PEARS: A Performance-Aware Static and Dynamic Framework for Heterogeneous Memory

Onkar Patil[1], Frank Mueller[1], Latchesar Ionkov[2], Jason Lee[2], Michael Lang[2]
[1]North Carolina State University, USA
[2]Los Alamos National Laboratory, USA
opatil@ncsu.edu,mueller@cs.ncsu.edu,{lionkov,jasonlee,mlang}@lanl.gov

ABSTRACT

Heterogeneous memory systems are becoming more prominent in high performance computing (HPC) with the introduction of High Bandwidth Memory (HBM) and Non-Volatile Memory (NVM) technologies. All these memory technologies along with DRAM, while being byte-addressable, have different latency, bandwidth, power and persistence characteristics that they bring to the table. NVM adds to the capacity of main memory and expands the address space of applications but lowers access speeds. HBM provides higher memory access bandwidth but has lower density than DRAM. Although there is software support for using all the above memory technologies in their individual capacity, there is a lack of runtime support that takes advantage of all the benefits of a heterogeneous memory system while being as user-transparent as possible. To this end, we present a memory-aware runtime system and programming paradigm called PEARS that enables HPC application programmers to write traditional HPC workloads while taking advantage of multiple benefits of a byte-addressable, heterogeneous memory system. We demonstrate PEARS abstractions for workloads of matrix-multiplication and a variety of stencils with different dimensionality while comparing performance to the standard baseline implementations of these workloads under various memory allocation scenarios.

CCS CONCEPTS

• **Computer Systems Organization** → **Architecture**; • **Computing methodologies** → **Massively parallel and high performance simulations**.

KEYWORDS

Optane DC, NVM, persistent memory, heterogeneous memory, Runtime system, static analysis, dynamic scheduling

ACM Reference Format:
Onkar Patil[1], Frank Mueller[1], Latchesar Ionkov[2], Jason Lee[2], Michael Lang[2]. 2021. PEARS: A Performance-Aware Static and Dynamic Framework for Heterogeneous Memory. In *The International Symposium on Memory Systems, Oct., 2021, Virtual Conference.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3488423.3488428

1 INTRODUCTION

Heterogeneous memory systems are becoming more prominent in HPC deployments. These memory systems are comprised of multiple memory technologies like HBM, NVM and traditional DRAM. With multiple processing units integrated into HPC nodes, the memory architecture of these compute devices also becomes part of the memory space that an application utilizes while exploiting heterogeneous compute systems. Intel's Knights Landing (KNL) introduced HBM along with traditional DRAM-based main memory [29]. Graphics devices utilizing the General Purpose Graphics Processing Units (GPGPU) have also been equipped with HBM [22]. Fujitsu uses a hybrid memory cube (HMC) for its A64FX ARM-based chips to deliver high bandwidth memory access to all the compute cores [12]. Recently, Intel launched their Optane DC Persistent Memory Modules (PMM), which are NVDIMMs that can be used as byte-addressable, non-volatile main memory (NVM) [14]. The Aurora supercomputer [3] will have support for Intel Optane DC PMMs. All these memory technologies differ from each other in terms of access latency and/or memory bandwidth provided to compute devices. HBM has higher bandwidth than DRAM due to a wider data access path whereas NVM is slower than DRAM in terms of latency. They also differ in terms of the memory density and their respective energy consumption.

Several software packages support these memory technologies. The Memkind library [8] provides support for memory management on HBM devices. Intel offers the PMDK library [31] that supports memory management on NVM devices. There are other software systems like Atlas [9] and Mnemosyne [33] that provide persistent memory programming abstractions for NVM devices. However, these software systems are not aware of the performance characteristics of specific devices they utilize. They neither have memory bandwidth nor capacity awareness of the memory devices they operate on. The tool hwloc [19] provides information on the performance characteristics of the memory system but this information is not available directly at runtime to the application. The burden of understanding performance impact of the underlying devices for every allocation and data movement is on the application programmer. This makes application design more complicated, i.e., applications often fall short in extracting the potential benefits from a heterogeneous memory system. Also, application programmers lack support for utilizing different memory devices using a single library API.

Today, all memory devices can be combined into a single flat byte-addressable address space, among other configuration options. Applications can allocate memory on any of the devices, however, due to the lack of performance and capacity awareness they choose association with specific memory regions during computational phases to optimize performance. This can cause load imbalance, particular for allocations crossing boundaries between disjoint memory devices, due to the inverse proportionality between memory bandwidth and memory capacity in a heterogeneous memory system. To remedy this problem, applications would require scheduling strategies assisting in balancing compute and memory resources in order to exploit the potential of higher memory bandwidth and higher capacity provided in heterogeneous memories. Currently, software systems lack such capabilities.

This work contributes a user-transparent framework for managing memory allocations/de-allocations and data movement for HPC workloads that utilize a heterogeneous memory system. Our framework provides a **PE**rformance-**A**ware **R**untime system with support for **S**tatic memory characterization (***PEARS***). It aims to fill the gaps in software support for utilizing heterogeneous memory systems for HPC workloads and relieve the programmer from complex application design. PEARS has the following features:

- static memory characterization;
- macro-based programming support for specific HPC workloads;
- a memory management system with performance awareness; and
- a runtime system to schedule data on multiple memory devices.

In this paper, we will present the details of our implementation of PEARS and an experimental evaluation of memory and compute-bound workloads on a heterogeneous memory system. In Section 2, we present the related work. We describe the heterogeneous memory architecture in Section 3 and present the design details of our framework in Section 4. We present our experimental setup in Section 5 and analyze the results in Section 6 followed by our conclusions in Section 7.

2 RELATED WORK

Recent studies after the launch of Intel's Optane DC PMMs evaluate its characteristics under different memory architecture configurations. Yang et al. and Izraelevitz et al. [15, 34] evaluated the read and write memory access characteristics of Optane DC PMM for different file-systems, database applications and performance benchmarks. They found that Optane DC improves the performance of file systems and database applications due to lower latencies than storage devices. Patil et al. [23] characterized the performance of a DRAM/NVM hybrid memory system for HPC applications. They measured the bandwidth performance and energy characteristics of HPC applications runs on Optane DC compared to pure DRAM and DRAM as cache for Optane DC. They also proposed a symbiotic hardware and software prefetching technique for DRAM-NVM hybrid memory system [24]. Peng et al. [25] evaluated Optane DC PMMs in all the configurations available and also measured the performance of separating read and write allocation on a DRAM/NVM memory system. Psaropoulos et al. [27] provided latency hiding for the difference between Optane DC and DRAM for database applications. All the above works focus on evaluating the basic performance characteristics of Optane DC under various execution contexts and workloads.

There have been several works that help manage a complex memory hierarchy. Leon et al. [19] proposed an interface to help manage the memory system complexity. It comprised of a set of memory attributes and an API to express and manage the diverse memory characteristics using high-level metrics. Oden et al. [20] propose a flexible memory allocator for complex memory architectures. There are other works that have focused on runtime based scheduling memory movement and management for different memory systems. Chandrashekar et al. [10] proposed a memory-heterogeneity-aware runtime system that guides data prefetch and eviction for a HBM-DRAM memory architecture. Alvarez et al. [1] similarly proposed a runtime approach to transparently manage stacked DRAM memories in task-based programming models. Sánchez et al. [28] proposed techniques at the runtime system level to mitigate the impact of NUMA effects on parallel applications' performance by leveraging runtime system metadata expressed in terms of a task dependency graph. Perarnau et al. [26] evaluated the performance implications of a scheme based on a software-managed scratchpad with coarse-grained memory-copy operations migrating application data structures between memory hierarchy levels. Benoit et al. [4] presented a realistic performance model to execute scientific workflows on high-bandwidth memory architectures such as Intel's Knights Landing. There have been several works that proposed static memory management, e.g., Greenspan et al. [13] combined a custom LLVM Pass with a custom C library to automatically handle memory allocations performed by function calls without the need for programmer input or hardware/OS level changes. Khaldi et al. [16] introduced a new LLVM analysis called Bandwidth-Critical Data Analysis (BCDA) to decide when it is beneficial to allocate data in High-Bandwidth Memory (HBM) and then transform allocation calls into specific HBM allocation calls to increase performance in parallel systems. Our work focuses on characterizing the memory architecture statically and using the heuristics to actively engage in dynamic memory management activities during runtime for given HPC workloads.

3 SYSTEM ARCHITECTURE

The system used in experiments is a single HPE Proliant DL360 node (see Table 1) with 2 CPU sockets equipped with Intel's Xeon 8260 (code-named Cascade Lake). Each chip has 24 cores with a clock frequency of 2.4 GHz. Each core has 2 processing units under hyperthreading for a total of 96 CPUs. Each core has a 32 KB private L1 instruction cache, a 32 KB private data cache, and a private 1 MB L2 cache. There is a 35.75 MB L3 cache shared between all cores. It has a DTLB cache with 64 entries, which is 4-way set associative.

Each socket has 12 DIMM slots. 6 of the slots are occupied by 16 GB DDR4 DRAM modules and the other 6 slots are occupied by 128 GB Optane DC modules for a total of 192 GB DRAM and 1.5 TB NVM. The node has 4 memory controllers in total, two are connected to 6 DRAM DIMMs each, and the other two, known as iMC, are connected to 6 NVDIMMs each. The processor uses the standard DDR4 protocol on the regular DRAM memory controller and the DDR-T protocol for Optane DC on the i-memory controller (iMC). Using this proprietary extension of the protocol, the Optane DC features asynchronous command/data timing and variable-latency memory transactions.

Table 1: Experimental Platform

Specifications	Optane Node
Model name	Intel(R) Xeon(R) 8260 @ 2.40GHz
Architecture	x86_64
CPUs	96
Sockets	2
Cores per socket	24
NUMA nodes	4
CPU MHz	3100
CPU max MHz	3900
CPU min MHz	1000
L1d cache	32 KB
L1i cache	32 KB
L2 cache	1 MB
L3 cache	35.75 MB
Memory Controllers	4
Channels/controller	6
DIMM protocol	DDR4
DRAM size	192 GB
Max. DRAM BW	104 GB/s
NVDIMM protocol	DDR-T
NVRAM size	1.5 TB
Max. NVRAM BW	40 GB/s
No. of nodes	1
Interconnect	No
Operating System	CentOS 7
Compiler	GCC 9.3.0

Table 2: Optane DC operation modes

Operation mode	Functionality
Memory mode	Optane DC PMMs act as volatile, byte-addressable main memory. DRAM acts as a cache for Optane DC and is not visible to the user
App Direct mode	Optane DC PMMs act as persistent storage separate from the primary memory hierarchy. Managed by file systems installed on it. DRAM acts as main memory
Mixed mode	Part of Optane DC PMMs can be used as main memory and the remaining part can be used as persistent storage. DRAM acts as cache for Optane DC
Flat mode	DRAM and Optane DC PMMs are part of the same address space and can be used as heap memory

Optane DC has an on-DIMM Apache Pass controller that handles memory access requests and the processing required on NVDIMM. The on-DIMM controller internally translates the addresses of all access requests for wear-leveling and bad-block management. It maintains an address indirection table on-DIMM that translates the DIMM's physical addresses to an internal device address. The table is also backed up on DRAM.

