ABSTRACT

PATIL, ONKAR BHALCHANDRA. Assessing the Performance of and Devising Optimization Strategies for Heterogeneous Memory under HPC Workloads . (Under the direction of Frank Mueller.)

NVM devices are becoming a part of the memory hierarchy with the introduction of technologies like PCM, MRAM, HRAM, etc., which enable hybrid memory systems. This addition presents an opportunity for applications to increase the size of problems currently running on HPC systems. However, the performance of the application will be affected when utilizing persistent memory due to different write and read speeds compared to DRAM. Hence, hybrid memory should be used in a close to optimal manner where the performance impact to the application is nearly minimal while taking advantage of the massive capacity and different access speeds to solve larger problems.

Application performance is influenced by the kernels that compose the application program. These kernels access different data structures within them with varying stride patterns such that their performance can be affected by the underlying memory device on which they are allocated on. Allocating the data structures on the correct memory device so that the application takes advantage of all the benefits of a heterogeneous system without compromising on performance is key. There exist mechanisms that allow programmers to allocate memory for data structures on persistent memory. These are supported by language extensions or macros that the programmer uses to decide on the allocation. However, the onus of utilizing the memory technologies efficiently falls on the skill and knowledge of the programmer. They are pushed to limits with scientific applications that can feature up to 1,000 or even more data structures. What's more, allocation strategies can change for heterogeneous memory hierarchies based on the usage pattern of the data structures at different stages of an application. It is difficult for the programmer to make the apt allocation decision for every data structure to ensure superior memory performance of the application, let alone reason about when to dynamically move data structures from one type of memory to another.

To alleviate this, we present a memory *performance aware runtime system and programming methodology, PEARS*, that enables HPC application programmers to write traditional HPC workloads while taking advantage of the multiple benefits that a byte-addressable, heterogeneous memory system can provide. We demonstrate multiple workloads implemented with PEARS while comparing their performance to the standard implementations of these workloads under various memory allocation scenarios.

We have evaluated the performance of a DRAM-NVM based hybrid memory system for scientific workloads in HPC systems. We compare its performance to traditional HPC systems and memory architectures, and then formalized the heuristics based on these evaluations. Analyzing performance of different access patterns and stream workloads along with the behavior of the cache hierarchy gives insight into the effect of varying access speeds on HPC applications. We will use this information along with static analysis of the memory system to identify different access patterns and streams to make automatic data allocation and movement decisions at runtime.

Assessing the Performance of and Devising Optimization Strategies for Heterogeneous Memory under HPC Workloads

by
Onkar Bhalchandra Patil

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2021

APPROVED BY:

Xipeng Shen	Huiyang Zhou
James Tuck	Frank Mueller Chair of Advisory Committee

DEDICATION

To my parents and brother who have always believed in me. And to all my teachers that have taught me everything I know.

BIOGRAPHY

The author hails from Mumbai, Maharashtra in India where he was born and brought up by his family. He completed his schooling from St. Anne‘s High School, Malad and Sathaye Junior College in Mumbai. In 2012, he completed his undergraduate degree from Fr. Conceicao Rodrigues College of Engineering, Mumbai, in the field of Information Technology. He was admitted to North Carolina State University, Raleigh, in USA in Fall 2012 to pursue his graduate studies in the field of Computer Science. He began working as a Graduate Research Assistant under the guidance of Dr. Frank Mueller in May 2013. He completed his Masters from North Carolina State University, Raleigh, in 2014 and had a 2 year stint at NetApp Inc. and Hewlett Packard Enterprise after which he returned back to pursue his doctoral degree. He takes keen interest in pursuing his hobbies in music, dance, photography, travel, sports and reading.

ACKNOWLEDGEMENTS

I would like to take this opportunity and thank everyone who has played a part in bringing success to me at this point in time.

I would thank my advisor Dr. Frank Mueller to have trust and faith in my capabilities and give me this golden chance to work on this research. His guidance and immense support has been the biggest factor in completion of this thesis. His encouragement and motivation to achieve the desired results has been inspiring. I appreciate his constant availability and help for his students. I would like to thank Dr. Xipeng Shen, Dr. Huiyang Zhou and Dr. James Tuck to take interest in my research and who agreed to be members of my advisory committee. I would also like to thank my lab mates and people of my research group for their help, support and encouragement whenever I needed it. It was a pleasure working with you all.

I also want to thank my mentors from Los Alamos National Laboratory, Dr. Latchesar Ionkov, Michael Lang and Jason Lee, who encouraged and guided my research all the way and helped me at every minute step along the way. I thank all my friends in USA, India and the ones spread all around the world for their constant support and well wishes. I wouldn't have come this far without anyone of them. I also want to thank all my teachers that have taught me at every point in my life and imparted invaluable knowledge to me for which I will be always grateful.

I want to thank my elder brother, Nishad, for all his love, care and motivation. His words of advice and guidance have been invaluable. Last but not the least, I want to thank my parents who trusted me the most, that I was capable of pursuing my dreams and ambitions and supported me with all that they have got. Their love and kindness have made difficult times seem very easy.

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 grants 1217748 and 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 numbers are LA-UR-19-25342, LA-UR-19-27887, LA-UR-20-27524 and LA-UR-21-25548.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

The principle of Von Neumann architectures is based on an execution concept: Instructions and data are placed in memory and then moved to registers in the Central Processing Unit (CPU) where they are executed on the Arithmetic Logical Unit (ALU). Modern day computers are still designed using this concept, but technology has always been evolving with the enhancements in hardware followed by software support to make best use of the enhancements. Over the years, different aspects of the architecture have evolved to achieve higher compute performance and efficiency. Increases in the speed of computation and memory size support larger datasets but compromise on access latency. To bridge the difference in access latencies between fast microprocessors and slow but high capacity memory, data caches were first used commercially in 1980s. Secondary caches were introduced to bridge the gap between compute and memory. Multi-core architectures were introduced to perform computations simultaneously and improve compute efficiency as processor clock speeds had reached their limits. Co-processors like the General Purpose Graphics Processing Units (GPGPUs) were introduced to achieve massive parallelism and process large amounts of data efficiently. To further help in accommodating and processing large amount of data, every compute device received its own specialized memory and cache. However, the memory density has not scaled up at the same rate as the scaling of cores. Today, all memory devices are primarily based on Dynamic Random Access Memory (DRAM) and caches are made up of Static Random Access Memory (SRAM). SRAM, although a faster memory technology, is significantly more expensive in terms of on-chip package space and cost due to additional transistors. DRAM, on the other hand, is a cheaper and denser memory technology compared to SRAM but is almost two orders of magnitude slower. But even with its higher density, DRAM does not scale well beyond 40-35 nm. This has led memory capacity per core to scale at only at a rate of 2/3rds of the core count over the last two

decades. As we apply principles of exascale (10^{12} floating point operations per second) compute capabilities and expand high performance compute (HPC) capabilities to newer avenues, such as machine learning, the need to support larger problem sizes (10-100 PBs) is imminent. Non-Volatile Memory (NVM) technologies like Phase Change Memory (PCM) can be used to increase the memory per core ratio and support larger problem sizes.

With larger datasets coming into play, one also faces the issue of efficiently moving large amounts of data between the memory and compute devices. One would require high bandwidth interconnects between compute and memory devices to address this issue. High-bandwidth Memory (HBM) is the result of this evolution. It is introduced to help massively parallel compute devices access large amounts of memory at a higher bandwidth.

Today, the architectures have become complex with the aforementioned enhancements to achieve efficient computation of ever increasing data sizes. HBM is available on many newer computing devices that process large amounts of application data simultaneously. The Intel Knight's Landing (KNL) architecture, although discontinued by Intel, is still being used in many compute clusters. It has HBM to support its tightly coupled massively parallel processors. AMD's Fiji GPGPU and Nvidia's Tesla GPGPUs were the first to provide HBM to aid its Streaming Multiprocessor (SMP) compute ability. HBM is provided on these architectures as an additional memory device to the existing Dynamic Random Access Memory (DRAM) devices. Fujitsu's A64FX ARM chips are connected to main memory using HBM. HBM is primarily used on GPGPUs where the streaming nature of computation demands such high bandwidth. NVM as a byte-addressable memory has recently been introduced with Intel's Optane DC Persistent Memory Modules (PMM), which can be plugged into the same DIMM slots as DRAM. Other technologies, e.g., Spin Torque Transfer Magnetic RAM (STT-MRAM), are being investigated to be introduced as byte-addressable persistent memory devices. These technologies have the potential to solve the memory capacity and bandwidth issues that current and future scientific applications face. Using all memory technologies in a single compute device gives rise to heterogeneous and hybrid memory architectures where memory devices differ in capacity, access latency, bandwidth and volatility.

A heterogeneous memory architecture presents users with a large number of challenges because current memory allocation and data movement policies were not designed for them. They assume a single access latency at every tier of the memory architecture, which is not the case for a heterogeneous memory architecture. Hence, there is a need for new policies that can help us take advantage of the different properties of a heterogeneous memory system.

Software support and enhancements have always followed the introduction of any technology advancement. They need to evolve with careful experimentation and evaluation of the new hardware to enhance the performance of applications that will use the new architecture. There exists software support to allocate memory regions on different memory devices with the help of Non-Uniform Memory Access (NUMA) architecture and the libraries built on top of it. However, while the existing libraries provide the basic allocation interfaces, they lack mechanisms to optimize the allocations for different applications such that the onus of optimizing the application performance falls on

the application developer. It is not a trivial problem either because all memories devices have their own trade-offs. For example, HBM provides the ability to bring large amounts of data to multiple compute devices in order to reduce execution time of the applications. But HBM is also a scarce and costly resource compared to DRAM, so the entire dataset cannot fit in HBM. HBM needs to be used conservatively and based on application characteristics to provide optimal performance benefits. It needs to be carefully evaluated with different application scenarios to see where the performance is enhanced and quantify it so that it can be formalized for further use. A similar scenario arises when we use DRAM and NVM in the same memory hierarchy and the capacity and access latency conflict with each other. The existing caching and prefetching policies are also problematic because now the underlying memory technology may have different cache line mappings and varying latency. Making all these decisions at the development level is very difficult given that every system can have its own custom memory hierarchy, and memory allocation policies will have to be customized based on it. One needs to relieve the application developer from this responsibility, yet be able to make all these decisions based on every memory architecture scenario that is encountered.

The aim of this work to automate the process of memory allocation, data placement and data movement. In order to automate the memory allocation, many factors need to be considered. Each memory device needs to be carefully analyzed and its behavior needs to be quantified in terms of access latency and memory bandwidth while keeping or even increasing the available capacity. These characteristics then need to be reflected in heuristics that can be used to quantify the performance impact of allocating a data structure on a given memory device. This performance impact needs to be considered in the context of the entire application so that each application can adapt its memory allocation for any hybrid memory architecture.

Given the information required to automate the data placement on a heterogeneous memory architecture, a macro-based solution that hints the runtime system can make the process convenient as most of the information we need can be made available at this stage. Every application consist of compute kernels, which dominated by loops (for, while, do..while) and conditionals (if..else, switch..case). Within these kernels there are computation statements that perform arithmetic operations, which translate into machine-level load or store operations. These code structures are known as streams of memory operations. Analyzing different types of compute kernels provides great insight into an application's memory performance and requirements. Measuring and quantifying the performance of different streams on a slow and fast memory helps in deriving a heuristic for memory allocations on such hybrid memory architectures. Heuristics can assist in the optimization of performance with just the knowledge of the application code but without having to evaluate applications repeatedly per different allocation. This analysis can be performed statically and all the memory allocation decisions can be made at runtime.

At runtime, more information, e.g., number of threads, allocation size and iteration counts, can be extracted. This can guide the data placements along with the statically collected heuristics. Hence, this work utilizes a macro-based programming solution that communicates the heuristic and application code information to the runtime system, which combines runtime analysis to

make the data placement and movement decisions. It relieves the application programmer from making all the memory allocation and data movement decisions while designing the application. In this work, the aim is to formalize a heuristic-based approach to help quantify the performance benefit of different allocations on a heterogeneous memory architecture based on the application characteristics, i.e., different streams. These heuristics help make memory allocation decisions at runtime with macro-based programming support for targeted HPC kernels and help decide how to best use the available memory resources.

Thesis hypothesis: To exploit the complexity of heterogeneous memory systems, an automated framework is required that considers static and dynamic characteristics of an application's memory footprint to drive memory allocation and data movement within HPC applications so that cost, energy and performance benefits can be realized for problem sizes in excess of singular homogeneous memories.

This thesis is divided into five principle chapters. Chapter 2 provides a thorough performance analysis of a DRAM-NVM based heterogeneous memory system using custom benchmarks and HPC applications and mini-apps. It also introduces the "flat mode" in which the heterogeneous memory system is completely byte-addressable. The memory system may operate in different modes, which are assessed in experiments that evaluate the performance. In this chapter, we gauge the characteristic differences between homogeneous and heterogeneous memory systems.

Chapter 3 evaluates the cost and energy benefits of using a heterogeneous memory system in large scale clusters. It compares the performance of the DRAM-NVM based memory system to a traditional DRAM based memory architecture with equivalent memory capacity by executing large scale runs of memory and compute bound HPC applications and min-apps. A projection of the cost and energy benefits of using heterogeneous memory systems for exascale machines is provided and a baseline performance for byte-addressable hybrid memory space is established.

Chapter 4 evaluates the performance of the existing HW and SW prefetching support for DRAM-NVM hybrid memory systems. It presents a new HW and SW symbiotic prefetching approach for multiple HPC kernels that is adaptive to the underlying memory devices and the allocations made on them. The performance with GCC's SW prefetching compiler pass is assessed and changes are proposed to make it adaptive for heterogeneous memory systems. In this chapter, we assess the data pipeline architecture support for hybrid memory systems to help improve the performance of heterogeneous memory system by efficiently moving data into the caches, thereby increasing data locality for HPC kernels.

Chapter 5 presents a performance-aware static and dynamic framework, called PEARS, that provides macro-based programming support for memory management and runtime scheduling for HPC workloads executing on heterogeneous memory systems. It provides the details of the framework that enables larger problem size executions than other allocation policies and helps schedule the memory workload across all memory devices in parallel. A comparison with traditional single memory device allocation policies and the OpenMP "collapse" clause is provided. In this chapter, we provide the automated framework that enables HPC applications to take advantage of

the cost, energy and performance benefits of heterogeneous memories by supporting performance aware memory management.

CHAPTER

2

PERFORMANCE CHARACTERIZATION OF A DRAM-NVM HYBRID MEMORY ARCHITECTURE FOR HPC APPLICATIONS USING INTEL OPTANE DC PERSISTENT MEMORY MODULES

2.1 Introduction

Memory hierarchies have been constantly evolving since computers were introduced and the von Neumann architecture was adopted. Today, semiconductor memory is dominated by dynamic random access memory (DRAM) as its density is high while its cost is low [Nai15]. DRAM is volatile, prone to soft errors, and more power-consuming due to the constant refreshing required to retain the stored data. As processor clock frequencies were scaled up, static random access memory (SRAM) was introduced as a caching layer to bridge the latency gap. The memory hierarchy kept expanding as multi-level caches were introduced, high-bandwidth memories were added, and the main memory sizes kept increasing. Data persistence, or non-volatility, is a feature of data storage, which is the secondary level of the current memory hierarchy. Most non-volatile devices are not on the memory bus like DRAM, but are much further away in terms of latency. The difference in access latencies between DRAM and other technologies used in storage made it cumbersome to scale the capacity

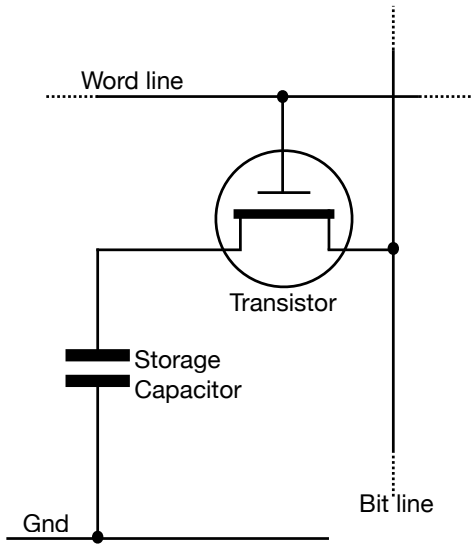
or add persistence to the primary memory hierarchy.

Supercomputers are built with individual nodes, which have their own memory hierarchy. The nodes are connected to other nodes by high-speed interconnects, allowing for direct memory access or remote direct memory access. The combined memory of a cluster is greater than that of a single node but requires complex software and additional hardware and the scale comes with its own share of problems. For example, Oak Ridge National Laboratory's Titan [Tit], a petaflop machine, which is now decommissioned, is one of the fastest supercomputers on the TOP500 November 2018 list [Topb]. Each node has 38 GB of DRAM. As a cluster of 18,688 nodes connected over an interconnect, Titan has 710 TB of DRAM. However, the sheer number of DRAM modules used in the system causes it to be susceptible to soft errors and hard faults. Additionally, memory is one of the main components contributing to the power consumption of Titan, which can reach up to 8.2 MW at its peak. Due to the higher number of DRAM modules that will be required to achieve exascale memory requirements, the cost to build and operate a larger machine with a similar memory architecture will increase significantly. This also increases the likelihood of failures [Gup17]. Thus, an exascale machine with a similar architecture may require hardware innovations to address the challenge of resiliency and power.

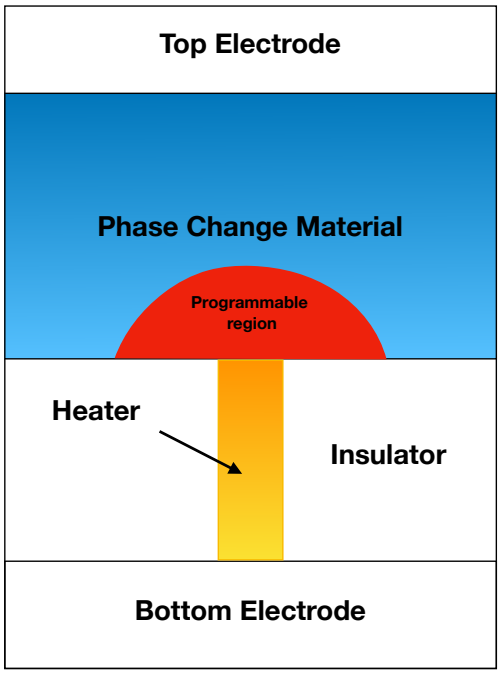
Over the past decade, memory technologies such as phase change memory (PCM) and spin-transfer torque RAM have been developed and are now used to make byte-addressable non-volatile memory devices [Rao14; Apa13]. Although they are slower, they have higher densities than DRAM. This trade-off requires a detailed analysis to evaluate the benefit of these new memory technologies. Intel has been the first-to-market with their Intel® Optane™ DC Persistent Memory Module (PMM), which is based on PCM technology. The Optane DC is plugged directly into the memory bus via traditional DIMM slots. It has 8 times higher density than DRAM and is cheaper per GB. The Optane DC PMMs can be used to expand the capacity of primary memory hierarchy with data persistence, or can be used as a traditional NVM block device.

Using memories with higher density that of DRAM will allow different design points for exascale computers. Fewer nodes can be used to reach higher aggregate memory capacities. Fewer nodes means fewer components, which in turn can lower the cost to build the system, reduce the overall power consumption of the system, and increase resiliency. Additionally, the data persistence of these new types of memory can also assist in the development of new fault-tolerance mechanisms.

In this chapter, we take a closer look at the Optane DC PMMs, its underlying technology, its operation, the different modes that it can operate in, and evaluate its performance for HPC applications. We focus on evaluating its use as main memory instead of part of the storage system. We have characterized the performance of Optane DC by using a custom benchmark inspired by the STREAM [McC95; McC91], which has access streams that are frequently used in HPC applications. We have evaluated the overall system performance with HPC applications like VPIC [Bow09], and proxy applications such as AMG [Yan02], LULESH [Kar13] and SNAP [Sna]. In Section 2.2, we review research related to persistent memory systems and their evaluation. In Section 2.3, we go over the background of non-volatile memory, and in Section 2.4 we focus on the Optane DC memory



(a) DRAM cell



(b) PCM cell

Figure 2.1 Memory cells of DRAM and PCM

architecture. In Sections 2.5, we evaluate the performance of Optane DC. In Section 2.7, we present potential future work and follow up with the conclusion in Section 2.8.

2.2 Related Work

Due to the recent launch of Optane DC PMMs, there are not many previous works that evaluate the performance characteristics of the device.

In [Izr19], Izraelevitz et al. evaluated the read and write characteristics of the Optane DC PMM. They evaluated the performance of Optane DC on all the modes available using the SPEC2017 benchmark suite. They found that applications experience a 15-61% slowdown with NVM-only allocations when compared to DRAM-only allocations. They also compared the performance of different filesystems and database applications like Mongo DB and MySQL by using Optane DC as persistent storage and persistent memory respectively. They found that Optane DC boosts the performance of filesystems and database applications due to lower latencies than storage devices. We are evaluating Optane DC for different streams that we encounter in different HPC applications and focus on using Optane DC as an extended address space for the same. Gill et al. [Gil19] used the Optane DC PMM to evaluate shared-memory graph frameworks like Galois on real world web-crawls. They found that Optane DC PMM yields performance and cost benefit for massive graph analytics when compared to a distributed graph frameworks on existing production clusters. Our work focuses on HPC problems that are mostly stencil codes and matrix operations. Psaropoulos et al. [Psa19] worked on hiding the latency difference between Optane DC and DRAM for database applications by interleaving the execution of parallel work in index joins and tuple reconstruction using co-routines. They accelerated end-to-end query runtimes on both NVM and DRAM by up to 1.7x and 2.6x, respectively. Van Renen et al. [Ren19] performed performance evaluations of Optane DC in terms of bandwidth and latency and developed guidelines for efficient usage of Optane DC and two tuned I/O primitives, namely log writing and block flushing. Their work is primarily based on the App-direct mode (modes are explains in Section 2.4) and is aimed to improve the performance of file-systems. Wu et al. [Wu17] studied the I/O performance of an early version of Optane DC, which was 3D-Xpoint with NFS and PVFS [Lat04] as the filesystems. It operated similar to the current App Direct Mode in Optane DC.

There has been a lot of work in providing software support for byte-addressable non-volatile memories. Volos et al. [Vol11] created a simple interface for programming with persistent memory called Mnemosyne. It allows programmers to allocate global persistent and dynamic data structures and also primitives to modify the data structures. Coburn et al. [Cob12] implemented a lightweight, persistent object system called NV-heaps. It provides transactional semantics that prevent errors and a persistence model for heap objects. Chakrabarti et al. [Cha14] proposed a system with durability semantics for a lock-based code called Atlas. It automatically maintains a globally consistent state in the presence of failures. Dulloor et al. [Dul14] implemented a POSIX file system, PMFS, that exploits persistent memory's byte-addressability to avoid overheads of block-oriented storage and enable direct persistent memory access by applications with memory-mapped I/O. Yang et al. [Yan15] implemented and evaluated the performance of a non-volatile B^+-Tree called NV-Tree and a key-value store based on it for NVDIMM-based servers. Shull et al. [Shu19] proposed a user-friendly

NVM framework for Java that ensures consistent stores for crash-recovery operations.

Various ways have been proposed to use NVM for HPC systems. Vetter et al. [VM15] evaluate the potential for NVM systems for extreme-scale HPC. They look at various persistence devices for integration of NVM in HPC and also look at integrating the functionalities of NVM. Kannan et al. [Kan13] optimized checkpoints for HPC application using NVM as a virtual memory and provide frequent, low overhead checkpoints. Patil et al. [Pat09] proposed a novel programming technique for stencil codes that guarantees fault tolerance against two hard failures on a shared non-volatile memory pool. Li et al. [Li12] proposed a fault tolerance process model based on NVRAM, which provides an elegant way for the applications to tolerate system crashes. Wang et al. [Wan12] proposed a novel approach for exploiting NVM as a secondary memory partition so that applications can explicitly allocate and manipulate memory regions therein. It had a library that enabled access to a distributed NVM storage system.

2.3 Background

The memory hierarchy in modern architectures is complex and deep [Nai15]. Register memory is closest to the processor, which is used to load and store operators, operands and instructions. It is implemented using flip-fops or an array of SRAM cells. It is the fastest memory, very expensive and consumes a lot of chip space and power. Most modern CPUs have 16 to 32 registers, which can hold 32 or 64 bits each. The access time for register memory is less than 1 ns. As computer programs require a lot more memory than there is on registers, we use a cheaper and higher density memory than registers as our main memory.

Main memory is comprised of arrays of DRAM cells as shown in Fig. 2.1a. It is a semiconductor-based memory technology that stores one bit of data in a capacitor within an integrated circuit. It is a rectangular array of cells that store a charge and are made of a capacitor and transistor per data bit. The number of cells define the capacity of a DRAM chip. There are positive and negative bit lines that connect all the cells in a column. A pair of cross connected inverters between the bit lines, called a sense amplifier, are used to stabilize the charges stored in the cells. DRAM has to be constantly refreshed to maintain its state due to the charge leak in the cells [Hid90]. The JEDEC standard [Jed] specifies that each row has to be refreshed every 64 ms or less. Due to this constant need of refreshing the charges, this memory uses much more energy. The access time to DRAM using the DDR4 protocol is approximately 50-100 ns. Access to DRAM is 10-15x slower than register access time, which can lead to many CPU stalls. Processors use SRAM caches as buffers to hide this latency. SRAM uses latches to store each bit. They are volatile and lose their state when the memory is not powered. It is termed as static because it does not require refreshes. Modern memory architectures have leveraged multi-level caches to reduce the effective latency between the main memory and the processor. The access time to SRAM cache is on an average 1-10 ns depending on the level of cache. SRAM is more expensive in chip space and power than DRAM.

This memory hierarchy has evolved and become more complex over multiple decades. It has

grown in size and also become faster due to the scaling of DRAM capacity and frequency over the last 2 decades. But DRAM scaling has been approximately 33% slower than core count scaling over the same period of time. Also, due to the increased number of memory cells and higher refresh rates, DRAM energy consumption has increased [Lim09; Rog09; Mut13]. Even though DRAM capacity has increased, due to higher densities, these memories have become less reliable [Gup17]. Extreme scale problems in HPC, machine learning, graph analytics, and other fields can exhaust in-node memory capacity and processing. [VM15]. Using NVM as a supplement to DRAM, in order to increase the size of main memory in a compute node, has been considered to be a viable option [Lee09]. Of all the NVM technologies, PCM has evolved best in terms of engineering [Lee09]. PCM is a resistive memory whereas DRAM is a charge memory. As shown in Fig. 2.1b, PCM has bit-line, which is a metal connected to a phase-changing material via a heater. When a current pulse is passed through the bit-line, a phase is set in the phase-change material and stored there until another current pulse is passed. The phase is read by detecting the resistance of the material through the access line. The phase-change material retains its phase for more than 10 years at ambient temperature [Lee09]. This property gives PCM its non-volatility. PCM is expected to scale down to 9 nm whereas scaling DRAM smaller than 40-35 nm is challenging [Mut13].

However, PCM has its own challenges and shortcomings. It has a higher write latency than that of DRAM due to the thermal activation required to change the phase-change material. PCM also suffers from wear due to thermal expansion and contractions of the contacts between the bit-line and phase-change material. The write-durability of PCM memory cell is approximately 10^8, which means frequent device replacements are required that can add to the cost [Lee09]. Despite these disadvantages, PCM provides the capability of scaling the main memory capacity required to match core count scaling. Intel's Optane DC PMMs are based on PCM technology.

2.4 Architecture

The system that we use for our experiments is a single node provided by the Intel Corporation. As described in Table 2.1, this node has 2 CPU sockets that are equipped with Intel's Xeon® 8260L (codenamed Cascade Lake). Each chip has 24 cores with a clock frequency of 2.4 GHz. Each core has 2 processing units 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 36 MB L3 cache shared between all cores. 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. That totals up to 192 GB of DRAM and 1.5 TB of non-volatile memory. The node has 4 memory controllers in total. Two of the memory controllers are connected to 6 DRAM DIMMs each, and the other two, known as iMC, are connected to 6 NVDIMMs each.

As shown in Figure 2.2, the processor communicates differently with Optane DC DIMMs than with DRAM. For DRAM, it uses the standard DDR4 protocol via the regular memory controller whereas for Optane DC it uses the DDR-T protocol via the i-memory controller (iMC). Using this

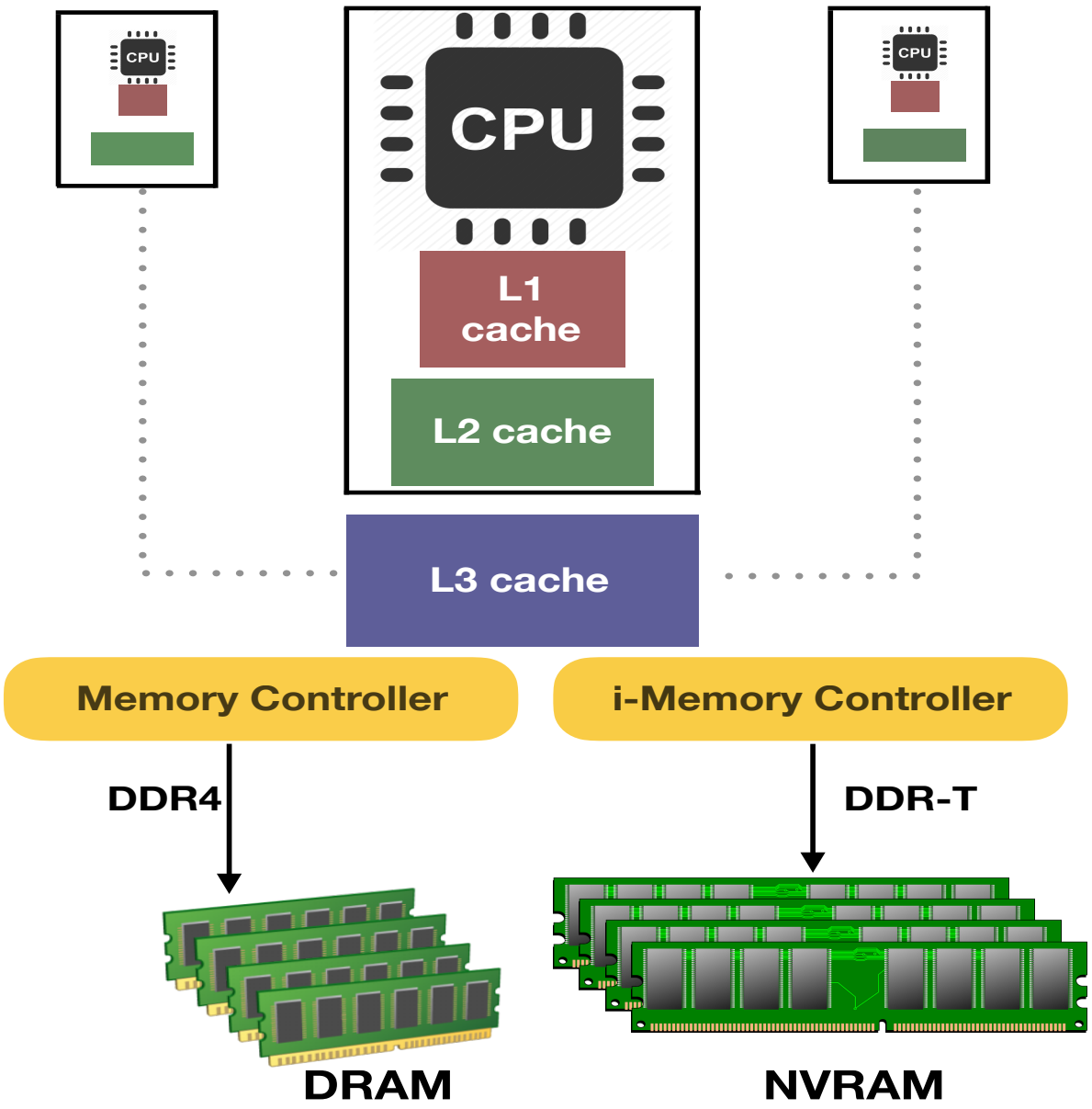


Figure 2.2 Memory architecture of Intel's Optane DC Node

Table 2.1 Experiment Platforms

Specifications	**Optane Node**
Model name	Intel(R) Xeon(R) 8260L @ 2.40GHz
Architecture	x86_64
CPUs	96
Sockets	2
Cores per socket	24
NUMA nodes	4
L1d cache	32 KB
L1i cache	32 KB
L2 cache	1 MB
L3 cache	35.3 MB
Memory Controllers	4
Channels/controller	6
DIMM protocol	DDR4
DRAM size	192 GB
NVDIMM protocol	DDR-T
NVRAM size	1.5 TB
Operating System	Fedora 27

proprietary add-on protocol to the DDR4 protocol, the Optane DC achieves asynchronous command/data timing and variable-latency memory transactions. To communicate with the iMC, the module controller in Optane DC PMM uses a request/grant scheme. The direction and timing of the data bus on which Optane DC resides is controlled by the processor. The processor sends a command packet per request to the Optane DC memory controller. The modules use 256 byte cache line access granularity which is larger than the 64 byte cache line access granularity used in DRAM.[Izr19]. Intel's asynchronous DRAM refresh (ADR) guarantees the CPU stores that reach it, will survive a power failure. The stores are flushed to NVDIMMs in less than 100 µs, which is the hold-up time. The iMC falls in the ADR domain but the caches do not. So, the stores will be persistent only after they reach iMC for which it uses a 72-bit data bus and transfers data in cache line granularity for CPU loads and stores. 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 [Izr19].

The Optane DC operates in three different modes. With a minor Linux kernel modification, we have configured the Optane DC to operate in a fourth mode. The configurations are described below.

2.4.1 Memory Mode

In Memory Mode, the Optane DC modules act as volatile main memory. DRAM acts as a direct-mapped cache for Optane DC with a block size of 4 KB and is managed by the CPU's memory controller. DRAM is no longer directly accessible, but allows for cache hits to be as fast as a DRAM access. Cache misses, however, can take as long as a DRAM cache miss plus an Optane DC access.

2.4.2 App Direct Mode

In App Direct Mode, the Optane DC modules act as persistent memory devices that are separate from main memory. DRAM is used as main memory. However, the Optane DC DIMMs are used through the block device entries that are created in the kernel. Once filesystems are installed on each device, the Optane DC modules are used as filesystems with significantly shorter access times than that of regular storage devices.

2.4.3 Mixed Mode

The Optane DC modules can also be partitioned to use part of the memory for persistent memory while the other part is used as volatile main memory. DRAM is still used as a cache for main memory rather than exposed as it would be in full App Direct Mode.

2.4.4 DRAM-NVM Hybrid Mode (Flat Mode)

Accessing both DRAM and Optane DC at the same time under a unified byte-addressable address space is not possible under the previous configurations. In [Izr19], Izraelevitz et al. ran experiments with their Optane DC modules not being cached by the system's DRAM by modifying the Linux kernel to recognize the Optane DC modules as RAM instead of persistent memory. We applied the changes to our node's kernel and set the DIMMs to App Direct mode, allowing us to see the Optane DC modules on NUMA nodes in addition to the DRAM NUMA nodes, which results in a combined main memory of the DRAM capacity plus the Optane DC capacity rather than only one or the other.

2.5 Experiments

Our aim is to evaluate Optane DC as an address space extender for main memory in HPC systems. We used an HPC application (VPIC), three HPC proxy-apps (AMG, LULESH, and SNAP) and a custom benchmark to evaluate the performance of Optane DC. We modified these applications[1] so we can allocate all the data either on the DRAM, or on Optane DC. We then compared the statistics we collected for both configurations. We performed a preliminary performance characterization using a custom STREAM-like benchmark that evaluates the performance of different types of kernels

[1] In order to have the applications allocate their data on Optane DC or DRAM only, we modified the applications to use the Simple Interface for Complex Memory (SICM) [Wil18] library, a NUMA-aware arena allocator for heterogeneous memory.

Table 2.2 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

encountered in HPC applications. Memory bandwidth information was collected for every stream used in a kernel that was parallelized using OpenMP [DM98]. The streams used in the benchmarks are representative of most of the streams found in HPC applications. We focus on different access patterns of data structures like sequential, strided, and random access. We also collect bandwidth numbers for matrix accesses and operations, for example, row-major access and stencil operations.

We have a test where we bypass the L3 cache by accessing elements that are apart by number of elements that fit in L3 cache.

Our experiments ran on the Optane DC node described in Section 2.4 and Table 2.1. We set the Optane DC modules in Memory Mode and the DRAM-NVM Hybrid Mode (Flat Mode), and compared the performance of all the applications for each mode. In Flat mode, we allocated memory on NVM and DRAM for different runs. In memory mode, memory was only allocated on NVM as DRAM was used as cache for NVM.

We performed strong and weak scaling for all of the HPC mini-apps and measured the total execution times, memory bandwidth, power consumption, last-level cache misses and double-precision floating point operations per second. We used LIKWID [Tre10] to collect performance counters. For our custom benchmark, we collected only the memory bandwidth for different kernels that we test.

Keeping in mind that we had Optane DC PMMs on only a single node, our experiments were not conducted on large number of processes or memory sizes. We ensured that our problem sizes were big enough to not fit into the last-level cache and so we can get a fair depiction of the performance of the different memories. Our problem sizes lie in a small/medium range as recommended by the authors of the mini-apps. We did not scale the number of processes to more than 48, i.e., half the number of processing units to avoid oversubscribing of resources. This was done in order to get correct performance numbers from the hardware performance counters. For the custom benchmark, we averaged the bandwidth measurements over 10 runs for every kernel which had a standard deviation up to 8%. For the HPC mini-apps, we average all measurements over 4 runs with a standard deviation of 11% for execution time. We describe the applications we use for our experiments below.

