane [19, 23, 27, 29, 35, 38, 41]. Patil et. al [23] measure performance of HPC mini-apps under different configurations of Optane DC PMM and reveal potential performance benefits of using PM-based heterogeneous memory in HPC. Different from above works, our work focuses on performance analysis and optimization of a production-level code (WarpX) for realistic simulations on real PM hardware.

Database and graph workloads. Recent works also propose various performance optimizations of databases and graph workloads on the Optane PM [5, 9, 15, 17, 30, 43]. Yang et. al [43] analyze the Optane architecture to optimize database and file system. TimeStone [15] solves the problem of poor scalability of durable transaction memory (DTM) on Optane by adopting multi-version concurrency control and a DRAM buffer. RECIPE [17] converts concurrent DRAM indexes to crash-consistent indexes on Optane. Gill et. al [9] evaluate four graph analytics frameworks and optimize performance by mitigating the NUMA effect of Optane. ATMem [5] employs a sampling-based profiler to select performance-critical data regions in graph applications on Optane.

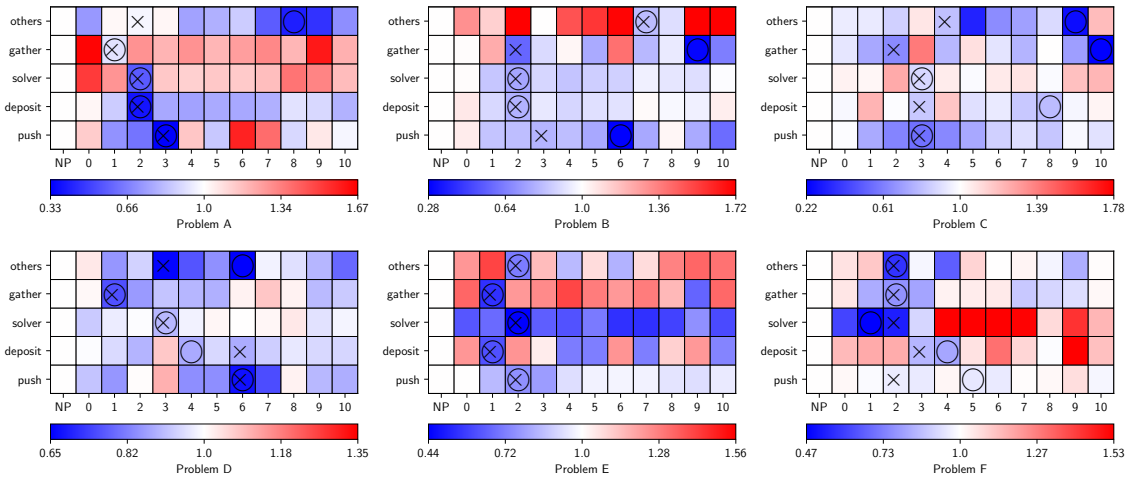

Figure 11: Performance at 1-10 prefetch distance normalized to that with no-prefetch (NP). X is the selection by the performance model and O is the result of exhaustive search. Blue indicates performance improvement and red for degradation.

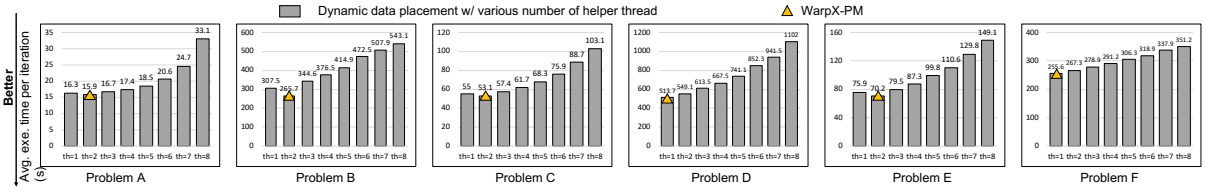

Figure 12: Compare the performance of WarpX-PM with the performance using one to eight helper threads. WarpX-PM always achieves the optimal performance.

8 CONCLUSIONS

The emerging large-capacity PM enables high-resolution large-scale scientific simulations. However, leveraging PM for production-level HPC codes on realistic problems remains to be investigated. In this paper, we focus on WarpX, a mission-critical plasma simulation code, as a use case to study PM implications on its performance. We demonstrate the PM benefits in simulation scales and propose a set of performance optimization strategies driven by detailed performance analysis. We improved the WarpX execution on Optane-only by 66.4% and outperformed DRAM-cached, the NUMA first-touch policy, and a state-of-the-art HM solution by 38.8%, 45.1% and 83.3%, respectively.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. We also thank Andrew Myers, Jean-Luc Vay, and Kevin Gott from Lawrence Berkeley National Lab for their kind answers to our questions on WarpX. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194), the Chameleon Cloud, and Intel hardware donation. LLNL-CONF-808998. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344.