Accessing data on Optane DC occurs after the translation. The controller translates 64 byte load/stores into 256 byte accesses due to the higher cache line access granularity of Optane DC, which causes write amplification [15]. Optane DC PMM can operate in different modes: (1) As an uncached byte-addressable memory (flat mode), (2) as DRAM cached main memory (Memory mode), or (3) as a block storage device (App-Direct mode). All modes (except for Flat) are provided by Intel. Flat is a custom mode introduced by patching the OS kernel to identify all DIMMs as DRAM, thereby creating a true hybrid memory address space. All our experiments are performed on the flat mode. The modes are described in Table 2.

3.1 SICM

The SICM (Simple Interface Complex Memory) [18] library provides an interface to allocate memory on different memory devices available on a given compute node. It is a bare-metal library that utilizes NUMA and jemalloc internally [2] to create arenas where memory can be allocated. The arenas can be moved between the different memory devices. The library has a two-tiered API approach: The high-level API gives a coarse grained control over the memory management of a heterogeneous memory system and the low-level API gives total control over memory management. SICM is an integral part of the software support required for our runtime solution. We utilize and extend the low-level SICM API to build and support our framework.

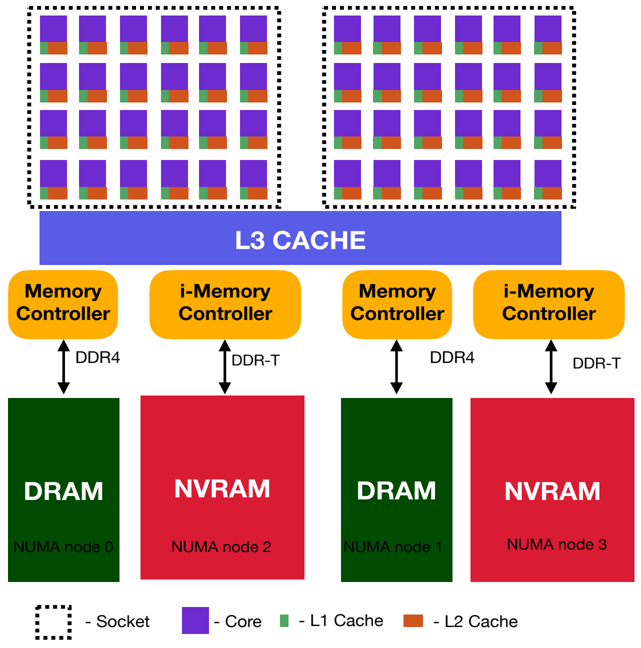


Figure 1: Heterogeneous memory architecture (flat mode)

4 THE PEARS SOFTWARE FRAMEWORK

PEARS has multiple components depicted in Figure 2. In combination, these components form a framework in support of memory management and scheduling for a heterogeneous memory system. We describe each component in detail in this section.

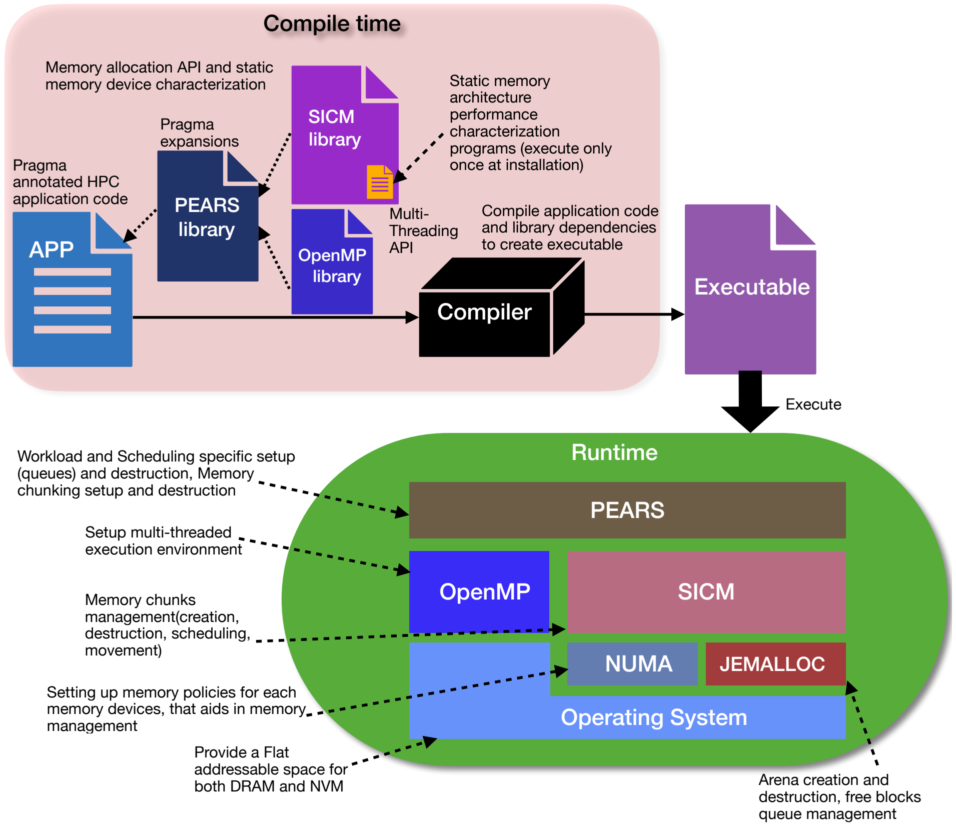
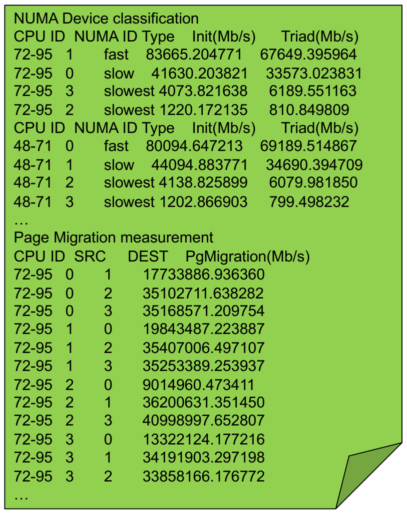

Compile time
Memory allocation API and static memory device characterization
Pragma expansions
Pragma annotated HPC application code
APP
PEARS library
SICM library
OpenMP library
Static memory architecture performance characterization programs (execute only once at installation)
Multi-Threading API
Compile application code and library dependencies to create executable
Compiler
Executable
Execute
Runtime
PEARS
OpenMP
SICM
NUMA
JEMALLOC
Operating System
Workload and Scheduling specific setup (queues) and destruction, Memory chunking setup and destruction
Setup multi-threaded execution environment
Memory chunks management(creation, destruction, scheduling, movement)
Setting up memory policies for each memory devices, that aids in memory management
Provide a Flat addressable space for both DRAM and NVM
Arena creation and destruction, free blocks queue management

Figure 2: PEARS: Block and flow diagram of static and dynamic components



```
NUMA Device classification
CPU ID  NUMA ID Type    Init(Mb/s)       Triad(Mb/s)
72-95   1       fast    83665.204771    67649.395964
72-95   0       slow    41630.203821    33573.023831
72-95   3       slowest 4073.821638     6189.551163
72-95   2       slowest 1220.172135     810.849809
CPU ID  NUMA ID Type    Init(Mb/s)       Triad(Mb/s)
48-71   0       fast    80094.647213    69189.514867
48-71   1       slow    44094.883771    34690.394709
48-71   2       slowest 4138.825899     6079.981850
48-71   3       slowest 1202.866903     799.498232
…
Page Migration measurement
CPU ID  SRC     DEST    PgMigration(Mb/s)
72-95   0       1       17733886.936360
72-95   0       2       35102711.638282
72-95   0       3       35168571.209754
72-95   1       0       19843487.223887
72-95   1       2       35407006.497107
72-95   1       3       35253389.253937
72-95   2       0       9014960.473411
72-95   2       1       36200631.351450
72-95   2       3       40998997.652807
72-95   3       0       13322124.177216
72-95   3       1       34191903.297198
72-95   3       2       33858166.176772
…
```

Figure 3: Example of the classification and measurements from the Memory characterization

4.1 Static Memory characterization

A heterogeneous memory system is comprised of multiple memory devices that differ in the underlying technology, e.g., DRAM, Phase Change Memory (PCM) or High Bandwidth Memory (HBM). These devices have different characteristics in terms of memory access latency, bandwidth, power consumption, capacity and persistence. Each characteristic can provide a unique advantage to the heterogeneous memory system when combining two or more of the above mentioned technologies. It is critical to identify and associate each memory device of the heterogeneous memory system with its specific characteristic. Currently, this association is unavailable at the BIOS or OS level. We simply see each memory device as a separate NUMA node at the user-level. Hence, there is a need to characterize each NUMA node in terms of its specific attributes and identify the underlying memory device.

To this effect, we devised a set of programs and scripts that run micro-benchmarks to measure, analyze and associate memory characteristics to each NUMA node of the system. We use three micro-benchmarks that measure the average write-only stream bandwidth, the average triad (1 write, 2 reads) stream bandwidth and the page migration bandwidth. The kernels comprising the micro-benchmarks feature sequentially-accessed streams, i.e., there is negligible cache locality, which provides an upper bound on the memory performance for a given access pattern. These micro-benchmarks are parallel workloads that use OpenMP to assess multi-threaded performance. We allocate stream buffers of 1 GB on each NUMA node separately and then execute the kernels for every NUMA node using the respective buffers. We measure the execution time for each kernel and then calculate the effective data bandwidth for them. The measurements are averaged over 10 runs and then recorded separately for each NUMA node. We run the benchmark separately for each core-NUMA node mapping using “numactl -m” and record the measurements. We use K-means clustering on the triad stream bandwidth measurements for all the NUMA nodes and core-NUMA node mappings before classifying the clusters as fast, slow and slowest memory devices with an assigned number from 0 to n, where n is the number of clusters and 0 is the fastest memory device. We choose this generalized classification in order to support future memory devices as it provides a more fine grained classification of the memory devices.

These micro-benchmarks and the associated scripts are part of our SICM library extensions. They are executed only once during the installation of SICM. All classifications and measurements are passed to the SICM library at runtime as a configuration file. An example of this file is depicted in Figure 3. These classifications and measurements are then used by the extended SICM library at runtime to perform memory management and scheduling.

4.2 Performance-Aware Memory management

We use the SICM library for low-level memory management. SICM provides support for creating arenas, which is a concept extended from jemalloc. The arenas help group memory pages in an encapsulation, i.e., they can be allocated on a specific memory device or be moved between memory devices. The arenas are moved between memory devices using the NUMA API, which help select the device they reside on. However, SICM lacks awareness of performance attributes for the underlying memory devices in a heterogeneous memory system. By complementing the framework without static memory characterization (described above), we bring performance awareness to SICM. Each NUMA node belongs a particular class of memory based on its performance.