2.5.1 AMG

AMG is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids [Yan02]. It was developed at Lawrence Livermore National Laboratory (LLNL). It is an SPMD code that uses MPI and OpenMP threading within MPI tasks. AMG is a highly synchronous code. The communication and computation patterns exhibit the surface-to-volume relationship common to many parallel scientific codes. We use the default Laplace type problem on a cube with a 27-point stencil.

2.5.2 LULESH

LULESH [Kar13] is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers. It is C++ based applications. It was developed at LLNL as a part of co-design efforts for exascale computations. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It uses MPI and OpenMP for parallelization and is also memory bound.

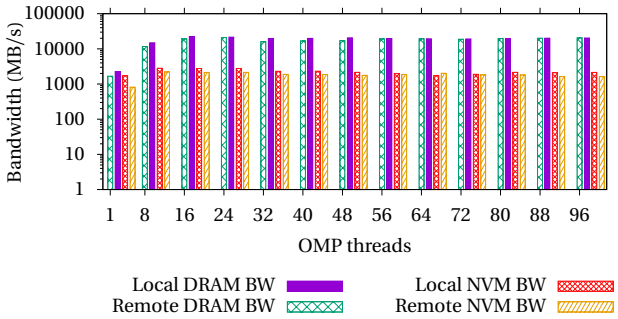
2.5.3 VPIC

Vector Particle-In-Cell (VPIC) [Bow09] is a simulation code developed at Los Alamos National Laboratory (LANL). It is an application that models kinetic plasmas in 1 to 3 dimensions. It uses MPI and OpenMP for parallelism. The code is comprised of kernels that compute multiple data streams at the same time and operate on entire data structures. The data structures scale based on the input decks and hence make VPIC memory bound.

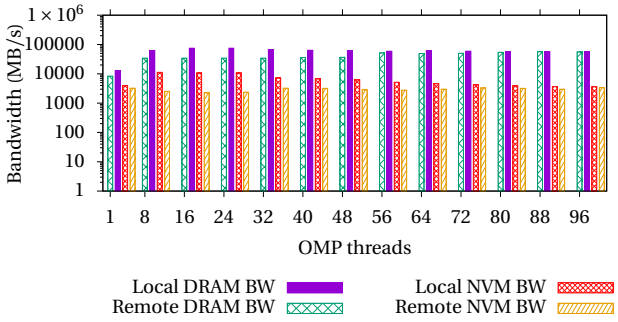
2.5.4 SNAP

SNAP [Sna] is based on the PARTISN code from LANL. SNAP mimics the computational workload, memory requirements, and communication patterns of PARTISN. The equation it solves has been composed to use the same number of operations, uses the same data layout, and loads elements of the arrays in approximately the same order. SNAP uses MPI to scale for HPC. We use the SNAP-C code. It is also a memory-bound application but is more bound by bandwidth than latency.

2.6 Results

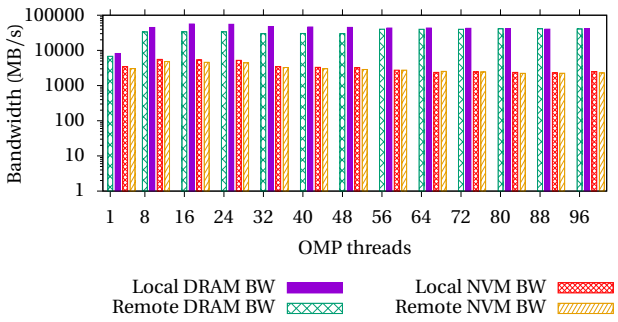


(a) Write-only stream bandwidth on the Optane node

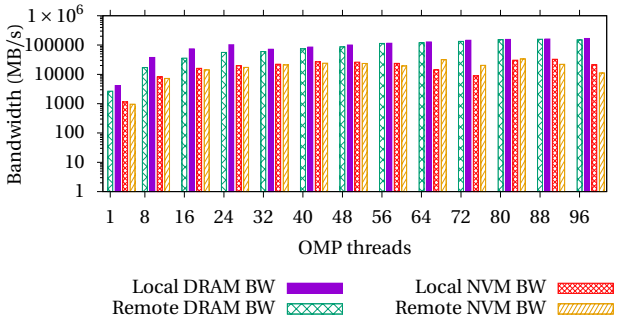


(b) 1 write+4 read stream bandwidth on the Optane node

Figure 2.3 Bandwidth measurement of Write-only and 1-write+4-read sequential access streams

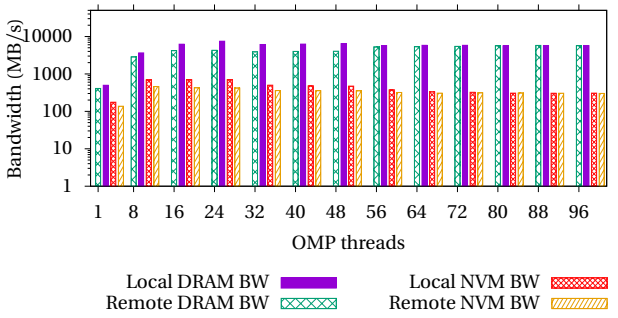


(a) Multiple stream kernel bandwidth on the Optane node

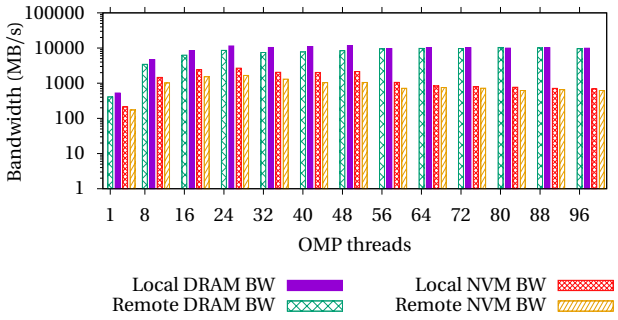


(b) Strided stream bandwidth on the Optane node

Figure 2.4 Bandwidth measurement of Multiple 1-write+4-read sequential access and Fixed stride access stream



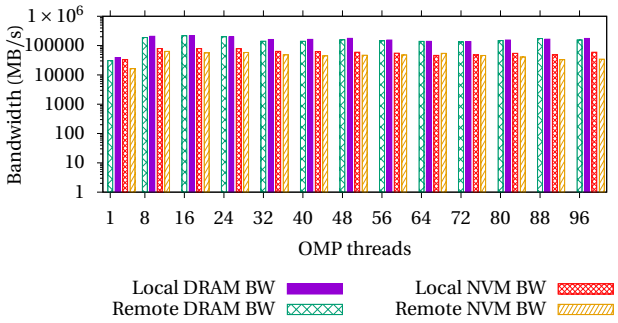
(a) Random stream bandwidth on the Optane node



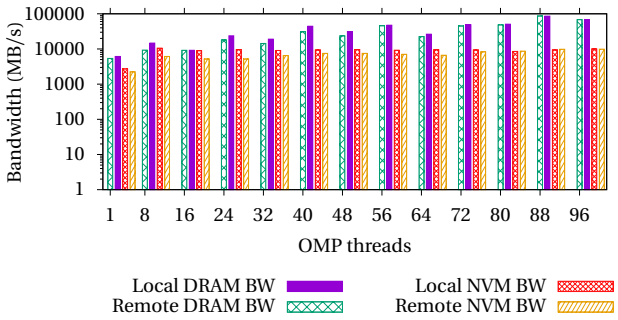
(b) LLC bypass stream bandwidth on the Optane node

Figure 2.5 Bandwidth measurement of Random and LLC bypass stride access stream

2.6.1 Performance evaluation of different streams on Optane DC



(a) Row major matrix stream bandwidth on the Optane node

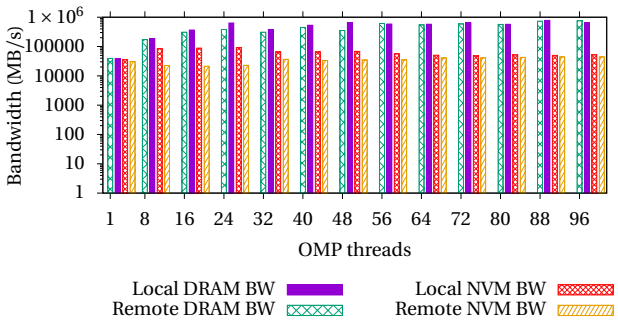


(b) Column major matrix stream bandwidth on the Optane node

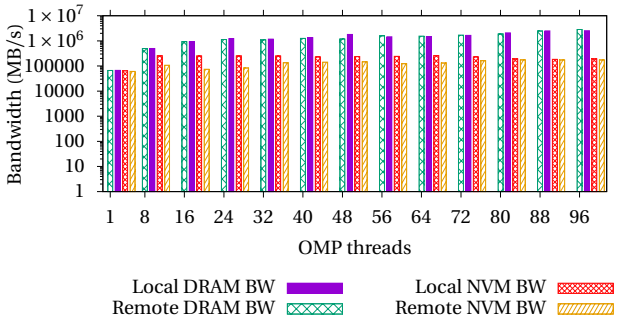
Figure 2.6 Bandwidth measurement of Row-major and Column-major matrix access stream

We evaluate the performance of various streams executed on Optane DC compared to DRAM. We perform strong scaling on the streams by increasing the OpenMP threads from 1 to 96 for different runs. We pinned the threads using `numactl -C` to specific processing units and then allocated the streams on every NUMA node to evaluate the effect of NUMA distances on memory bandwidth. We collected the effective bandwidth for all processing units and their local and remote NUMA node combinations and averaged the bandwidth results over 10 runs with a standard deviation of 7%. Every stream or data structure used in the experiment was 1 GB in size. All the plots have bandwidth on the Y axis which is depicted on a log scale and the number of threads on X axis.

Fig. 2.3a depicts results for strong scaling of the write-only stream. The average bandwidth for local NUMA nodes peaks at 48 threads and then it plateaus. The bandwidth of remote NUMA nodes also increases with the number of OpenMP threads and peaks at 24 threads after which it plateaus as well. This effect is caused by oversubscribing of resources beyond 48 threads which causes the memory controller queues to overflow. This can lead to serialization of loads and stores due to back pressure and nullifies the benefit of bank parallelism. We observe that the NUMA distances affect bandwidth of NVM up to 22% and up to 16% for DRAM in case of strong scaling for the write only stream. Fig. 2.3b shows results for the kernel with a single write and four read stream. The same effect is observed as for the write-only stream but the relative bandwidth for each run is more than 3x



(a) Row+column major matrix stream bandwidth on Optane node



(b) 9-cell stencil stream bandwidth on the Optane node

Figure 2.7 Bandwidth measurement of Row-Column matrix and 9 point stencil stream

the bandwidth of the write-only stream. This effect is observed for both DRAM and NVM, but DRAM achieves 8x higher bandwidth than NVM before oversubscribing. The difference between DRAM and NVM bandwidth worsens beyond 48 threads where they start to plateau. Such sequentially accessed streams are used mostly during initialization or problem generation phases of HPC applications. The above results indicate that utilization of local DRAM nodes in this phase is critical while not oversubscribing to compute resources.

Fig. 2.4a provides results for a kernel with multiple single write plus 4 read streams. The performance is similar to what is observed for the single write and four read stream but the effective bandwidth is slightly lower. The bandwidth for DRAM is 30% lesser than the single write and four read stream and 50% lesser for NVM. Such streams don't need a lot of parallelism to achieve maximum bandwidth however the access latency of the memory device will affect the performance. Fig. 2.4b depicts results for a kernel with a fixed stride access for increasing OpenMP threads. The stride is bigger than the cache line size. For this stream the NUMA distances have no effect on the memory bandwidth, except for fewer than 24 threads for DRAM. NVM scales similarly to the previous streams but achieves higher memory bandwidth. The memory bandwidth of DRAM keeps increasing with the number of threads until it peaks at 48 threads for NVM. This indicates that NVM bandwidth may be constrained by the core count irrespective of the access pattern. This stream achieves up to 8x the bandwidth of write-only stream for DRAM and 10x the bandwidth of the same for NVM. Fig. 2.5a shows results from a randomly accessed single write plus 4 read stream, which achieves the lowest effective bandwidth of all streams. The random accesses are determined by

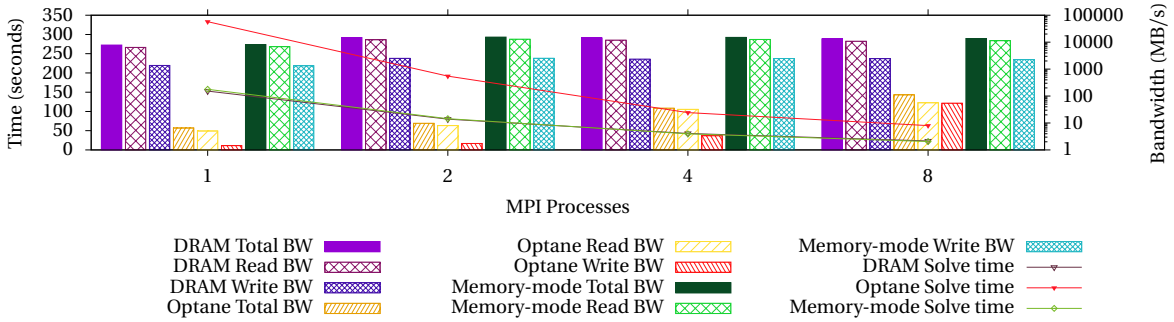
an indirection array that is initialized by the rand() function that generates the order of indices to access. The memory bandwidth for local DRAM and NVM nodes is up to 40% higher till 48 threads after which it plateaus and is unaffected by NUMA distances. This effect is observed because of not allowing the HW prefetcher to take advantage of any temporal locality. Hence, effective bandwidth is so low. Fig. 2.5b illustrates results for a single write plus 4 read stream forced to bypass the L3 cache for every access. In this stream, we observe that the bandwidth achieved by DRAM is up to 5x higher than NVM till 48 threads. The bandwidth remains plateaus for all NUMA nodes beyond that. This result indicates that if we take the caches out of the picture, the effective bandwidth achieved by Optane DC is not affected as much as DRAM despite the higher access latencies. Such streams with varying access patterns are common in the computation phase of HPC applications. It becomes essential to identify the access patterns of each linear stream at every stage of the program and place the stream in the most effective memory node given the amount of compute resources subscribed to.

Fig. 2.6a depicts results from a stream that accesses a single write and two read matrices in row-major order. The scaling pattern is similar to the write-only stream but the effective bandwidth is almost 10x the bandwidth for DRAM and up to 40x the bandwidth for NVM when compared to the same. This high bandwidth may be observed due to the large cache size and prefetching which take advantage of the spatial locality. Again, such a stream is a common occurence during the initialization phase of HPC applications and it can benefit from being placed in local DRAM memory. Fig. 2.6b assesses a stream accessing a single write and two read matrices in column-major order. Here, the NVM bandwidth remains steady with the increase in threads with a slight advantage for local NVM nodes. However, for DRAM, NUMA distances do not make a big difference and the bandwidth keeps increasing with the increasing number of threads. The bandwidth is 3x lower for DRAM and 8x lower for NVM than the row-major stream for most threads, except for the 2 highest thread counts, where DRAM bandwidth jumps up to 86 GB/s. Fig. 2.7a shows results for a kernel that accesses a single write plus two read matrices stream, all in row-major except for the last read stream in column-major order. It achieves 4x higher memory bandwidth than the row major stream for both DRAM and NVM. It achieves such high bandwidth due to spatial locality and prefetching in the cache. The scaling pattern is similar to the row major access stream but achieves substantially higher bandwidth for both memories. Lower thread counts give advantage to local NUMA node but beyond 48 threads, there is no difference. Fig. 2.7b depicts results for a 9-point stencil kernel, which scales similarly Fig. 2.7a but with approximately 40x the bandwidth for all thread counts and memories. This stream achieves the highest bandwidth of all the streams due to a lot of spatial and temporal locality in the cache. The bandwidth observed is effectively the bandwidth of the cache. Such matrix streams occur during the computation phase of a HPC application. Although NUMA distances do influence the effective bandwidth of these streams, the memory device used affects the bandwidth significantly with increasing number of threads. Also the effective use of cache locality can help in achieving higher performance for both memory devices.

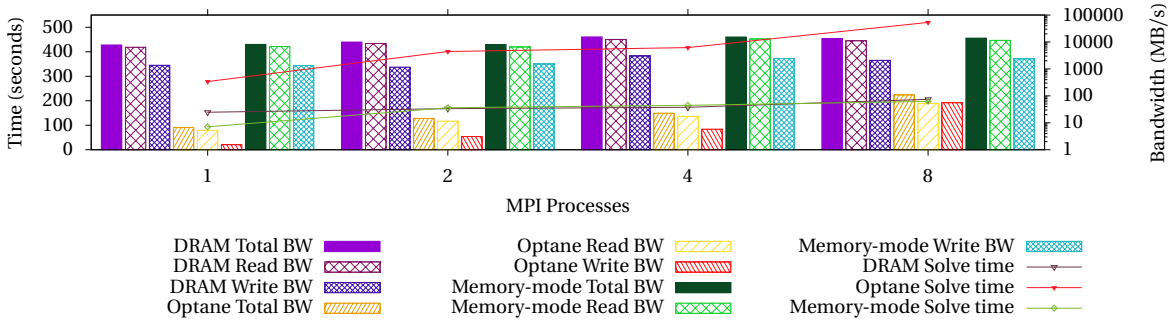
Taking all the results into account, we can infer that the higher latency of Optane DC and the lack

of optimal cache support causes it to not perform as well as DRAM. We observe that with effective caching and prefetching, Optane DC can deliver much better performance than what is observed in our evaluation. However, these results give a fair idea of which workloads can benefit from NVM and gives a quantification of the performance impact by using NVM in place of DRAM.

2.6.2 HPC Benchmark evaluation



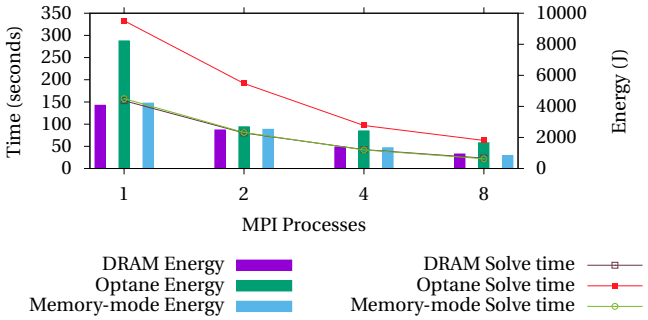
(a) AMG Bandwidth Strong Scaling



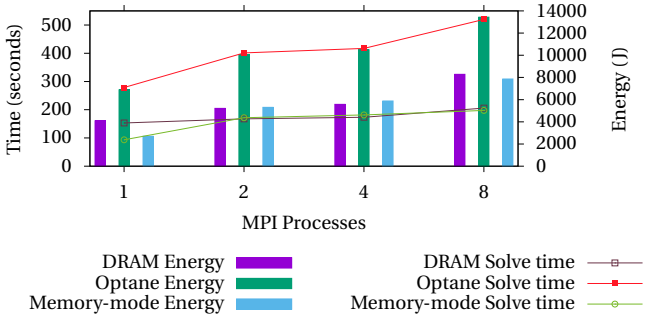
(b) AMG Bandwidth Weak Scaling

Figure 2.8 Bandwidth measurement for AMG

For benchmarks, we plot the application bandwidth and execution time measurements together in a single graph to observe their correlation. Similarly, we plot the energy consumption and execution time measurements together. We also plot the cycles/instruction (CPI) and L3 miss ratio together. We plot these graphs for both strong and weak scaling experiments. For bandwidth and execution time graphs, we plot execution time on the left-hand side y-axis in seconds and depicted as lines. The bandwidth is plotted on the right-hand side y-axis in megabytes/seconds (MB/s) and depicted as a bar chart. For energy consumption and execution time graphs, we again plot our execution time on the left-hand side y-axis as lines. The energy is plotted on the right-hand side y-axis in Joules (J) as a bar chart. For CPIs and L3 miss ratio graphs, we plot the CPIs on the left-hand side y-axis as lines. The L3 miss ratio is plotted on the right-hand side y-axis as a bar chart. Both CPIs and L3 miss ratio have no unit. Bandwidth is plotted on a logarithmic scale whereas all other

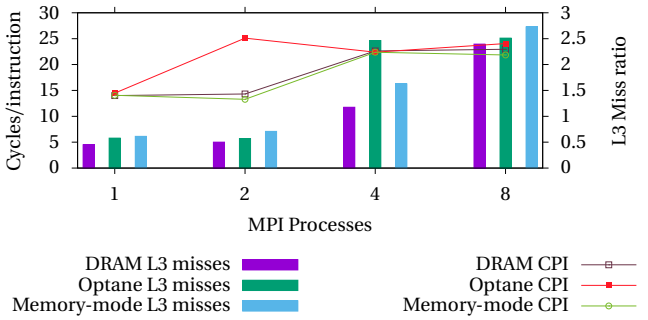


(a) AMG Memory Energy Consumption Strong Scaling

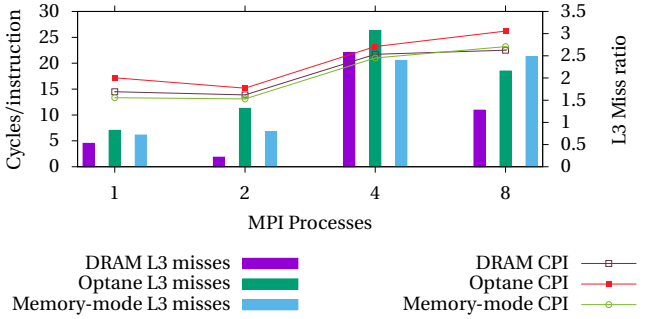


(b) AMG Memory Energy Consumption Weak Scaling

Figure 2.9 Memory energy consumption for AMG



(a) AMG L3 Miss Ratio and CPI Strong Scaling



(b) AMG L3 Miss Ratio and CPI Weak Scaling

Figure 2.10 L3 miss ratio and CPI for AMG

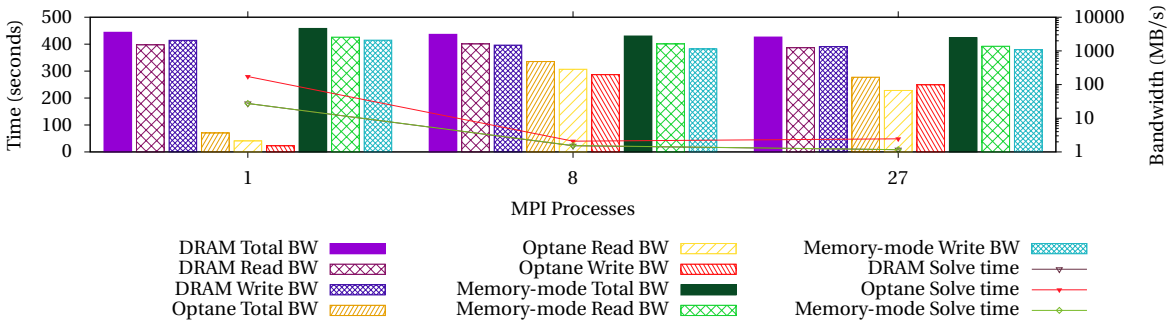
measurements are plotted on a linear scale. The x-axis depicts the number of MPI processes for a given execution.

2.6.2.1 AMG results

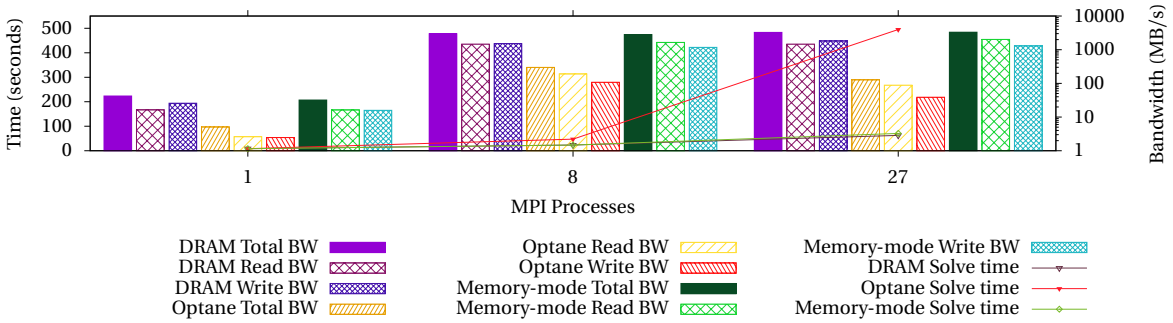
Fig. 2.8a and Fig. 2.8b depict the graphs for strong and weak scaling of AMG, respectively. For strong scaling, we scale the processes from 1 to 8 using MPI and keep the data size constant by reducing the size per processor from 256 to 128. For weak scaling, we again scale the processes from 1 to 8 and keep the size per processor same at 256 as we scale the data size proportionally. We observe that memory bandwidth for Optane-only execution is 2 to 3 orders of magnitude lower than DRAM-only and Memory-mode executions. This results in more than 2x higher execution times for Optane-only execution. This result is observed for both strong and weak scaling cases. The observed bandwidth remains fairly constant for both strong and weak scaling across all number of processes for DRAM-only and Memory-mode executions but rises for Optane-only execution. The lower bandwidth for Optane-only execution is a result of the higher access latency of Optane DC. The Memory-mode execution matches the performance of DRAM-only execution because it uses DRAM as a cache. The problem sizes for this experiment are small enough to fit into DRAM. Hence, there is minimal difference between the performance of DRAM-only and Memory-mode executions. Fig. 2.9a and Fig. 2.9b depict the energy consumption and execution time of all three executions of AMG for strong and weak scaling. We observe that the energy consumption of Optane-only execution is 2x higher than DRAM-only and Memory-mode executions. This is due to its higher execution time even though the power consumed by Optane-only execution is lower than the other executions. Fig. 2.10a and Fig. 2.10b depict the L3 cache misses and Cycles/Instruction(CPI) for strong and weak scaling of AMG for all 3 executions. In strong scaling, we observe that the CPIs for Optane-only execution are higher for low number of processes. For higher number of processes they are almost equal to the other 2 executions. However, the L3 cache misses increase rapidly with the number of processes. This also explains the difference in execution times of Optane-only execution and other 2 executions. The increase in L3 cache misses in Optane-only execution is observed under weak scaling as well but the CPIs are consistently higher than the CPIs for the other 2 executions. AMG is a memory bound application that is heavily affected by memory access speeds. Hence, the Optane-only executions suffer from heavy performance degradation in terms of execution time and energy consumption. Such applications that require faster access speeds will suffer from a NVM-only approach.

2.6.2.2 LULESH results

Fig. 2.11a and Fig. 2.11b depict the graphs for strong and weak scaling of LULESH, respectively. We increase the number of processors from 1 to 27 using MPI, as LULESH accepts only cubes of natural numbers as a valid configuration. For strong scaling, we keep the problem size constant at 125000 data points and increase the number of processors. We observe that when LULESH is running only on Optane DC in flat mode, it has a 50% higher execution time than DRAM-only and Memory-mode configurations for a single process and 8 processes. For 27 processes, the execution time is approximately 6x higher. This effect is observed because the memory bandwidth is almost

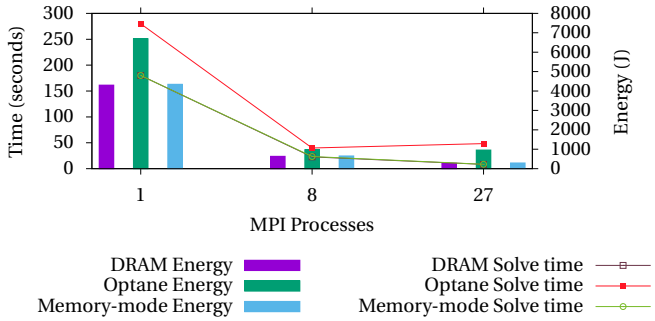


(a) LULESH Bandwidth Strong Scaling

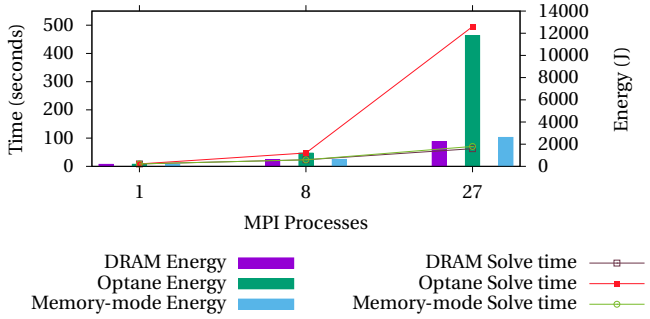


(b) LULESH Bandwidth Weak Scaling

Figure 2.11 Bandwidth measurement for LULESH

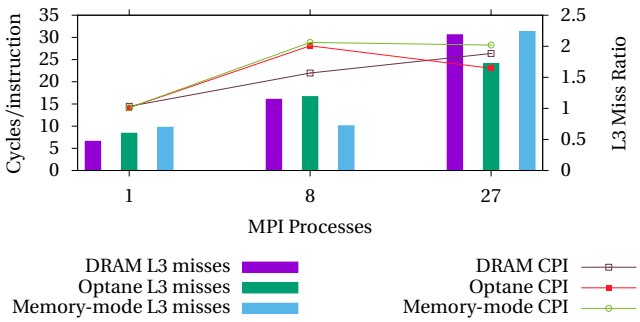


(a) LULESH Memory Energy Consumption Strong Scaling

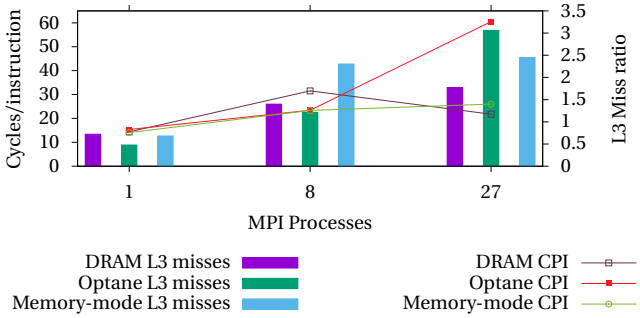


(b) LULESH Memory Energy Consumption Weak Scaling

Figure 2.12 Memory energy consumption for LULESH



(a) LULESH L3 Miss Ratio and CPI Strong Scaling



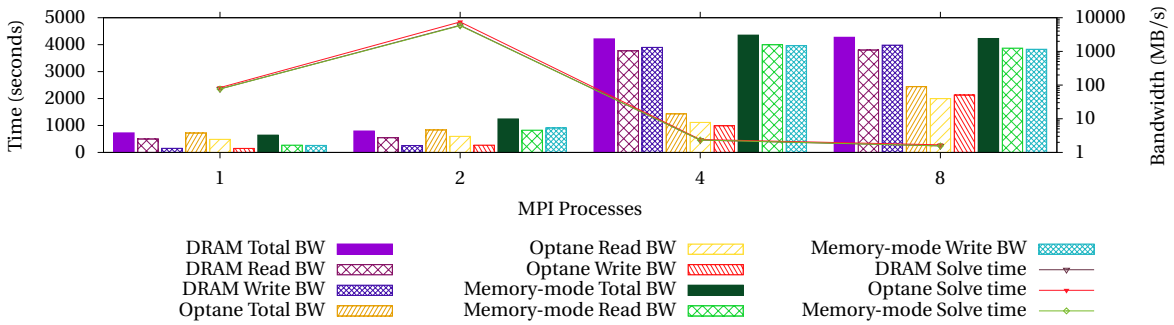
(b) LULESH L3 Miss Ratio and CPI Weak Scaling

Figure 2.13 L3 miss ratio and CPI for LULESH

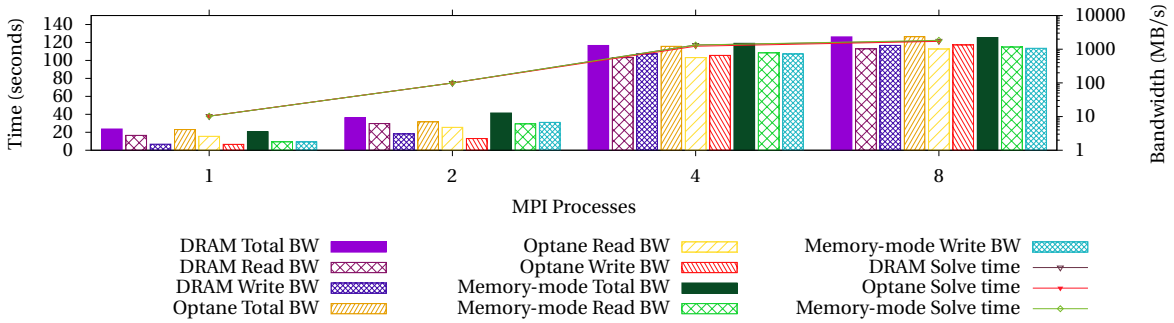
an order of magnitude lower than DRAM-only and Memory-mode executions. The difference in execution time is more amplified in weak scaling for 27 processes due to lower bandwidth. In weak scaling, we keep the number of data points per process constant at 15,625. The low bandwidth on Optane-only execution is due to the high access latency of Optane DC. We do not observe this effect in Memory-mode because of the DRAM caching in Memory-mode. Hence, there is no difference in execution time for DRAM-only and Memory-mode execution as the access latency would be the same. In flat mode, Intel's ADR is at work to guarantee persistence of data, which can be hampering performance and the write amplification might increase access latencies, too. The memory bandwidth in weak scaling is similar to strong scaling except for single process execution in strong scaling, where Optane DC has 2 orders of magnitude lower bandwidth. Fig. 2.12a and 2.12b depict the memory energy consumption of LULESH in each mode for strong and weak scaling, respectively. We observe that the energy consumption for Optane DC is up to 60% higher than DRAM-only and Memory-mode executions. This is in direct correlation to the execution time because the power consumption for Optane DC is up to 30% less than DRAM. For smaller problem sizes under weak scaling with fewer threads the energy consumption of Optane DC is similar to the other executions. A trade-off between capacity, problem size and performance needs to be achieved to keep the application execution within desired energy budgets. Fig. 2.13a and Fig. 2.13b depict the L3 cache miss ratio and CPIs for LULESH strong and weak scaling, respectively. For strong scaling, the L3 cache misses increase with the number of processes but they are lower for Optane-only execution on 27 nodes. In weak scaling, the CPIs are 3x higher compared to DRAM-only and the L3 cache misses are significantly higher. These add to the execution time, explaining the difference in

execution time for weak scaling of LULESH. Applications like LULESH are dependent on memory bandwidth for performance. These applications can reduce their energy consumption with NVM when they are running smaller problem sizes and fewer threads.