REFERENCES

[1] 2020. WarpX-PM. https://anonymous.4open.science/r/4ca6ddfe-f76d-457e-b8a7-c136bcdf739e/.

[2] 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, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. https://doi.org/10.1145/3037697.3037706

[3] Charles K Birdsall and A Bruce Langdon. 2004. *Plasma physics via computer simulation*. CRC press.

[4] Kevin J Bowers, BJ Albright, L Yin, B Bergen, and TJT Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas* 15, 5 (2008), 055703.

[5] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*.

[6] 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 *European Conference on Computer Systems*.

[7] Pradeep Fernando, Ada Gavrilovska, Sudarsun Kannan, and Greg Eisenhauer. 2018. NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/3208040.3208061

[8] Samuel F.Martins, Ricardo A.Fonseca, Luís O.Silva, Wei Lu, and Warren B.Mori. 2010. Numerical Simulations of Laser Wakefield Accelerators in Optimal Lorentz Frames. *Computer Physics Communications* 181, 5 (2010), 869–875.

[9] 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. http://arxiv.org/abs/1904.07162

[10] David P Grote, Alex Friedman, Jean-Luc Vay, and Irving Haber. 2005. The warp code: modeling high intensity ion beams. In *AIP Conference Proceedings*, Vol. 749. American Institute of Physics, 55–58.

[11] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. https://doi.org/10.1145/3373376.3378465

[12] 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 (SoCC '16)*. ACM, New York, NY, USA. https://doi.org/10.1145/2987550.2987570

[13] 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. CoRR abs/1903.05714 (2019). *arXiv preprint arXiv:1903.05714* (2019).

[14] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. 2017. HeteroOS — OS design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080245

[15] 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 (ASPLOS '20)*.

[16] Lawrence Berkeley National Lab. [n.d.]. Laser-Plasma Accelerators for High-Energy Physics. https://bella.lbl.gov/research/bella-center-research-high-energy-physics/.

[17] 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 (SOSP '19)*.

[18] Linux. [n.d.]. numactl-Linux man page. https://linux.die.net/man/8/numactl

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

[20] 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 (PPoPP'21)*.

[21] Larry McVoy and Carl Staelin. [n.d.]. LMbench - Tools for Performance Analysis. http://lmbench.sourceforge.net/.

[22] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-Scale Adaptive Mesh Simulations through Non-Volatile Byte-Addressable Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.

[23] 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)*.

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

[25] 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

[26] I. B. Peng and J. S. Vetter. 2018. Siena: Exploring the Design Space of Heterogeneous Memory Systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 427–440. https://doi.org/10.1109/SC.2018.00036

[27] Ivy Bo Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. 2020. Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems. In *IPDPS*.

[28] picsar. [n.d.]. The PICSAR project. https://picsar.net/

[29] Jie Ren, Kai Wu, and Dong Li. 2020. Exploring Non-Volatility of Non-Volatile Memoryfor High Performance Computing Under Failures. In *IEEE International Conference on Cluster Computing*.

[30] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Neurips*.

[31] Andy Rudoff. 2013. Programming Models for Emerging Non-Volatile Memory Technologies. 38, 3 (2013).

[32] Thomas Willhalm, Roman Dementiev, Patrick Fay. 2017. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[33] J-L Vay, A Almgren, J Bell, L Ge, DP Grote, M Hogan, O Kononenko, R Lehe, A Myers, C Ng, 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 (2018), 476–479.

[34] John P Verboncoeur. 2005. Particle simulation of plasmas: review and advances. *Plasma Physics and Controlled Fusion* 47, 5A (2005), A231.

[35] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. 2019. Evaluation of Intel 3D-Xpoint NVDIMM Technology for Memory-Intensive Genomic Workloads. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*.

[36] WX Wang, ZTWM Lin, WM Tang, WW Lee, S Ethier, JLV Lewandowski, G Rewoldt, TS Hahm, and J Manickam. 2006. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas* 13, 9 (2006), 092505.

[37] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[38] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

[39] 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*.

[40] 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*.

[41] 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 *International Conference on Supercomputing (ICS)*.

[42] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*.

[43] 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)*.

[44] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (2019).