During the initialization of SICM, we obtain the memory classification from the configuration file. We acquire information on the capacity of each device from the NUMA device files in the Operating System (OS). We extend the SICM API to allow the user to specify the desired class of memory (fast, slow, slowest) for any given allocation. The memory nodes for each class of memory have a fixed preference order. When the first preferred device of the specified class does not have enough memory, allocation moves on to the next preferred device. If no device in the desired class has enough memory for the new allocation, then the next slowest class is selected for memory and allocation. We also extend the allocator API to accept a group id to associate related allocation to each other. We create a new arena for each group specified by the user. The metadata (arena, group id, start address of an allocation, class of memory, NUMA device id, size) about a given allocation is stored in a linked list within SICM and utilized later during its life cycle. The metadata is destroyed after a particular arena is de-allocated. We also extend the SICM API to move memory allocation between different classes of memory. Memory is moved based on the availability of space within the devices of a given memory class. If a class were to run out of memory, the allocation is not moved. Any metadata of a particular allocation is updated accordingly.

We also extend the SICM API to support various utility functions that are used internally, e.g., to check the available capacity of a given device, to query the group id of an allocation etc. The SICM extended API can be used on its own for any kind of workload/application but requires specific knowledge on the programmers part, i.e., it is primarily designed in support of frameworks that establish higher-level memory abstractions.

4.3 Macro-based Programming support for HPC workloads

To utilize our framework, we depend on the SICM library to perform memory management and then orchestrate the runtime scheduling of tasks. This involves initialization and finalization of many supporting data structures that maintain the metadata for memory management. The framework also adds additional buffers to the data structures allocated by the user to maintain logical consistency of the workloads. Hence, the framework provides targeted support for a set of HPC workloads. Adding the respective API calls to achieve the runtime support as described above would require significant changes to the source code of existing HPC applications. It will also hinder development of new HPC applications that utilize kernels as the programmer would have to consider many factors affecting performance due to memory placement of data structures.

To facilitate heterogeneous memory management, we provide a macro-based approach in support of our framework. We define workload-specific macros that perform the initialization and finalization of the framework while enabling the programmer to focus strictly on the workload kernel. The macros expand into low-level API calls to the extended SICM library for initialization, memory management, runtime scheduling and consistency, finalization and OpenMP pragma support. The macros provide an easy methodology to program with complex memory systems while giving the programmer control over the runtime scheduling of the workload kernels.

```
int main(int argc, char *argv[]){
   …
BoT_TMM_INIT(row, col, matrix_size, no_of_procs) //Initialization
CHUNKED_ALLOCATE(&a, SICM_DOUBLE)     //Allocation
CHUNKED_ALLOCATE(&b, SICM_DOUBLE)
CHUNKED_ALLOCATE(&c, SICM_DOUBLE)
BoT_PROLOGUE(3,a,b,c)
for(i = 0; i < CHUNK_ROWS; i++){  //Initialization kernel
 for(j = 0; j < CHUNK_COLUMNS; j++){
   a[SEQ_ACCESS][i*CHUNK_COLUMNS+j] = (double)(rand())    \
                                    /((double)(RAND_MAX)+1);
   b[SEQ_ACCESS][i*CHUNK_COLUMNS+j] = (double)(rand())    \
                                    /((double)(RAND_MAX)+1);
   c[SEQ_ACCESS][i*CHUNK_COLUMNS+j] = 0.0;
 }
}
BoT_EPILOGUE(3,a,b,c)
   …
BoT_PROLOGUE(3,a,b,c)  //Prologue
TMM_KERNEL_LOOP
 for(i = 0; i < CHUNK_ROWS; i++){        //Main kernel
  for(j = 0; j < CHUNK_COLUMNS; j++){
   double tmp=0.0;
    for(k = 0; k < CHUNK_ROWS; k++){
     tmp += a[ROW_ACCESS][(i*CHUNK_COLUMNS) + k]  \
           *b[COL_ACCESS][(k*CHUNK_COLUMNS) + j];
    }
    c[SEQ_ACCESS][(i*CHUNK_COLUMNS) + j] += tmp;
   }
  }
BoT_TMM_EPILOGUE(3,a,b,c)  //Epilogue
   …
CHUNKED_DEALLOCATE(&a, SICM_DOUBLE)  //De-allocation
CHUNKED_DEALLOCATE(&b, SICM_DOUBLE)
CHUNKED_DEALLOCATE(&c, SICM_DOUBLE)
BoT_TMM_FINALIZE()  //Finalization
}
```

Figure 4: Sample code of tiled matrix multiply with PEARS macros

The runtime scheduling macros are to be used to delimit the target kernel, i.e., as a prologue and an epilogue to the kernel. The initialization and finalization macros encompass the entire program. An example of how the code shape with our macros extensions is given in Figure 4. The macros also provide support for adjusting loop limits and identifying access patterns. We support accessing the memory chunks in sequential or row- or column-order based on the kernel access pattern. Currently, we provide macro-based programming support for tiled matrix multiplication as well as one-, two, and three-dimensional (1D/2D/3D) stencil kernels.

4.4 Runtime Scheduling Support for heterogeneous memory system

In a heterogeneous memory system, memory performance and capacity of a given device are inversely proportional to each other.

Fast memory has low capacity while slow memory tends to be larger in capacity, sometimes by almost an order of magnitude. The slower memory supports in-core computation for larger problem representations but also hampers performance due to slower access latency besides the simple cost of accessing larger problem sizes. Application performance can suffer significantly as it is bounded by the performance of the slowest memory device. This problem is particularly aggravated if allocated memory remains in the same memory device throughout the execution of the entire application. To remedy this, we provide a runtime data movement scheduling mechanism through which we ensure that both the capacity of the heterogeneous system and the memory performance of the fastest memory device in the system are striking a balance.

Table 3: Statically Fixed chunked sizes in (MB)

Chunk size	TMM	2D dataset	3D dataset
F1	0.5	128	512
F2	1	256	1024

We achieve this by dividing a given allocation into multiple memory chunks. Each chunk is allocated in a separate arena and the chunks are distributed across all memory classes based on the capacity ratios of each memory class. The workload kernel iterates over all chunks to perform the desired computation. Chunks are subjected to parallel execution within an OpenMP parallel region. Right before and right after a scheduled chunk is executed, its associated memory arenas can be moved between memory devices. We move each memory chunk (arena) using the SICM API, which utilizes the mbind() system call to select the memory policy. The objective during execution is to allow the maximum number of memory chunks to utilize the fastest memory device in the system while balancing the workload for all participating threads, even when some of them are subjected to chunks on different memory devices. To this end, we provide two memory scheduling techniques that a programmer may select, namely, the “Inspector-Executor” and the “Bag of tasks” methods. A simple change in macro prefix name selects the scheduling technique, e.g., the `BoT` prefix denotes Bag of tasks in the code snippet in Figure 4, which can be changed to `IEC` for Inspector-Executor.

The Inspector-Executor method first uses the main thread as an inspector thread that assigns a small set of the total memory chunks to each thread (including itself) subject to computation within the kernel. Once executed, memory chunks with computational results are moved to the slowest memory device. After a chunk was operated on during execution, the main thread then assigns more chunks to each thread based on their order of completion and chunk throughput. The earliest finishing thread receives proportionately more chunks to process based on its throughput, and these chunks are moved to the fast memory device. The last finishing thread receives fewer chunks and moves them to the fast memory device only if capacity is available. If any memory chunks remain to be processed after this, then each thread that finishes computation on its assigned memory chunks can grab a set of the remaining chunks. Such a thread then iterates over new chunks while also moving them to a fast memory device based on the availability of capacity. Threads exit the parallel region once all the memory chunks have been processed.

In the Bag of tasks method, each thread starts with a small subset of the total number of memory chunks. After processing them, the threads move the memory chunks to the slowest memory device and grab more chunks on a first-come-first-served basis. An earliest arriving thread has a higher probability of being assigned more memory chunks, which it can move to the fast memory device based on the availability of capacity. Again, the threads exit the parallel region once all the memory chunks have been processed.

Data movement ceases if all memory chunks can fit inside fast memory. A critical part of the runtime scheduling support is the creation of memory chunks, the number of chunks, the size of each chunk, as well as consistency and atomicity of the data distributed across these chunks. We provide two methods determine chunking granularity, i.e., the size of chunks and their total number.

Method 1: The chunk size is determined by the problem size, i.e, the number of memory regions within a kernel, the allocation size of each kernel, and the capacity ratios of the fast, slow and slowest devices.

Method 2: The chunk size is fixed based on the private cache size of each core (see next section for implementation choices) in the system. Internally, the macros issue the necessary API calls that determine memory chunking and allocation with respect to device types. To ensure deterministic and logically correct accesses of every data element of a memory chunk, specific macros are provided. As data is partitioned across memory chunks, we further provide algorithmic support for a given workloads. The macros for tiled matrix multiplication ensure that the row-wise and column-wise access of data across chunks is consistent and semantically correct.

Stencil operations are supported by providing halo value communication across memory chunks via buffering each memory chunk and partitioning the data logically according to its dimensionality. Due to this, we incur a memory and communication overhead for 1D, 2D and 3D stencil workloads. For 1D and 2D stencil workloads, the memory overhead for an n^2 matrix is

$$4h \times (\frac{n}{m} + 1) \times m$$

, where h is the size of the halo, n is the dimension of the matrix and m is the number of memory chunks. Similarly, for an n^3 matrix, the memory overhead is

$$4h \times (3\frac{n}{m} + 2h) \times m$$

, where h is the size of the halo, n is the dimension of the cube and m is the number of memory chunks.

5 EXPERIMENTAL SETUP

We measure the effectiveness of our memory management and runtime scheduling solution in a set of experiments on our heterogeneous memory system described in Section 3. The objective of our experiments is to assess the performance impact of our memory management and runtime scheduling methods under different configurations for critical HPC kernels are used in many applications.

We developed custom benchmarks for a set of representative HPC kernels including dense tiled matrix multiplication (TMM), 3- and 7-point 1D stencil kernels, 5-, 9-point, and 9-point with diagonal values 2D stencil kernels, and 7-, 13- and 27-point 3D stencil kernels. The TMM kernel consists of 5 2D matrices that perform multiplication in two stages. First, the product of two matrices $C = A \times B$ is calculated, second, its result matrix is used in another multiplication, $E = C \times D$. All stencil kernels consist of a write stream and a read stream of a 2- or 3-dimensional dataset of linearly laid out arrays accessed in row-major order.