2.6.2.3 VPIC results



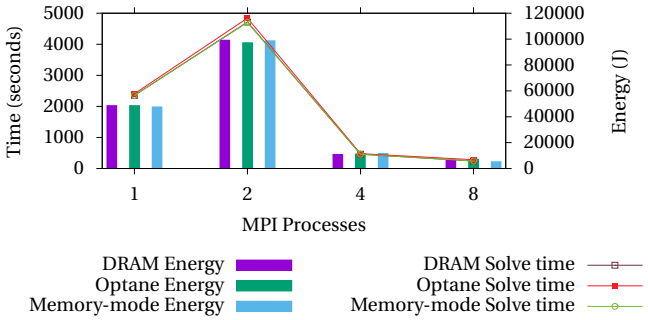
(a) VPIC Bandwidth Strong Scaling



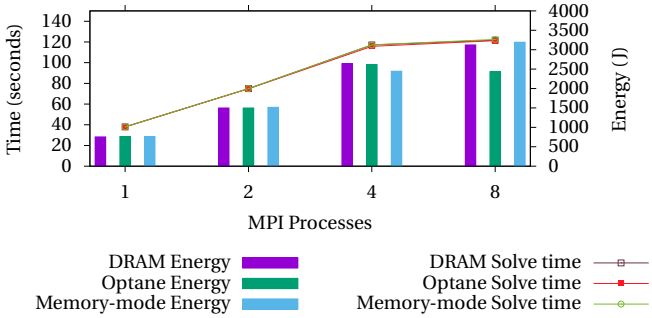
(b) VPIC Bandwidth Weak Scaling

Figure 2.14 Bandwidth measurement for VPIC

Fig. 2.14a and Fig. 2.14b depict the execution time and memory bandwidth for strong and weak scaling of VPIC. We use the 'lpi' input deck provided by the authors of the benchmark for our experiments. For strong scaling, we increase the number of processes from 1 to 8 using MPI and keep the problem size the same by changing the 'nppc' value from 2048 to 256. The 'nppc' variable in the input deck determines the number of particles/cell for each species in the plasma. We observe that NVM-only execution of VPIC is 2 to 16% slower than DRAM-only and Memory-mode executions. For weak scaling, we keep the problem size per process same by keeping the 'nppc' value at 512. This slowdown is caused by the lower bandwidth observed for Optane-only execution. Optane-only memory bandwidth is at least an order of magnitude lower than DRAM-only and Memory-mode bandwidth for larger number of processes in case of strong scaling. For weak-scaling, the memory bandwidth of Optane-only execution is similar to DRAM-only execution and Memory-mode execution and hence there is no difference in execution times either. Fig. 2.15a and Fig. 2.15b

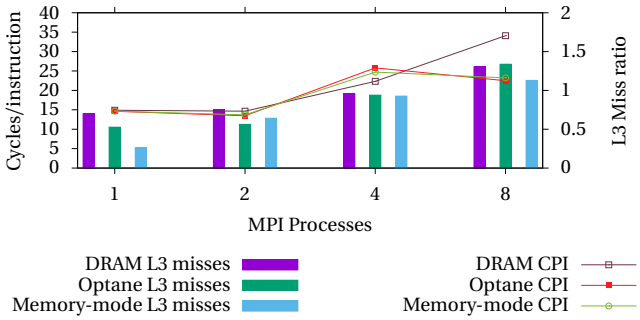


(a) VPIC Memory Energy Consumption Strong Scaling

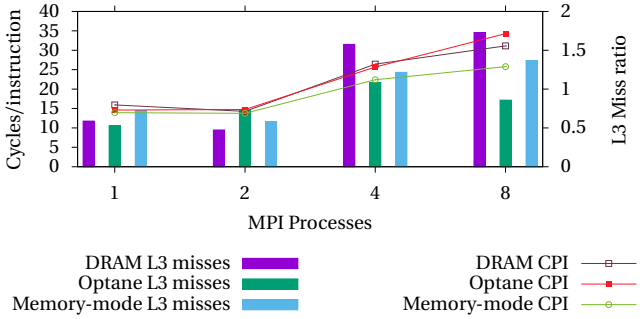


(b) VPIC Memory Energy Consumption Weak Scaling

Figure 2.15 Memory energy consumption for VPIC



(a) VPIC L3 Miss Ratio and CPI Strong Scaling



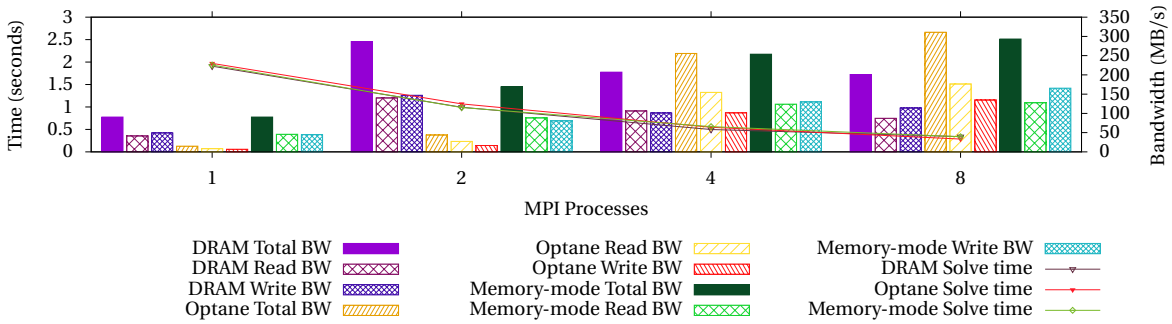
(b) VPIC L3 Miss Ratio and CPI Weak Scaling

Figure 2.16 L3 miss ratio and CPI for VPIC

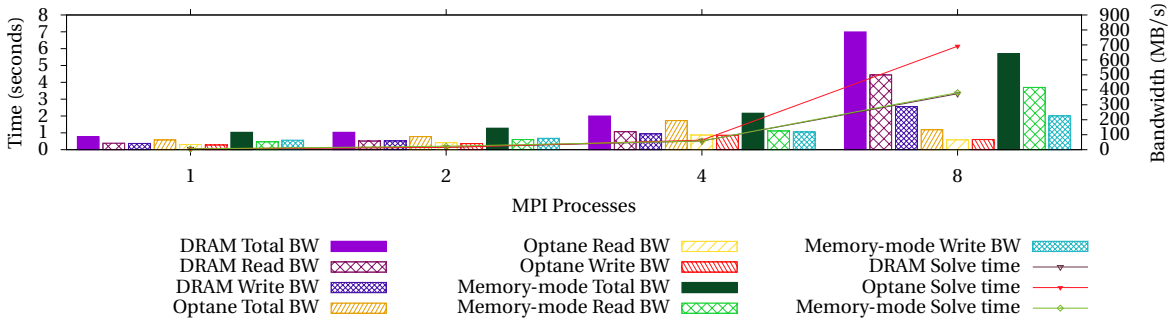
depict that the memory energy consumption for VPIC strong and weak scaling, respectively. The Optane-only execution's energy consumption remains constant with strong scaling of VPIC similar to the DRAM energy consumption. However under weak scaling, Optane-only execution's energy

consumption rises at slower rate than the other 2 executions. As the execution times are similar for all execution under weak scaling, the energy consumed by all three executions is also similar. Fig. 2.16a and Fig. 2.16b depict the L3 cache miss ratio and CPIs for strong scaling and weak scaling of VPIC. Even though the cache misses increase with strong scaling for Optane DC execution, the CPIs remain lower than DRAM-only execution. The cache miss ratio for Memory-mode execution rises at a slower rate than NVM-only and DRAM-only execution. In weak scaling, Optane-only execution results in fewer L3 cache misses than DRAM-only and Memory-mode execution but higher CPIs. This keeps the execution times of Optane-only execution low for weak scaling. VPIC optimizes its cache hits, as seen in the results, to achieve higher performance and hence there is minimal difference in execution times of all three executions even though there is a significant difference in bandwidth. Such applications can benefit from Optane DC by reducing energy consumption while not compromising on performance.

2.6.2.4 SNAP results



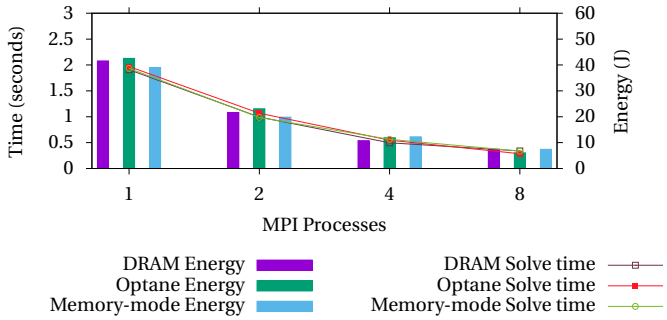
(a) SNAP Bandwidth Strong Scaling



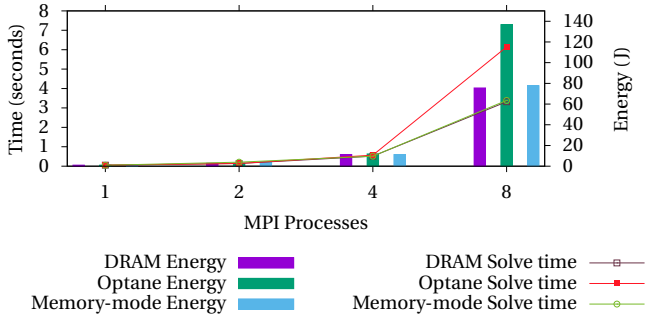
(b) SNAP Bandwidth Weak Scaling

Figure 2.17 Bandwidth measurement for SNAP

Fig. 2.17a and Fig. 2.17b depict the memory bandwidth and execution time for strong and weak scaling of SNAP in Optane-only, DRAM-only, and Memory-mode executions. We use MPI to scale the number of processes from 1 to 8. We use the C-version of SNAP that is compiled with mpicc. Here,

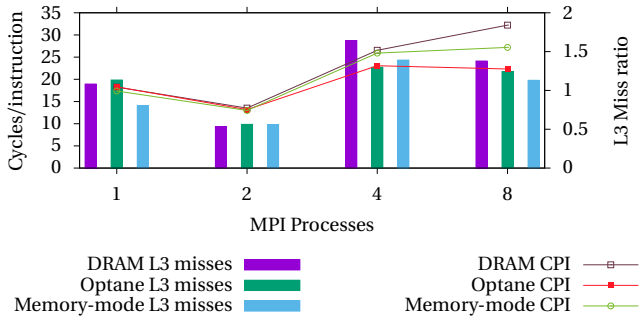


(a) SNAP Memory Energy Consumption Strong Scaling

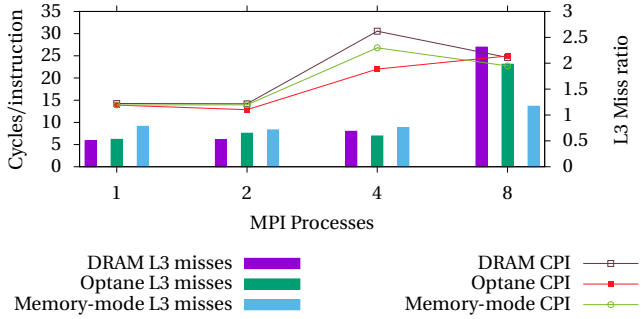


(b) SNAP Memory Energy Consumption Weak Scaling

Figure 2.18 Memory energy consumption for SNAP



(a) SNAP L3 Miss Ratio and CPI Strong Scaling



(b) SNAP L3 Miss Ratio and CPI Weak Scaling

Figure 2.19 L3 miss ratio and CPI for SNAP

we observe that the execution times in strong scaling vary only slightly for all the three executions. Optane DC memory bandwidth increases with increasing number of processes and is the highest out of all three executions for 8 processes. This is reflected in the execution times of all 3 executions.

However, for weak scaling the execution time increases for Optane-only execution when we scale up to 8 processes. We also observe that the memory bandwidth for Optane-only execution does not increase with weak scaling. This explains the 2x higher execution time for Optane-only execution compared to DRAM-only and Memory-mode execution. Fig. 2.18a and Fig. 2.18b depict the energy consumption of SNAP execution in Optane-only, DRAM-only, and Memory-mode executions. The energy consumption remain fairly constant across all three executions for both strong and weak scaling. As the execution times are similar for these executions the energy consumption of Optane-only execution is the least amongst the 3 executions, when the number of processes are 4. Under weak scaling, Memory-mode consumes the least power overall but due to larger execution times it consumes the maximum energy. Fig. 2.19a and Fig. 2.19b depict the plot of L3 cache miss ratios and CPIs for strong and weak scaling of SNAP. We observe that CPIs differ significantly for strong scaling at higher number of processes, where Optane-only execution experiences the least CPIs. The L3 cache misses actually rise with increasing number of processors for all three executions. The L3 cache misses scale with increasing number of processes for weak scaling, too. However, the changes in CPIs for higher number processes are erratic. The C-version of SNAP was created to take advantage of the vector operations in the Intel microarchitecture and is highly optimized to take advantage of the cache hierarchy and prefetching methods. Due to this, the performance degradation is minimal with SNAP, which results in reduction in energy consumption on the low power Optane DC memory.

2.7 Future Work

Intel's Optane DC PMM opens up a range of possibilities for use of NVM in various applications. We plan to explore the use of NVDIMMs in optimizing HPC applications. The different modes in which you can operate Optane DC PMMs have potential to optimize many HPC applications. With our performance characterizations, we plan to develop allocation policies for NVM for different configurations. This will help accommodate large problems with fewer compute nodes and operate under a required compute and energy budget. There are many variable components to Optane DC that need to be characterized as well to take full advantage of the technology, for example, its variable latency and power consumption. We plan to study the longevity and varying latencies of NVDIMMs when used with HPC workloads and their susceptibility to faults and failures. This will help improve the resiliency of the supercomputers that would use NVDIMMs.

In addition to the massive memory capacity, NVDIMMs have data persistence, which can help develop novel resiliency techniques. They can be used to store lightweight checkpoints and restart processes that fail. We plan to explore the possibility of building a fast and lightweight checkpoint/restart mechanism for exascale supercomputers [Pat17]. It can also be used to maintain metadata of large-scale systems and help in lookup operations. The data stored on NVDIMMs can be used to detect and correct soft errors by using checksums for increasing reliability. We will explore the use of NVDIMMs to increase the reliability of computations. We also plan to investigate the

kernel and user-level support required for efficient use of NVDIMMs. Compiler-based analysis and profiling information can help optimize the use NVDIMMs for various applications. We will also assess support for other memory technologies that can be incorporated into the DRAM-NVM hybrid memory hierarchy.

2.8 Conclusion

In this chapter, we performed characterization of a hybrid memory system comprising of a slower NVM device and a faster DRAM device. We conclude that using a slower byte-addressable memory device hampers the performance of memory-bound HPC applications due to higher access latencies and lower memory bandwidth. However, using the DRAM as a cache for the slower NVM device maintains the performance of HPC applications observed on DRAM-only memory systems while increasing the memory capacity of the system, which needs to be further verified on large problem sizes. Although using NVM as main memory directly hampers the performance, it has the potential to reduce the energy consumption of HPC applications with reasonable trade-offs. Optane DC PMMs enables us to close the gap between core count and memory capacity scaling. With this assessment, part of hypothesis has been shown, namely that a byte-addressable heterogeneous memory system requires static and dynamic analysis to optimize compute-bound HPC applications and that there is a potential for energy benefits.

CHAPTER

3

NVM-BASED ENERGY AND COST EFFICIENT HPC CLUSTERS

Table 3.1 Comparisons of related work (BW: memory bandwidth, L: latency, ET: exec. time, EN: energy, CB: cache behavior)

Publication	Characteristics	Hybrid Memory	Allocation Policy	Applications	Problem sizes	Cost Analysis
Our work	BW ET EN	Yes	**Memory agnostic**	AMG,LULESH,VPIC	Large	**Yes**
M. Ben et al. [Ols19]	BW ET	Yes	Memory aware	AMG,LULESH,SNAP,QMCPACK	Large	No
Patil et al. [Pat19]	BW ET EN CB	Yes	Memory aware	AMG,LULESH,SNAP,VPIC,Custom	Small, Medium	No
Izraelevitz et al. [Izr19]	BW L	Yes	Memory aware	SPEC 2006,2017,PARSEC	Small	No
Peng et al. [Pen19]	BW EN L	Yes	Memory aware	GAP	Large	No

3.1 Introduction

HPC systems have been instrumental in running large-scale simulations of scientific problems in almost every field. These scientific applications require large compute and memory resources, e.g., the Summit supercomputer [Hin18] at Oak Ridge National Laboratory (ORNL), to perform the simulations within reasonable amounts of time and with sufficient accuracy. According to the latest TOP500 list [Topa], we are currently within the petascale era of compute capability (PetaFLOPS), where problem sizes run into petabytes (PBs). By 2021, we expect to achieve exascale compute capability which will increase the problem sizes due to more complex simulations including whole slide image analysis [Aur; Qiu14]. To run these massive simulations and analysis, along with exaFLOP compute capability, we also need a large amount of memory resources.

HPC systems today are large clusters of nodes [Hpc] with compute and memory resources. The

compute resources include the primary CPU and often additional co-processors or accelerators such as GPUs [LL15]. Memory is usually comprised of DRAM but recently accelerators have included separate high bandwidth DRAM (HBM) modules. The compute and memory resources across nodes are connected to each other with high-speed switches and interconnects. The clusters also have cooling and power equipment.

According to the TOP500 list released in June 2020 [Topa], the 10 fastest supercomputers in the world have a large number of nodes, which deliver 27 to 513 peak petaFLOPS and have main memory ranging between 200 TB to 4.85 PB. Their peak power consumption ranges from 1.5 to 28 MW. The total number of compute nodes in these clusters are in the multiples of thousands. Due to the use of accelerators in recent years, the number of compute nodes in a cluster has reduced with the increase in compute density per node. DRAM based main memory has scaled, too, but only at *half* the rate of increase in compute capability [Mut13].

In order to achieve exascale capability, memory density needs to increase at the same rate as core count to solve larger problems. The Frontier supercomputer that will be operational in 2022 has an expected aggregate system memory of 10 PB [Fro] with approximately 6,400 nodes and a power envelope of 30 MW. Its acquisition cost is $600 million. However, if only traditional DRAM memory is used then memory density of the Frontier machine will not see any significant increase compared to Summit. This will restrict the problem sizes that can be run on the Frontier machine and reduce the ability to achieve results with higher accuracy and analysis with finer granularity. We focus on increasing the memory density of compute nodes and memory per core ratio in future HPC systems to support such large problem sizes with cost and energy efficiency.

NVM is a byte-addressable, high-density memory that can help increase the amount of memory in a single node of a cluster [Nai15]. NVM can be produced using various technologies like Phase Change Memory (PCM) [Lee09], spin-torque transfer RAM (STT-RAM)[Apa13] or Resistive RAM (Re-RAM). NVM has a lower power consumption, as constant refreshes to maintain state are not required. However, all of these memories are slower than DRAM in terms of access latencies. Intel has been first-to-market with a NVDIMM form factor that can be used in conjunction with DRAM. The NVM device is based on PCM technology known as Optane DC PMM. The Aurora supercomputer will also have support for Optane DC PMM. It is eight times denser than DDR4 DRAM but is also approximately six times slower in terms of write latency [Pat19; Izr19]. Also, Optane DC is cheaper than DDR4 in terms of cost per GB of memory by a factor of 1.5 to 2 depending on the capacity of the DIMM [Obj].

Contributions: This work evaluates the hypothesis that using NVM devices like Intel's Optane DC PMM as an extension to the existing memory hierarchy can support larger problem sizes in fewer number of nodes, keeping cost and power consumption within a target budget, compared to DRAM-only nodes. The slower write speeds of NVM and lower aggregate compute capabilities over all nodes may reduce the performance of the system, but the reduced inter-node communication over the interconnect, increased memory per core ratio, reduced acquisition cost and energy efficiency provide be a trade-off for the slowdown [GM11]. The increased memory size that NVM can bring to

a node will result in maintaining the required overall memory capacity across a cluster with fewer number of nodes. This hypothesis is tested by measuring the performance characteristics and power consumption of compute nodes with and without NVM by executing large problem sizes of a set of HPC applications (VPIC, AMG and LULESH).

Recent works [Pat19; Izr19] characterize the performance of Optane DC for different HPC workloads and benchmarks compared to DRAM. However, they lack the performance characterization of a system with a DRAM-NVM based hybrid memory address space where the application allocates data structures on both DRAM and NVM for a given execution combined with a comparison to the same execution on a traditional DRAM-based HPC systems. Our work fills this gap and provides novel insights for running applications on large memory nodes. It also provides the baseline comparison for procuring future HPC systems while considering different memory systems.

In section 3.2, we discuss related work. In section 3.3, we review the background of NVM in HPC. In section 3.4, we provide the details of the architecture of our proposed experimental setup. In section 3.5, we evaluate the experiments we conduct to test our hypothesis and in section 3.6, we assess the observed results and project the benefits of NVM in HPC. In section 3.7, we discuss our future work and we conclude the chapter in section 3.8.

3.2 Related Work

Tab. 3.1 provides a feature comparison of our work to recent studies that characterize the performance of NVM technologies especially with Intel's Optane DC PMM.

Izraelevitz et al. [Izr19] evaluated the memory access characteristics of Optane DC PMM for different file-systems, database applications and performance benchmarks. They found that Optane DC boosts the performance of file-systems and database applications due to lower latencies than storage devices. In contrast, we focus on using Optane DC as an extended memory address space for the existing memory hierarchy for HPC applications. Patil et al. [Pat19] characterized the performance of a DRAM-NVM hybrid memory system for HPC applications. They measured the performance of frequently occurring HPC kernels and HPC applications executing on Optane DC with executions on DRAM and executions with DRAM as cache for Optane DC. They compared the performance of all memory devices whereas we are measuring the performance of two different memory systems. Peng et al. [Pen19] evaluated the Optane DC PMMs in all the configurations available and also measured the performance of separating reads and writes on a DRAM-NVM memory system. Our work focuses on comparing the performance of memory system agnostic executions of HPC applications on DRAM-NVM and DRAM-only memory systems with a flat address space.

Recent studies focus on creating systems and policies to enable hybrid memory architectures. M. Ben et al. [Ols19] used profiling information obtained from PEBS based characterizations to guide memory allocations on complex memory systems. Zhou et al. [Zho19] proposed a victim-aware cache policy to improve the lifetime of NVM in a hybrid memory system. Rodriguez et al. [RR14] examined write-aware replacement policies of data in PCM-based systems.

There have been many studies to identify challenges related to achieving exascale systems. Bergman et al. [Ber08] detailed numerous challenges that need to be overcome to reach exascale HPC capability. They outlined the memory challenges the current systems face and survey many options to overcome them. They also examine the challenges related to capping the power of exascale systems. Peterka et al. [Pet19] identified similar challenges to achieve exascale computing and focused on memory bandwidth and capacity issues and how to mitigate them. Ashraf et al. [Ash19] examined methods of improving HPC performance under desirable power caps. Gamatié et al. [Gam19] empirically surveyed NVM technologies for energy efficient HPC systems. In contrast, we focus on evaluating available technology to increase the memory density and keep power consumption and operating costs of HPC systems down.

3.3 Background

Supercomputer architecture has been constantly evolving over the last six decades. These changes were driven by the need to solve larger and more complex problems, technological advances and cost justification. In the 1970s, supercomputers were leading edge processor machines specially designed to perform floating point operations as fast as possible [LL15]. However, within a decade, supercomputers were built using multiple processors connected by different means as the cost of building faster processors rose astronomically. This eventually led to the development of cluster computing, where commodity hardware was connected over a fast network (e.g., Beowulf Clusters). The clusters were further sped up by introduction of multi-core processors, which were designed to reduce the speed gap between memory and processors, increase instruction level parallelism and improve power efficiency. A decade ago, GPUs were introduced in clusters in order to utilize Single Instruction Multiple Data (SIMD) parallel processing and boost the performance of the supercomputers. Today, large clusters with processors and GPUs, connected by a high speed interconnect, are leading the TOP500 list of supercomputers.

Memory architecture also went through changes along with the changes in supercomputer architecture [Nai15]. With monolithic supercomputers, the memory system was simple with a single type of main memory. As the processing speeds increased, more memory was required to support the large problems that were executed on the faster processors. However, memory access speeds did not increase at the same rate as processor speeds. This led to the introduction of different memory technologies that are used as buffers or caches to reduce the performance impact of larger but slower memory. The memory architecture was split into different tiers, from faster but smaller memory to slower memory with massive capacities, in order to support multi-core processors and parallelism. This has created the memory hierarchy we have today.

The changes in supercomputer architecture were also sparked by the evolution of different or more efficient numerical solvers and methods for scientific problems over the decades as well as development of software that enabled the efficient use of new hardware. Supercomputers eventually became very specialized, i.e., to solve only certain types of problems efficiently [Hpc]. The Summit

supercomputer is very efficient in running artificial intelligence and big data problems, which are predominantly SIMD-based problems. It may not be the best suited platform for other types of problems.

Memory technology has not scaled well in terms of capacity compared to the scaling of processing speed [Mut13]. This has led to inefficiency in the design of clusters as the memory capacity is critical in order to run large problem sizes. The system designer has to strike a balance between the number of nodes in a cluster, the amount of memory, and the size and speed of the interconnect when given a specific cluster cost, power budget and target performance. Due to inefficient scaling of DRAM capacity, a large number of nodes will be required in a cluster to support the target problem sizes at exascale. This will also add more communication infrastructure to the cluster, which acts as a performance bottleneck and consumes significant amounts of power. We hypothesize that the increased memory capacity that NVM can bring to a node will result in maintaining the required memory capacity of the cluster with fewer number of nodes. This will help in solving large problem sizes while staying within target cost and power budgets.

3.4 Architecture

To test our hypothesis, we set up an experimental architecture with actual servers. We have a single node with DRAM-NVM hybrid memory and 4 nodes with only DRAM memory that can support the same problem size. The 4 nodes are connected to each other using an Infiniband switch. We also used a setup of 2 DRAM-NVM hybrid memory nodes connected over Infiniband to measure and isolate the effect of communication on the performance of applications. The specifications of each node are described in Table 3.2.

We refer to the nodes with NVM as the Optane nodes. In the Optane nodes, we have two sockets, each with Intel's 24 core Cascade-Lake processor with hyper-threading turned on, which effectively provides 96 processing units. Each core has a 32 KB private L1 instruction cache, a 32 KB private L1 data cache, and a private 1 MB L2 cache. There is a 35.3 MB L3 cache shared between all cores. 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. This adds up to 192 GB of DRAM and 1.5 TB of non-volatile memory. The nodes have 4 memory controllers in total. Two of the memory controllers are connected to 6 DRAM DIMMs each, and the other two, known as iMC, are connected to 6 NVDIMMs each.

The nodes without NVM are referred to as DRAM nodes. Every DRAM node has two sockets of Intel's 22 core Skylake processor with hyper-threading turned on, which effectively gives 88 processing units. Each core has a 32 KB private L1 instruction cache, a 32 KB private data cache, and a private 1 MB L2 cache similar to the Optane nodes. There is a 30.25 MB L3 cache shared between all cores. Each socket has 6 DIMM slots each with a 32 GB DDR4 DRAM. That adds up to 384 GB of DRAM. Each node has 2 memory controllers connected to 6 DRAM DIMMs and all nodes are connected via single port to a Mellanox EDR 100 GB/s switch. We also utilize a 8 node DRAM-only setup with Skylake nodes with half the memory capacity.

Table 3.2 Experiment Platforms

Specifications	**Optane Node**	**DRAM Node**(x4)
Model name	Intel Xeon 8260L	Intel Xeon 6152
Architecture	x86_64	x86_64
CPUs	96	88
Sockets	2	2
Cores per socket	24	22
NUMA nodes	4	2
CPU MHz	3100	2900
CPU max MHz	3900	3700
CPU min MHz	1000	1000
L1d cache	32 KB	32KB
L1i cache	32 KB	32KB
L2 cache	1 MB	1 MB
L3 cache	35.3 MB	30.25 MB
Memory Controllers	4	2
Channels/controller	6	6
DIMM protocol	DDR4	DDR4
DRAM size	192 GB	384 GB
Max. DRAM BW	104 GB/s	104 GB/s
NVDIMM protocol	DDR-T	None
NVRAM size	1.5 TB	None
Max. NVRAM BW	40 GB/s	None
No. of nodes	2	4
Interconnect (BW)	Yes (100 GB/s)	Yes (100 GB/s)
Avg. Power Consumption	440 W	330 W
Operating System	CentOS 7	CentOS 7
Acquisition Cost factor	1	0.65

We use a different architecture for DRAM nodes because of the lack of availability of Cascade-Lake based machines as they have only recently been introduced on the market. Cascade-Lake architecture is essentially the same as Skylake with a larger package size, 4% higher CPU frequency and slightly higher data rate [Cas]. Although the Optane nodes has a higher L3 cache size, the cache size/core ratio is equivalent for both architecture. The DRAM nodes have slightly slower cores but they are in higher number for 4 nodes in total compared to the Optane node. The Optane node consumes 25% more power than a single DRAM node due to additional DIMMS but has more than 4x memory capacity. We used the LMbench 3.0 [MS96] benchmark to compare the memory bandwidth and access latency of both the Optane node and a DRAM node. We used a buffer size of 4 GB with a single instance of the benchmark and averaged all the readings over 3 runs. The results are presented in Table 3.3. The DRAM node has up to 2% better read bandwidth compared to Optane node but up to 5% lower write bandwidth. The same difference is reflected in the sequential and random access latencies as well. The acquisition cost of a single DRAM node is approximately 0.65

times the cost of a single Optane node as shown in Table 2.1. Hence, the total acquisition cost of 4 DRAM nodes is 2.5x the cost of a single Optane node, which provides an equivalent memory address space. These cost ratios are devised from the actual quotes of the nodes used for experimentation in our paper which cannot be revealed due to non-disclosure agreements. All the nodes are connected to rack mounted power distribution units (PDU) provided by Hewlett Packard Enterprise, which send Protocol Data Units to the administrator node. We capture the basic power data of the data units from the rack for DRAM nodes. These data units are timestamped data at the outlet granularity so we get per-node power data as well. We capture the same data for the Optane nodes.

Table 3.3 Performance comparison using lmbench3.0

Node	Rd BW(MB/s)	Wr BW(MB/s)	Sequential access(ns)	Random access(ns)
Optane	10926 ± 879	8792 ± 15	27.2 ± 0.1	85.4 ± 1.1
DRAM	11153 ± 712	8353 ± 15	26.8 ± 0.1	92.5 ± 1.1

3.5 Experiments

We evaluate the performance of a HPC application and 2 HPC proxy-apps on the Optane nodes and DRAM nodes. Optane DC can be configured to run in different modes: it can be used as a byte-addressable memory with DRAM as a direct-mapped cache (Memory mode), or as a persistent memory device separate from the memory (App-direct mode), or as a combination of the two (Mixed mode). In order to have the applications allocate their heap data on Optane DC and DRAM agnostically, we modified two files in the Linux OS kernel, arch/x86/platform/efi/efi.c and arch/x86/boot/compressed/eboot.c, to treat NVDIMM as DRAM [Izr19], which unifies the Optane DC and DRAM under one large address space in true DRAM-NVM hybrid memory fashion. We refer to this mode as the "Flat" mode hereafter.

We compare the performance of the single Optane node operating in Flat and Memory-mode separately with a cluster of 4 DRAM nodes with an equivalent amount of DDR4 DRAM main memory as described in Table 2.1. We also perform the evaluation of the same benchmarks running over 2 Optane nodes operating in Flat mode connected by a high speed interconnect. The aim of these experiments is to compare the performance of homogeneous address spaces connected over high-speed interconnects to massive address spaces with a hybrid but slower memory architecture with fewer nodes. We want to evaluate whether using NVM as a part of the primary memory architecture can deliver reasonable performance on fewer number of nodes compared to the conventional DRAM address spaces spread across a larger number of nodes. We scale the problem sizes of our

benchmarks to ensure they do not fit entirely in the DRAM of a single Optane node so that we do not end up measuring only the performance of DRAM. However, in the multi-node Optane setup, smaller problem sizes do execute only on DRAM. Our problem sizes can be described as large HPC problems ranging from 200 GB to 1.3 TB. The memory footprint of each application is indicated in Table 3.4. The upper limit of the problem sizes were fixed such that the applications did not run into an out of memory error on any of the setups. The memory allocation for all our benchmarks is memory technology agnostic. In Flat mode, it exhausts local DRAM memory before spilling onto NVM. The amount of compute we scale up to is bound by the amount of compute available on the Optane node to have a fair comparison. We use hyper-threading on the single Optane node in order to scale the compute on the DRAM nodes to resemble a similar core-to-working-set-size ratio compared to bare DRAM nodes and the 2 Optane nodes.

Table 3.4 Benchmark configuration

Benchmark	No. of iterations	Memory Footprint	
		Strong Scaling(GB)	Weak Scaling(GB/process)
AMG	Variable(19-21)	388	9.6
VPIC	600 timesteps	572	12
LULESH	10	590	22

We perform strong and weak scaling of applications by using MPI on both Optane and DRAM nodes. We use Open MPI 3.1.3. We use LIKWID [Tre10] to collect performance characteristics such as energy consumption and memory bandwidth on all nodes for all the experimental runs. All processes are pinned to a particular core individually for every run on both setups. Hence, the memory mapping for every process is consistent across runs even if the processes mapped to DRAM will have different performance compared to processes mapped to NVM. We also calculate the total energy consumed by the nodes based on rack managed PDU data. The racks were provided by HP and utilize model H8B50A full-length PDUs. These PDUs do not collect the power consumption for the Mellanox switch and the cooling system used for the nodes. The PDUs report the power consumption of the node at a 10 second interval. We utilize “ipmitool” to collect node power numbers.All applications are compiled using GCC 7.3.0 on both setups with -O2 optimization. We do not use any special libraries that are not provided with the source code of these applications and the page size used on both setups is 4 KB. We next describe our applications used in experiments.

VPIC:

Vector Particle-In-Cell (VPIC) [Bow08b] models kinetic plasmas in 1 to 3 dimensions and employs a variety of short-vector, single-instruction-multiple-data (SIMD) intrinsics for high performance

and has been designed so that the data structures align with cache boundaries making it compute bound. However, for our experiments we do not focus on vector operations. The code comprises of kernels that compute multiple data streams and operate on entire data structures. We use the 'lpi' input deck for our experiments.

AMG:

AMG is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids [Yan02]. The driver provided with AMG builds linear systems for various 3-dimensional problems. It it is an SPMD code that uses MPI and OpenMP threading within MPI tasks. AMG is memory bound with only about 1-2 computations per memory access, so memory-access speeds will also have a large impact on performance. We use the default problem, which is a Laplace type problem on a cube with a 27-point stencil.

LULESH

LULESH [Kar13] is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers. It features numerical algorithms, data motion, and programming style typical for scientific C or C++ based applications. It uses MPI and OpenMP for parallelization and is also memory bound.

These applications are representative of the workloads of common HPC applications that are currently used. Hence, evaluating the above applications will provide good approximation of the energy and cost efficiency of using a DRAM-NVM hybrid memory compared to a traditional memory architecture. We compare the inter-node and intra-node point-to-point MPI communication performance on DRAM and Optane, respectively, using the OSU MPI micro-benchmark suite 5.6.2 [al.] as depicted in Figure 3.1. The y axes are on a logarithmic scale, with the left y-axis indicating bandwidth as a boxplot and the right y-axis showing latency as a line graph. We observe that the shared memory buffer communication on the Optane node has up to half the latency and 3x the bandwidth than the Infiniband-connected DRAM-only nodes. By using a Flat mode memory, we consolidate all MPI communication on the intra-node network (e.g., Intel Quickpath [Zia10]), which can reduce the inter-node communication overhead for applications. This is also reflected in Table 3.5, where we present the MPI profiling information for all benchmarks evaluated in this work. We used EZTrace 1.1-9 [Tra11] to collect the communication traces and observe that all applications spend less than 19% of their total execution time on MPI communication. The majority of the communication time is spent waiting (MPI_WaitAll) and during collective operations like MPI_AllReduce. As processes are spread over multiple nodes, there will be some node local and remote communication and the overall progress (e.g., timestep between stencil updates) would be upper bounded by the slowest link, i.e., any remote node point-to-point communication (for an MPI_WaitAll handle or collective). This will be true for any interconnect irrespective of the memory architecture on the nodes. However, the ratio of average communication time to total execution time for a given application remains constant for any number of nodes irrespective of the interconnect.

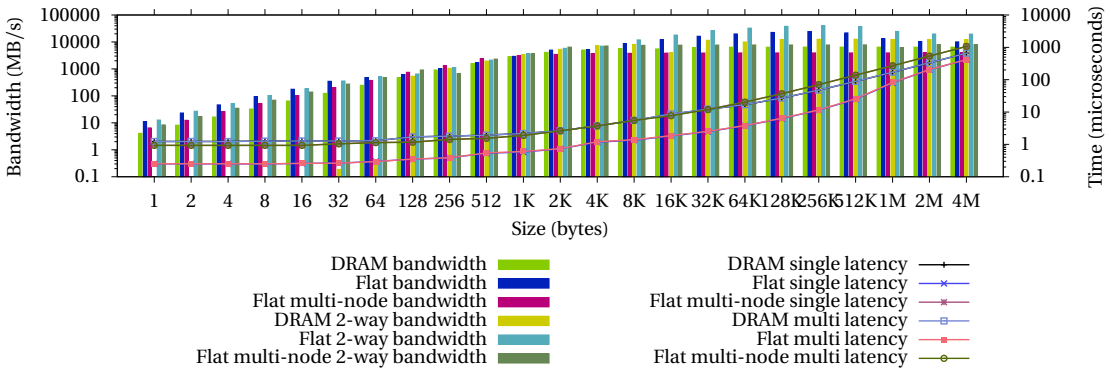


Figure 3.1 Performance of MPI communication for DRAM-only vs. Optane nodes

Table 3.5 MPI Communication profile (QP - Quickpath, IB - Infiniband)

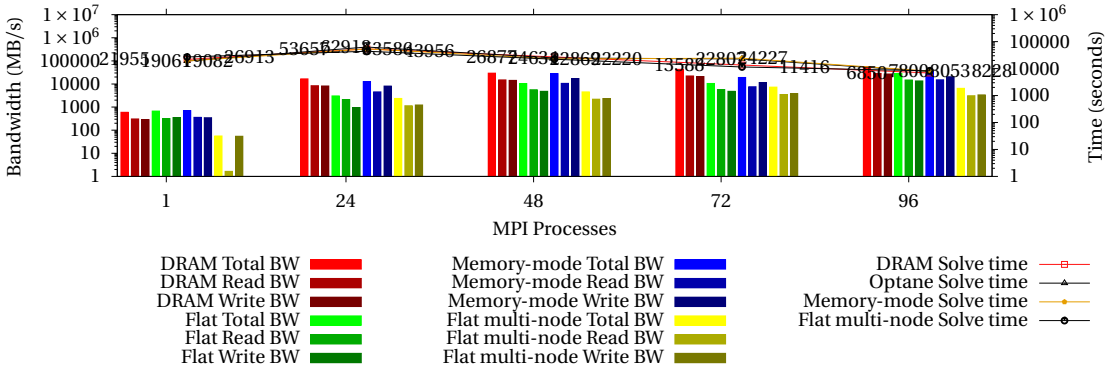
Benchmark	MPI Processes	Comm. time(%)		Avg. msg size(MB)		Functions consuming most time	
		QP	IB	QP	IB	QP	IB
VPIC	48	4.73	2.8	0.38	0.31	MPI_WAIT (93%)	MPI_WAIT (53%)
AMG	48	18.8	27.38	0.022	0.025	MPI_WAITALL (72%)	MPI_WAITALL (85%)
LULESH	27	10.73	7.93	1.37	1.5	MPI_ALLREDUCE (65%)	MPI_ALLREDUCE (85%)

Due to the scale of the problem sizes, the execution times of the applications tends to be very high. In order to finish the experiments for VPIC and LULESH in a reasonable amount of time but also have a fair representation of the compute and memory operations, we reduce the number of iterations of the main loop. This reduction has no effect on the performance of the application. The total number of iterations each experiment runs is given in Table 3.4. We plot the average of execution times and energy consumption over three runs for each execution of the benchmarks with a standard deviation of 1 to 3% for execution time and 2 to 5% for energy. These applications are a fair representation of the HPC workloads that run on the current HPC machines.

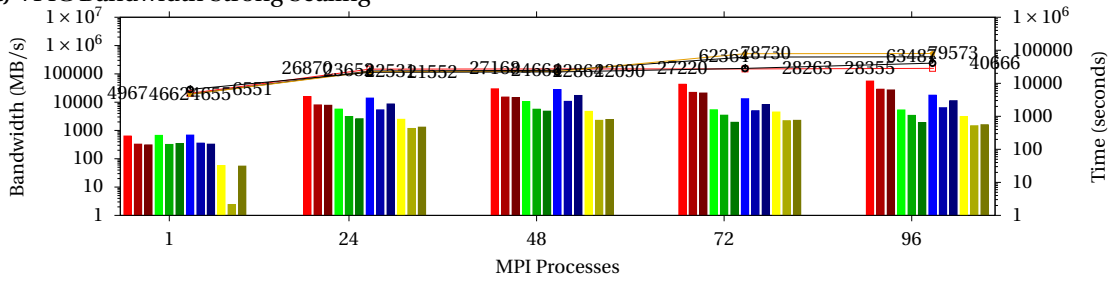
3.6 Results

Application bandwidth and execution time measurements are plotted together in a single graph to observe their correlation. Similarly, energy consumption and execution time measurements are plotted together. These graphs are provided for both strong and weak scaling experiments. In all graphs, the execution time is indicated on the right-hand side y-axis with units as seconds and depicted as lines, complemented by execution time values per data point in series order (DRAM/Flat mode/Memory mode/Multi-node). In the first set of graphs, bandwidth is indicated on the left-hand side y-axis with units as megabytes/seconds (MB/s) and depicted as a bar chart. In the second set, energy is indicated on the left-hand side y-axis with units as Joules (J) and depicted as a bar chart. All axes are on a logarithmic scale, except the time axes in Figures 3.8a, 3.8b, 3.8c and 3.8d. The x-axis depicts the number of MPI processes for a given execution. We refer to the results of 4 nodes without

NVM as “DRAM” and the two modes in which the single node with Optane DC PMMs operates by their respective names, “Flat-mode” and “Memory-mode”. The 8 nodes with NVM are referred to as “DRAM(8x)” and the 2 nodes with Flat-mode are referred to as “Flat multi-node”.



(a) VPIC Bandwidth Strong Scaling



(b) VPIC Bandwidth Weak Scaling

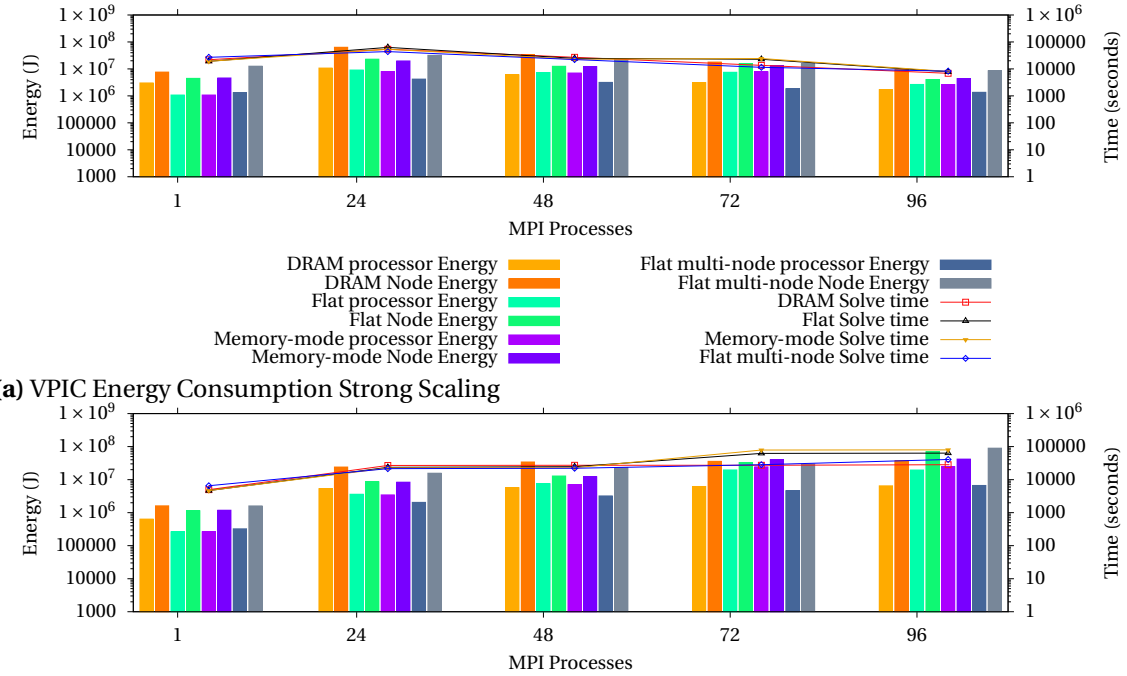
Figure 3.2 Bandwidth measurement for VPIC

3.6.1 VPIC results

Figure 2.14a depicts the execution times and application bandwidth for strong scaling of VPIC. We increase the number of processes from 1 to 96 and the overall problem size remains constant. To achieve this, we decrease the ’nppc’ parameter from 131072 to 8192 in the input file.

Observation 1: *The execution times for Flat-mode, Memory-mode, Multi-node and DRAM are similar for VPIC with only up to 15% difference for all process counts up to the number of native cores (48), and time is further reduced by oversubscribing (hyper-threading beyond 48 cores). However, the bandwidth achieved by Flat-mode is lower than Memory-mode and DRAM barring the single process execution due to serialization. Nonetheless, Flat-mode delivers comparable performance to DRAM and Memory-mode provides a slight benefit over Flat-mode. We find a similar effect for execution time in weak scaling, except for oversubscriptions, which prolongs execution.*

This is due to the nature of VPIC to optimize cache hits (as mentioned in Section 3.5) and have strided access pattern. It is a compute-bound application that aligns data accesses with the cache line size. Due to its access pattern, VPIC obtains significant spatial locality in the cache and does not fetch memory frequently. This helps in hiding the memory access latency of NVM. VPIC also benefits



(a) VPIC Energy Consumption Strong Scaling

(b) VPIC Energy Consumption Weak Scaling

Figure 3.3 Energy consumption for VPIC

from a larger L3 cache in the node with Optane DC and intra-node communication. Memory-mode executions try to hide latency for memory accesses by using the DRAM as a cache but page swapping between NVM and DRAM causes it to have only 15% decreased execution time compared to Flat-mode. There is an increase in execution time under strong scaling at 24 processes for all executions relative to the serial execution for a single process. When multiple processes are executing at the same time, an overhead for parallelization is introduced first seen at 24 processes. As we increase the number of processes further, the benefit of parallelization outweighs the overhead. Memory mode has lower execution time than Flat Mode due to significantly higher write bandwidth achieved over Flat mode. We also see that the Flat-mode bandwidth increases at a faster rate than Memory-mode beyond 72 processes. This effect can be attributed to that fact that Flat-mode utilizes all 4 memory controllers at the same time whereas Memory mode utilizes only the 2 DRAM memory controllers. This can increase the load on the DRAM controllers with higher number of processes and restrict the bandwidth.

Figure 3.2b depicts weak scaling, where the number of processes increases from 1 to 96 but the problem size per core remains constant. To achieve this, we keep nppc=32,768 in the input file for all runs. We observe that Flat-mode and Memory-mode have 10-17% lower execution time than DRAM up to 48 processes. The application bandwidth achieved by Flat-mode is lower than DRAM and Memory-mode executions. However, beyond 48 processes, the execution time doubles up due to oversubscribing of CPUs (hyper-threading). When oversubscribing, the memory controller queues overflow, which leads to serialization of loads and stores due to back pressure and nullifies the benefit of bank parallelism so that further scaling has no effect on the performance. This is also the

reason for the longer times observed for 72 processes in strong scaling.

Figure 3.3a depicts the energy consumption and execution time for strong scaling of VPIC.

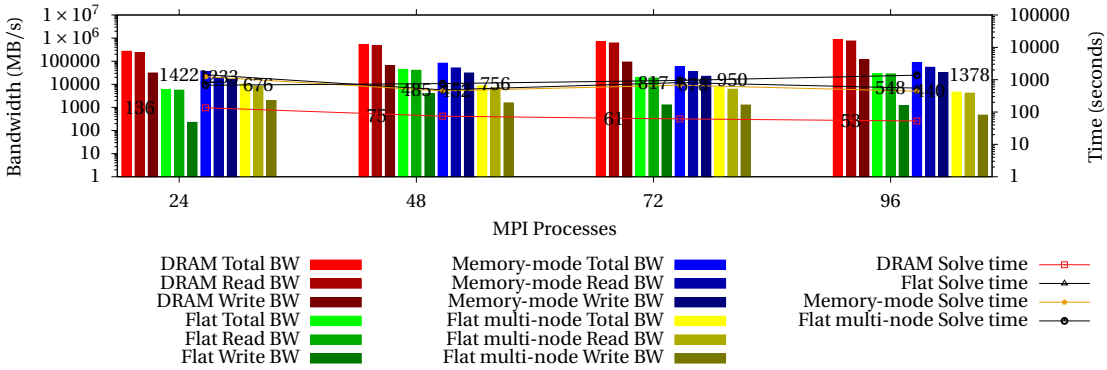
Observation 2: *Flat-mode, Multi-node and Memory-mode consume up to 3x less energy than DRAM for VPIC in terms of node energy due to similar execution times. These energy savings are observed for all executions in spite of similar or higher processor energy consumption for Flat and Memory-mode than DRAM.*

The node energy is also contributed to by other auxiliary components, e.g., cooling fans and power supplies in addition to the processor and DIMMs. We measure the energy consumed by the processors separately using the RAPL driver through LIKWID but we do not measure the energy consumed by the auxiliary components. We account for that energy in the node energy measurement obtained from the PDUs. The node energy for Flat and Memory-mode and Multi-node remains lower due to the fewer number of nodes involved in the execution of the application, i.e., fewer components that consume energy. DRAM has a larger difference between its node energy and its processor energy due to more auxiliary components. The same effect is observed for weak scaling depicted in Figure 2.15b. In multi-node execution, as the number of nodes is twice than of Flat-mode, the energy consumed is also higher. The energy consumption of Flat-mode is up to 3x lower than DRAM for all executions barring the 96 processes case, where the higher execution time of Flat-mode causes it to consume more energy than DRAM. The higher execution time for Flat-mode is again a result of oversubscription. The Memory-mode consumes a similar amount of node energy compared to Flat-mode despite a slight improvement in execution time due to increase in energy consumption of DIMMS where DRAM is used as cache. The Flat multi-node execution has higher energy consumption than DRAM due to its higher execution time but increasing problem size the energy benefits should scale with capacity: With a problem size at the capacity of the 2 nodes with Flat memory, 8 DRAM-only nodes are required to fit the problem size in memory, and the energy consumption of the DRAM-only nodes is higher than the 2 nodes with Flat memory when extrapolated.

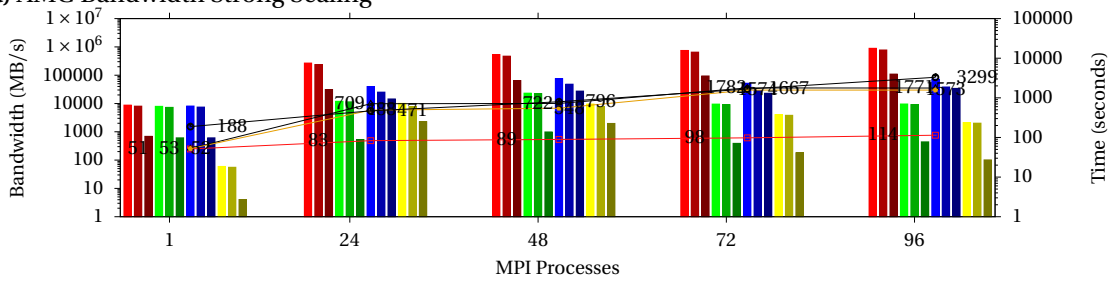
Inference 1: *Applications that optimize their cache hits (VPIC) or are compute-bound can benefit from a low power, high capacity memory device in terms of energy given a large enough last-level cache and enough compute resources on a single node. Their execution will result in low energy consumption under minimal to no performance degradation for a DRAM-NVM hybrid memory platform with lower acquisition and operation costs than multiple nodes using traditional DRAM memory. Exploiting DRAM as a cache for a NVM based memory can provide a slight advantage over the DRAM-NVM hybrid memory space in terms of execution time at the cost of a 20% reduction in problem size.*

3.6.2 AMG results

Figure 3.4a depicts the results for strong scaling of AMG. We scale the number of processes from 24 to 96 and keep the aggregate problem size constant. This is done by using the input value of 768 in the x,y and z problem dimensions for a single process and reducing it to 192, 192 and 128 in x,y and z problem dimensions. We do not provide results for a single process execution as the problem

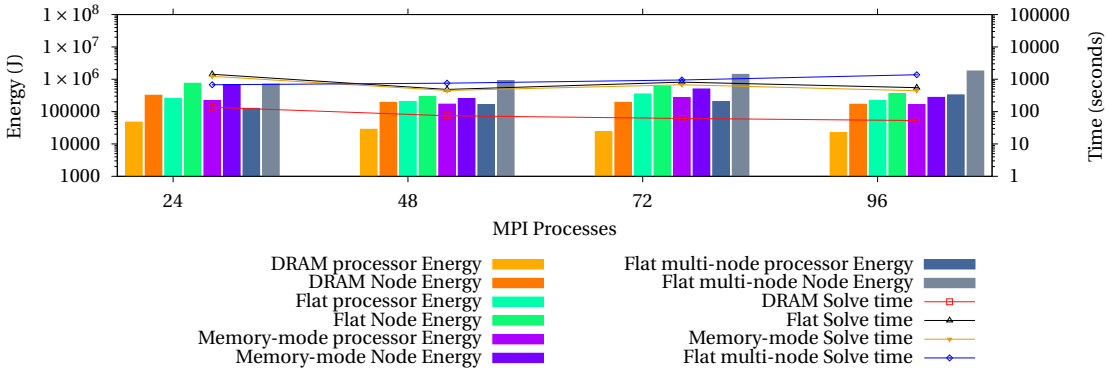


(a) AMG Bandwidth Strong Scaling

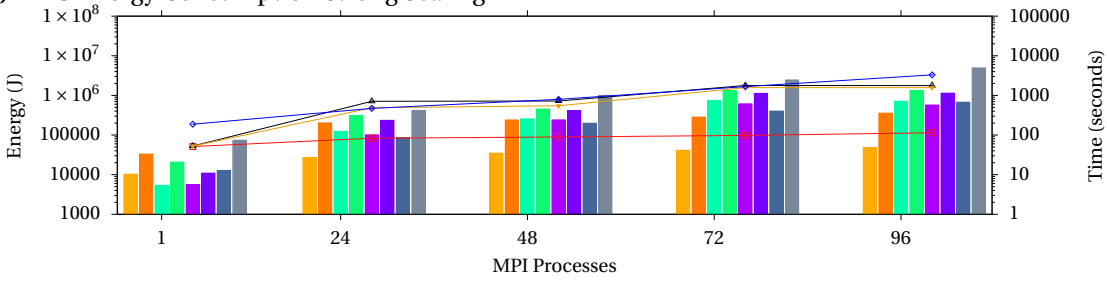


(b) AMG Bandwidth Weak Scaling

Figure 3.4 Bandwidth measurement for AMG



(a) AMG Energy Consumption Strong Scaling



(b) AMG Energy Consumption Weak Scaling

Figure 3.5 Energy consumption for AMG

size would exceed the memory address space of a single DRAM node. Weak scaling is depicted in Figure 3.4b, where we scale up the number of processes from 1 to 96 keeping the size per processor constant. This is achieved by keeping the input value of all dimensions at 224 for every run.

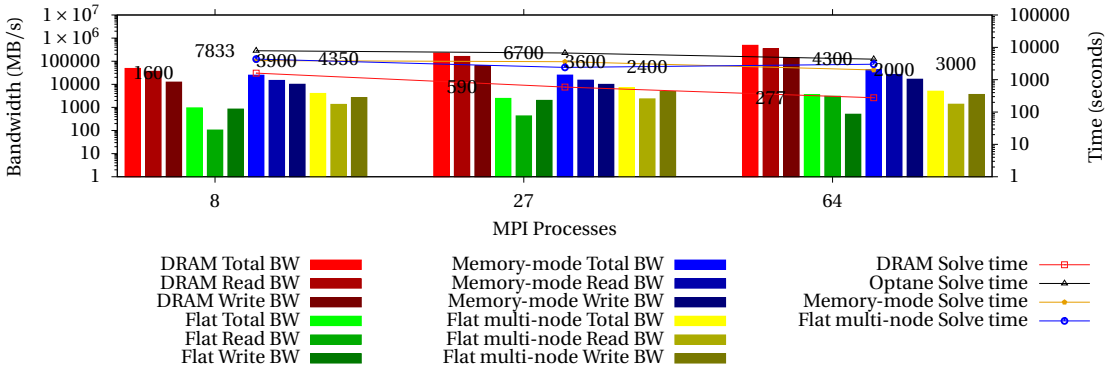
Observation 3: *The execution time of Flat-mode is an order of magnitude slower than DRAM for AMG. The total bandwidth achieved by Flat-mode is two orders of magnitude lower than DRAM, which causes the execution time to be an order of magnitude higher for both strong and weak scaling. Memory-mode provides up to a 20% reduction in execution time compared to Flat-mode for strong scaling with a higher memory bandwidth due to using DRAM as a cache. For weak scaling, Memory-mode has a 30% lower execution time compared to Flat-mode before core oversubscription and a 60% reduction in execution time after oversubscription.*

AMG is a memory-bound benchmark and its performance is heavily dependent on the write latency of the memory device. The access pattern is very irregular, where array indices are often indirectly referenced from other arrays. Due to higher write latency of Optane DC, Flat and Memory-mode bandwidth remains lower than DRAM and the execution time is higher. The gap between execution time widens as we scale up the number of processes for weak scaling. As the process count exceeds 48 cores for Flat-mode, the execution time increases dramatically due to oversubscription of resources. Memory-mode hides memory access latency by using DRAM as a cache but due to constant swapping of pages between DRAM and NVM the execution time is 6 to 9x higher than DRAM for strong scaling. In weak scaling, for lower number of processes the problem size is also lower, which results fewer page swaps between DRAM and NVM. Hence, the execution time is 30% lower compared to Flat-mode but 6 to 15x slower than DRAM. Figure 3.5a and Figure 3.5b depict the energy consumption of Flat-mode, Memory-mode and DRAM executions for strong and weak scaling of AMG, respectively.

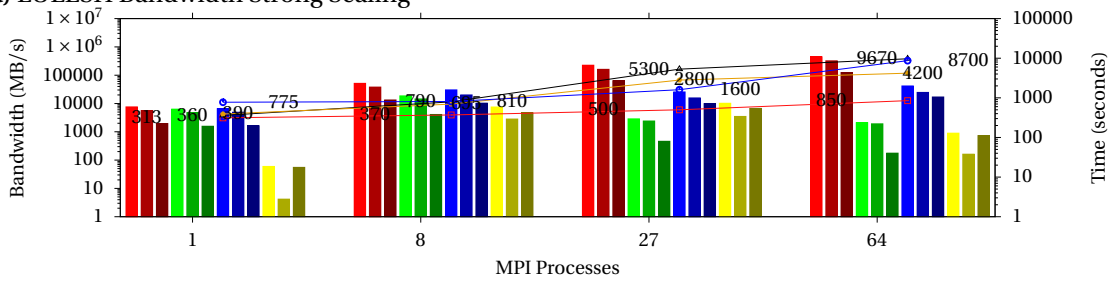
Observation 4: *For strong scaling of AMG, the total energy consumption of Flat-mode is up to 2x higher than the total energy consumption of DRAM. For weak scaling, the energy consumption for a single process execution in Flat and Memory-mode is lower than DRAM. In contrast, for all multi-process executions the energy consumption of Flat and Memory-mode is 3-4x higher than the energy consumption of DRAM.*

Energy consumption of an application is directly correlated to its execution time. Even though Optane DC DIMMs consume less power than DRAM DIMMs, they consume higher energy due to longer execution times. The DRAM DIMMs plus their additional processors consume a fraction of the aggregate energy of all 4 DRAM nodes. In contrast, the Optane DC DIMMs consume a large fraction of the total energy consumption of the Optane node. Memory-mode uses DRAM as cache causing the DIMM energy consumption to rise such that only a 12-15% reduction in node energy is observed compared to Flat-mode.

Inference 2: *Memory-bound applications like AMG, which are dependent on fast access latency of memory devices, will suffer in performance and energy when executed on a DRAM-NVM hybrid memory architecture. Using DRAM as a cache for NVM can help reduce execution time and energy consumption up to 15-20% at the cost of a 20% reduction in problem size. However, the acquisition costs of the multiple DRAM nodes required to run larger problems is more than 2x higher than a single Optane node.*

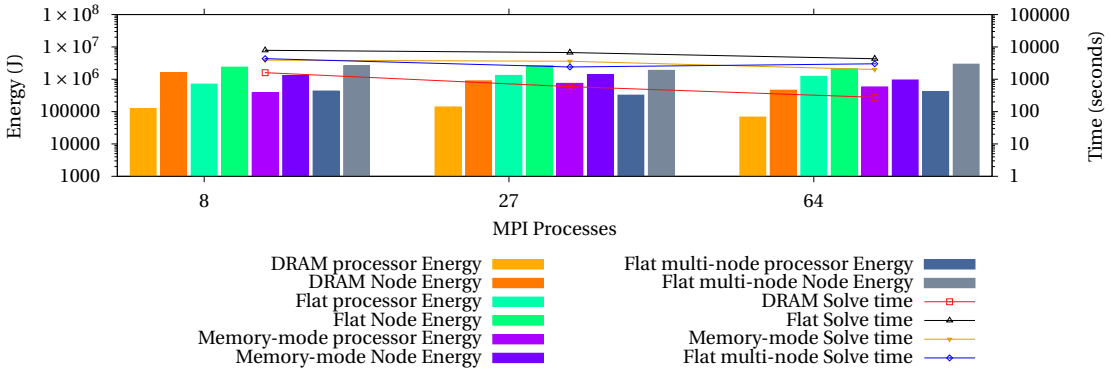


(a) LULESH Bandwidth Strong Scaling

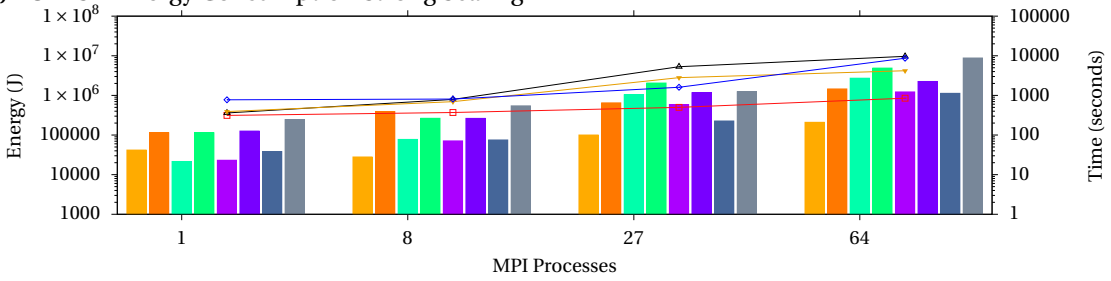


(b) LULESH Bandwidth Weak Scaling

Figure 3.6 Bandwidth measurement for LULESH



(a) LULESH Energy Consumption Strong Scaling



(b) LULESH Energy Consumption Weak Scaling

Figure 3.7 Energy consumption for LULESH

3.6.3 LULESH results

Figure 3.6a depicts the bandwidth and execution time for strong scaling of LULESH from 8 to 64 MPI processes. We do not provide results for the single process execution as the problem size

exceeds the memory capacity of a single DRAM node. The aggregate problem size remains the same. In order to achieve this, the input parameter changes from 1024 to 256 in each dimension keeping the number of elements constant. Figure 3.6b depicts the bandwidth and execution time for weak scaling, i.e., scaling the number of processes from 1 to 64 while keeping the problem size per processor constant. This is achieved by keeping the input parameter at 320 in every dimension, which results in increasing numbers of elements with the number of processes.

Observation 5: *For LULESH, the execution time for Flat-mode is up to an order of magnitude higher than DRAM; and Memory-mode is more than 50% faster than Flat-mode but 3-7x slower than DRAM. Due to the higher access latencies of Optane DC, Flat and Memory-mode do not achieve reasonable read or write bandwidth. DRAM, on the other hand, achieves high read and write bandwidth, which results in lower execution times. This effect is observed for both strong and weak scaling. For weak scaling, the execution time of Flat-mode is up to an order of magnitude higher than DRAM while Memory-mode is more than 60% faster than Flat-mode for larger numbers of processes.*

LULESH, like AMG, is also heavily dependent on access latency on the underlying memory device making it a memory-bound application. The memory access patterns of LULESH are non-unit stride due to its region based solvers [Kar13]. This increases the LLC misses and wastes precious cycles waiting for memory fetches as observed in [Pat19]. For weak scaling, fewer processes on the Optane node for Flat and Memory-mode achieve comparable bandwidth to DRAM. Hence, the difference in execution time is not significant. However, as we scale up the processes beyond 27, the execution time increases exponentially due to low bandwidth. The application bandwidth for LULESH is also affected by problem size. In strong scaling, as the problem size per process reduces, the memory bandwidth increases with higher number of processes. However, in weak scaling where the problem sizes per process remains the same, the application memory bandwidth reduces. This could be a result of the non-unit stride references of LULESH. Memory-mode executions perform significantly better than Flat-mode due to DRAM caching but are still 3-7x slower than DRAM. This is due to an order of magnitude higher bandwidth than Flat mode execution.

Figures 3.7a and 3.7b depict energy consumption and execution times for strong and weak scaling of LULESH respectively.

Observation 6: *LULESH consumes up to 3x more energy on Flat-mode than on DRAM due to longer execution time for strong scaling. For weak scaling, Flat and Memory-mode consumes 30% less energy than DRAM up to 8 processes.*

The Optane DIMMs contribute heavily to the total energy consumption of Flat and Memory-mode executions compared to DRAM energy consumption on DRAM nodes in strong scaling. Compared to Flat-mode, Memory-mode consumes up to 50% less node energy due to the significantly lower execution times. In weak scaling, due to lower power consumption of Optane DIMMs and fewer processors with similar execution times, Flat-mode consumes less energy. The energy consumption increases drastically for Flat-mode beyond 8 processes due to higher execution times. However, Memory-mode executions have similar energy consumption to DRAM for higher number of processes. These results show that applications dependent on both read and write bandwidth for

performance can expect reasonable performance with some energy savings for smaller problem sizes and fewer processes on a single node with Optane DC combined with lower acquisition costs. Using Memory-mode, we can obtain significant performance benefits compared to Flat-mode.

Inference 3: *From the above observations, we infer that compute-bound applications, which utilize cache locality well, are able to run larger problem sizes on fewer compute nodes with a DRAM-NVM hybrid memory system. They experience minimal performance degradation while providing significant cost and energy savings. Conversely, memory-bound applications suffer from performance degradation and higher energy cost. Using DRAM as a cache for NVM can reduce the performance degradation for memory-bound applications. There may also be an opportunity for memory-bound applications on Flat-mode if memory latency can be hidden using aggressive prefetching. Hence, a single node with a DRAM-NVM hybrid memory system can support large problem sizes with reasonable trade-offs that traditionally would require approximately 4 DRAM-only nodes to run.*

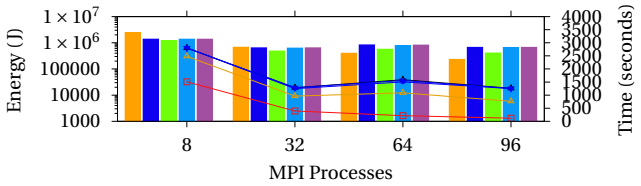
Observation 7: *The executions with split allocations utilizing more DRAM have lower execution times and consequently lower energy consumption.*

Figure 3.8 depicts execution time and energy consumption for all three benchmarks while splitting the dynamic memory across DRAM and NVM in fixed ratios of 1:1, 1:3 and 1:7 (DRAM:NVM). We achieve this by overloading the allocation wrappers in the benchmarks and utilizing 'numa_-alloc_onnode()' instead of 'malloc()'. The problem sizes used for these experiments are smaller than the previous experiments as we are limited by DRAM capacity. For VPIC, we observe that DRAM(8x) consumes more energy than all other executions despite having lower execution time until the other executions start using hyper-threading. This can again be attributed to the cache locality of VPIC and fewer nodes to execute on due to the consolidated memory capacity of Flat-mode. For AMG, any execution that utilizes NVM has higher execution times and the ratio of DRAM memory used has no effect. Hence, the energy consumption for all Flat-mode executions is higher than DRAM(8x). For LULESH, we see that DRAM(8x) has lower execution times and energy consumption compared to other executions but splitting memory across DRAM and NVM mitigates the performance degradation.

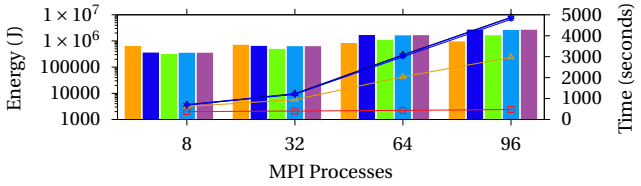
Inference 4: *From the above observations, we infer that applications with cache locality benefit from a memory allocations split across DRAM and NVM. Higher utilization of DRAM can mitigate the performance degradation due to NVM.*

3.6.4 Projection

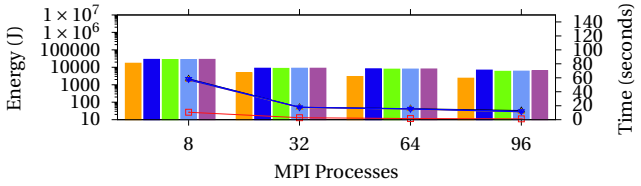
To assess the cost benefit of a DRAM-NVM memory system based supercomputer, we refer to Frontera, a cluster with Cascade Lake processors, predecessors to the Sapphire Rapids processors to be used in Aurora [Aur]. It has an acquisition cost of $60 million [Topa] and achieves 24 petaPFLOPS of LINPACK performance on 8,008 nodes connected by Mellanox HDR-100 switches with dual sockets and 192 GB of DRAM on each node that amounts to 1.46 PB in total. Each node costs approximately $7,500 and we spread the cost across Compute, Memory and Interconnect with ratios of 0.66 ($5,000), 0.2 ($1,500) and 0.14 ($1,000), respectively. We assume these ratios based on an informal survey of costs for the respective components that is available publicly as the exact quotes



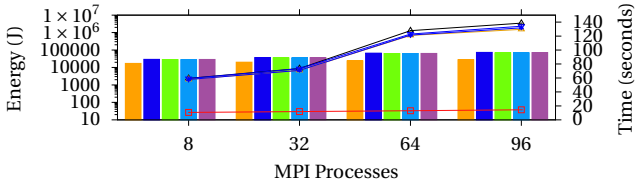
(a) VPIC Strong Scaling



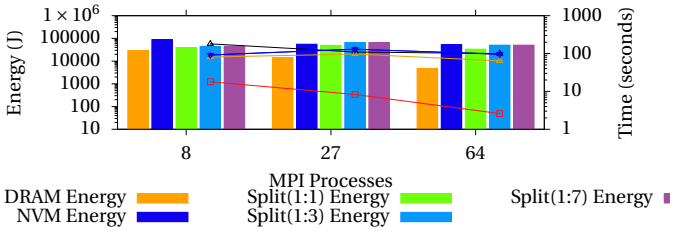
(b) VPIC Weak Scaling



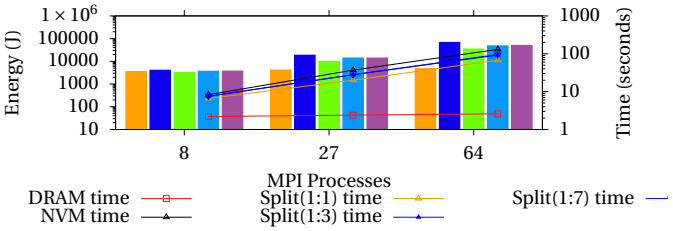
(c) AMG Strong Scaling



(d) AMG Weak Scaling



(e) LULESH Strong Scaling



(f) LULESH Weak Scaling

Figure 3.8 Energy consumption of VPIC, AMG and LULESH under different memory allocation policies

for the system are unavailable.

The Cascade Lake nodes can support up to 768 GB of DRAM using the 32 GB DDR4 DIMMs and 6 TB of Optane DC using the 512 GB NVDIMMs based on memory channels shown in Table 2.1. This brings the support to a total of approximately 55 PB of byte-addressable memory in Flat mode.

Based on the cost ratio for Optane DC compared to DRAM in Section 3.1, Optane DC on a single node costs around $25,000 and the extra DRAM adds another $4,500, which increases the per-node cost to $37,000. Just the addition of Optane DC PMMs and the required DRAM to support it will bring the cost of Frontera up to $296 million. If compared to the projected acquisition cost ($500 million) of Aurora [Aur], we still would have around $200 million to purchase additional compute to increase additional FLOPS for this machine, which is equivalent to the entire budget of Summit [Sum]. However, if the current DRAM based architecture of Frontera was to be scaled up to support a problem size of 55 PB, it would require 290,290 nodes and would cost $2.2 billion. Even though the compute performance of such a machine will be approximately 708 petaFLOPS assuming a linear speed up, the acquisition cost and power required to operate such a large cluster makes it infeasible. We can bring down the number of nodes by upgrading the amount of DRAM in every node to 768 GB. Such a machine requires approximately 72,573 nodes and the cost of the cluster will be approximately $871 million, where each node costs $13,000. This configuration will achieve a compute performance of approximately 177 petaFLOPS assuming a linear speed up, but it will come at approximately 1.25x the cost of Aurora, and its power and space requirements will still be infeasible. By just increasing the capacity of the current cluster by adding NVM, one can support 20x larger problem sizes than Summit at just 1.5x the cost. As seen in Fig. 2.15b, compute-bound application will suffer minimal performance degradation while increasing the problem size support by approximately 37x and being significantly more energy efficient than any DRAM based configurations. Memory-bound applications will suffer while using a memory agnostic allocation scheme as seen in Fig. 2.8b. One can sacrifice 11% of the total memory capacity by using DRAM as cache for NVM to limit the slowdown up to 50% for some applications while still supporting 19x larger problem sizes than Summit [Pat19]. Also, a memory aware allocation scheme can limit the slowdown by keeping memory bound data structures in DRAM [Pen19] for a hybrid memory system.

3.7 Future Work

In the future, we want to run similar experiments in an environment where we can measure and observe the energy characteristics for multiple nodes of Optane at a finer granularity, which can help design energy budgets for future HPC systems. We wish to develop novel data placement and data movement policies for the same that can optimize the performance of NVM. We will investigate static and dynamic approaches to optimize memory allocations for HPC applications executing on DRAM-NVM hybrid memory systems. We wish to assess larger cache sizes combined with different Optane memory sizes. This may make Optane benefits accessible to a wider range of applications.

3.8 Conclusion

Our work assessed the performance and energy characteristics of DRAM-NVM memory systems for large problem sizes under different modes and compared it to DRAM-only memory systems. For memory-bound applications, using a DRAM-NVM memory can hamper performance due to higher

access latencies but using DRAM as a cache for NVM can restrict the performance degradation to a certain extent with the tradeoff in smaller problem sizes. However, with enough compute resources, compute-bound applications achieve similar performance and lower energy on Flat-mode compared to traditional HPC systems. Flat-mode has marginally lower performance than Memory-mode for most cases. But Flat-mode allows significantly larger problems to be brought into memory, a tradeoff future HPC system designers will have to consider when choosing between fast DRAM or slower but larger NVM at equal acquisition cost. Memory allocation policies that can utilize DRAM more can also help take advantage of NVM. It provides a promising path to extend memory address spaces of HPC systems without compromising performance and energy for certain codes, which translates into lower operational cost combined with lower acquisition costs. We show that our hypothesis holds in terms of memory capacity, i.e., a heterogeneous memory system does support larger problem sizes compared to traditional DRAM-based system while offering energy and cost benefits with different design trade-offs for future HPC clusters.

CHAPTER

4

SYMBIOTIC HARDWARE CACHE AND SOFTWARE DTLB PREFETCHING FOR HPC WORKLOADS ON DRAM/NVM HYBRID MEMORY

4.1 Introduction

Hybrid memory architectures are being increasingly adopted in modern computing systems. Intel's Knights Landing (KNL) introduced High Bandwidth Memory (HBM) along with traditional DRAM-based main memory [Sod16]. General Purpose Graphics Processing Units (GPGPU) are also equipped with HBM [OâĂ14]. Fujitsu is using a hybrid memory cube (HMC) for its A64FX ARM based chips to deliver high bandwidth memory access to all the compute cores [Fuj]. Recently, Intel launched their Phase Change Memory based Optane DC Persistent Memory Modules (PMM), which are byte-addressable NVDIMMs used as non-volatile main memory (NVM) [Hru18]. The Aurora supercomputer [Aur], which will be launched in 2021, will have support for Intel Optane DC PMMs. Hence, such a DRAM/NVM based hybrid memory architecture will become more prevalent in future HPC systems. One of the reasons why memory architectures are becoming heterogeneous is because each memory technology brings different characteristics to the table with pros and cons [Mut13]. HBM allows more data to be moved to the processor with the same access latency as DRAM, while NVM provides higher memory capacity supporting application runs with larger problem sizes on fewer systems (nodes), which can result in lower energy consumption and cheaper acquisition

costs.