The stencil codes are implemented as Jacobi iterative kernels, which are common in Computational Fluid Dynamics (CFD) applications, Partial Differential Equations (PDEs), and pointular automata [11]. Some examples of stencil code-based HPC applications are Vector Particle In Cell (VPIC) [5–7] and Algebraic Multi-grid (AMG) [35], which are compute- and memory-bound applications, respectively. The 3-, 5-, 7-, 9- and 13-point stencils use the Von Neumann neighborhood, whereas the 7-point (1D), 9-point with diagonal values and 27-point stencils use the Moore neighborhood [17]. The 3-point stencil is a 1D stencil, where for every iteration the previous element and the next element are read along with the current one. Another 7-point 1D stencil reads the previous 3 elements and the next 3 elements along with the current one per iteration. The 5-point stencil is a 2D stencil, where along with adjacent elements in the same row of the current element, adjacent elements in the same column of the current element are also read. The 9-point stencil has a similar shape to the 5-point stencil but it access twice as many adjacent elements in the same row and column. The 9-point stencil is a 2D stencil including diagonal elements beyond the 5-point stencil. The 7-point stencil is a 3D stencil, where along with the adjacent elements in the same row (x) and column (y) of the current element, adjacent elements in the next and previous plane (z) are read. The 13-point stencil is also a 3D stencil that accesses twice as many adjacent elements than the 7-point stencil along each dimension. Similarly, the 27-point stencil is a 3D stencil with diagonals on every dimensional pair beyond the 7-point stencil. These stencils comprise one or more read streams, plus a write stream accessed sequentially. We run each benchmark under multiple memory allocations and runtime scheduling policies. The different memory allocation policies used are DRAM-only allocation, NVM-only allocation, memory chunking with a dynamically determined chunks size and memory chunking with a statically fixed chunk size. The DRAM and NVM-only allocations are realized via numa_alloc_onnode(). Memory chunks are allocated using the macro-based programming API of PEARS, with the SICM library underneath. We compare our memory chunking management over the above allocation policies to compare the performance of distributed memory allocation across heterogeneous memory devices with a homogeneous memory allocation policy. We also assess dynamic and fixed chunk sizes with their respective macro variants. Dynamic chunk sizes are determined by the problem size of the HPC kernel and the capacity ratios of all the memory devices at runtime. Static chunk sizes are fixed relative to the L2 and L3 cache sizes and the maximum chunking that is supported on the system for matrix multiplication, 2D datasets and 3D datasets. Fixed chunk sizes for each benchmark are described in Table 3.

The different runtime scheduling policies used in our experiments are regular OpenMP for loop-based work sharing (referred to as DRAM and NVM based on allocation), the collapse clause of OpenMP (referred to as DRAM-collapse and NVM-collapse, based on the allocation), our Inspector-Executor (IEC) model and our Bag of tasks (BoT) model. The BoT and IEC models use the PEARS framework. The OpenMP collapse [21, 30] clause specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause over the number of available threads. Any sequential execution of the iterations in associated loops determines the order of the iterations in the collapsed iteration space. We compare with this scheduling clause because the concurrency of the iteration of the all memory chunks together as a whole determine the order of distribution of all memory chunks (iteration partitioning) among all threads in BoT and IEC models. This is similar to the collapse clause.

We use OpenMP for all our experiments. Benchmarks are compiled using -O3 optimization with GCC 9.3.0. The maximum problem sizes of our experiments are limited by the DRAM size of the system (192 GB). We measure the total execution time for each benchmark using clock_gettime() and average over 5 runs for each configuration with a standard deviation of no more than 3%. To obtain cache performance and memory performance metrics, we use LIKWID [32] to measure metrics for every benchmark that are obtained from hardware performance counters. We perform strong and weak scaling for all our benchmarks and scaled the number of OpenMP threads from 2-32. We proportionally change problem sizes for each benchmark depending on the type of scaling. For TMM, the problem sizes varies from a 2048^2 element matrix to a 8192^2 element matrix. For 1D and 2D stencil benchmarks, we vary the problem size from a 16384^2 to a 65536^2 element matrix. For 3D stencils, we vary problem sizes from 820^3 to 2048^3 elements. For TMM without any chunking, the tiling size is fixed at 64^2 elements.

6 RESULTS

In the following, we present our experimental results and findings. We performed experiments for both strong and weak scaling and plot results in separate graphs. Each graph features box plots. We provide graphs for three different metrics, namely, the memory bandwidth, cache bandwidth and DTLB-load store miss ratio, which are plotted as bar charts with bandwidth (MB/s) on the left y-axis. Each graph indicates the number of OpenMP (OMP) threads on the x-axis.

We conduct two different sensitivity studies that are plotted separately. First, the different memory management and scheduling techniques are investigated in separate data series within the same plots. Second, different chunk sizes are analyzed in plots. For the memory bandwidth analysis, we provide (1) read, (2) write and (3) total bandwidth measured at the memory controllers using LIKWID. Similarly, for Cache Bandwidth analysis we provide (a) L1-L2 cache bandwidth, (b) L2-L3 cache bandwidth and (c) L3-System (Sys) bandwidth measured using the performance counters with the help from LIKWID. For DTLB miss ratio analysis, we provide (i) load and (ii) store store misses measured using LIKWID. We measure solve time of all our benchmarks, which encapsulates the entire application execution time, including the overhead of initialization

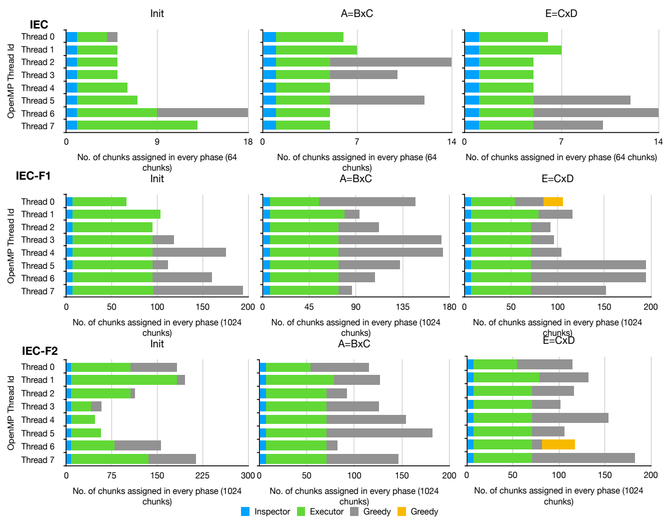


Figure 5: Memory chunk distribution of IEC scheduling for dynamic and static chunk sizes

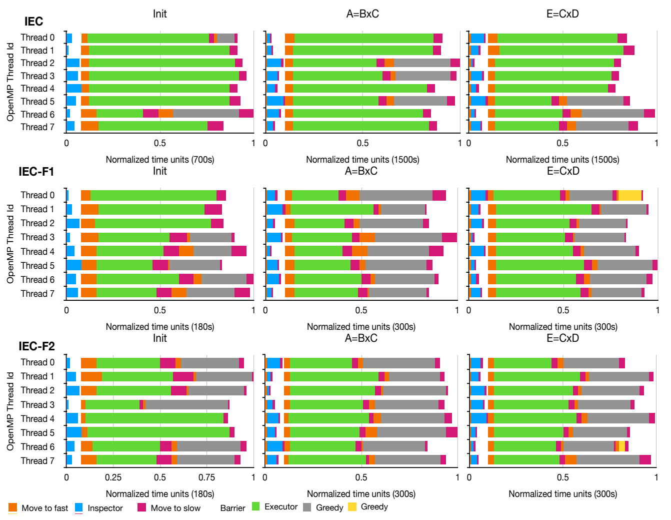


Figure 6: Time series distribution of IEC scheduling for dynamic and static chunk sizes

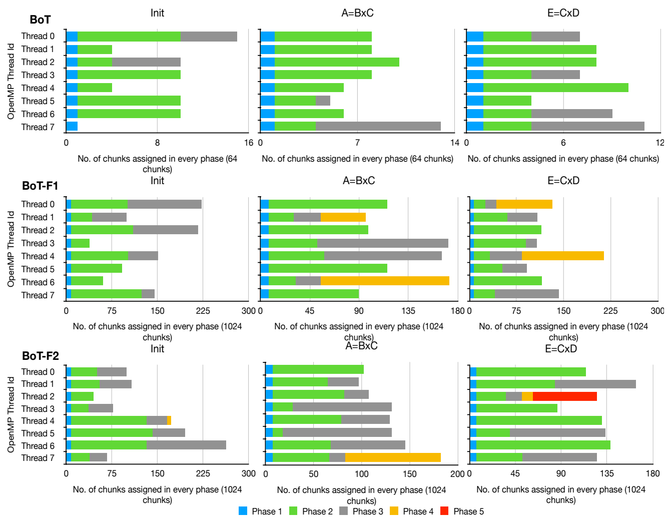


Figure 7: Memory chunk distribution of BoT scheduling for dynamic and static chunk sizes

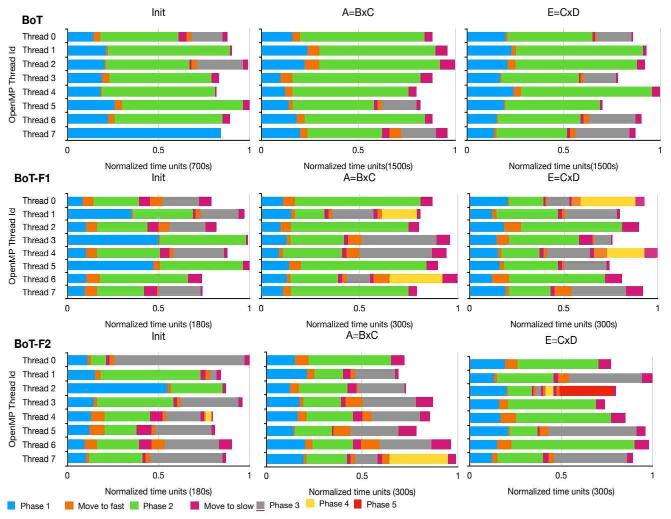


Figure 8: Time series distribution of BoT scheduling for dynamic and static chunk sizes

and finalization of the PEARS framework. Due to space limitations, we provide a subset of representative graphs from the vast dataset that was collected. Other graphs have similar trends, which are also discussed.

Figures 5 and 7 depict the memory chunk distribution graphs of our IEC and BoT runtime scheduling policies for static and dynamic chunk sizes with the TMM workload. A stacked bar graph with the x-axis shows the number of memory chunks assigned to any given thread. The y-axis indicates each individual thread from 1..8. Figures 6 and 8 depict the time series distribution for our IEC and BoT runtime scheduling policies for static and dynamic chunk sizes with the TMM workload. It is again depicted as a stacked bar chart with 8 OpenMP threads on the y-axis and normalized time on x-axis. The figures provide distributions for each of the three kernels of TMM (Init, A=BxC, E=CxD). For both IEC and BoT with dynamic chunking, the number of chunks created is lower than fixed chunk sizes (IEC-F1, IEC-F2, BoT-F1 and BoT-F2) due to the dependence on problem sizes and memory capacity. However, the workload distribution of of both IEC and BoT scheduling policies is not affected by the number and size of chunks. For IEC scheduling, a uniform memory chunk distribution is observed up till the executor phase for dynamic and fixed chunk size across all threads (Figure 5) and across all kernels. The Inspector phase distributes a very small fraction of the total memory chunks to each thread and, based on the completion of each thread, distributes the chunks for the Executor phase. This results in a quite balanced time series distribution (Figure 6) with some imbalance during the Executor

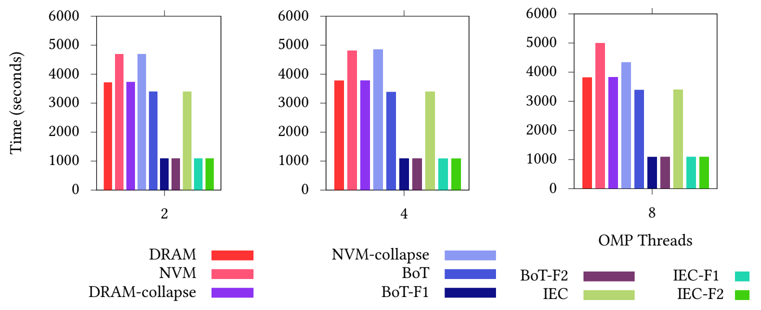


Figure 9: Solve Time comparison of scheduling and chunking policies for TMM with Strong scaling

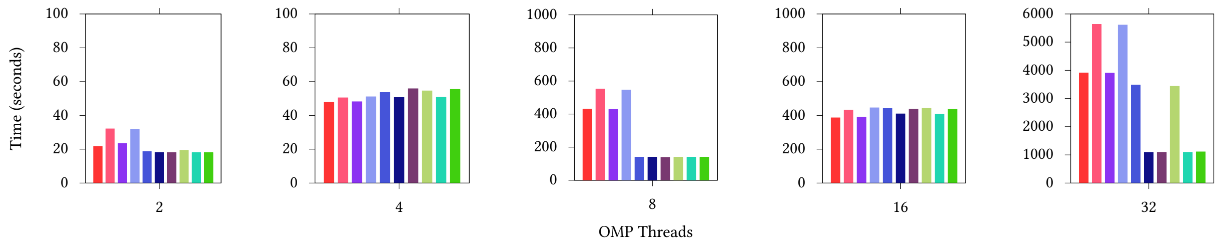


Figure 10: Solve Time comparison of scheduling and chunking policies for TMM with Weak scaling

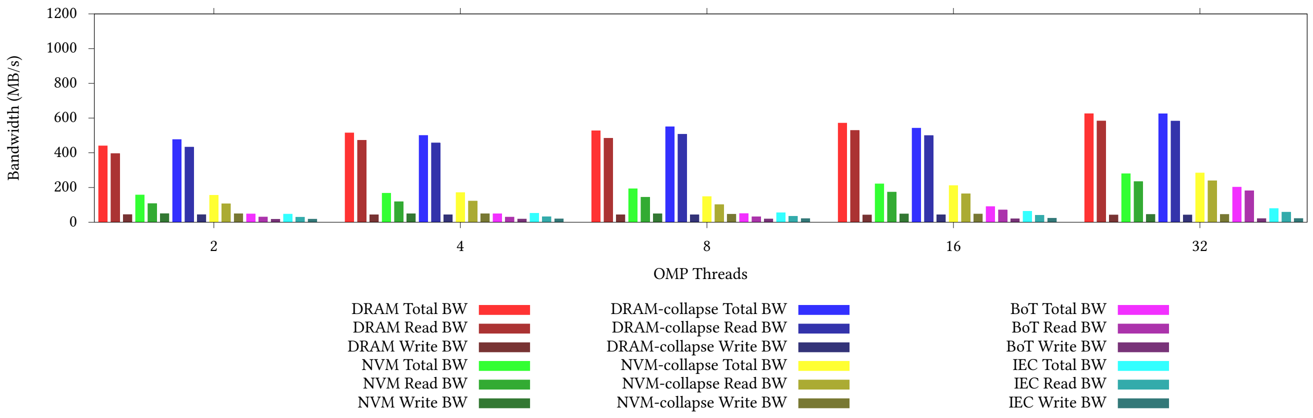
phase of IEC. However, the greedy phase helps in reducing the imbalance by distributing more chunks to the threads that finish early. These threads finish executing the chunks quickly on the faster memory while other threads are working on the slower memory chunks. For BoT scheduling, the tasks are greedy in grabbing a subset of total memory chunks right from the outset. In phase 1, BoT does not move any chunks from where they reside. Hence, for the Init kernel, we observe that a few threads take a lot of time to finish phase 1 as some of the memory chunks were residing in the slower memory device. However, as we move ahead in phases, the workload distribution becomes more balanced. The data movement overhead increases with larger subsets of memory chunks but it is compensated when the chunks are executed on the faster memory as seen for the A=BxC kernel. Both scheduling policies ensure that if a thread takes longer time to finish in the earlier stages, other threads move ahead in phases quickly to balance the workload as seen in IEC-F2 for E=CxD kernel.

Observation 1: *The runtime scheduling and memory management of PEARS outperforms other memory allocation and runtime scheduling methods for the TMM workload.*

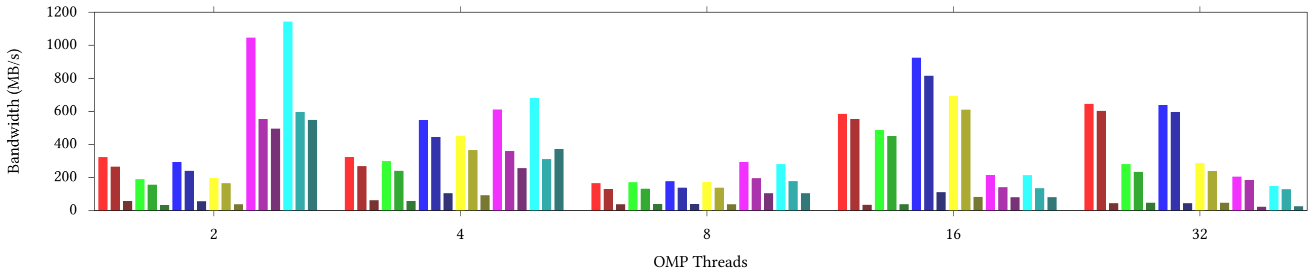
Figures 9 and 10 depict the time comparison of all scheduling and chunking policies and Figures 11a and 11b depict the memory bandwidth results for strong and weak scaling of the TMM benchmark. For Figures 9 and 10, we plot every OpenMP thread configuration separately due to the large difference in solve time over varying number of threads under weak scaling. We observe that, under strong scaling, BoT and IEC runtime scheduling outperform all other memory allocation and runtime scheduling methods. The execution times of BoT and IEC are identical, which are up to 40% faster than pure NVM allocations and up to 15% faster than pure DRAM allocations in execution time.

Under weak scaling, BoT and IEC are on par or faster than other runtime scheduling methods in terms of execution time. They achieve this in spite of having lower memory bandwidth than other methods, especially for larger numbers of threads. This can be attributed to the L2-L3 cache bandwidth of BoT and IEC for strong scaling and the L1-L2 cache bandwidth for weak scaling, as shown in Figures 12a and 12b, respectively. This indicates that there more cache hits are due to the memory chunking, which reduces the dependency on memory access latency. Figures 13a and 13b indicate higher load and store misses for BoT and IEC. This, along with lower memory bandwidth, means that both BoT and IEC runtime scheduling prefetch pages into the DTLB beforehand while utilizing all memory controllers simultaneously, which gives PEARS a performance boost over other memory allocation and runtime scheduling methods. For TMM weak scaling, the problem sizes grow linearly with the number of threads but the algorithm has a complexity of (O^3). Hence, the execution time still grows at a cubic rate.

Figures 14a and 14b depict the memory bandwidth for different memory chunk sizes under BoT and IEC scheduling. Both scheduling policies determine chunk sizes dynamically at runtime based on the problem size. In contrast, BoT-F1, IEC-F1, BoT-F2 and IEC-F2 impose statically fixed chunk sizes for comparison, where sizes are indicated in Table 3. We observe that fixed chunk sizes that align with the private cache achieve lower execution time, i.e., outperform dynamically determined chunk sizes in Figures 9 and 10. Both IEC and BoT result in almost 3x times faster execution with their