Modern architectures use hardware (HW) prefetchers to increase the amount of cache hits of applications and reduce the effective latency of L1 and L2 cache misses. This can improve overall application performance. HW prefetchers rely on hardware to fetch data by tracking cache line addresses accessed in the N most recently accessed 4KB pages for each core and then predicting future access locations during execution [Mes15; Isl17]. The HW prefetchers are limited by the 4KB page boundary and so-called huge pages do not alleviate this problem as the HW prefetcher is unaware of the larger page boundary [Mar19]. Also, the number of huge page Data Translation Look-aside Buffer (DTLB) entries are very limited, and their allocation requires specific API calls. Software (SW) prefetching is performed by using the prefetch instructions that are part of the instruction set architecture (ISA) on many processors. Compilers perform SW prefetching by adding prefetch instructions to the code after analyzing memory access patterns statically, which increases the amount of cache hits on the critical path. Each prefetch mechanism has its own advantages and disadvantages. However, currently both mechanisms are designed and fine-tuned for only DRAM-based main memory, and SW prefetching is very conservatively utilized. For instance, the "prefetch-loop-arrays" pass in GCC [Sta02] determines the read and write prefetch distances based on mostly the L1 cache size, DRAM access latency and the 64-byte cache line granularity used for DRAM on Intel architectures. It is very rigid in terms of the heuristics it utilizes to decide if prefetching is profitable or not and explicitly tries to avoid clashing with the HW prefetcher.

SW DTLB prefetching was proposed almost three decades ago [Bal94] but is not implemented in any modern systems or compiler frameworks. The DTLB is an address translation cache that contains entries mapping page numbers to page frames. DRAM/NVM based hybrid memory systems aim to support larger datasets for applications, which means more memory pages will be fetched by the CPU [Mar19]. This will create additional pressure on the DTLB because every DTLB-miss is served by a page walk across a 4 or 5 level radix tree, which can incur a high cost on performance at times. Further performance degradation can occur if the page walk results in a page fault and main memory has to be accessed to retrieve page table indirections. NVM access latency can further add to this problem. As we move towards hybrid memory architectures, prefetch mechanisms need to adapt in order to mitigate the performance degradation of applications.

High performance computing (HPC) applications frequently use different solvers, e.g., Partial Differential Equations (PDE), Fast Fourier Transform (FFT) and Conjugate Gradient (CG) [Lus06]. The solvers consist of kernels that perform heavy compute and memory operations on large structured arrays. They perform computations, e.g., using various stencil shapes, which display strong spatial and temporal locality. It is critical to take advantage of the locality of references for good performance of the HPC applications on hybrid memory systems. Upcoming HPC systems will have hybrid memory devices that require a more effective prefetch methodology to achieve good performance.

This chapter analyzes the performance of a DRAM/NVM based hybrid memory system for different HPC kernels. Computational kernels are enhanced by SW prefetch instructions and assessed in their relative effect on performance. The objective here is to characterize the prefetch performance

for hybrid memory systems while executing HPC workloads so that the current prefetch mechanisms can adapt to new and potentially hybrid memory architectures.

This work does not evaluate Memory-mode performance as we have observed in Chapters **??** and **??** that the page fault overhead of DRAM cache is much higher than the L3 cache miss overhead of using NVM in Flat mode. This work provides more dynamism to a hybrid memory architecture than Memory mode by interacting directly with the SRAM caches. Section 4.2 summarizes previous work on hybrid memory systems and SW prefetching. Section 4.3 provides an overview of the "prefetch-loop-arrays" pass in GCC and explores the architecture of a hybrid memory system. Section 4.4 presents our experimental setup and the custom benchmark developed in this work for evaluation. Section 4.5 presents and discusses results and observations. Section 4.6 proposes modifications to the "prefetch-loop-arrays" pass to achieve SW adaptive prefetching. Section 4.8 summarizes our contributions.

4.2 Related Work

A number of recent studies have been conducted recently after the launch of Intel's Optane DC PMMs. Yang et al. and Izraelevitz et al. [Yan15; Izr19] 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. [Pat19] 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. Peng et al. [Pen19] 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. All the above works focus on evaluating the basic performance characteristics of Optane DC under various execution contexts and workloads. Our work primarily focuses of characterizing the effects of prefetching and utilization of the cache in in a byte-addressable DRAM/NVM hybrid memory address space for HPC workloads.

Several works have been conducted on utilizing SW prefetching in order to improve performance on traditional DRAM based memory systems. Callahan et al. [Cal91] were the first to introduce SW prefetching as non-blocking prefetch instruction to eliminate cache miss latency. Mowry et. al [Mow92] introduced and evaluated compiler based SW prefetching that worked in coordination with the HW prefetcher. Bala et al. [Bal94] introduced SW prefetching for DTLB to decrease both the number and cost of kernel DTLB misses. Margaritov et al. [Mar19] proposed a HW-based DTLB prefetch mechanism to reduce the address translation times in modern processor systems. Badawy et al. [Bad04] evaluated the use of SW prefetching and locality optimizations for various HPC workloads for DRAM-based memory systems and found that for some cases SW prefetching has more benefits.

Fuchs et al. [Fuc14] designed a HW prefetcher for code block working sets that predicted the future memory accesses of stencil based codes. Swamy et al. [Swa14] introduced a hardware/software framework to support efficient helper threading on heterogeneous manycores, where the helper

thread would perform SW prefetching to achieve higher sequential performance for memory-intensive workloads. Zhao et al. [Zha05] used a dynamic approach to pool memory allocations together and fine tuned the data prefetching based on the access patterns observed in the profiler. Islam et al. [Isl17] evaluated hardware prefetching in a flat-addressable heterogeneous memory comprising HBM and phase change memory (PCM), where a large buffer was placed in HBM to hide the access latency of PCM. Meswani et al. [Mes15] explored various schemes to manage the heterogeneous memory architectures and then refined the mechanisms to address a variety of HW and SW prefetch implementation challenges. Lee et al. [Lee12] evaluated both SW and HW prefetching for various workloads and suggested techniques for cooperative HW/SW prefetching. We aim to utilize DTLB- and cache line-based software prefetching in order to adapt to the upcoming hybrid memory systems with fast and slow access memory devices while improving the performance of HPC workloads.

4.3 Architecture

4.3.1 GCC prefetch-loop-arrays compiler pass

Mowry et al. [Mow92] designed the GCC compiler pass to optimize HPC workloads with SW prefetch hints that work in coordination with the HW prefetcher. This section analyzes the operational characteristics of their prefetch algorithm. The algorithm aims to be fine tuned for DRAM-based memory systems. All constants and heuristics are fixed to values that conform with DRAM specifications, which differ from system to system. The algorithm works on a per loop basis:

1. Gather all memory references in the loop and convert them into a $base+step*iter+delta$ form. Classify them as read or write references and form groups of references based on $base+step$ to identify different access patterns.

2. Calculate the profitability for each memory reference using a heuristic-based cost model that takes into account the total number of instructions, memory references, the miss rate, trip count, prefetch-ahead distance, unroll factor, the maximum prefetch-slots and temporal locality.

3. Determine the references that can be prefetched based on maximum prefetch-slots and prefetch-modulo (relative to the loop iterator).

4. Unroll the loops to satisfy prefetch-modulo and prefetch-before constraints in prologue size, but without loop peeling.

5. Emit the prefetch instructions.

The cost model used to determine the profitability of prefetching for different memory reference groups is as follows:

- First determine the prefetch-modulo and prefetch-before for every memory reference using the $step$ and $delta$ values. Their temporal locality is determined data dependence analysis on the memory references.
- The prefetch-ahead is determined by using the ratio of $\frac{fetch\_time}{iteration\_time}$ indicating how far ahead to prefetch. Both these values are unavailable at compile time. Instead, target-specific constants are used to make an "educated guess".
- The acceptable miss rate is calculated based on the references that have the same $base + step * iter$ but different $delta$. If $delta$ exceeds cache line size then it is determined to be a miss. If the probability of this miss is less than 0.95, then prefetching is considered to be unprofitable.
- It determines if the loop has enough iterations to justify prefetching ahead using the trip-count-to-ahead-ratio with a cut-off threshold of 4 (i.e., no prefetching below this threshold).
- It also calculates the ratio between $\frac{total\_instruction\_count}{memory\_references}$ to determine if the loop has enough CPU instructions to overlap the cache misses. If the ratio is smaller than the machine specific threshold, no prefetching is done.
- It also calculates the prefetch-cost via the ratio of $\frac{total\_prefetch\_count}{total\_instructions\_count}$. If this cost is too high, then no prefetching is performed.
- The ratio of $\frac{prefetch\_mod}{unroll\_factor}$ is calculated with a threshold value of 4, below which no prefetching occurs.

Some of these thresholds are overly strict even by DRAM standards. If the same prefetching parameters were used for NVM memory accesses, the algorithm would fail to gauge the most efficient prefetch configuration or not perform prefetching at all. Although the algorithm acknowledges different types of streams/access patterns in a kernel, it does not consider varying thresholds. Hence, it cannot adapt to different memory access patterns or memory technologies. Also, all parameters are defined based on cache and cache line sizes. No DTLB prefetching is considered.

4.3.2 DRAM-NVM hybrid memory architecture platform

The system used in experiments is a single HPE Proliant DL360 node 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 achieves asynchronous command/data timing and variable-latency memory transactions. 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 [Izr19]. 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 experiments are performed on the Flat-mode.

4.4 Experimental Setup

The aim of this experiment is to characterize the performance of SW prefetching for different prefetch distances under temporal and non-temporal prefetching with allocations on DRAM and NVM separately. We switch HW prefetching on and off for all experiments to evaluate its effect, which requires a reboot after toggling on/off three BIOS setting: HW Prefetch, Adjacent Sector Prefetch, and DCU Stream Prefetch. We compare the SW prefetch performance with and without "prefetch-loop-arrays" compiler optimization of GCC 9.3.0 while using the O3 flag for all compilations. The symbiotic SW prefetching runs are compiled with "no-unroll-loops" in order to measure the effect of varying unroll distances.

We developed a custom benchmark that allows us to measure the prefetch performance for different kernels frequently occurring in HPC applications. These kernels include a write-only (Wr-only) stream, single-write-multiple-read stream (1W4R), and 3-, 5-, 7-, 9- and 27-point stencil streams. The Wr-only stream kernel consists of 5 sequential write streams of linear arrays. The 1W4R kernel has one write stream and four read streams, which are also accessed sequentially. All stencil kernels consist of a write stream and a read stream of a 3-dimensional (3D) dataset of linearly laid out arrays accessed in row-major order using three nested for loops.

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 [CAP14]. Some examples of stencil code-based HPC applications are Vector Particle In Cell (VPIC) [Bow08b; Bow09; Bow08a] and Algebraic Multi-grid (AMG) [Yan02], which are compute- and memory-bound applications, respectively. The 3-, 5- and 7-point stencils use the Von Neumann

neighborhood whereas the 9- and 27-point stencils use the Moore neighborhood [KS07]. The 3-point stencil is a one-dimensional (1D) stencil, where for every iteration the previous element and the next element are read along with the current one. The 5-point stencil is a two-dimensional (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 7-point stencil is a 3D stencil, where along with the adjacent elements in the same row and column of the current element, adjacent elements in the next and previous plane are read. The 9-point stencil is a 2D stencil including diagonal elements beyond the 5-point stencil. Similarly, 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. Each stream is 4 GB in size and is allocated separately on each NUMA node using numa_alloc_onnode() for every run.

We manually unroll and peel the kernels. Each kernel has a prologue and an epilogue loop. The prologue loop prefetches each element of the stream sequentially until up to given read or write prefetch distance. The main compute kernel is unrolled up to the unroll distance and the next elements are prefetched after unrolling is complete. The main loop stops when there are no more elements to prefetch and the remaining iterations are completed in the epilogue loop. Due to the variability in the read and write distances, the prologue and epilogue loops are split and separated by conditional statements to avoid over-prefetching and segmentation faults.

We use the GCC __builtin_prefetch() function to prefetch the desired cache line at every iteration [Gcc], which automatically calls the corresponding intrinsic for a given instruction set architecture. We change the read and write prefetch distance explicitly from 32 bytes to 16,384 bytes using nested loops that encompass all the kernels. The upper bound of the prefetch distance is kept at 16,384 bytes to avoid L1 cache contention at higher distances. We change the parameters of the prefetch call to perform non-temporal and temporal prefetching for linear and stencil read streams, respectively. We use non-temporal prefetching for all write streams. For non-temporal prefetching, the data is fetched into a non-temporal cache structure to avoid cache pollution; whereas for temporal prefetching the data is prefetch into L2 cache or higher [Inta; Intb]. We also vary the unroll distance in another nested loop from 4 to 64. We limit the unroll distance to 64 to restrict additional pressure on available CPU registers. We perform cache line prefetching with distances from 32 to 2,048 and perform DTLB page prefetching from distances from 4,096 to 16,384. This is enabled by adding a conditional statement to the prefetch statement block, which prefetches only after certain number of iterations have elapsed nearing the page boundary. DTLB caching causes the address translation to be stored in the 4-tier DTLB cache, which reduces future page faults that are expensive in terms of CPU cycles [Sch19].

We refer to our technique as “symbiotic prefetching” from here on. We execute the benchmark on 48 processes running individually on each core launched by MPI, but without imposing communication via message passing, i.e., just to provide realistic workloads in separate address spaces with contention on shared resources (last-level cache, DRAM/NVM). We divide the stream size equally between all processes and allocate them separately. We pin all processes to the cores and

then calculate the data bandwidth for DRAM and NVM allocations using the ratio of total memory of all data structures accessed and wall clock time measured for each kernel. Each measurement is averaged over 10 runs, where a standard deviation of 4%-6% is observed for all kernels. To obtain cache performance metrics, we use the LIKWID Marker API [Tre10] to measure metrics for every individual kernel obtained from HW performance counters. The performance counters are obtained for the configuration that delivers highest performance benefit and then reported relative to measurements of the same kernel without any prefetching as a baseline. We also perform the same comparison for the kernels when they are compiled using "prefetch-loop-arrays". We term this as compiler prefetching from here on.

4.5 Results

This section discusses the results of experiments and present observations. Prefetch performance is depicted as percentage changes over different compilation options. We depict percentage changes in data bandwidth observed with symbiotic prefetching (sp) as a heat map over a 3D graph relative to the baseline bandwidth observed when compiled with no prefetching (np). The x-axis depicts the write prefetch distance in bytes, the y-axis the read prefetch distance in bytes, and the z-axis the unroll distance in number of iterations. The heatmap colors represent percentage changes in bandwidth relative to the baseline. We plot the graphs for DRAM (left) and NVM (right) separately.

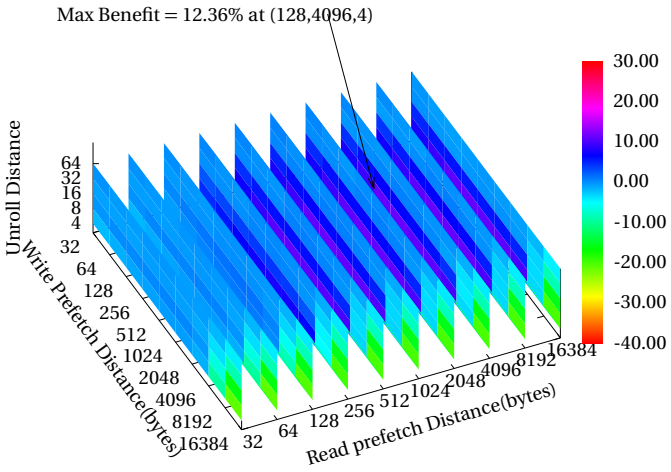
We only present heatmaps for the 7-point stencil in Figures 4.1 and 4.2 due to space limits, but we report and discuss the results (data bandwidth and performance metrics) for all kernels as all have similar graphs, albeit with different best values (subject of a forthcoming technical report). Figure 4.1 depicts results of symbiotic and compiler prefetching for the 7-point stencil kernel without HW prefetching. Figure 4.1a and 4.1b depict performance changes under symbiotic prefetching for DRAM and NVM, respectively, where an arrow indicates the highest benefit configuration (write, read, unroll distances).

Observation 1: For DRAM, bandwidth increases for larger read prefetch distances (x-axis) but abruptly drops close to the largest write prefetch distances (y-axis) and is best for a small unroll distance (z-axis).

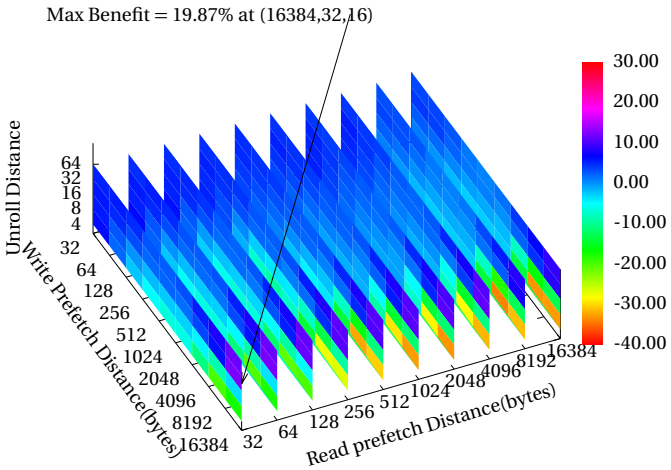
Observation 2: For NVM, the bandwidth decreases toward higher write and read prefetch distances (x+y axes) and, in contrast to DRAM, is best for largest unroll distance (z-axis).

Observation 3: For DRAM, symbiotic prefetching results in slightly higher L1-DTLB load misses, and even slightly more for compiler prefetching. For NVM, L1-DTLB load misses are significantly higher by a factor of 11x (1100%) under symbiotic prefetching compared to no SW prefetching — where compiler prefetching actually *reduces* the L1-DTLB load misses for NVM. Symbiotic prefetching also reduces the L1-DTLB store misses for NVM whereas compiler prefetching increases them.

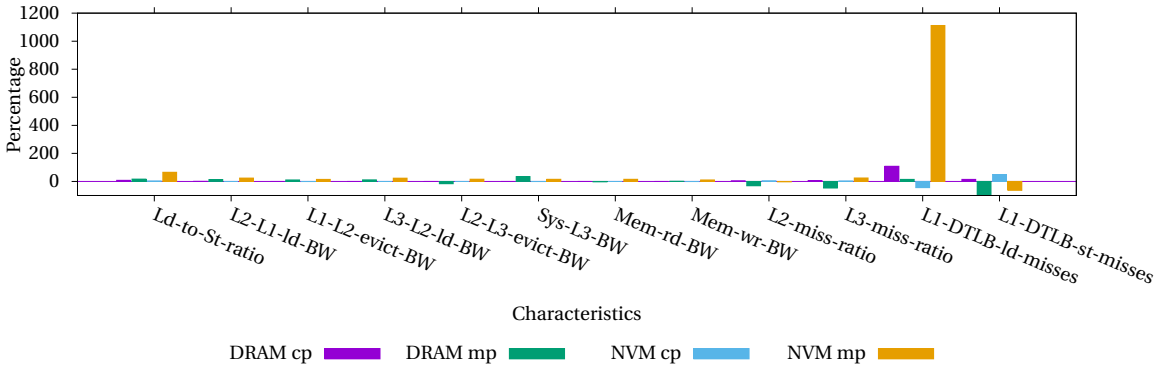
Given the structure of a 3D 7-point stencil, a total of 5 read streams and 1 write stream exist. Hence, performance of the kernel critically depends on the availability of read data in the cache. As the stencil moves across the data set, there is reuse of all neighboring read data points. Hence, it is prefetched as temporal data into the L2 and L1 caches. The write data points do not have reuse, and are hence prefetched as non-temporal data into a separate cache structure. However, if the latency of



(a) DRAM sp % change



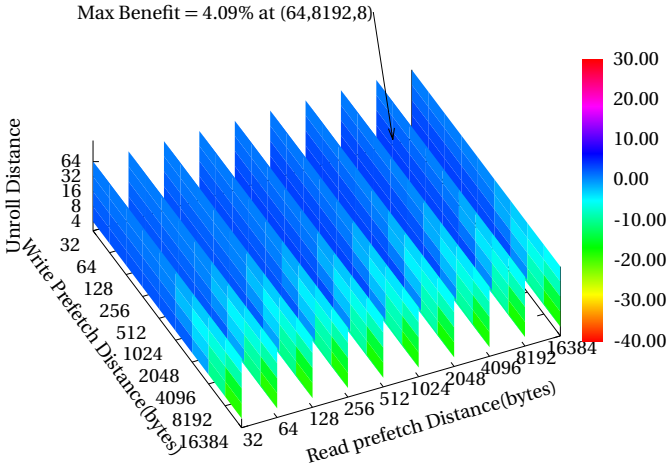
(b) NVM sp % change



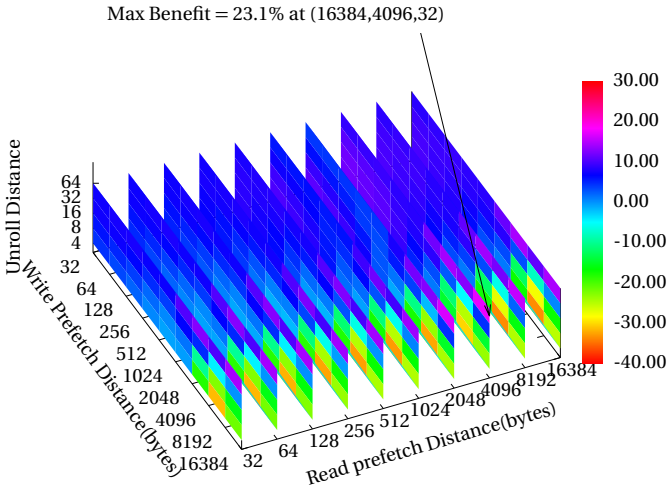
(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (mp)

Figure 4.1 Performance of 7-pt stencil (3D) stream with Temporal prefetching relative to no HW prefetching as a baseline

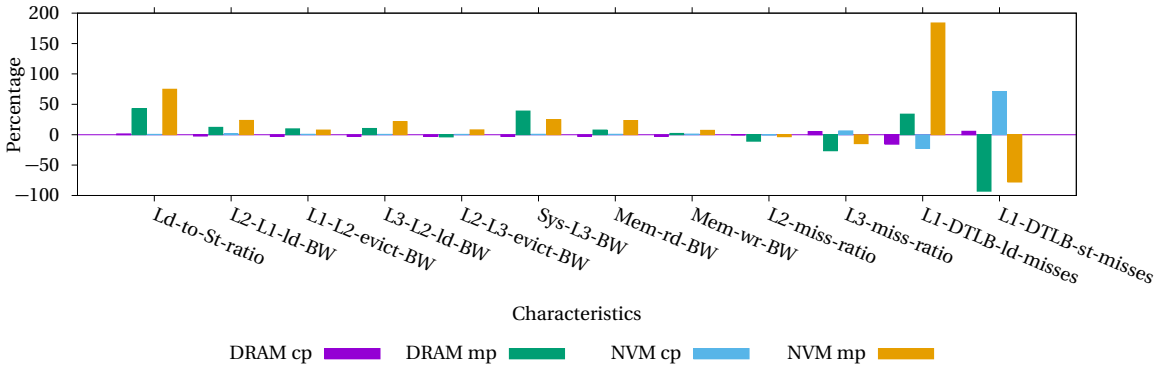
the memory accesses is large then a high unroll distance is required to reduce CPU stall cycles. The dependence on unroll distance is reflected in the NVM heatmap where the performance benefits



(a) DRAM sp % change



(b) NVM sp % change



(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

Figure 4.2 Performance of 7-pt stencil (3D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline

from higher unroll distances, which increases temporal reuse of read data and overlaps prefetch latency of the write stream with computation. Nonetheless, a short unroll distance is sufficient for

faster DRAM memory accesses under DTLB prefetching for the read streams. The pages for read data are quickly cached into the DTLB, which increases the reuse within read streams by reducing page walks and page faults. Further, short unroll distances are sufficient to overlap with prefetches spaced according to the write stream. This is also reflected in hardware counter metrics, where a reduction in L2, L3 and L1-DTLB store misses is observed for symbiotic prefetching on DRAM, which is the source of the performance benefit. For NVM, the L1-DTLB load misses are high due to a smaller read prefetch distance, and the reduction in store misses provides the main performance benefit. Cache and memory bandwidths increase as a result of reduced L1-DTLB misses.

Let us also consider Figure 4.5a in this context, which depicts the performance of all prefetching methods relative to no prefetching as a baseline per kernel; and Figure 4.5b, which depicts the performance comparison of HW prefetching relative to no HW prefetching as a baseline. The y-axis depicts the percentage change in data bandwidth and the x-axis lists all benchmark kernels for both figures. For the 7-point stencil, compiler prefetching is not able to provide any performance benefit as seen in Figure 4.5a whereas symbiotic prefetching on the other hand provides a 12% and 19% performance benefit for DRAM and NVM, respectively, over no prefetching. This is also reflected in the hardware counters, where compiler prefetching shows the smallest changes over these metrics. This results from tight bounds plus low margins on heuristics and greater dependence on HW prefetching, where its absence harms performance.

Inference 1: *The prefetching configurations show diametric behavior for the same kernel when its streams are allocated on DRAM and NVM. SW prefetching provides benefits for both DRAM and NVM without relying on or being complemented by the HW prefetcher.*

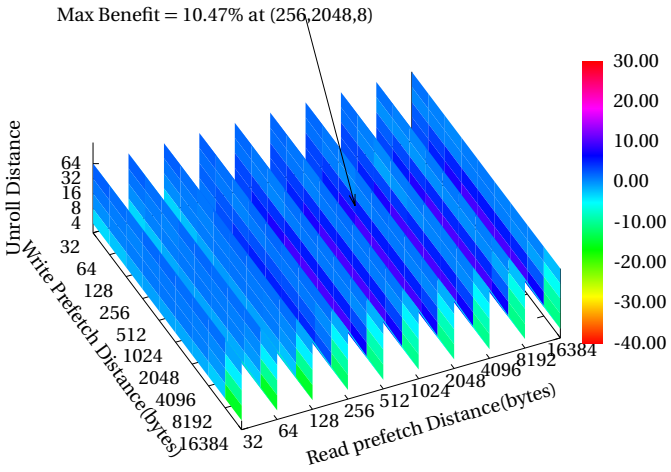
Figure 4.2 depicts the results for symbiotic and compiler prefetching for the 7-point stencil kernel with HW prefetching, with the same subfigures as before.

Observation 4: For DRAM, the bandwidth increases toward lower write and also slightly toward higher read prefetch distances, and it slightly increases for smaller unroll distances.

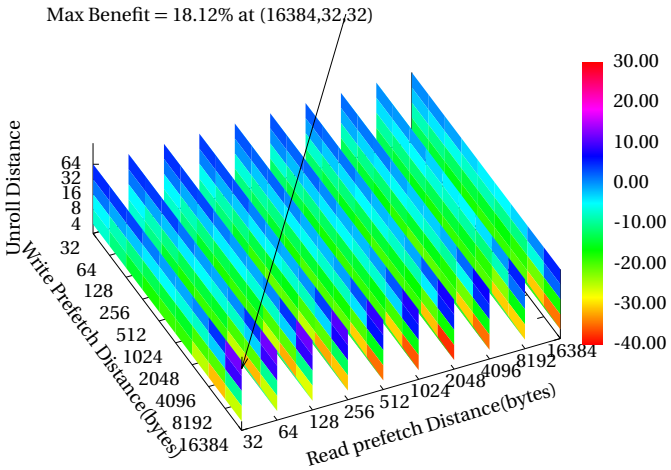
Observation 5: For NVM, the data bandwidth increases as we move higher on all three axes.

Observation 6: L1-DTLB load misses for DRAM increase under symbiotic prefetching but decrease under compiler prefetching. L1-DTLB store misses are reduced by symbiotic prefetching, whereas compiler prefetching increases it. A similar but more profound value change is observed for NVM.

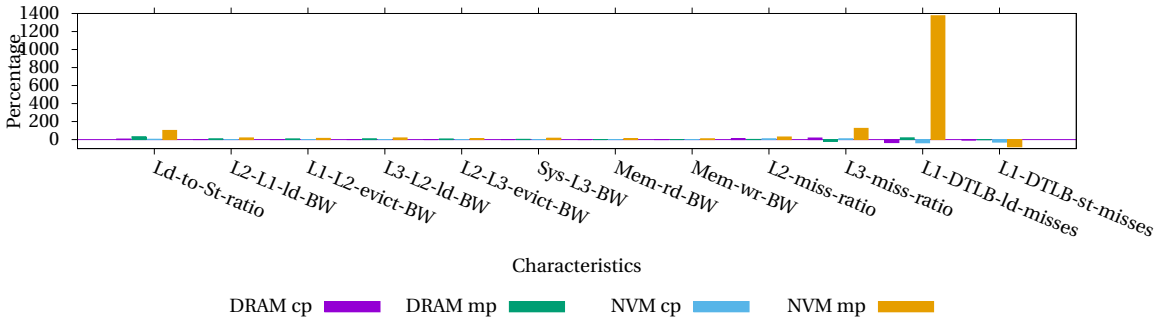
As symbiotic prefetching operates independently of HW prefetching, hardware counters change according to bandwidth trends of Figures 4.2a and 4.2b. The increase in darkness on the heatmap indicates that HW and SW prefetching *together* are able to provide the best performance benefit. For DRAM, additional HW prefetching does not significantly affect symbiotic prefetching but complements its performance by enabling higher read prefetch and unroll distances to better take advantage of temporal locality for this 7-point stencil kernel. Every dispatched load operation by the HW prefetcher is first dispatched to the L1 cache and DTLB. Upon an L1 miss, it is relayed to lower caches, which can incur up to 80 cycles to be fetched. However, a TLB miss can result in a page walk or page fault accounting for thousands of cycles. However, due to the DTLB prefetching,



(a) DRAM sp % change



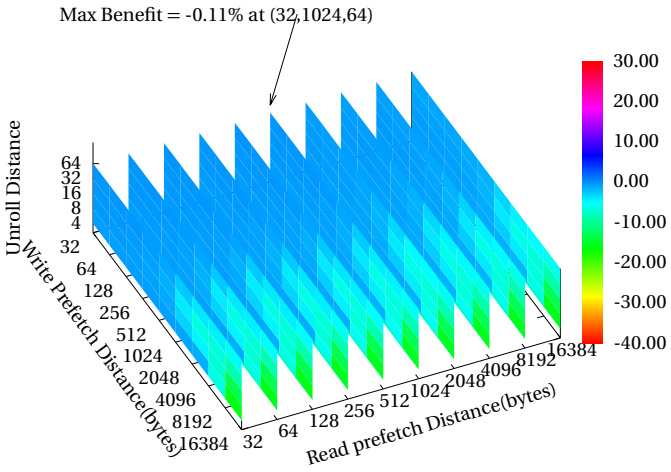
(b) NVM sp % change



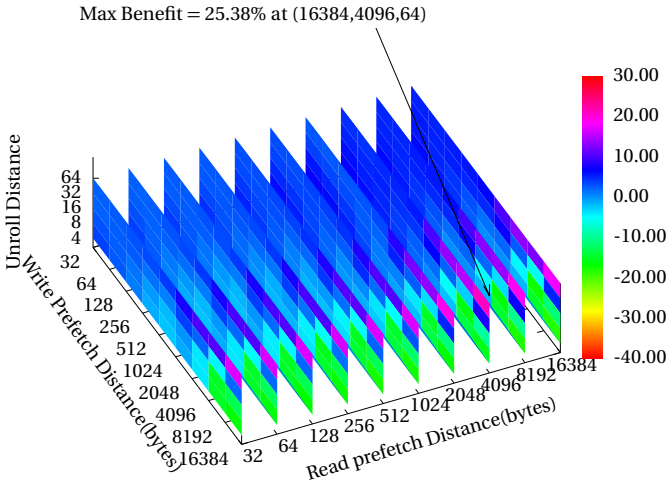
(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (mp)

Figure 4.3 Performance of 9-pt stencil (2D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline

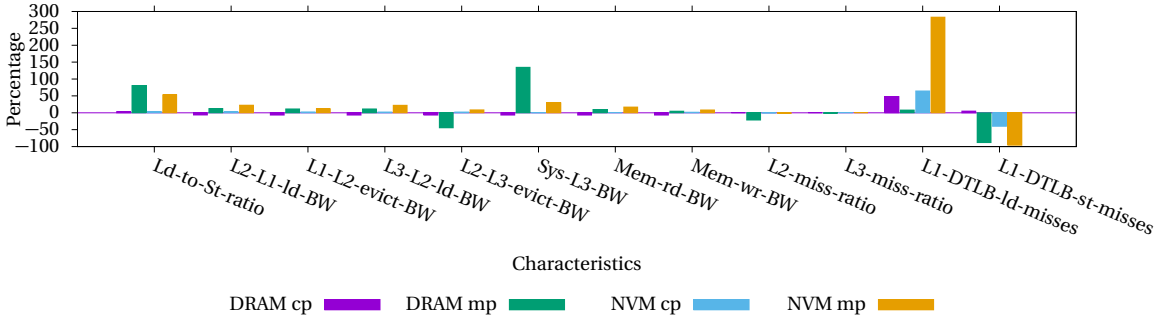
the dispatched load encounters a TLB hit most of the time reducing the overhead of page walks and page faults.



(a) DRAM sp % change



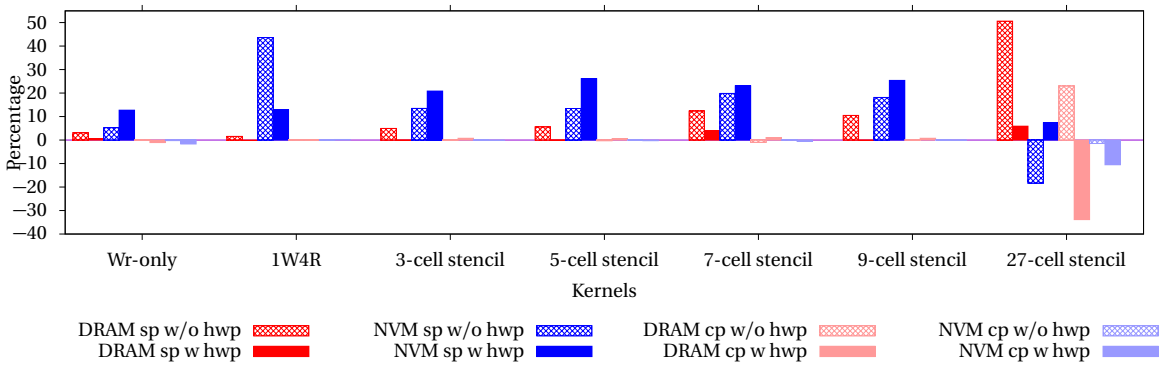
(b) NVM sp % change



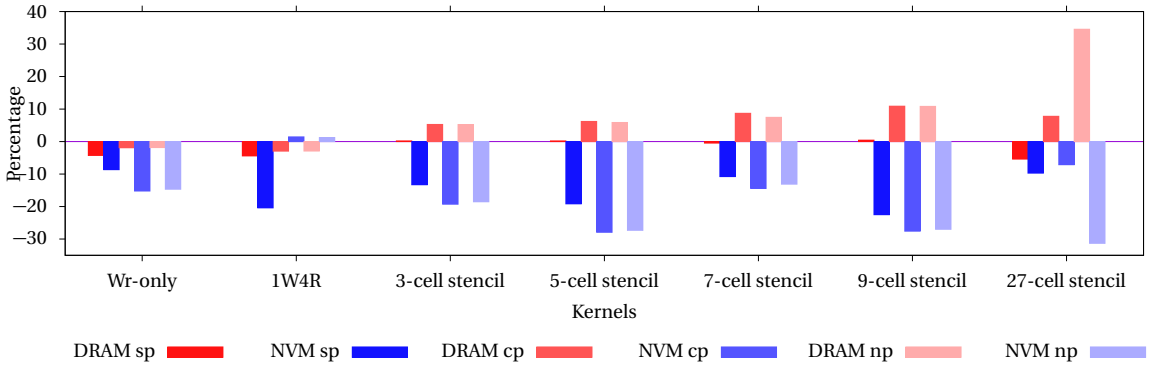
(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

Figure 4.4 Performance of 9-pt stencil (2D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline

Nonetheless, the addition of HW prefetching provides benefits (see Figure 4.5a). NVM is also assisted by HW prefetching with larger prefetch and unroll distances for symbiotic prefetching



(a) Percentage difference in performance compared to no prefetching (-O3 only) as a baseline



(b) Percentage difference in performance with HW prefetching compared to no HW prefetching as a baseline

Figure 4.5 Performance Comparison between all prefetching methods (sp = symbiotic prefetching, cp = Compiler prefetching, np = no prefetching, hwp = HW prefetching)

compared to no HW prefetching. SW read prefetching affects the DTLB cache, which results in fewer L1-DTLB load misses relative to the baseline (only HW prefetching). Here, HW prefetching serves the temporal locality of the access pattern. This frees up the SW prefetcher, which is now utilized to reduce TLB misses. Hence, the performance benefit of symbiotic plus HW prefetching results in twice the performance gain compared to no HW prefetching for NVM (see Figure 4.5a). Compiler prefetching cannot improve performance by much without DTLB prefetching; it is DTLB prefetching that provides the main source of improvement (see heatmaps of DRAM and NVM).

Inference 2: *SW prefetching improves DTLB performance in a manner symbiotic to HW prefetching driving cache benefits for the kernel for both DRAM and NVM.*

Figures 4.3 and 4.4 depicts results for symbiotic and compiler prefetching for the 9-point stencil kernel without and with HW prefetching, respectively. The hardware counters behave similarly to the 7-point stencil, except for increased L1-DTLB load misses for compiler prefetching when used in conjunction with HW prefetching. This is due to the 2D layout of the 9-point stencil, which has only 3 read streams compared the 5 streams of the 7-point stencil, which incurs aggressive short distance prefetches for reads by the HW prefetcher resulting in over-running the page boundary

Table 4.1 Symbiotic prefetching configurations and performance benefits

Kernel	Prefetch Distances(bytes), Unroll length and Performance Improvement(%)															
	DRAM								NVM							
	With HW prefetch				Without HW prefetch				With HW prefetch				Without HW prefetch			
	Read	Write	Unroll	%	Read	Write	Unroll	%	Read	Write	Unroll	%	Read	Write	Unroll	%
Write-only	-	256	64		-	128	32	3.13	-	4096	32	12.70	-	4096	32	5.27
1W4R	128	128	64	0.02	128	1024	32	1.57	16384	4	32	12.93	16384	8192	4	43.62
3-pt stencil	64	64	64	-0.07	512	128	4	4.94	4	16384	32	20.83	4096	16384	16	13.49
5-pt stencil	64	64	64	-0.01	16384	512	8	5.62	4	16384	64	26.15	32	16384	32	13.41
7-pt stencil	8192	64	8	4.09	4	128	4	12.36	4096	16384	32	23.1	32	16384	16	19.87
9-pt stencil	1024	32	64	-0.11	206	256	8	10.47	4	16384	64	25.38	32	16384	32	18.12
27-pt stencil	4	64	32	5.89	16384	128	8	50.6	206	16384	64	7.33	64	4096	16	-18.36

inflicting a DTLB miss. Any requests of the HW prefetcher benefit upper caches while symbiotic prefetching effectively becomes DTLB prefetching and results in reduced L1-DTLB load misses. Although this is observed for NVM, symbiotic prefetching is not able to increase the L1-DTLB load misses for DRAM, as the latter (DRAM) does not benefit in performance. Due to fewer read streams, HW prefetch requests populate the DTLB cache quickly enough that symbiotic prefetching becomes redundant. The heatmaps of all the kernels show a similar behavior with subtle differences owing to the number of read and write streams present in the kernel. The SW prefetch configurations indicated by the arrows that deliver the highest performance benefit for each kernel are summarized in Table 4.1. The table reinforces the inference that SW prefetching should be utilized alongside HW prefetching to combine DTLB caching and data caching due to the symbiosis between former and latter prefetch techniques, respectively. But configurations have to be adapted to the underlying memory technology (DRAM vs. NVM) and specific access patterns.

Observation 7: We observe that symbiotic prefetching by itself provides more performance benefit than compiler prefetching for DRAM and NVM over all kernels. The highest performance benefit is observed when symbiotic prefetching is used in conjunction with HW prefetching from 4 to 26%. The performance for NVM degrades when combined with HW prefetching for all kernels, except for 1W4R. But notably, symbiotic prefetching mitigates this degradation (except for the rather simplistic Wr-only and 1W4R kernels). In contrast, compiler prefetching remains ineffective, i.e., any degraded performance cannot be reduced. The HW prefetcher degrades the performance of linear array streams on DRAM, and SW prefetching is unable to mitigate this problem.

Inference 3: *HW prefetching only serves short-distance read prefetches that have high temporal locality, which are typical signatures of 1D or 2D stencil kernels. HW prefetching also needs to be adaptive to access patterns in order avoid performance degradation. Here, SW prefetching comes to the rescue as it mitigates these performance degradations on NVM.*

4.6 Adaptive SW prefetching as a compiler pass

Based on our observations and inferences, we propose the following changes to the "prefetch-loop-arrays" compiler pass in GCC:

- To adapt to a hybrid memory, compilers should become memory allocation-aware. This can be accomplished by overloading malloc() where the programmer along with the size can also

specify the desired memory device and the NUMA nodes need to be mapped to the correct memory device to complete the allocation.

- The pass already identifies different streams and their access patterns in any given loop. Hence, classifying specific workloads is feasible and should be incorporated.
- Instead of a single set of constants and threshold values to guide heuristics and the cost model, a static table for each memory technology should be maintained with specific constants/thresholds per access pattern. Our contributions in this work lay the foundation to automatically derive these constants and thresholds in a calibration run, which then allows the derivation of values similar to Table 4.1. Once access pattern and memory type of streams have been determined, the compiler pass can readily decide on prefetches given the specific constants/thresholds.
- Condition check for non-temporal locality should be removed, and non-temporal prefetches should be supported. Checking of the HW prefetcher stride needs to be lifted to allow for both symbiotic SW and HW prefetching.
- The acceptable miss rates need to be lowered for large distances to accommodate DTLB prefetching. Similarly, many thresholds (trip-count-to-ahead-ratio, prefetch-mod-to-unroll-factor-ratio, memory-ref-count-reasonable and insn-to-prefetch-ratio-too-small also) need to be higher for symbiotic prefetching to allow DTLB prefetching in software — as well as support for higher unroll distances that are adaptive for slow and fast memories.
- With predictable access patterns, priority ordering of prefetches based on lowest prefetch-modulo can be replaced by the (more easily) predicted performance benefit.
- Finally, loop peeling is required when emitting prefetch instructions at the end of the pass.

The changes are beyond the scope of this work, but only become feasible due to the contributions of our memory-hybrid, adaptive and symbiotic prefetching.

4.7 Future Work

In the future, we would like to perform similar characterizations for other specialized workloads and identify the ones that can benefit from SW prefetch support while using a DRAM/NVM hybrid memory system. We plan to implement our adaptive prefetcher design and evaluate its performance and feasibility for a variety of HPC applications and mini-apps. The compiler performs many loop optimizations as part of the O1, O2 and O3 optimization sets. We plan to study the effect of all relevant optimizations on the performance of HPC workloads executing on hybrid memory systems and modify them to improve performance. We aim to incorporate the adaptive prefetch methods with an intelligent data placement policy that can extract the best performance for a hybrid memory system and completely automate memory allocation and SW prefetching for HPC workloads.

4.8 Conclusion

Our work provides novel insight that the existing rigid and conservative approach to SW prefetching leaves ample performance potential on the table for HPC workloads. We show that existing HW prefetchers are neither optimized for NVM memory nor for non-temporal workloads. We contribute HW and SW prefetch methods that are more adaptive and show that they succeed in extracting symbiotic performance while being sensitive to hybrid memory systems. Our DTLB-based symbiotic SW prefetching improves the performance of HPC kernels from 4 to 26% for data streams allocated on both DRAM and NVM, and our SW prefetching complements HW prefetching rather than competing with it. We also present a simple design to modify an existing SW prefetch compiler pass to implement our prefetch configurations with the potential to automatically improve performance for HPC workloads on future hybrid memory systems. In this chapter, we show part of our hypothesis as we design a symbiotic HW/SW prefetching approach for heterogeneous memory systems that outperforms current prefetching approaches and results in additional performance improvements.

CHAPTER

5

PEARS: A PERFORMANCE-AWARE STATIC AND RUNTIME FRAMEWORK FOR HPC APPLICATIONS USING HETEROGENEOUS MEMORY SYSTEM

5.1 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 [Sod16]. Graphics devices utilizing the General Purpose Graphics Processing Units (GPGPU) have also been equipped with HBM [OâĂ14]. Fujitsu uses a hybrid memory cube (HMC) for its A64FX ARM-based chips to deliver high bandwidth memory access to all the compute cores [Fuj]. 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) [Hru18]. The Aurora supercomputer [Aur] 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 [Can15] provides support for memory management on HBM devices. Intel offers the PMDK library [Teh20] that supports memory management on NVM devices. There are other software systems like Atlas [Cha14] and Mnemosyne [Vol11] 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 [Le619] 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/deallocations 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.

This work does not evaluate Memory-mode performance as we have observed in Chapters 2 and 3 that the page fault overhead of DRAM cache is much higher than the L3 cache miss overhead of

using NVM in Flat mode. This work provides more dynamism to a hybrid memory architecture than Memory mode by interacting directly with the SRAM caches. 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 5.2, we present the related work. We describe the heterogeneous memory architecture in Section 5.3 and present the design details of our framework in Section 5.4. We present our experimental setup in Section **??** and analyze the results in Section 2.6 followed by our conclusions in Section 5.8.

5.2 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. [Yan15; Izr19] 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. [Pat19] 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 [Pat20]. Peng et al. [Pen19] 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. [Psa19] 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. [Leó19] 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. [OB17] 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. [Cha17] proposed a memory-heterogeneity-aware runtime system that guides data prefetch and eviction for a HBM-DRAM memory architecture. Alvarez et al. [Alv18] similarly proposed a runtime approach to transparently manage stacked DRAM memories in task-based programming models. Sánchez et al. [SB18] 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. [Per16] 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. [Ben18] presented a realistic performance model to execute scientific workflows on high-bandwidth memory architec-

tures such as Intel's Knights Landing. There have been several works that proposed static memory management, e.g., Greenspan et al. [Gre19] 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. [KC16] 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.

5.3 System Architecture

Table 5.1 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
NVDIMM protocol	DDR-T
NVRAM size	1.5 TB
No. of nodes	1
Interconnect	No
Operating System	CentOS 7
Compiler	GCC 9.3.0

The system used in experiments is a single HPE Proliant DL360 node (see Table 5.1) 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.

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 [Izr19]. 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 5.2.

5.3.1 SICM

The SICM (Simple Interface Complex Memory) [Lan19] 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 [Apr06] 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.

Table 5.2 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

5.4 The PEARS Software Framework

PEARS has multiple components depicted in Figure 5.2. 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.

5.4.1 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

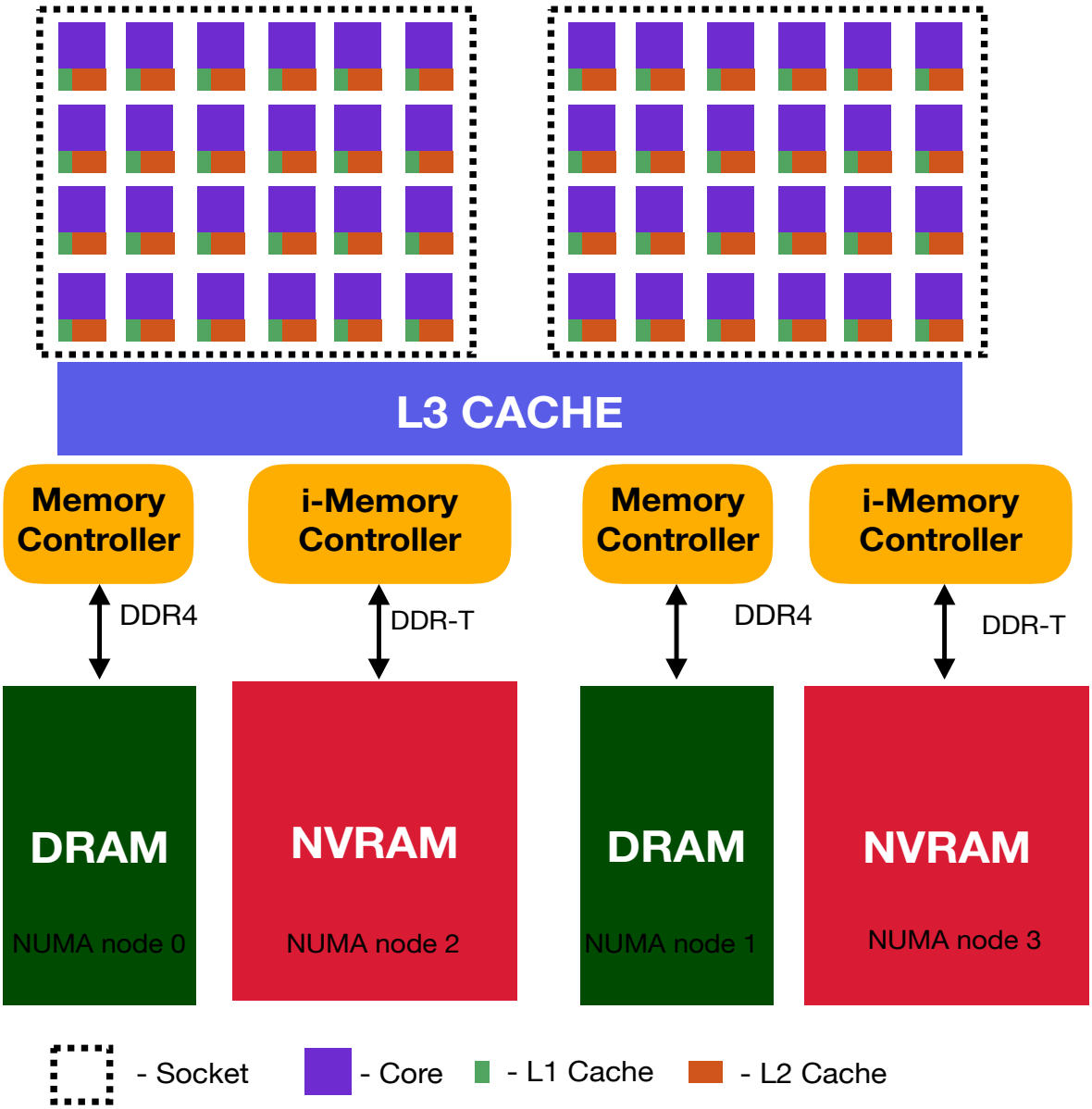


Figure 5.1 Heterogeneous memory architecture (flat mode)

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

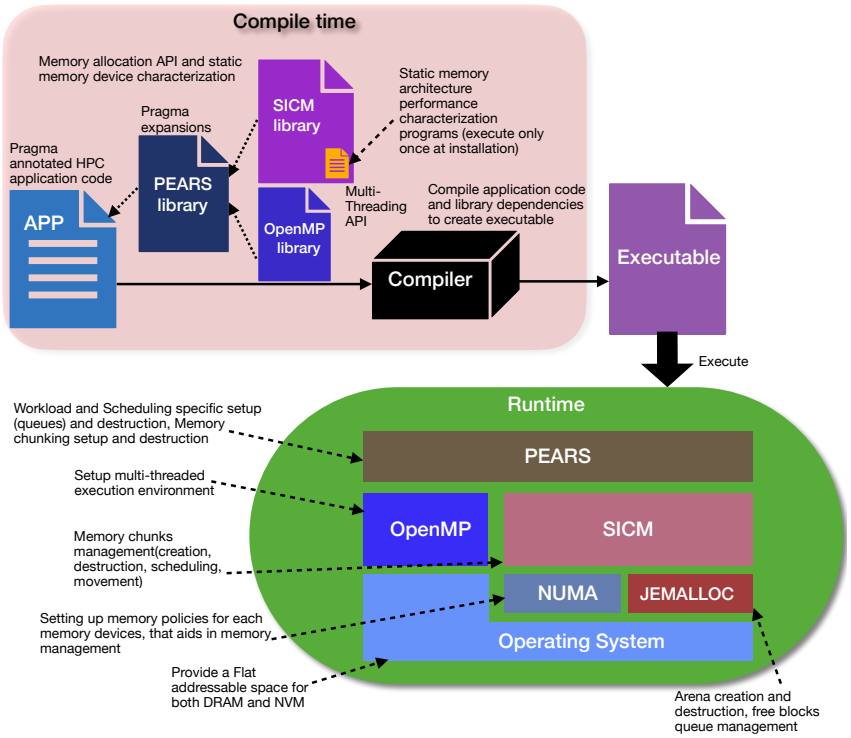


Figure 5.2 PEARS: Block and flow diagram of static and dynamic components

```
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 5.3 Example of the classification and measurements from the Memory characterization

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 a 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 5.3. These classifications and measurements are then used by the extended SICM library

at runtime to perform memory management and scheduling.

5.4.2 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.

5.4.3 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.

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 5.4. The macros also provide support for adjusting loop limits and identifying access patterns. Currently, we provide macro-based programming support for tiled matrix multiplication as well as one-, two, and three-dimensional (1D/2D/3D) stencil kernels.

5.4.4 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 5.3 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

```
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 5.4 Sample code of tiled matrix multiply with PEARS macros

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 of 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 5.4, 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.

5.5 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 5.3. 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 [CAP14]. Some examples of stencil code-based HPC applications are Vector Particle In Cell (VPIC) [Bow08b; Bow09; Bow08a] and Algebraic Multi-grid (AMG) [Yan02], 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 [KS07].

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 5.3.

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 [Ope; Sre19] 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 [Tre10] 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

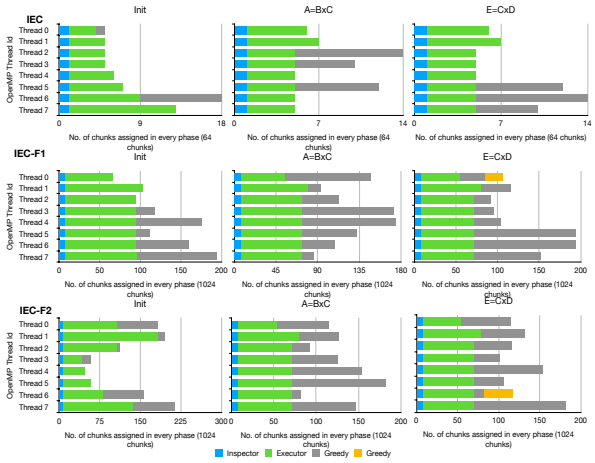


Figure 5.5 Memory chunk distribution of IEC scheduling for dynamic and static chunk sizes

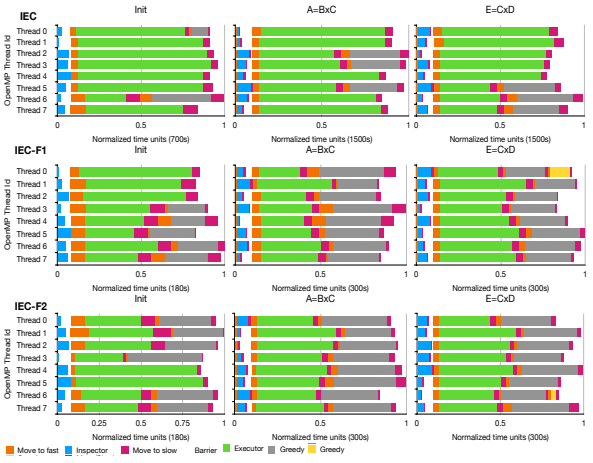


Figure 5.6 Time series distribution of IEC scheduling for dynamic and static chunk sizes

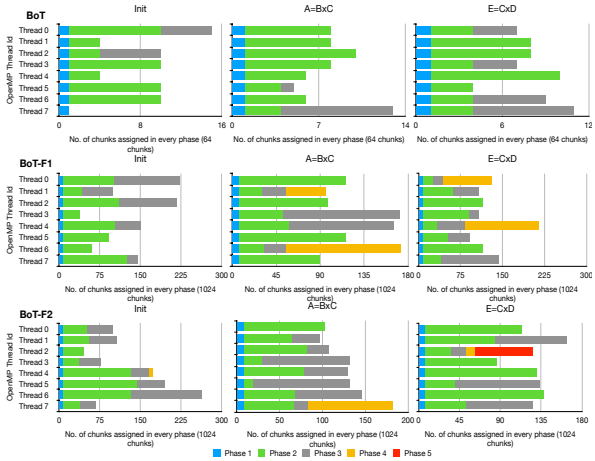


Figure 5.7 Memory chunk distribution of BoT scheduling for dynamic and static chunk sizes

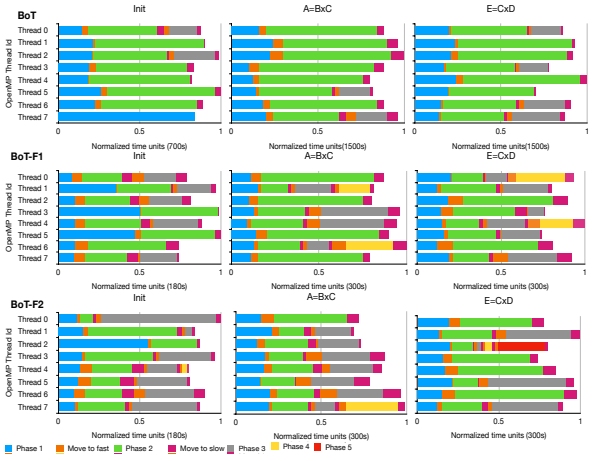


Figure 5.8 Time series distribution of BoT scheduling for dynamic and static chunk sizes

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.

5.6 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 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 5.5 and 5.7 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 5.6 and 5.8 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 5.5) 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 5.6) with some imbalance during the Executor 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*

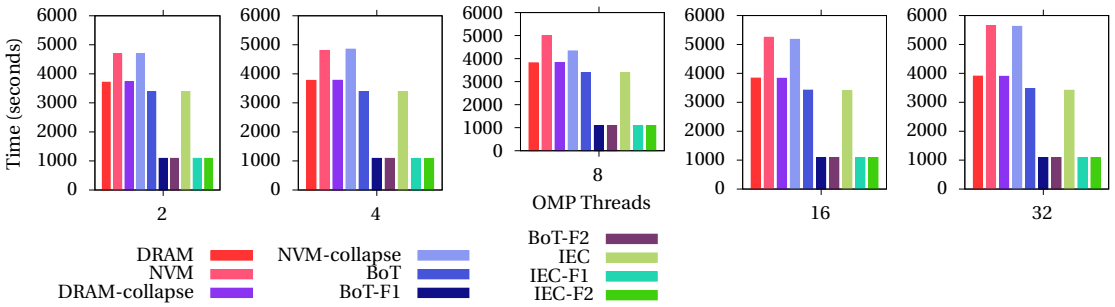


Figure 5.9 Solve Time comparison of scheduling and chunking policies for TMM with Strong scaling

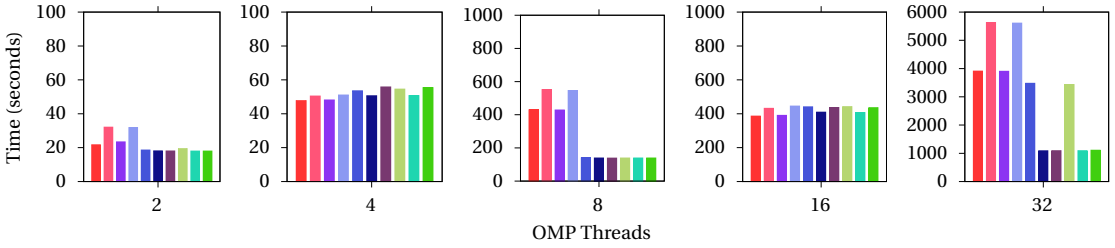
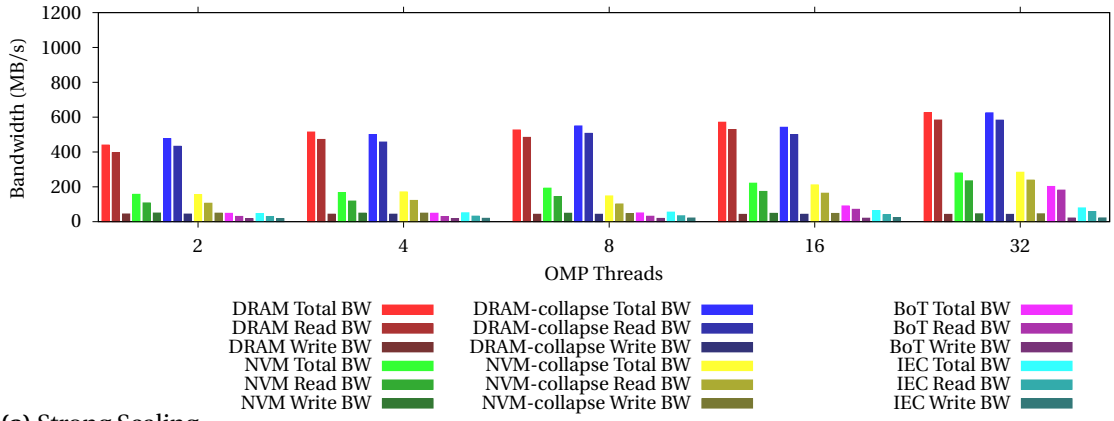


Figure 5.10 Solve Time comparison of scheduling and chunking policies for TMM with Weak scaling

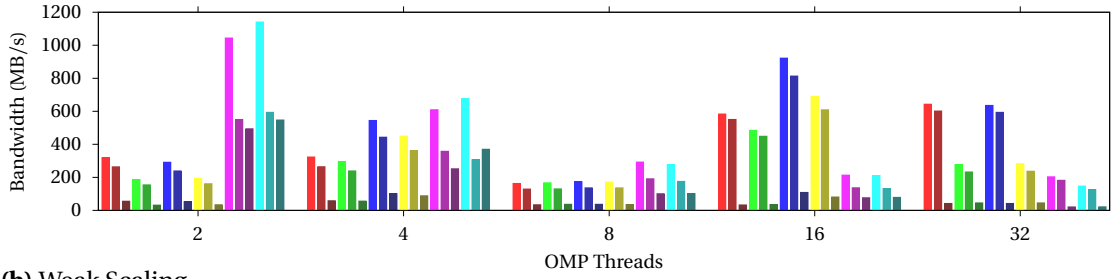
memory allocation and runtime scheduling methods for the TMM workload.

Figures 5.9 and 5.10 depict the time comparison of all scheduling and chunking policies and Figures 5.11a and 5.11b depict the memory bandwidth results for strong and weak scaling of the TMM benchmark. For Figures 5.9 and 5.10, 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 5.12a and 5.12b, respectively. This indicates that there more cache hits are due to the memory chunking, which reduces the dependency on memory access latency. Figures 5.13a and 5.13b 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).

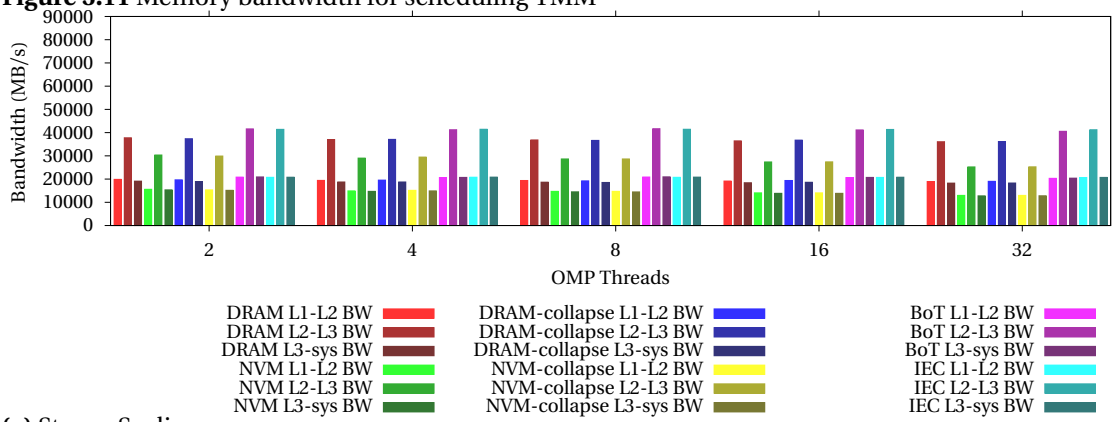


(a) Strong Scaling

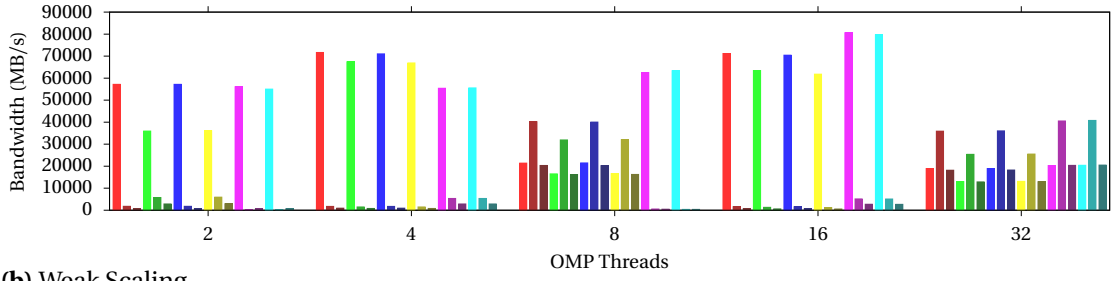


(b) Weak Scaling

Figure 5.11 Memory bandwidth for scheduling TMM



(a) Strong Scaling

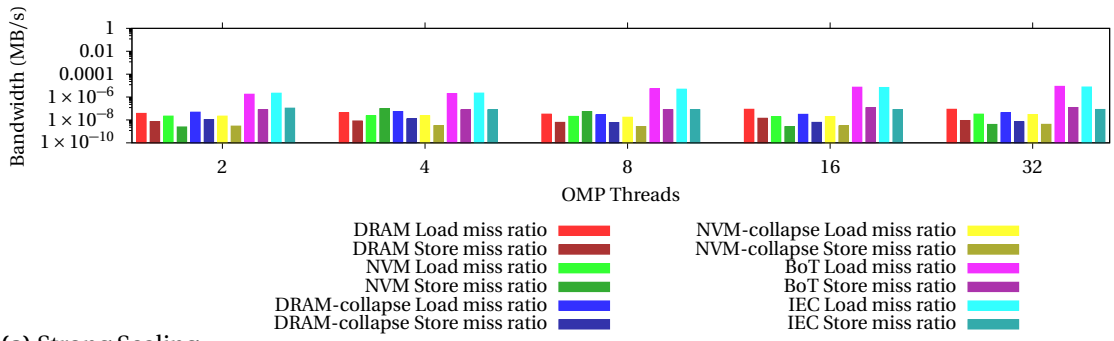


(b) Weak Scaling

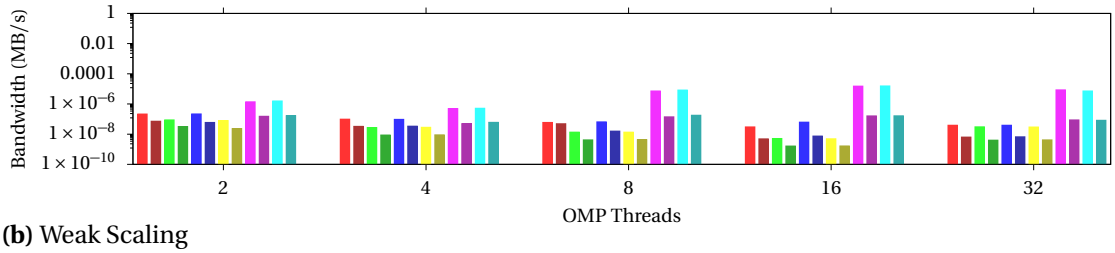
Figure 5.12 Cache bandwidth for scheduling TMM

Hence, the execution time still grows at a cubic rate.

Figures 5.14a and 5.14b depict the memory bandwidth for different memory chunk sizes under BoT and IEC scheduling. Both scheduling policies determine chunk sizes dynamically at runtime

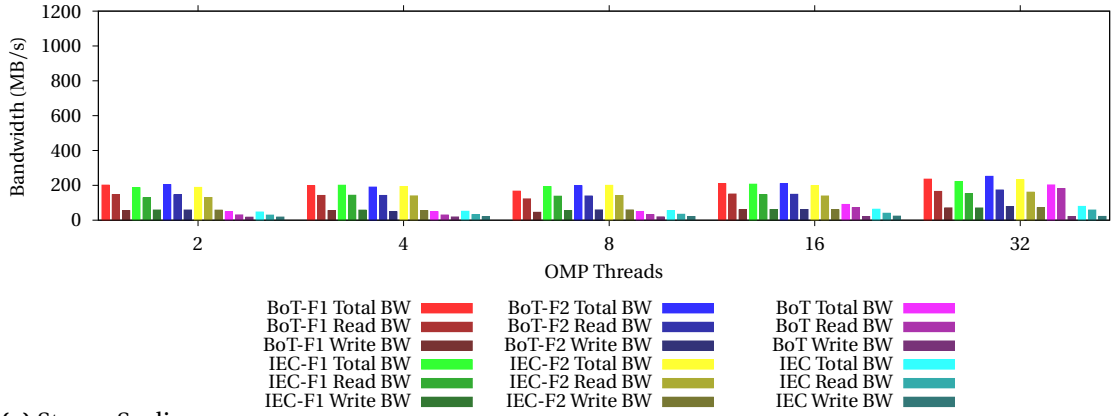


(a) Strong Scaling

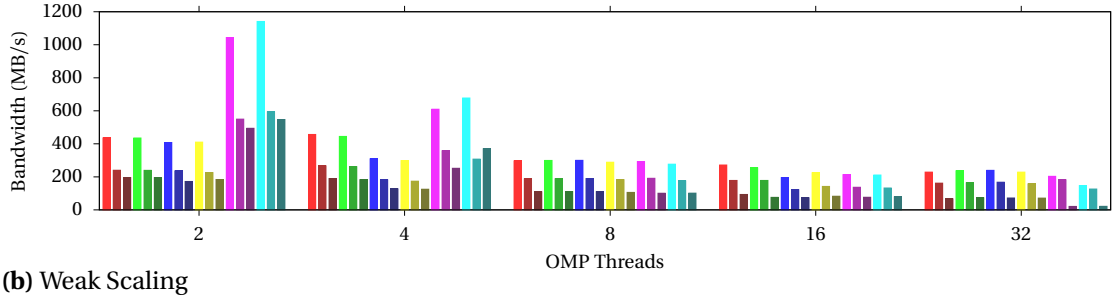


(b) Weak Scaling

Figure 5.13 DTLB load store miss ratios for scheduling TMM



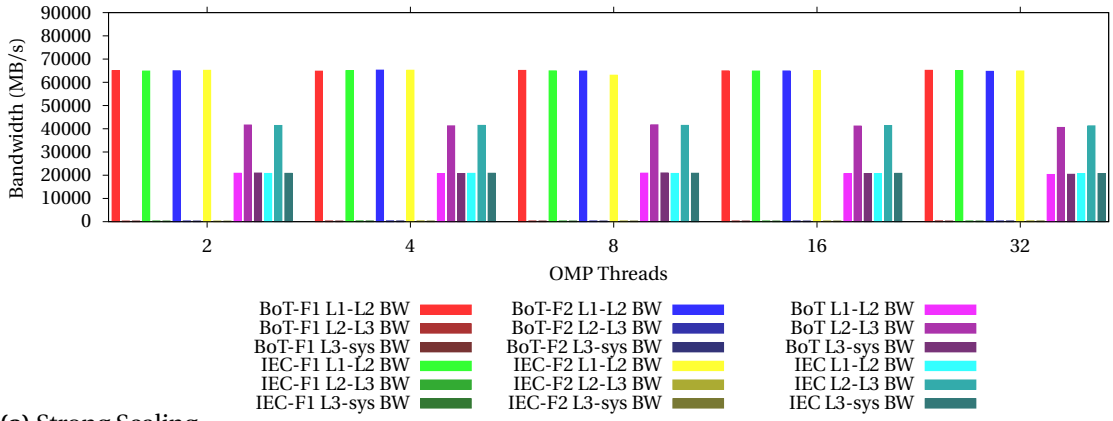
(a) Strong Scaling



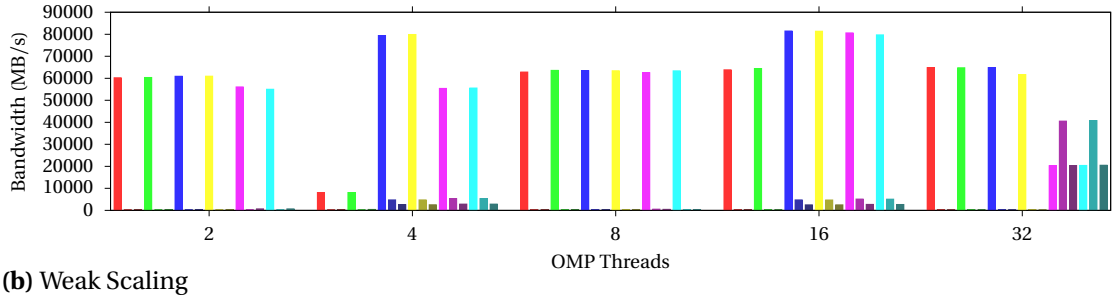
(b) Weak Scaling

Figure 5.14 Memory bandwidth over different chunk sizes for TMM

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 5.3. We observe that fixed chunk sizes that align with the private cache achieve lower execution time, i.e., outperform dynamically

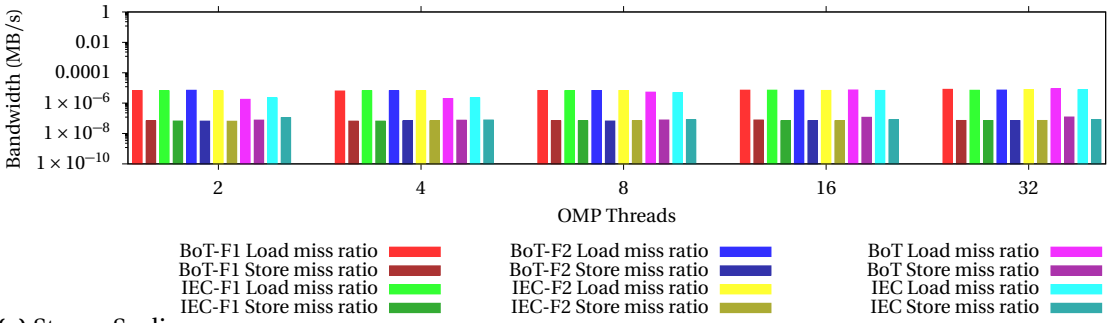


(a) Strong Scaling

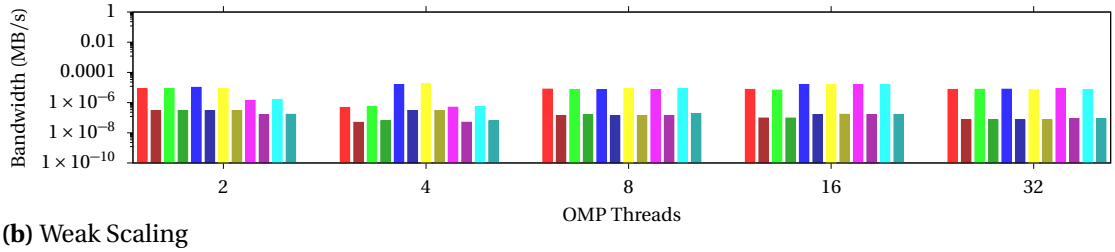


(b) Weak Scaling

Figure 5.15 Cache bandwidth over different chunk sizes for TMM



(a) Strong Scaling



(b) Weak Scaling

Figure 5.16 DTLB load store miss ratios over different chunk sizes for TMM

determined chunk sizes in Figures 5.9 and 5.10. Both IEC and BoT result in almost 3x times faster execution with their 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

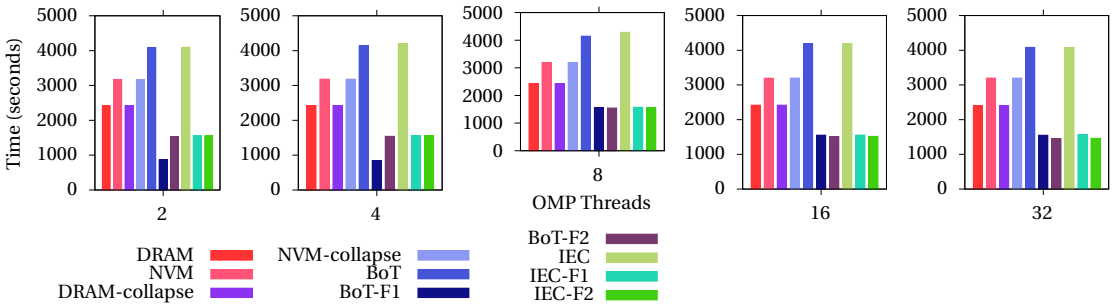


Figure 5.17 Solve Time comparison of scheduling and chunking policies for 27-pt stencil with Strong scaling

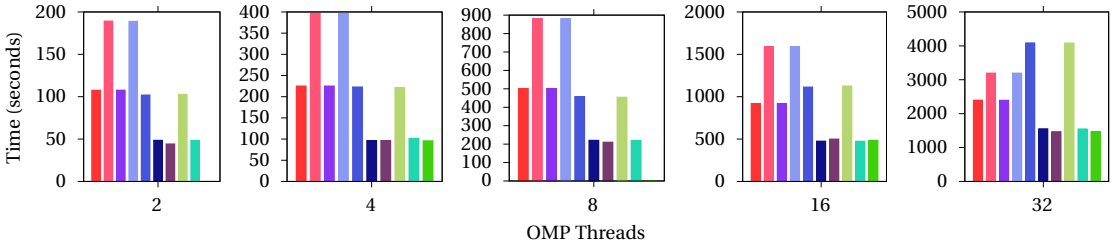


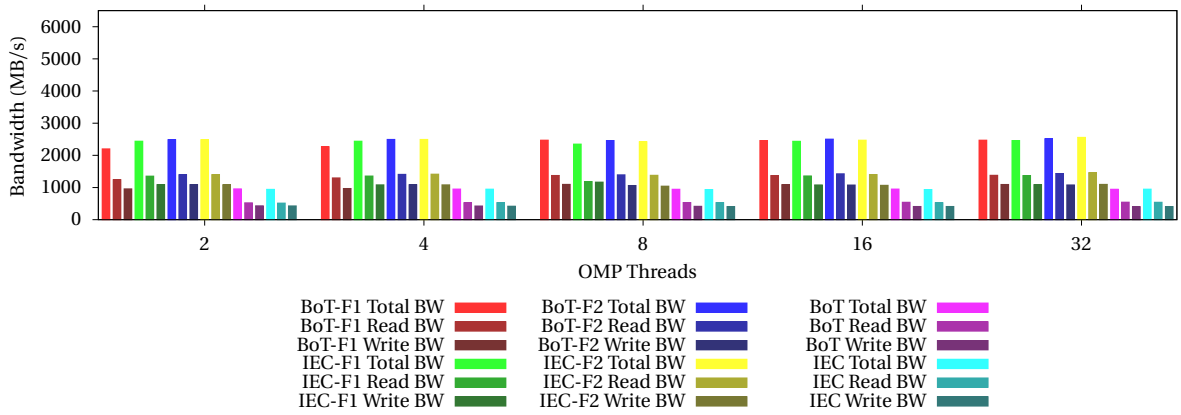
Figure 5.18 Solve Time comparison of scheduling and chunking policies for 27-pt stencil with Weak scaling

L1-L2 cache bandwidth (as shown in Figures 5.15a and 5.15b) along with higher DTLB load misses (as shown in Figures 5.16a and 5.16b), which contributes to the faster performance.