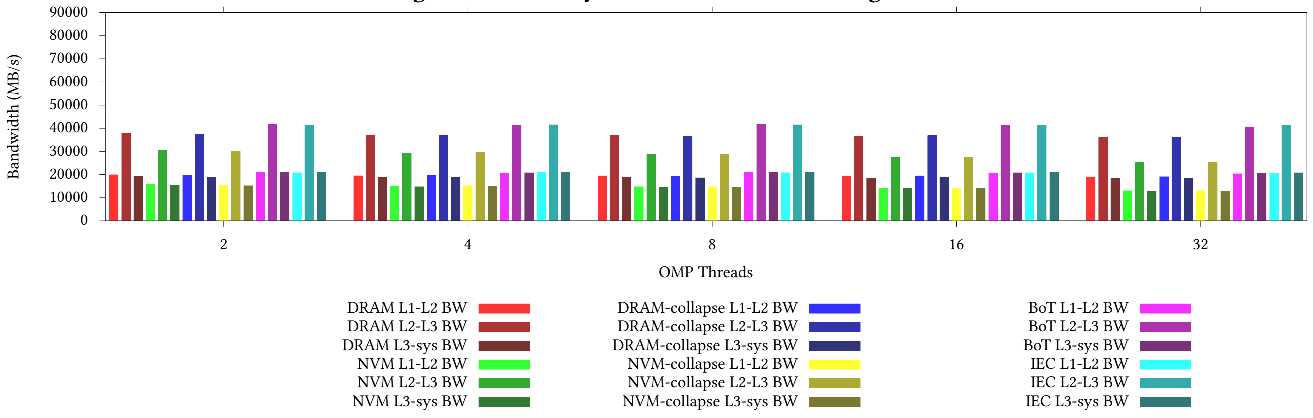


(a) Strong Scaling

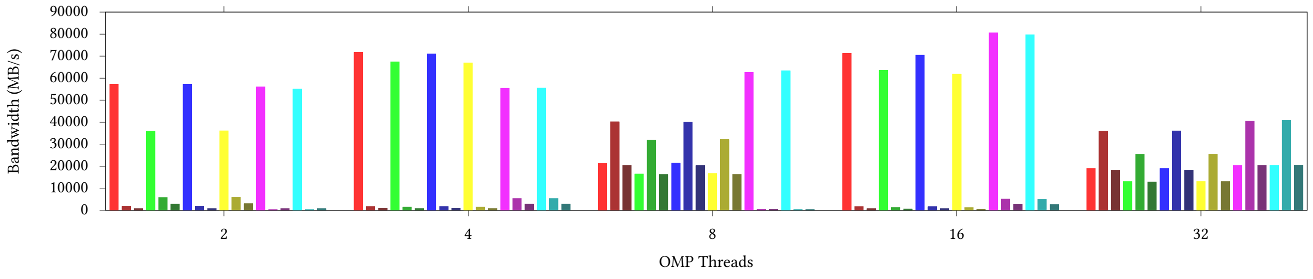


(b) Weak Scaling

Figure 11: Memory bandwidth for scheduling TMM

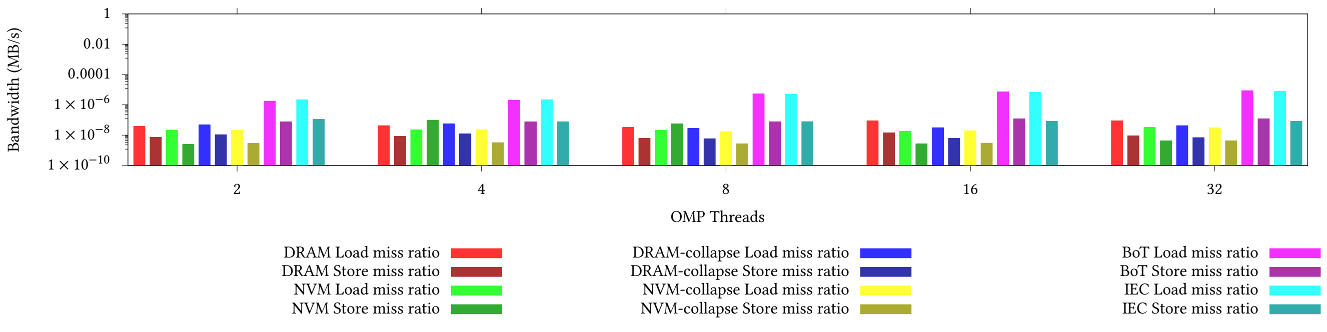


(a) Strong Scaling

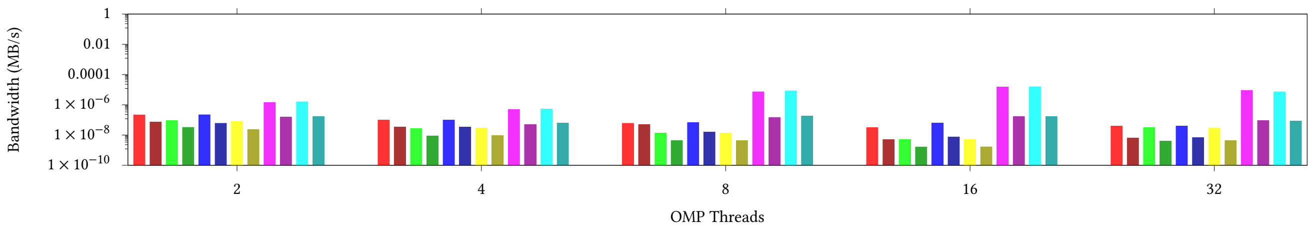


(b) Weak Scaling

Figure 12: Cache bandwidth for scheduling TMM

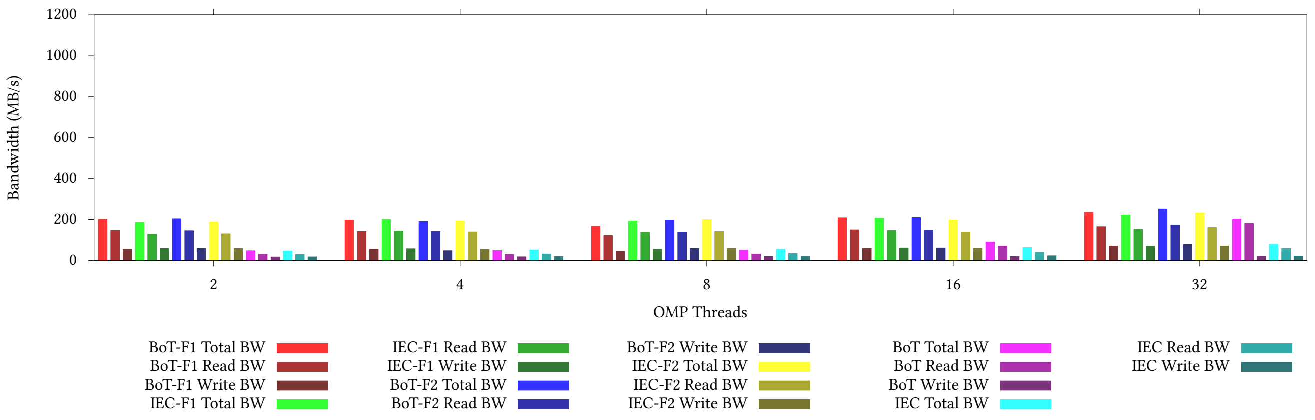


(a) Strong Scaling

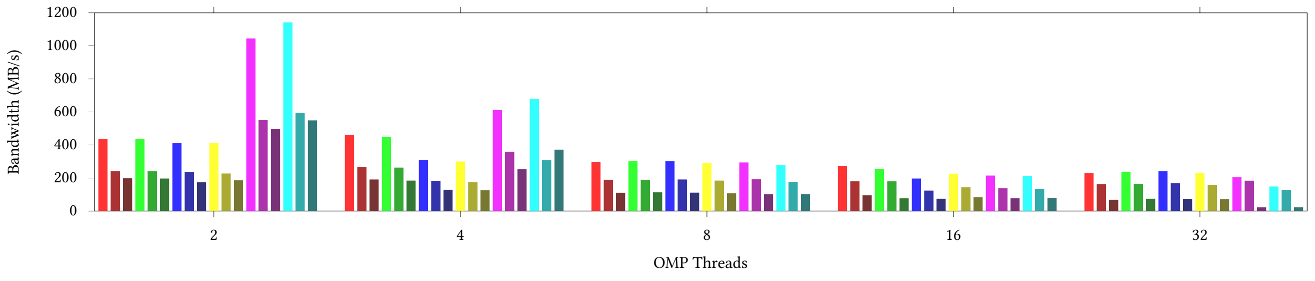


(b) Weak Scaling

Figure 13: DTLB load store miss ratios for scheduling TMM

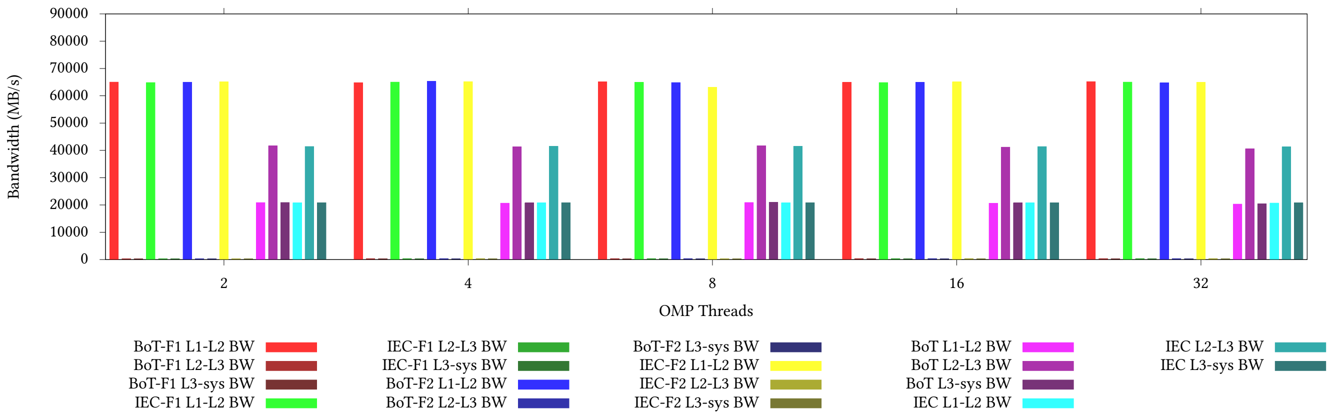


(a) Strong Scaling

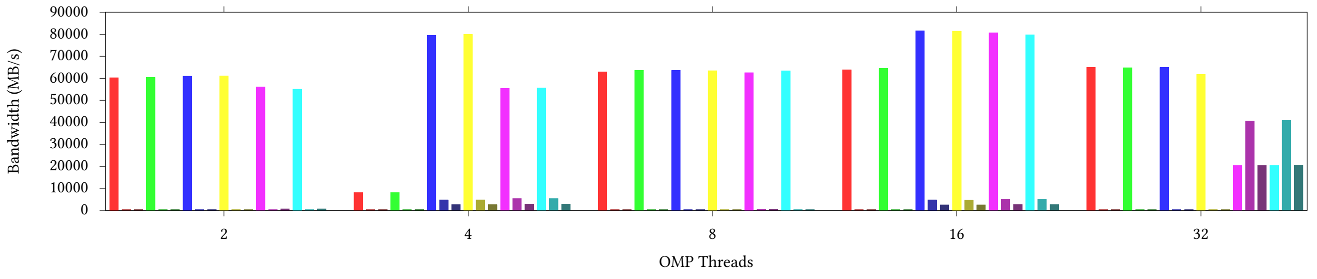


(b) Weak Scaling

Figure 14: Memory bandwidth over different chunk sizes for TMM

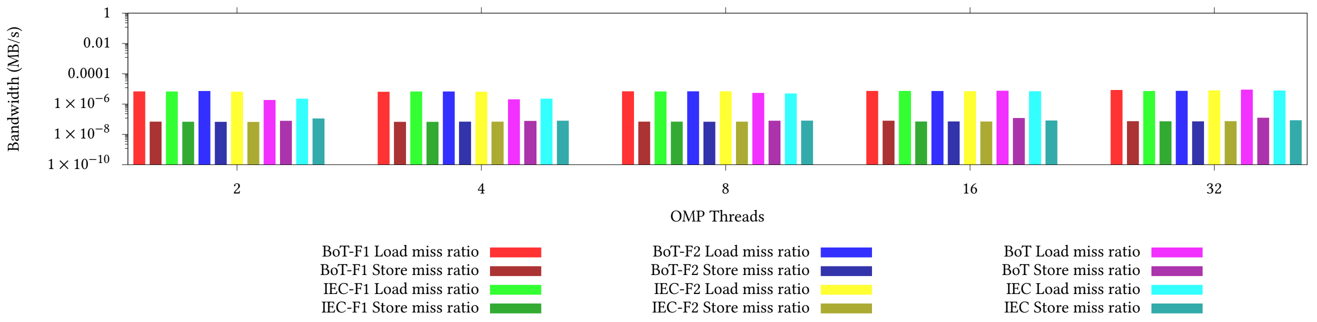


(a) Strong Scaling

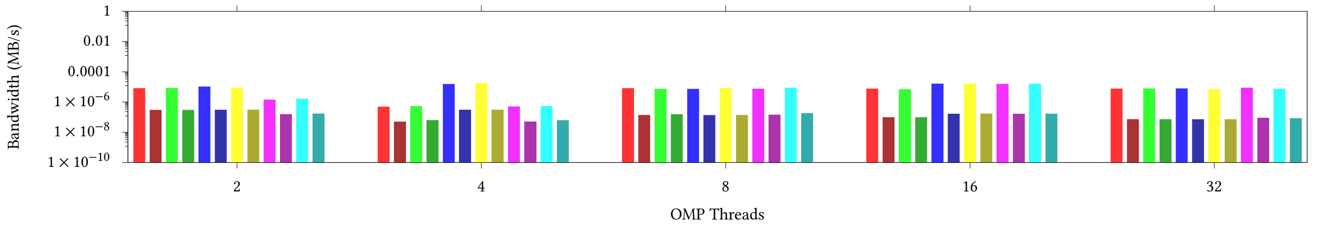


(b) Weak Scaling

Figure 15: Cache bandwidth over different chunk sizes for TMM



(a) Strong Scaling



(b) Weak Scaling

Figure 16: DTLB load store miss ratios over different chunk sizes for TMM

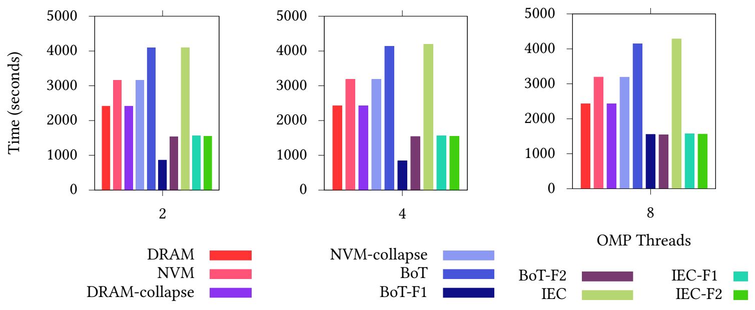


Figure 17: Solve Time comparison of scheduling and chunking policies for 27-pt stencil with Strong scaling