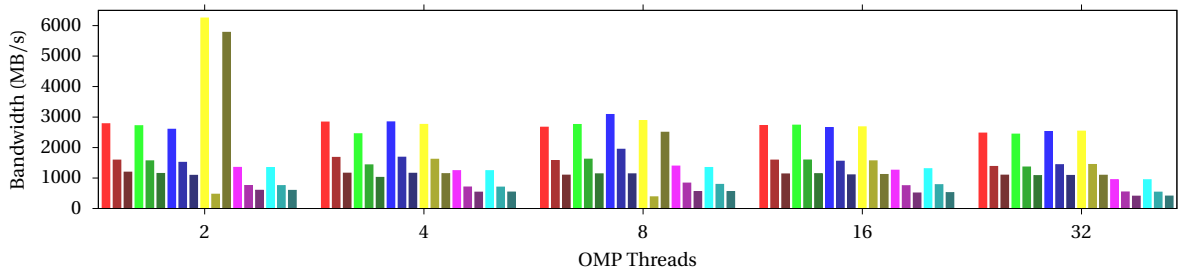
Inference 1: *Splitting the memory allocation into chunks and scheduling them across memory devices simultaneously while fully utilizing fast memory and all memory controllers at the same time helps achieve higher performance for TMM on a heterogeneous memory system.*

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 is 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.



(a) Strong Scaling



(b) Weak Scaling

Figure 5.19 Memory bandwidth over different chunk sizes for 27-pt stencil

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

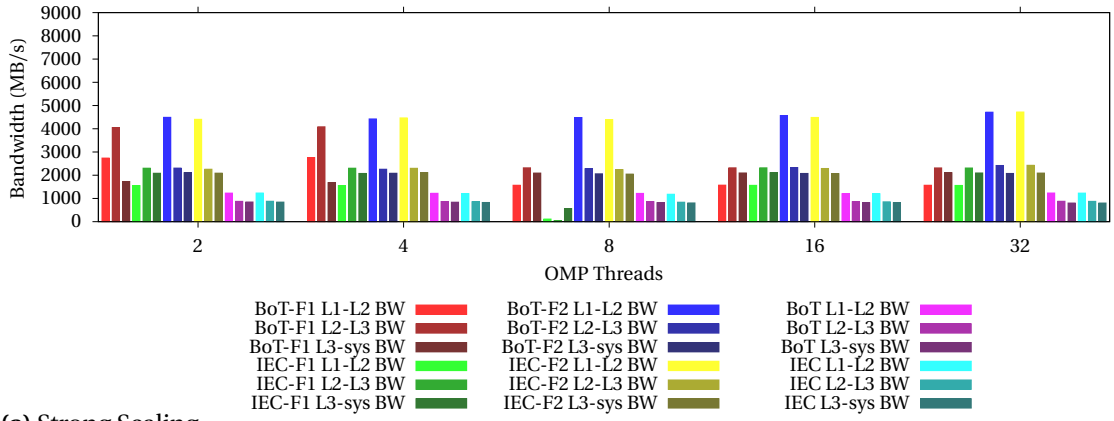
DRAM allocation and 40% slower than NVM allocation for strong scaling. Similarly, the 2D 9-point stencil (2 halo) benchmark and the 2D 9-point stencil (with diagonals) benchmark result in 3x and 2.5x times longer execution, respectively, under BoT compared to DRAM allocation, and 30% slower execution 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.*

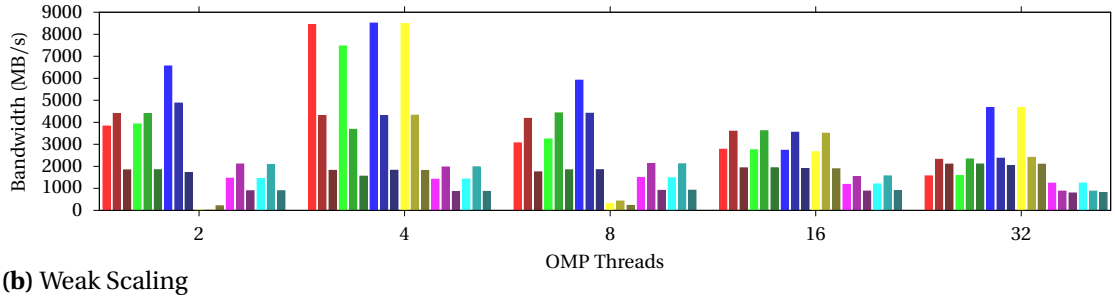
Observation 4: *For 3D stencil codes under both strong and weak scaling, runtime scheduling and memory management of PEARS static chunking outperforms the pure DRAM methods, while PEARS with dynamic chunking is on par with the DRAM methods.*

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 5.17 and 5.18), which means it is outperforming DRAM allocation. The same improvement in performance is not observed for BoT for 7- and 13- 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 5.19a, 5.19b, 5.20a



(a) Strong Scaling



(b) Weak Scaling

Figure 5.20 Cache bandwidth over different chunk sizes for 27-pt stencil

and 5.20b.

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.*

5.7 Future Work

Currently, the PEARS framework support only TMM and stencil workloads. We would want to support all types workloads on the framework and make it more generic. We would want to incorporate more static analysis at the compiler level to further reduce the onus on the programmer to manage workloads on a heterogeneous memory system. This could entail automatic insertion of code blocks for memory management and runtime scheduling. We could utilize more static analysis of the codes that can help us determine the best memory chunk size of any given workloads. On the runtime side, we currently utilize NUMA mbind() to move pages between memory devices. This has an overhead moving pages at a coarser granularity and kernel dependencies. We would like to have a custom DMA based memory movement mechanism that could help move memory chunks asynchronously at a finer granularity and achieve higher parallelism and low overhead. We would look to reduce memory and halo value communication overhead due to memory chunking and also incorporate prefetching techniques to improve cache performance on heterogeneous memory systems.

5.8 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. Our hypothesis holds in so far as our automated framework deliv-

ers optimal performance for many HPC workloads executing on heterogeneous memory systems, which exceeds the performance of existing allocation and scheduling strategies and relieves the application developer from embedding complex optimizations in their applications.

CHAPTER

6

CONCLUSION

In this work, a thorough characterization of a true byte-addressable heterogeneous memory system is presented that uses DRAM and NVM for HPC workloads. Although using NVM as main memory directly hampers the performance, it is shown that NVM has the potential to reduce the energy consumption of HPC applications with reasonable trade-offs between performance and energy. Optane DC PMMs can enable application executions to close the gap between core count and memory capacity scaling. A DRAM-NVM hybrid system was evaluated at a large memory capacity scale by comparing its performance to traditional memory system based clusters of equivalent memory size. Results indicate that such heterogeneous architecture provide a promising path to extend memory address spaces of HPC systems without compromising performance and energy for certain codes, which translates into lower operational cost combined with lower acquisition costs.

The heterogeneous memory performance of contemporary systems was evaluated under existing HW and SW prefetching support for data pipelines. This work provides novel insight indicating that the existing rigid and conservative approach to SW prefetching leaves ample performance potential on the table for HPC workloads. It is shown that existing HW prefetchers are neither optimized for NVM memory nor for non-temporal workloads. Both HW and SW prefetch methods are contributed that are more adaptive and show that they succeed in extracting symbiotic performance while being sensitive to hybrid memory systems.

Finally, using all characterizations and evaluations, a solution for optimally utilizing a heterogeneous memory system for HPC applications is presented. This features a memory management and runtime scheduling framework called PEARS that enable HPC application programmers to write programs to take advantage of the resources provided by a heterogeneous memory system. It helps reduce the performance impact of a slower memory in the memory system while, at the same

time, helping to support larger problem sizes than a homogeneous memory systems in synergy with reducing the dependency on the application programmer to optimize memory performance for a heterogeneous memory system. The PEARS framework has the potential to 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.

In summary, we have shown that the hypothesis holds as we present an automated framework that considers static and dynamic characteristics of an application's memory footprint to exploit the complexity of heterogeneous memory systems. It enables memory allocation and data movement within HPC applications so that cost, energy and performance benefits can be realized for problem sizes in excess of singular homogeneous memories.

BIBLIOGRAPHY

[al.] al., D. K. P. et. *OSU Micro-Benchmarks*. https://mvapich.cse.ohio-state.edu/benchmarks/.

[Alv18] Alvarez, L. et al. "Runtime-guided management of stacked dram memories in task parallel programs". *Proceedings of the 2018 International Conference on Supercomputing*. 2018, pp. 218–228.

[Apa13] Apalkov, D. et al. "Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM)". *J. Emerg. Technol. Comput. Syst.* **9**.2 (2013), 13:1–13:35.

[Apr06] April, J. E. "A Scalable Concurrent malloc(3) Implementation for FreeBSD". 2006.

[Ash19] Ashraf, M. U. et al. "IMPROVING PERFORMANCE IN HPC SYSTEM UNDER POWER CONSUMPTIONS LIMITATIONS". *International Journal of Advanced Research in Computer Science* **10**.2 (2019).

[Aur] *Aurora - May 2019*. https://press3.mcs.anl.gov/aurora/.

[Bad04] Badawy, A.-H. et al. "The efficacy of software prefetching and locality optimizations on future memory systems". *Journal of Instruction-Level Parallelism* **6**.7 (2004).

[Bal94] Bala, K. et al. "Software prefetching and caching for translation lookaside buffers". *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association. 1994, p. 18.

[Ben18] Benoit, A. et al. "A performance model to execute workflows on high-bandwidth-memory architectures". *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[Ber08] Bergman, K. et al. "Exascale computing study: Technology challenges in achieving exascale systems". *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* **15** (2008).

[Bow08a] Bowers, K. J. et al. "0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner". *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 63:1–63:11.

[Bow08b] Bowers, K. J. et al. "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation". *Physics of Plasmas* **15**.5 (2008), p. 055703. eprint: https://doi.org/10.1063/1.2840133.

[Bow09] Bowers, K. J. et al. "Advances in petascale kinetic plasma simulation with VPIC and Roadrunner". *Journal of Physics: Conference Series* **180** (2009), p. 012055.

[Cal91] Callahan, D. et al. "Software prefetching". *ACM SIGARCH Computer Architecture News* **19**.2 (1991), pp. 40–52.

[Can15] Cantalupo, C. et al. “memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.” (2015).

[CAP14] Cruz, R. de la & Araya-Polo, M. “Modeling stencil computations on modern HPC architectures”. *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer. 2014, pp. 149–171.

[Cas] *Cascade Lake - Microarchitectures - Intel, WikiChip 2019*. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake#Memory_Hierarchy.

[Cha14] Chakrabarti, D. R. et al. “Atlas: Leveraging locks for non-volatile memory consistency”. *ACM SIGPLAN Notices*. Vol. 49. 10. ACM. 2014, pp. 433–452.

[Cha17] Chandrasekar, K. et al. “A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications”. *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, pp. 1293–1300.

[Cob12] Coburn, J. et al. “NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories”. *ACM Sigplan Notices* **47**.4 (2012), pp. 105–118.

[DM98] Dagum, L. & Menon, R. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. *IEEE Comput. Sci. Eng.* **5**.1 (1998), pp. 46–55.

[Dul14] Dulloor, S. R. et al. “System software for persistent memory”. *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 15.

[Fro] *Frontier - Nov 2019*. https://www.cray.com/company/news-and-media/doe-frontier-press-release.

[Fuc14] Fuchs, A. et al. “Loop-aware memory prefetching using code block working sets”. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 533–544.

[Fuj] *Fujitsu HPC Roadmap*. URL: \url{https://www.fujitsu.com/global/Images/fujitsu-hpc-roadmap-beyond-petascale-computing.pdf}.

[Gam19] Gamatié, A. et al. “Emerging NVM Technologies in Main Memory for Energy-Efficient HPC: an Empirical Study” (2019).

[Gcc] *GCC Data Prefetching Support*. URL: https://gcc.gnu.org/projects/prefetch.html.

[Gil19] Gill, G. et al. “Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory”. *CoRR* **abs/1904.07162** (2019). arXiv: 1904.07162.

[GM11] Gupta, A. & Milojicic, D. “Evaluation of hpc applications on cloud”. *2011 Sixth Open Cirrus Summit*. IEEE. 2011, pp. 22–26.

[Gre19] Greenspan, D. “LLAMA-automatic memory allocations: an LLVM pass and library for automatically determining memory allocations”. *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 363–372.

[Gup17] Gupta, S. et al. “Failures in large scale systems: long-term measurement, analysis, and implications”. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, p. 44.

[Hid90] Hidaka, H. et al. “The cache DRAM architecture: A DRAM with an on-chip cache memory”. *IEEE Micro* **10**.2 (1990), pp. 14–25.

[Hin18] Hines, J. “Stepping up to Summit”. *Computing in science & engineering*. **20**.2 (2018), pp. 78–82.

[Hpc] *HPC evolution*. `https://insidehpc.com/2016/08/the-evolution-of-hpc/`.

[Hru18] Hruska, J. “Intel announces new optane DC persistent memory year 2018”. *ExtremeTech https://www. extremetech. com/extreme/270270-intel-announces-new-optane-dc-persistent-memory* (2018).

[Inta] *Intel64 and IA-32 Architectures Optimization Reference Manual*. URL: `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf?countrylabel=Colombia`.

[Intb] *IntelÂő 64 and IA-32 Architectures Software DeveloperâĂŹs Manual*. URL: `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4`.

[Isl17] Islam, M. et al. “Hbm-resident prefetching for heterogeneous memory system”. *International Conference on Architecture of Computing Systems*. Springer. 2017, pp. 124–136.

[Izr19] Izraelevitz, J. et al. “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module”. *CoRR* **abs/1903.05714** (2019). arXiv: `1903.05714`.

[Jed] *JEDEC DDR4 SDRAM standards year 2017*. (2017). URL: `https://www.jedec.org/standards-documents/docs/jesd79-4a`.

[Kan13] Kannan, S. et al. “Optimizing checkpoints using nvm as virtual memory”. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 29–40.

[Kar13] Karlin, I. et al. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, 2013, pp. 1–9.

[KC16] Khaldi, D. & Chapman, B. “Towards automatic HBM allocation using LLVM: a case study with knights landing”. *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE. 2016, pp. 12–20.

[KS07] Kretz, T. & Schreckenberg, M. “Moore and more and symmetry”. *Pedestrian and evacuation dynamics 2005*. Springer, 2007, pp. 297–308.

[Lan19] Lang, M. K. “Simplified Interface to Complex Memory (SICM) FY19 Project Review” (2019).

[Lat04] Latham, R. et al. “A next-generation parallel file system for Linux cluster.” *LinuxWorld Mag.* **2** (2004).

[Lee09] Lee, B. C. et al. “Architecting phase change memory as a scalable dram alternative”. *ACM SIGARCH Computer Architecture News* **37**.3 (2009), pp. 2–13.

[Lee12] Lee, J. et al. “When prefetching works, when it doesnâĂŹt, and why”. *ACM Transactions on Architecture and Code Optimization (TACO)* **9**.1 (2012), pp. 1–29.

[Leó19] León, E. A. et al. “M&MMs: navigating complex memory spaces with hwloc”. *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 149–155.

[Li12] Li, X. et al. “NV-process: a fault-tolerance process model based on non-volatile memory”. *Proceedings of the Asia-Pacific Workshop on Systems*. ACM. 2012, p. 1.

[Lim09] Lim, K. et al. “Disaggregated memory for expansion and sharing in blade servers”. *ACM SIGARCH computer architecture news*. Vol. 37. 3. ACM. 2009, pp. 267–278.

[LL15] Li, B. & Lu, P. “The Evolution of Supercomputer Architecture: A Historical Perspective”. *CCF Conference on Computer Engineering and Technology*. Springer. 2015, pp. 145–153.

[Lus06] Luszczek, P. R. et al. “The HPC Challenge (HPCC) benchmark suite”. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Vol. 213. 10.1145. Citeseer. 2006, pp. 1188455–1188677.

[Mar19] Margaritov, A. et al. “Prefetched Address Translation”. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 1023–1036.

[McC91] McCalpin, J. D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 1991-2007.

[McC95] McCalpin, J. D. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995), pp. 19–25.

[Mes15] Meswani, M. R. et al. “Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories”. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 126–136.

[Mow92] Mowry, T. C. et al. “Design and evaluation of a compiler algorithm for prefetching”. *ACM Sigplan Notices* **27**.9 (1992), pp. 62–73.

[MS96] McVoy, L. W., Staelin, C., et al. “lmbench: Portable tools for performance analysis.” *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.

[Mut13] Mutlu, O. “Memory scaling: A systems architecture perspective”. *2013 5th IEEE International Memory Workshop*. IEEE. 2013, pp. 21–25.

[Nai15] Nair, R. “Evolution of memory architecture”. *Proceedings of the IEEE* **103**.8 (2015), pp. 1331–1345.

[OB17] Oden, L. & Balaji, P. “Hexe: A toolkit for heterogeneous memory management”. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2017, pp. 656–663.

[Obj] *PCM become a reality*. `http://www.objective-analysis.com/uploads/2009-08-03_Objective_Analysis_PCM_White_Paper.pdf`.

[Ols19] Olsen, M. B. et al. “Portable Application Guidance for Complex Memory Systems”. *Proceedings of the fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE. 2019, pp. 156–166.

[Ope] *OpenMP 3.1 specification - July 2011*. URL: `https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf`.

[OâĂŹ14] OâĂŹConnor, M. “Highlights of the high-bandwidth memory (hbm) standard”. *Memory Forum Workshop*. 2014.

[Pat09] Patil, O. et al. “Persistent Regions that Survive NVM Media Failure” (2009).

[Pat17] Patil, O. et al. “Exploring use-cases for non-volatile memories in support of hpc resilience”. *SC Poster Session* (2017).

[Pat19] Patil, O. et al. “Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using Intel Optane DC Persistent Memory Modules”. *Proceedings of the fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE. 2019, pp. 288–303.

[Pat20] Patil, O. et al. “Symbiotic HW Cache and SW DTLB Prefetching for DRAM/NVM Hybrid Memory”. *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2020, pp. 1–8.

[Pen19] Peng, I. B. et al. “System evaluation of the Intel Optane Byte-addressable NVM”. *Proceedings of the fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE. 2019, pp. 304–315.

[Per16] Perarnau, S. et al. “Exploring data migration for future deep-memory many-core systems”. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 289–297.

[Pet19] Peterka, T et al. “4.4 Exascale Systems”. *In Situ Visualization for Computational Science* (2019), p. 18.

[Psa19] Psaropoulos, G. et al. “Bridging the latency gap between NVM and DRAM for latency-bound operations”. *Proceedings of the 15th International Workshop on Data Management on New Hardware*. CONF. ACM. 2019.

[Qiu14] Qiu, J. et al. “Towards HPC-ABDS: an initial high-performance big data stack”. *Building Robust Big Data Ecosystem ISO/IEC JTC* **1** (2014), pp. 18–21.

[Rao14] Raoux, S. et al. “Phase change materials and phase change memory”. *MRS Bulletin* **39**.8 (2014), 703âĂŞ710.

[Ren19] Renen, A. van et al. “Persistent Memory I/O Primitives”. *Proceedings of the 15th International Workshop on Data Management on New Hardware*. ACM. 2019, p. 12.

[Rog09] Rogers, B. M. et al. “Scaling the bandwidth wall: challenges in and avenues for CMP scaling”. *ACM SIGARCH Computer Architecture News* **37**.3 (2009), pp. 371–382.

[RR14] Rodríguez-Rodríguez, R. et al. “Write-aware replacement policies for pcm-based systems”. *The Computer Journal* **58**.9 (2014), pp. 2000–2025.

[SB18] Sánchez Barrera, I. et al. “Reducing data movement on large shared memory systems by exploiting computation dependencies”. *Proceedings of the 2018 International Conference on Supercomputing*. 2018, pp. 207–217.

[Sch19] Schwarz, M. et al. “ZombieLoad: Cross-privilege-boundary data sampling”. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 753–768.

[Shu19] Shull, T. et al. “Designing a User-Friendly Java NVM Framework” (2019).

[Sna] *SNAP: SN (Discrete Ordinates) Application Proxy*. URL: `https://github.com/lanl/SNAP`.

[Sod16] Sodani, A. et al. “Knights landing: Second-generation intel xeon phi product”. *Ieee micro* **36**.2 (2016), pp. 34–46.

[Sre19] Sreenivasan, V. et al. “A framework for enabling OpenMP autotuning”. *International Workshop on OpenMP*. Springer. 2019, pp. 50–60.

[Sta02] Stallman, R. M. “GNU compiler collection internals”. *Free Software Foundation* (2002).

[Sum] *SUMMIT*. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`.

[Swa14] Swamy, B. N. et al. “Hardware/software helper thread prefetching on heterogeneous many cores”. *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE. 2014, pp. 214–221.

[Teh20] Tehrany. “Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack”. 2020.

[Tit] *TITAN*. (2019). URL: https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

[Topa] *TOP500 List - Nov 2019*. https://www.top500.org/lists/2019/11/.

[Topb] *TOP500 List - November 2018*. (2018). URL: https://www.top500.org/list/2018/11/.

[Tra11] Trahay, F. et al. "EZTrace: a generic framework for performance analysis". *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2011, pp. 618–619.

[Tre10] Treibig, J. et al. "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments". *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 207–216.

[VM15] Vetter, J. S. & Mittal, S. "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing". *Computing in Science & Engineering* **17**.2 (2015), pp. 73–82.

[Vol11] Volos, H. et al. "Mnemosyne: Lightweight persistent memory". *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 91–104.

[Wan12] Wang, C. et al. "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines". *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE. 2012, pp. 957–968.

[Wil18] *SICM: Simple Interface Complex Memory*. ACM, 2018.

[Wu17] Wu, K. et al. "Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads". *CoRR* **abs/1708.02199** (2017). arXiv: 1708.02199.

[Yan02] Yang, U. M. et al. "BoomerAMG: a parallel algebraic multigrid solver and preconditioner". *Applied Numerical Mathematics* **41**.1 (2002), pp. 155–177.

[Yan15] Yang, J. et al. "NV-Tree: reducing consistency cost for NVM-based single level systems". *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 2015, pp. 167–181.

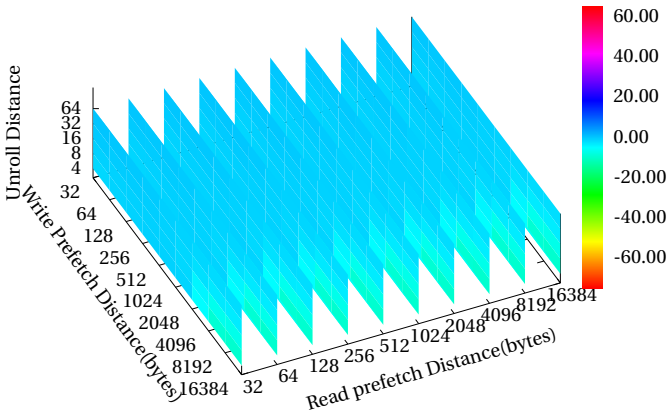
[Zha05] Zhao, Q. et al. "Dynamic memory optimization using pool allocation and prefetching". *ACM SIGARCH Computer Architecture News* **33**.5 (2005), pp. 27–32.

[Zho19] Zhou, F. et al. "VAIL: A Victim-Aware Cache Policy to Improve NVM Lifetime for Hybrid Memory System". *Parallel Computing* (2019).

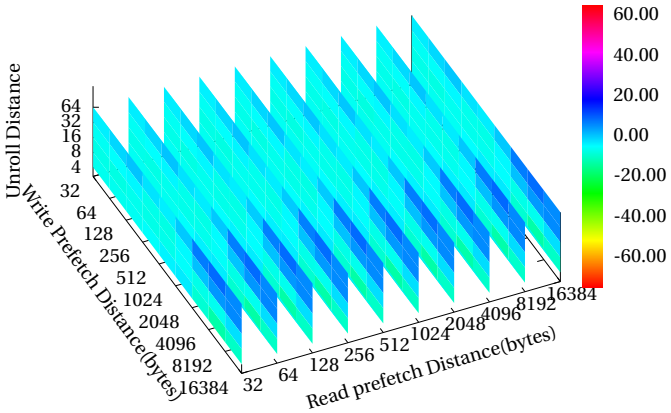
[Zia10] Ziakas, D. et al. "Intel® quickpath interconnect architectural features supporting scalable system architectures". *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE. 2010, pp. 1–6.

APPENDICES

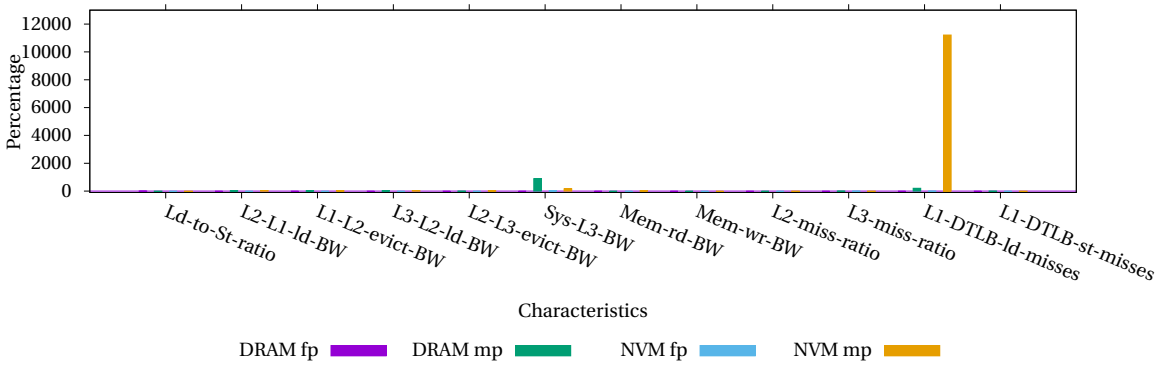
.1 Additional graphs from Chapter 4



(a) DRAM sp % change

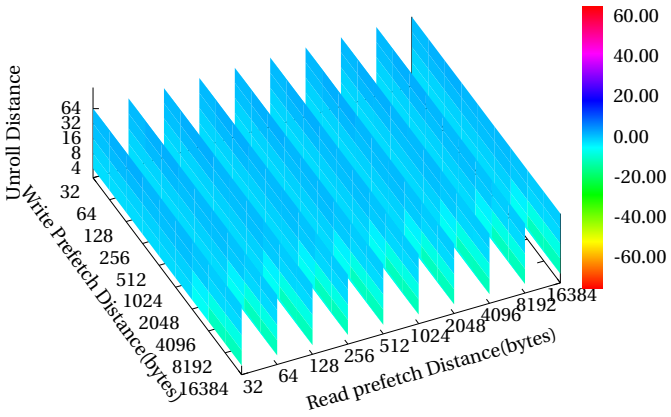


(b) NVM sp % change

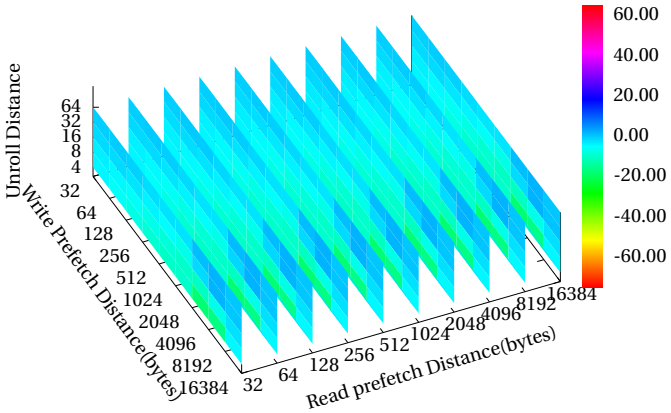


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

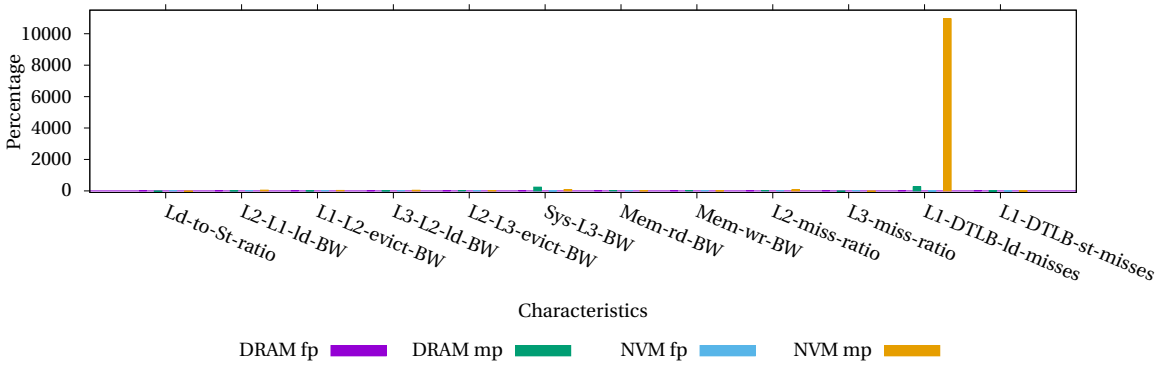
Figure 1 Performance of Write-only (1D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) DRAM sp % change

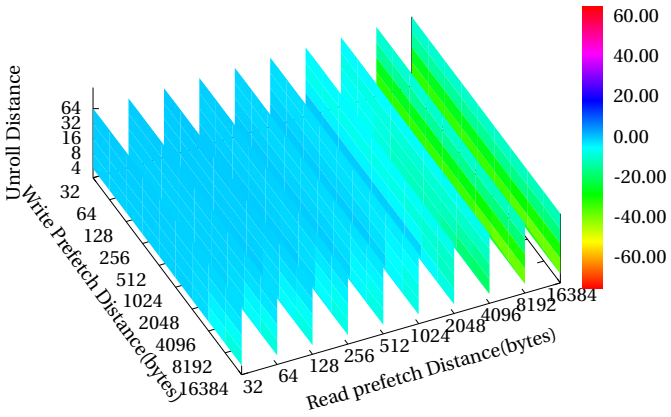


(b) NVM sp % change

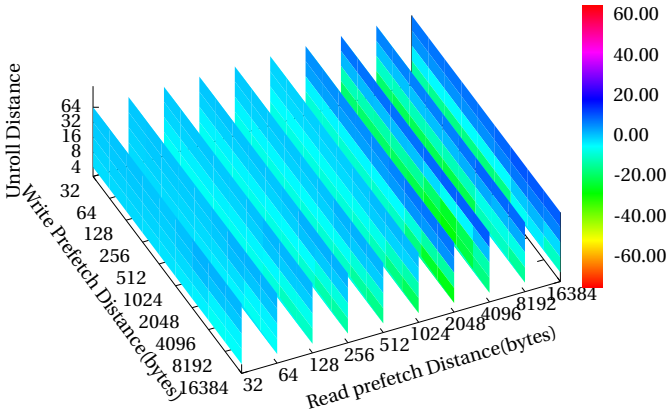


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

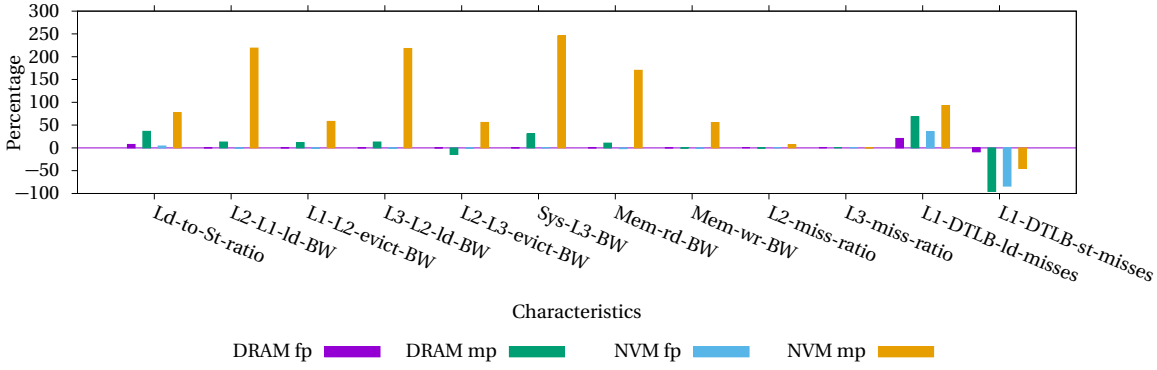
Figure 2 Performance of Write-only (1D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline