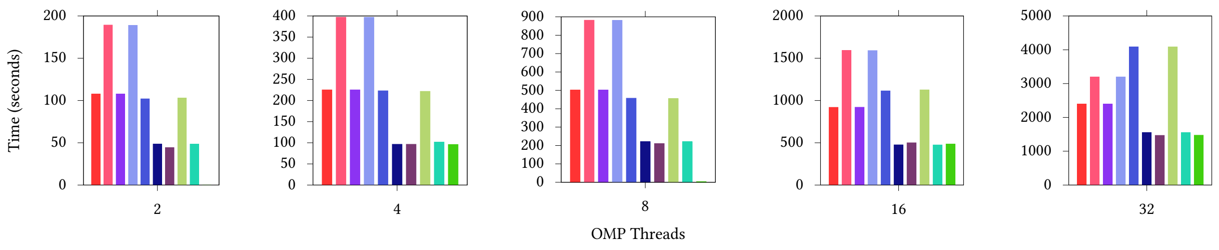


Figure 18: Solve Time comparison of scheduling and chunking policies for 27-pt stencil with Weak scaling

statically fixed F1 and F2 chunk variants under strong and weak scaling compared to dynamic chunk size variants. The fixed chunk sizes show higher memory bandwidth and L1-L2 cache bandwidth (as shown in Figures 15a and 15b) along with higher DTLB load misses (as shown in Figures 16a and 16b), which contributes to the faster performance.

Inference 1: *Splitting the memory allocation into chunks and scheduling them across memory devices simultaneously while fully utilizing fast memory and all memory controllers at the same time helps achieve higher performance for TMM on a heterogeneous memory system.*

Observation 2: *For 1D stencil codes, PEARS scheduling and memory management can support larger data sizes than DRAM or NVM allocations by themselves. It also outperforms all NVM scheduling policies. However, for the smaller problem sizes that fit into DRAM, it has inferior performance to DRAM memory allocation and runtime scheduling methods.*

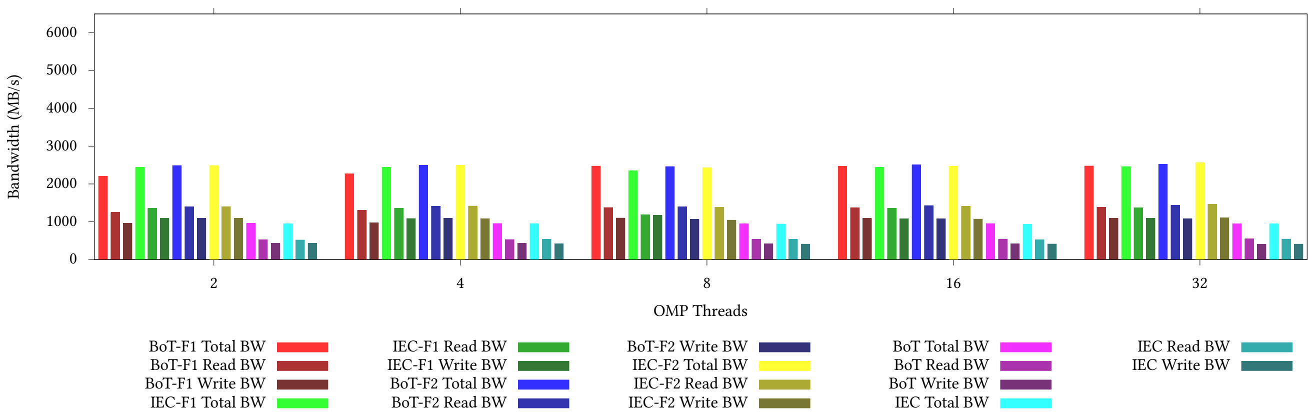
For a 1D 3-point stencil kernel (figures omitted due to space), we observe that BoT and IEC perform up to 3x times slower than DRAM allocation for both strong and weak scaling. For lower numbers of threads in weak scaling, BoT and IEC perform slightly faster than NVM allocation. This can be attributed to the lower memory reuse of very small stencils, which result in high memory bandwidth demand than cannot be met by NVM (used by BoT and IEC) while DRAM provide higher bandwidth. We observe that BoT and IEC also achieve lower cache bandwidth than other allocation and runtime scheduling methods. Nonetheless, BoT and IEC do achieve the same DTLB load and store misses compared to others. This means that BoT and IEC do not help in achieving high locality in the caches for 3-point stencil workloads. Fixed chunk sizes improve the execution time by only 10%. This is true for both strong and weak scaling.

For a 1D 7-point 3 halo stencil kernel (figures omitted), BoT and IEC (both using NVM) again are 3x times slower than DRAM but perform on par with NVM-collapse allocation and scheduling for strong scaling. For weak scaling, they perform slightly better than NVM and NVM-collapse allocations and runtime scheduling for lower numbers of threads. BoT and IEC have slightly higher memory, cache bandwidth and DTLB load miss ratios than NVM and NVM-collapse allocation and runtime scheduling methods. Due to the denser stencil (increased spatial locality) and higher reuse (increased temporal locality), BoT and IEC are able to achieve higher cache locality. Fixed chunk sizes do not have any impact on the performance.

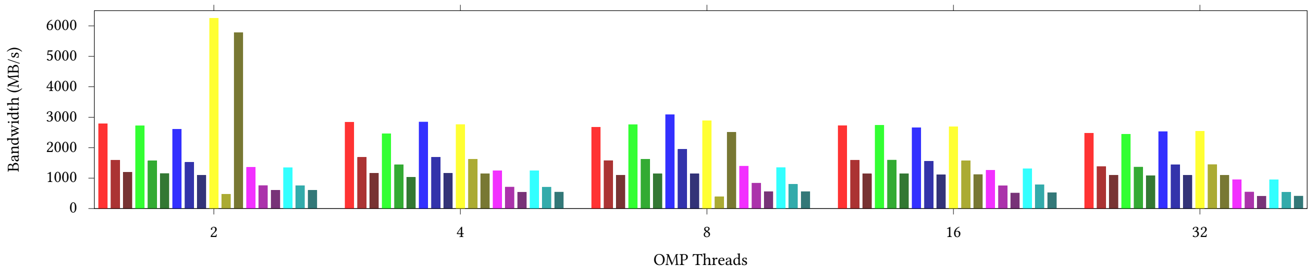
Inference 2: *The PEARS framework can accommodate larger 1D stencil problem sizes than any other allocation policy. However, for smaller problem sizes with sparse 1D stencils, it has inferior performance to DRAM , which is due to lower memory and cache bandwidth. For denser stencils, PEARS outperforms NVM only allocations and runtime scheduling.*

Observation 3: *For 2D stencil codes, PEARS can again support larger problem sizes than other policies. But for allocations that fit into DRAM, PEARS is slower than DRAM only. Yet under weak scaling, it is on par or faster than the collapse runtime scheduling methods for both DRAM and NVM allocation policies.*

For a 2D 5-point stencil kernel, we observe that BoT and IEC perform almost 3x time worse than DRAM allocation and 40% slower than NVM allocation for strong scaling. Similarly, the 2D 9-point stencil (2 halo) benchmark and the 2D 9-point stencil (with



(a) Strong Scaling



(b) Weak Scaling

Figure 19: Memory bandwidth over different chunk sizes for 27-pt stencil

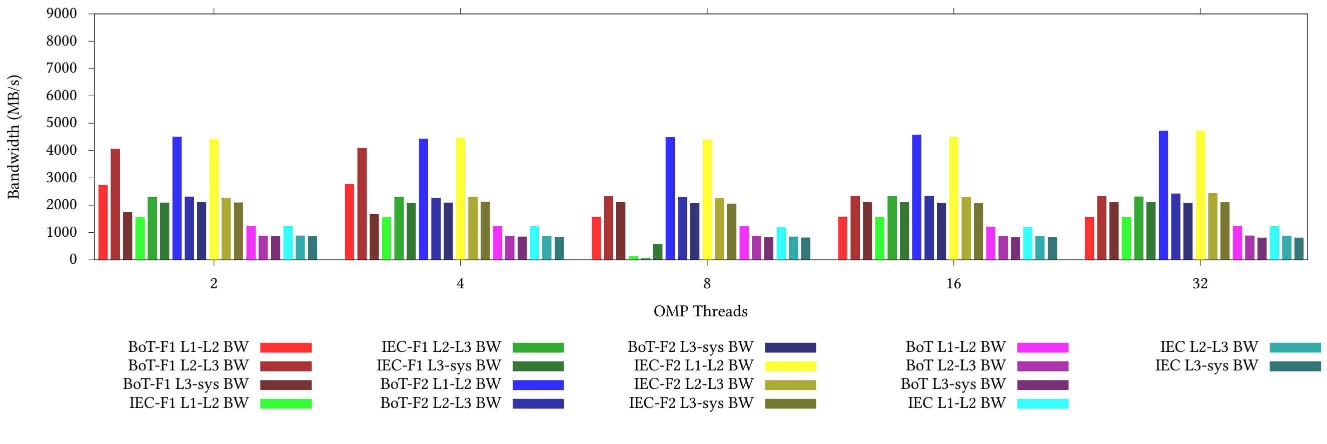
diagonals) benchmark result in 3x and 2.5x times longer execution, respectively, under BoT compared to DRAM allocation, and 30% slower execution than NVM allocation. However, BoT and IEC outperform NVM allocation in case of all three 2D stencil codes for lower numbers of threads under weak scaling. For both the 2D 9-point stencil codes, BoT and IEC outperform NVM-collapse allocation and runtime scheduling and are on par with DRAM-collapse allocation and runtime scheduling. The performance of BoT and IEC can be attributed to the change in memory bandwidth observed for all three benchmarks. However, neither PEARS variant is able to extract any benefit from cache locality, which is seen by their low cache bandwidth and high DTLB load and store misses. For all three benchmarks, the fixed chunk sizes for BoT and IEC offer only slight performance improvement over the dynamically determined memory chunk sizes, except for larger numbers of threads, where both fixed chunk sizes perform around 10% faster. This improvement is again attributed to higher memory bandwidth achieved by fixed size memory chunking.

Inference 3: *PEARS can accommodate larger 2D stencils than any other method. But for smaller problem sizes, a lack of improvement in cache locality for 2D stencils under PEARS results in lower performance than with DRAM allocations. Nonetheless, PEARS still provides some benefit over NVM allocations and the OpenMP collapse runtime scheduling, even for smaller data sizes.*

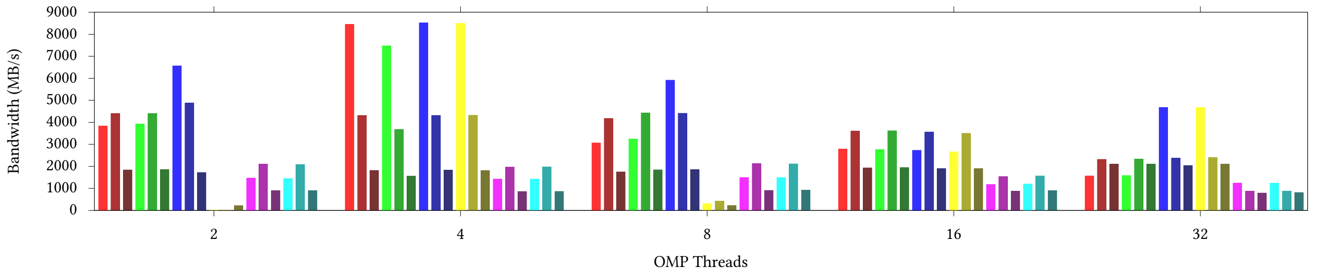
Observation 4: *For 3D stencil codes under both strong and weak scaling, runtime scheduling and memory management of PEARS static chunking outperforms the pure DRAM methods, while PEARS with dynamic chunking is on par with the DRAM methods.*

For a 3D 7-point stencil benchmark, BoT and IEC perform approximately 3x times slower than other memory allocation and runtime scheduling methods. However, IEC performs 10% faster than BoT. This is the case for both strong and weak scaling. Similarly, for a 3D 13-point stencil (2 halo) BoT performs 3.5x slower than DRAM allocation and IEC performs 2.5x slower. IEC is again 30% faster than BoT for both strong and weak scaling. For a 3D 27-point stencil, BoT and IEC are both 2x slower than DRAM for strong scaling. In weak scaling, BoT and IEC are on par with DRAM allocation performance and better than NVM allocation. For all 3 benchmarks, this performance can be attributed to high memory bandwidth and DTLB load and store misses.

A different picture is seen for fixed chunk sizes for both BoT and IEC for all the 3D-stencil codes. For the 3D 7-point stencil, we observe that IEC performs 3x times faster with fixed chunk sizes, which puts it on par with DRAM allocation performance. Similarly, for the 3D 13-point stencil (2 halo) we again see a 3x time performance improvement for IEC and for the 3D 27-point stencil, we see a 4x times performance improvement (as shown in Figure 17 and 18), which means it is outperforming DRAM allocation. The same improvement in performance is not observed for BoT for 7- and 13-



(a) Strong Scaling



(b) Weak Scaling

Figure 20: Cache bandwidth over different chunk sizes for 27-pt stencil

point (2halo) 3D stencils where the fixed chunk sizes only bring minor improvements in performance. For 27-point stencil, we see a 4x times performance improvement. The performance impact of fixed chunk size allocation can be attributed to the higher memory and cache bandwidth observed over the dynamically determined chunk sizes as shown in Figures 19a, 19b, 20a and 20b.

IEC runtime scheduling has a clear advantage over BoT scheduling for sparse 3D stencil codes. In BoT, all threads grab chunks in a greedy manner. This can lead to cases where a single thread can grab more chunks than they can fit in the fast memory leading to sub-optimal results in a heterogeneous memory system due to low cache reuse. This situation is avoided with IEC as the main thread aims to balance the workload across all threads and ensures optimal use of both the fast memory and all memory controllers within the memory subsystem. Hence, with the large problem and chunk sizes of sparse 3D stencil codes, IEC performs faster than BoT and, in some cases, faster than (or at least on par with) DRAM allocation.

Inference 4: *The performance of IEC runtime scheduling and memory management under PEARS is on par or even exceeds all other allocation and scheduling methods for 3D stencil codes, due to IEC's ability to balance workload across memory devices and controllers in the memory system. This helps take advantage of the capacity and the speed available in a heterogeneous memory system.*

Observation 5: *The PEARS framework delivers better performance for denser stencils than sparse stencils and for higher dimensional datasets than lower dimensional datasets.*

Denser stencils that operate on large data sets tend to have lower temporal cache locality due to larger reuse distance between accesses and the streaming-like access pattern. This leads to lower cache hits and cache contention. When only a single type of memory is used for allocation in a heterogeneous memory system, not all memory resources are utilized, which can leave some performance gains on the table. Due to memory chunking in PEARS, the datasets get divided into smaller memory streams, which have higher temporal cache locality due to a shorter reuse distance. When the memory chunk sizes align with the private cache sizes, the number of cache hits for each memory chunk also increases. This helps workloads such as dense TMM. As the memory chunks are distributed and executed simultaneously across all memory devices, all resources are utilized, which improves the memory systems performance as well. This also reduces memory request back pressure from the memory controllers. The memory chunking does incur a memory overhead and halo value communication overhead for stencil codes. But with large datasets, this overhead becomes a small fraction of the total workload and its overhead is easily recouped.

Inference 5: *Overall, the PEARS framework is an ideal solution achieve balanced memory management and runtime scheduling for dense stencil or TMM workloads with large datasets.*

7 CONCLUSION

This work contributes a memory management and runtime scheduling framework called PEARS enabling HPC application programmers to write programs that can take advantage of the resources provided by a heterogeneous memory system. It helps targeted HPC workloads like TMM and stencil kernels to achieve better memory performance than traditional memory management and scheduling techniques. It helps reduce the performance impact of a slower memory in the memory system. At the same time, it helps support larger problem sizes than can fit into a homogeneous memory while reducing the dependency on the application programmer to optimize memory performance for a heterogeneous memory system.The PEARS framework will accelerate the adoption and use of heterogeneous memory systems for large scale HPC workloads, which has many cost and energy benefits for future HPC systems.

ACKNOWLEDGMENTS

This material is based upon work supported by United States Department of Energy National Nuclear Security Administration prime contract #89233218CNA000001 subcontract #508854 dated November 1, 2018 for Los Alamos National Laboratory, NSF grant 1525609 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The unclassified release number is LA-UR-21-25548.

REFERENCES

[1] Lluc Alvarez, Marc Casas, Jesus Labarta, Eduard Ayguade, Mateo Valero, and Miquel Moreto. 2018. Runtime-guided management of stacked dram memories in task parallel programs. In *Proceedings of the 2018 International Conference on Supercomputing*. 218–228.

[2] Jason Evans April. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD.

[3] Aurora - May 2019 [n. d.]. Aurora - May 2019.

[4] Anne Benoit, Swann Perarnau, Loïc Pottier, and Yves Robert. 2018. A performance model to execute workflows on high-bandwidth-memory architectures. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[5] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 2008. 0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413435

[6] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas* 15, 5 (2008), 055703. https://doi.org/10.1063/1.2840133 arXiv:https://doi.org/10.1063/1.2840133

[7] K J Bowers, B J Albright, L Yin, W Daughton, V Roytershteyn, B Bergen, and T J T Kwan. 2009. Advances in petascale kinetic plasma simulation with VPIC and Roadrunner. *Journal of Physics: Conference Series* 180 (jul 2009), 012055. https://doi.org/10.1088/1742-6596/180/1/012055

[8] 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. (3 2015). https://www.osti.gov/biblio/1245908

[9] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 433–452.

[10] Kavitha Chandrasekar, Xiang Ni, and Laxmikant V. Kale. 2017. A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1293–1300. https://doi.org/10.1109/IPDPSW.2017.168

[11] Raúl de la Cruz and Mauricio Araya-Polo. 2014. Modeling stencil computations on modern HPC architectures. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 149–171.

[12] Fujitsu [n. d.]. Fujitsu HPC Roadmap.

[13] Derrick Greenspan. 2019. LLAMA-automatic memory allocations: an LLVM pass and library for automatically determining memory allocations. In *Proceedings of the International Symposium on Memory Systems*. 363–372.

[14] J Hruska. 2018. Intel announces new optane DC persistent memory. *ExtremeTech https://www. extremetech. com/extreme/270270-intel-announces-new-optane-dc-persistent-memory* (2018).

[15] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[16] Dounia Khaldi and Barbara Chapman. 2016. Towards automatic HBM allocation using LLVM: a case study with knights landing. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 12–20.

[17] Tobias Kretz and Michael Schreckenberg. 2007. Moore and more and symmetry. In *Pedestrian and evacuation dynamics 2005*. Springer, 297–308.

[18] Michael Kenneth Lang. 2019. Simplified Interface to Complex Memory (SICM) FY19 Project Review. (10 2019). https://doi.org/10.2172/1569724

[19] Edgar A León, Brice Goglin, and Andres Rubio Proaño. 2019. M&MMs: navigating complex memory spaces with hwloc. In *Proceedings of the International Symposium on Memory Systems*. 149–155.

[20] Lena Oden and Pavan Balaji. 2017. Hexe: A toolkit for heterogeneous memory management. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 656–663.

[21] OpenMP 3.1 specification - July 2011 [n. d.]. OpenMP 3.1 specification - July 2011.

[22] Mike O'Connor. 2014. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*.

[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 fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE, 288–303.

[24] Onkar Patil, Frank Mueller, Latchesar Ionkov, Jason Lee, and Michael Lang. 2020. Symbiotic HW Cache and SW DTLB Prefetching for DRAM/NVM Hybrid Memory. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[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 fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE, 304–315.

[26] Swann Perarnau, Judicael A Zounmevo, Balazs Gerofi, Kamil Iskra, and Pete Beckman. 2016. Exploring data migration for future deep-memory many-core systems. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 289–297.

[27] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between NVM and DRAM for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. ACM.

[28] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. 2018. Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 2018 International Conference on Supercomputing*. 207–217.

[29] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[30] Vinu Sreenivasan, Rajath Javali, Mary Hall, Prasanna Balaprakash, Thomas RW Scogland, and Bronis R de Supinski. 2019. A framework for enabling OpenMP autotuning. In *International Workshop on OpenMP*. Springer, 50–60.

[31] Tehrany. 2020. Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack.

[32] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 207–216.

[33] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 91–104.

[34] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 167–181.

[35] Ulrike Meier Yang et al. 2002. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 1 (2002), 155–177.