(a) DRAM sp % change

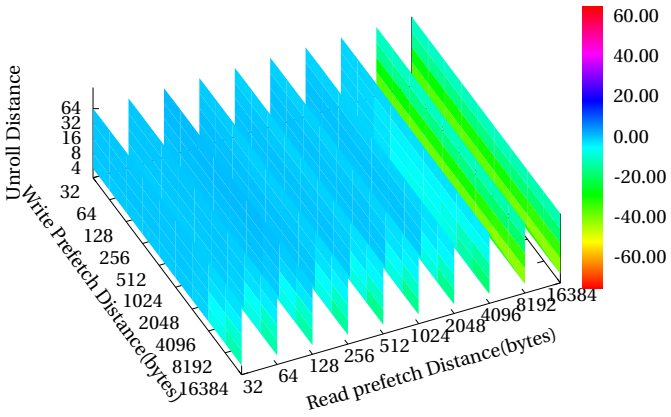


(b) NVM sp % change

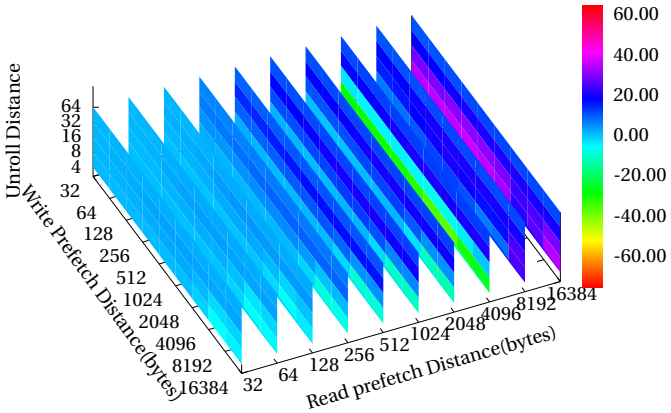


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

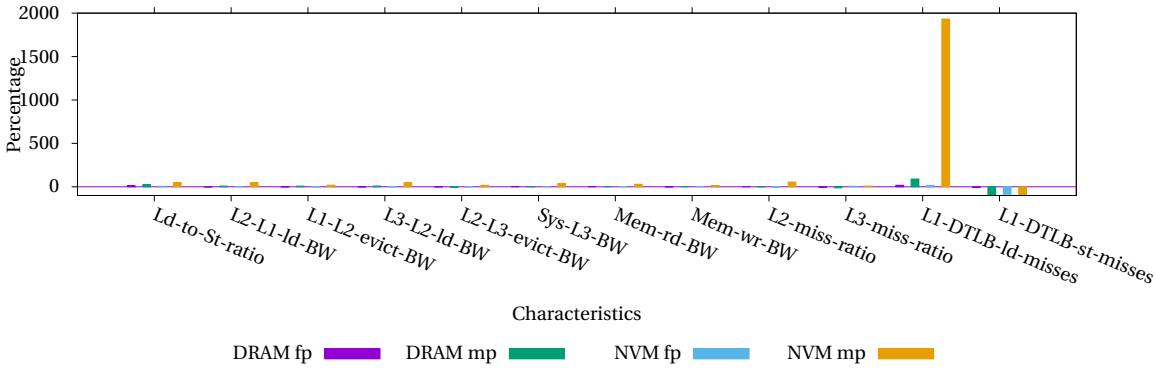
Figure 3 Performance of Single write four read (1D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) DRAM sp % change

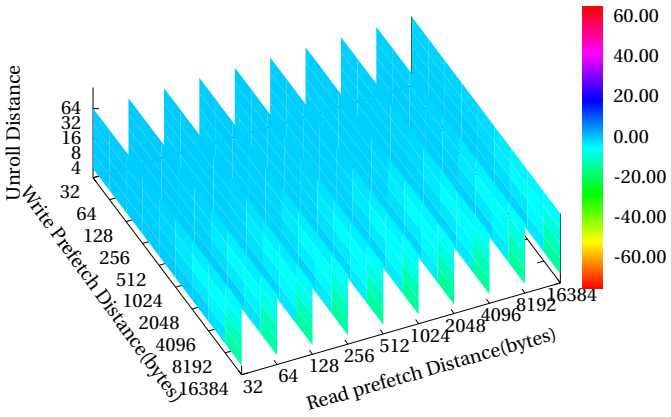


(b) NVM sp % change

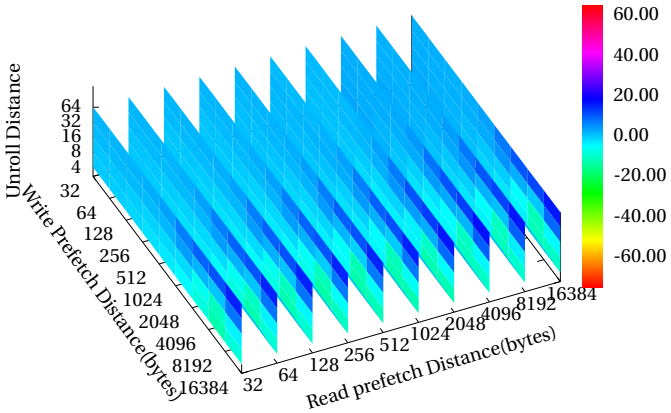


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

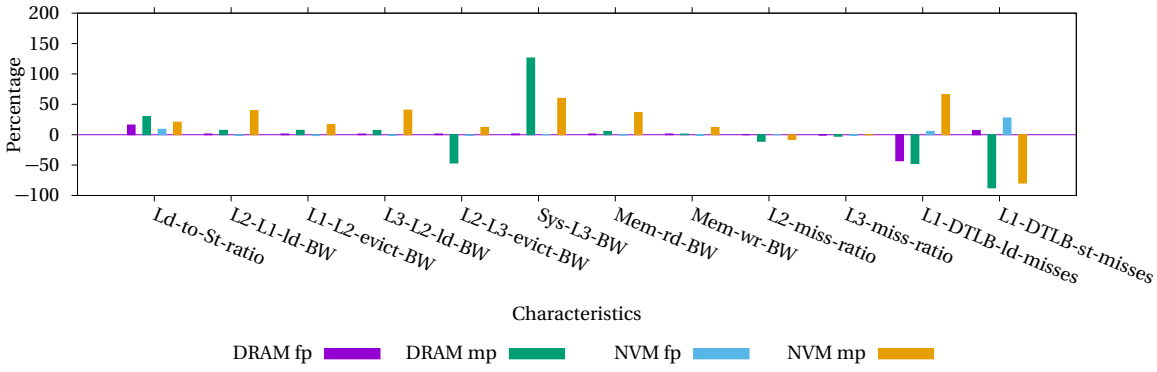
Figure 4 Performance of Single Write four read (1D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline



(a) DRAM sp % change

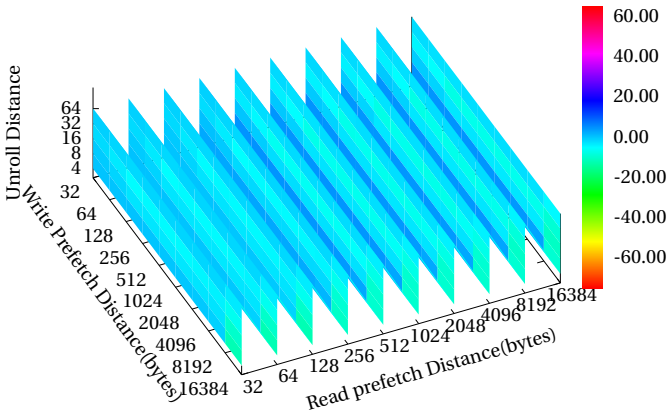


(b) NVM sp % change

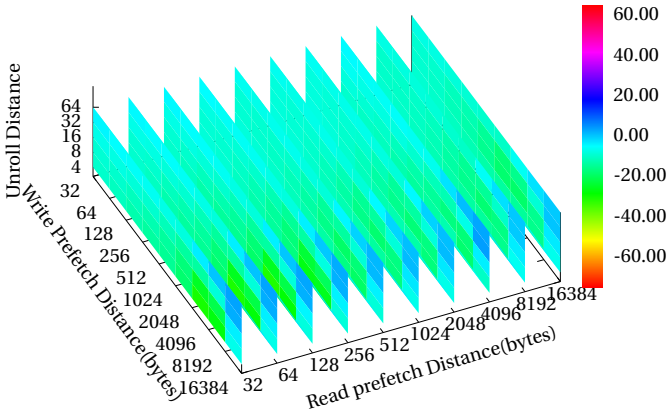


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

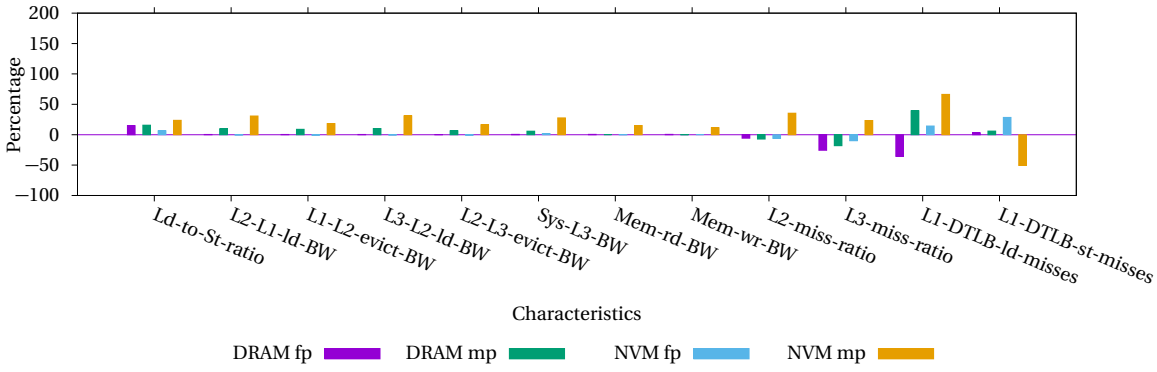
Figure 5 Performance of 3-pt stencil (2D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) DRAM sp % change

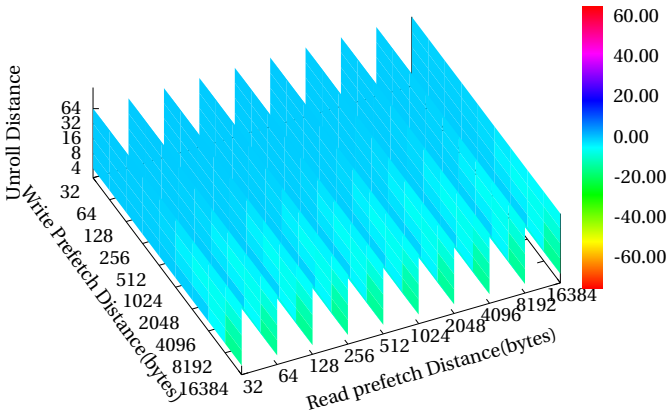


(b) NVM sp % change

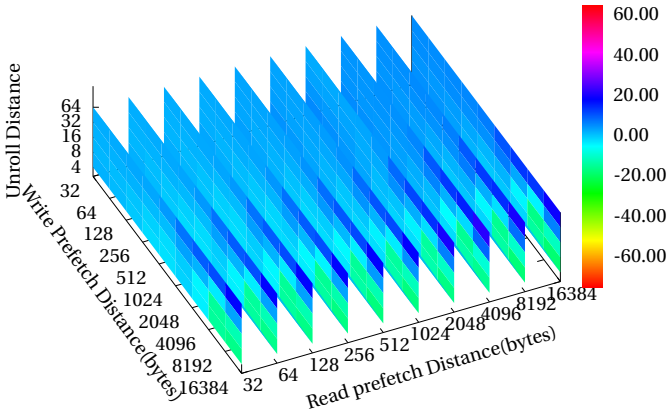


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

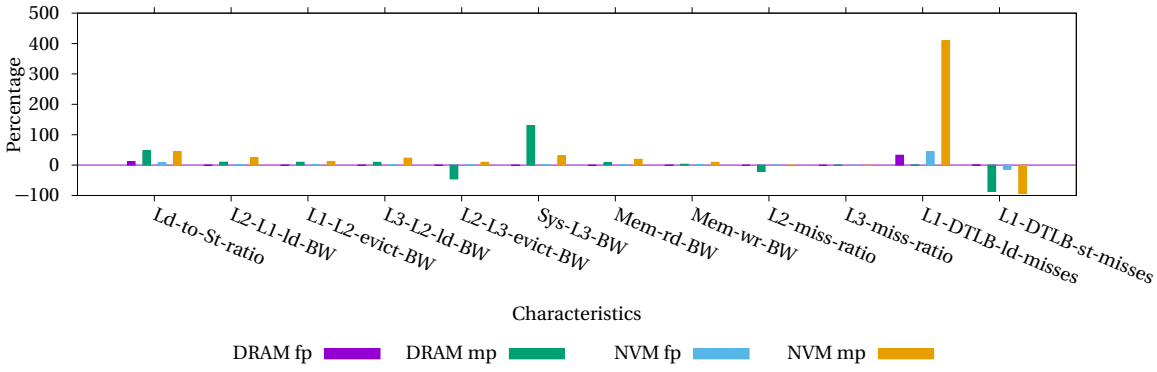
Figure 6 Performance of 3-pt stencil (2D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline



(a) DRAM sp % change

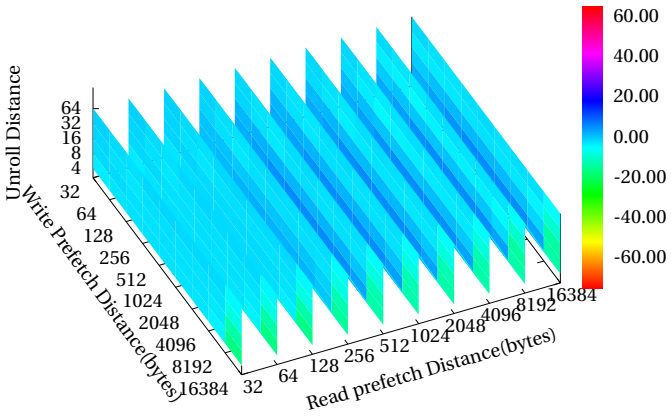


(b) NVM sp % change

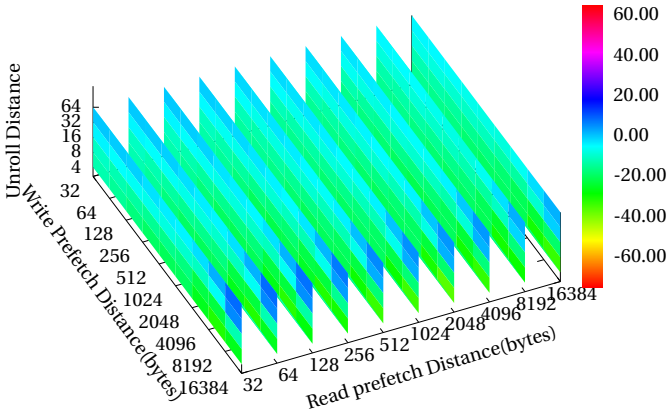


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

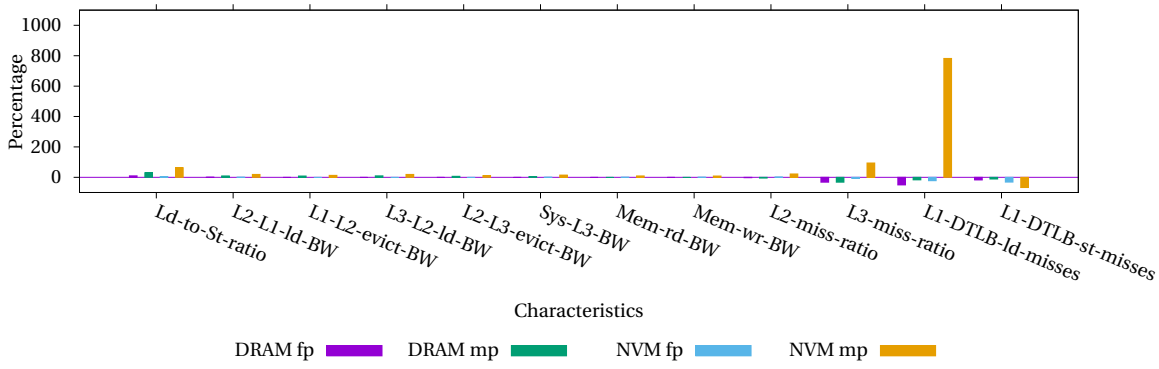
Figure 7 Performance of 5-pt stencil (2D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) DRAM sp % change

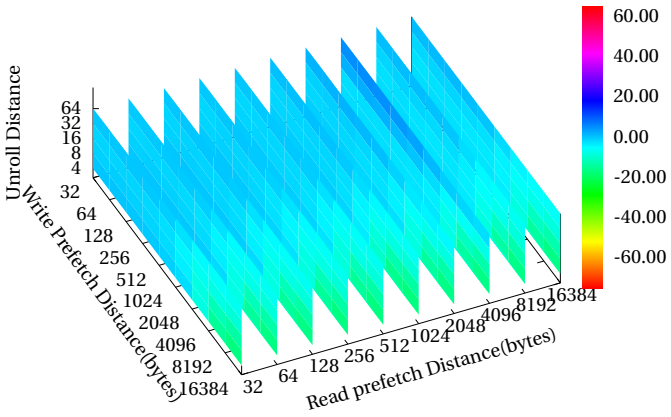


(b) NVM sp % change

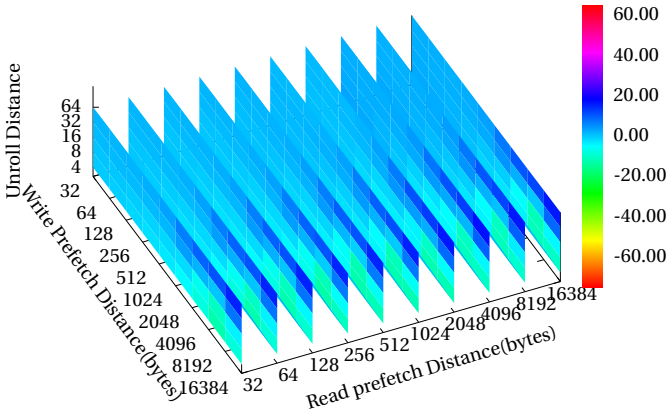


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

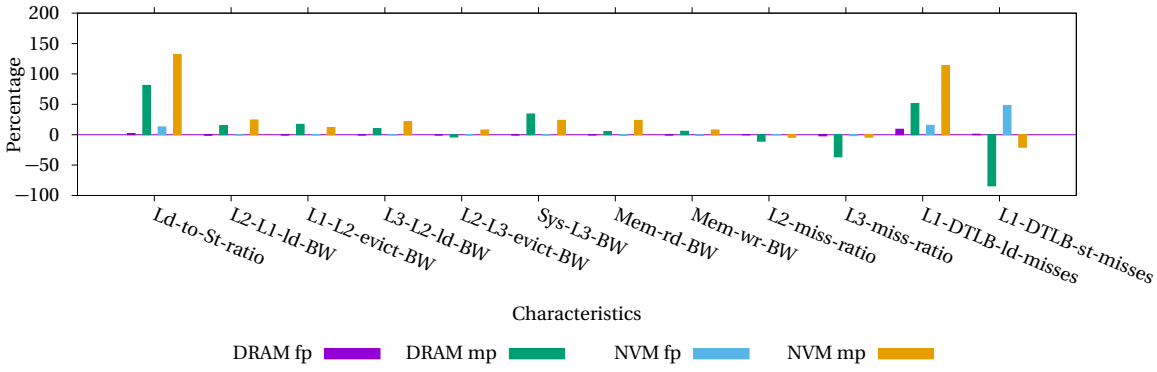
Figure 8 Performance of 5-pt stencil (2D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline



(a) DRAM sp % change

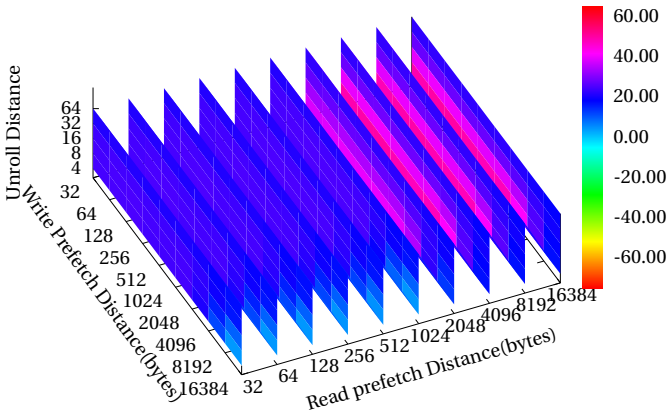


(b) NVM sp % change

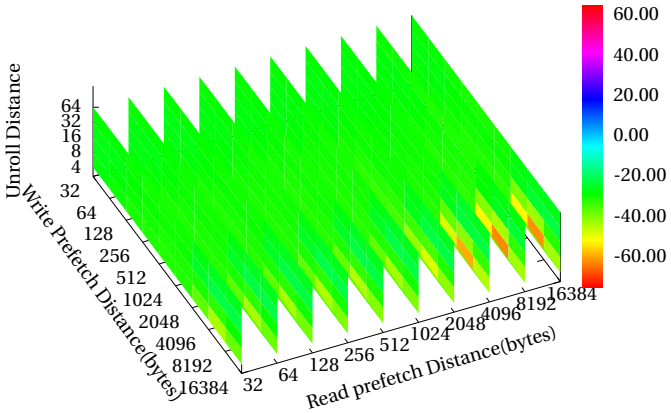


(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

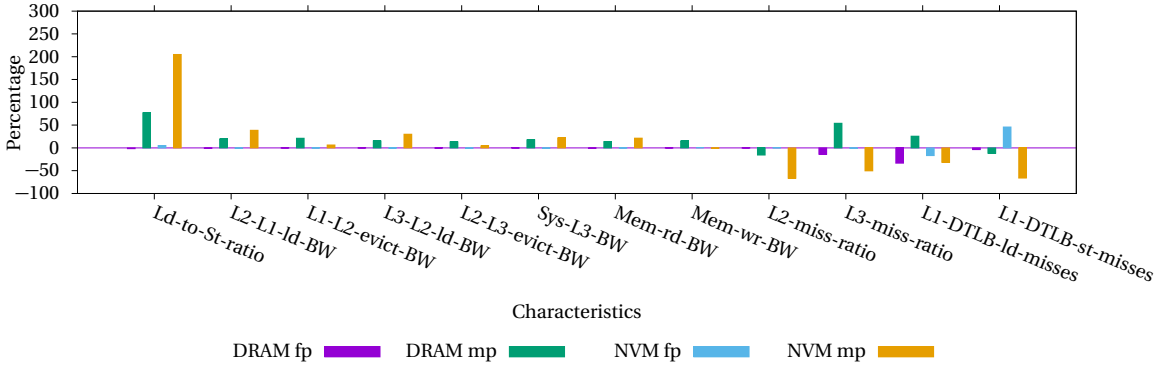
Figure 9 Performance of 27-pt stencil (3D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) DRAM sp % change



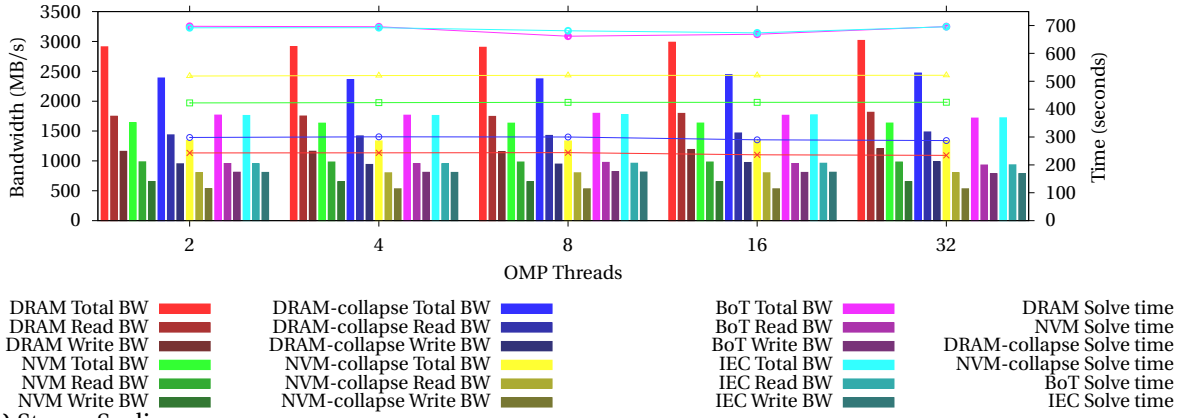
(b) NVM sp % change



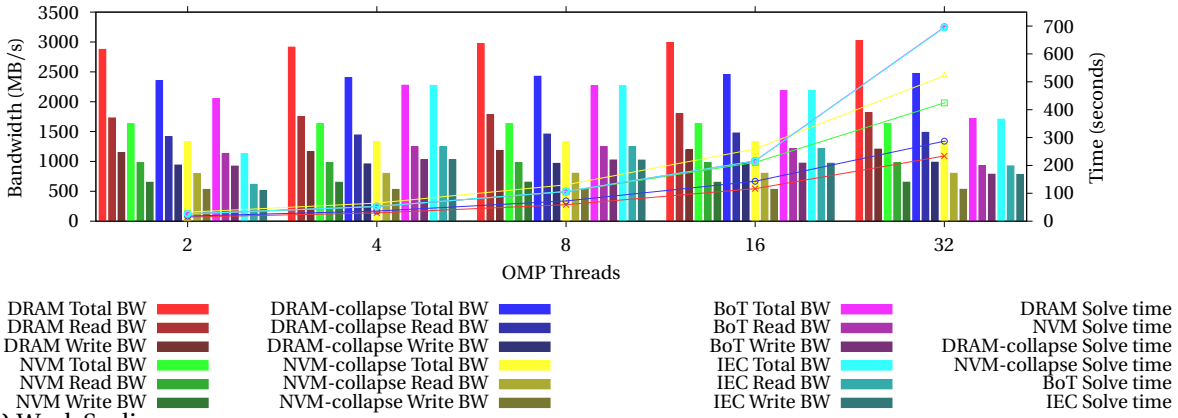
(c) Cache and Memory performance metrics comparison between compiler (cp) and symbiotic prefetching (sp)

Figure 10 Performance of 27-pt stencil (2D) stream with L2 Temporal prefetching relative to no HW prefetching as a baseline

.2 Additional graphs from Chapter 5

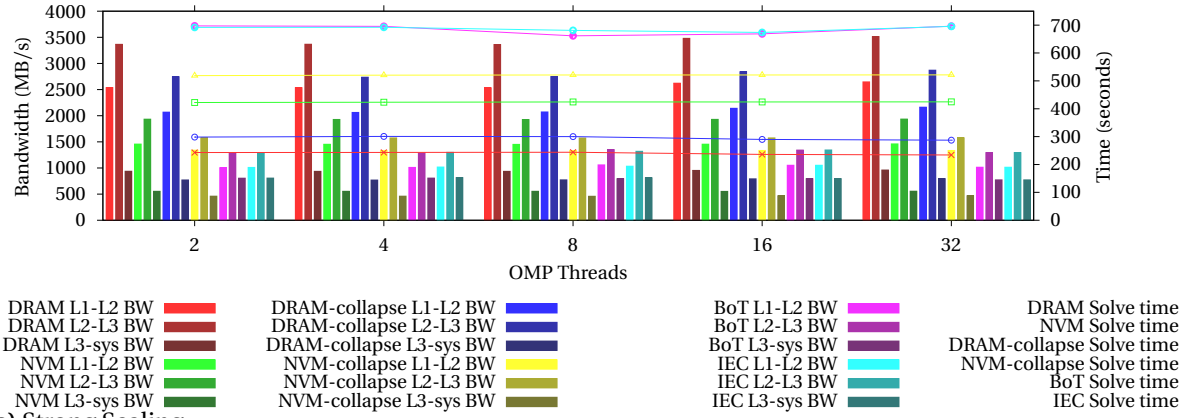


(a) Strong Scaling

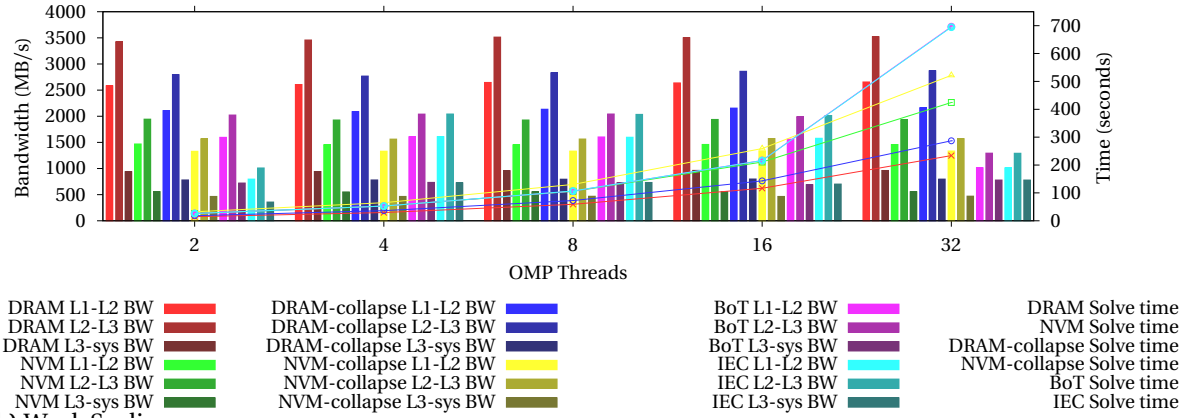


(b) Weak Scaling

Figure 11 3-pt stencil scheduling comparison Time and Bandwidth

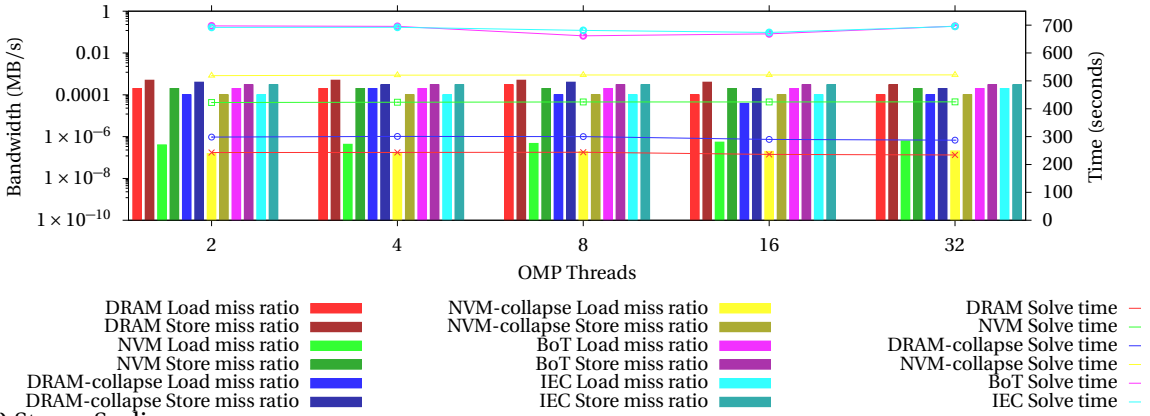


(a) Strong Scaling

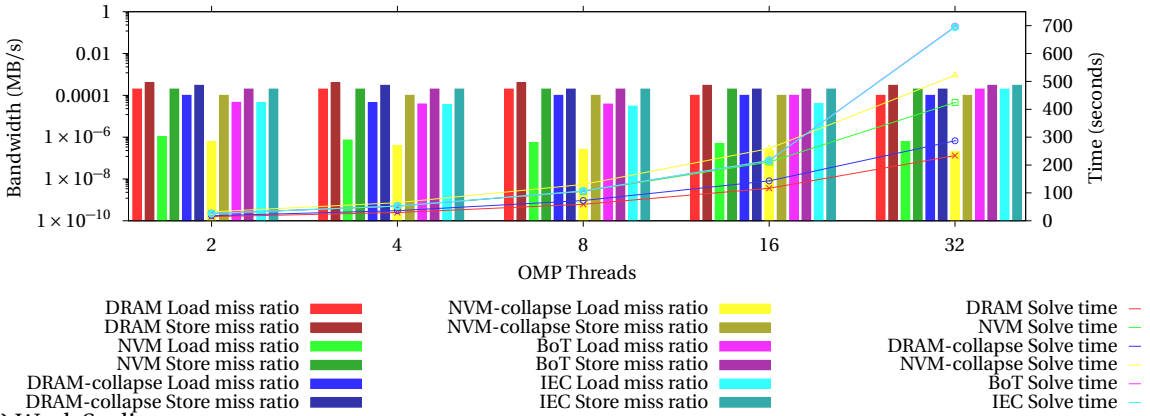


(b) Weak Scaling

Figure 12 3-pt stencil scheduling comparison Time and Cache Bandwidth

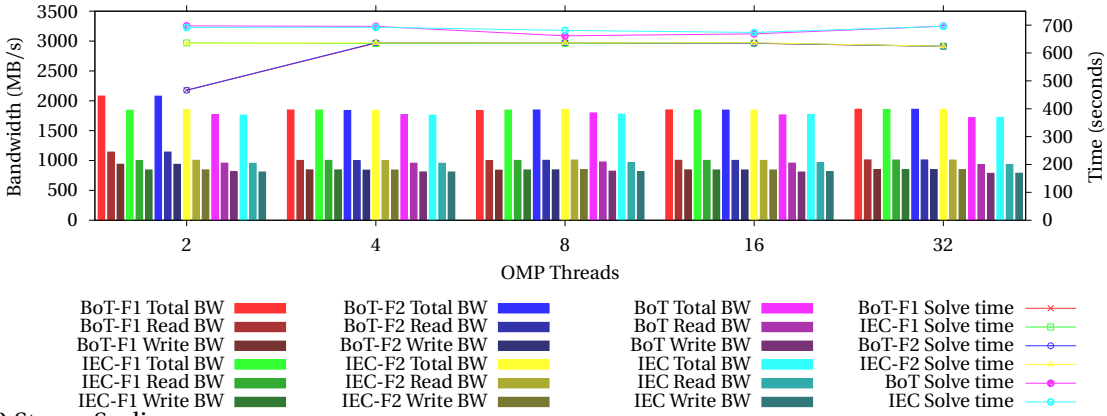


(a) Strong Scaling

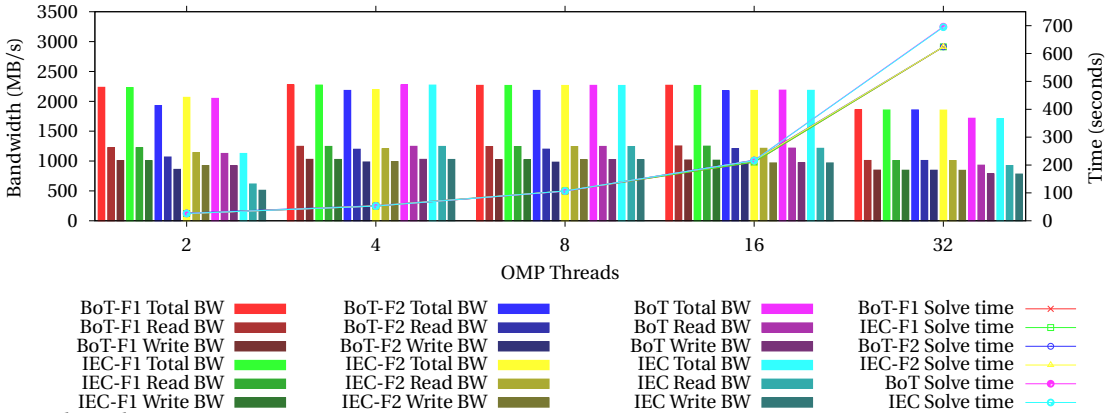


(b) Weak Scaling

Figure 13 3-pt stencil scheduling comparison Time and DTLB Load Store miss ratios

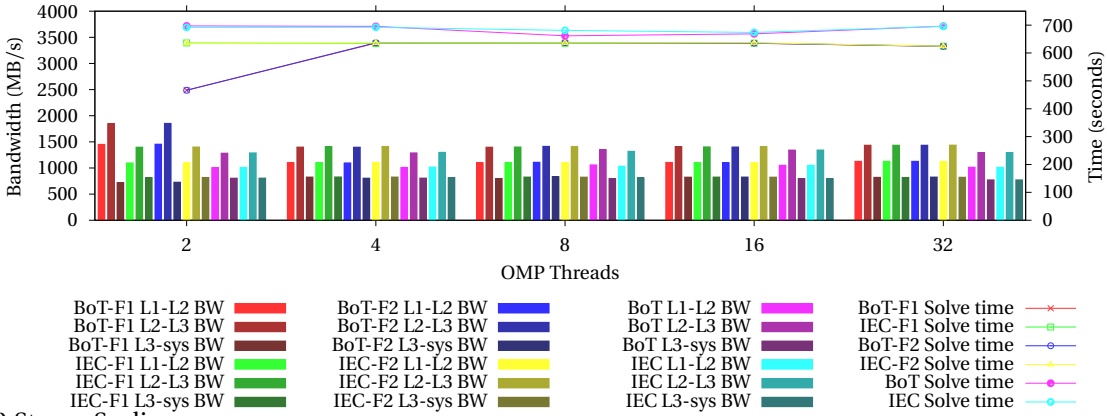


(a) Strong Scaling

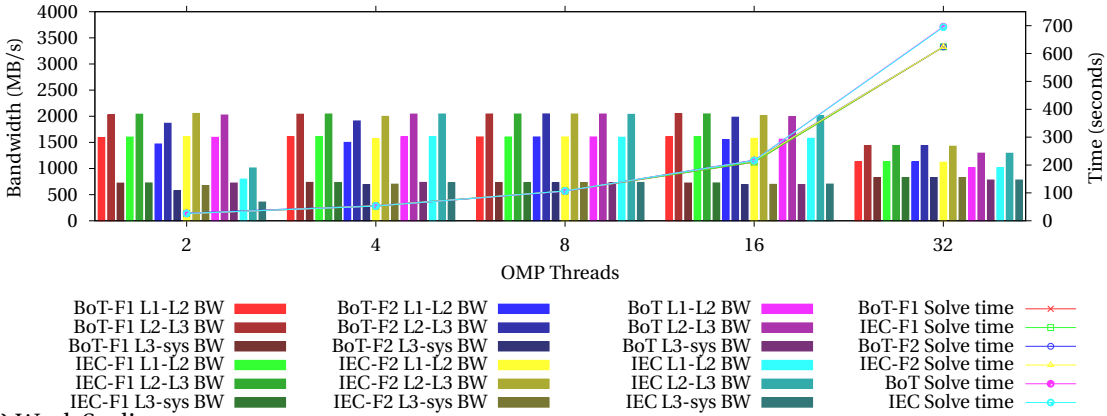


(b) Weak Scaling

Figure 14 3-pt stencil scheduling comparison Time and Bandwidth

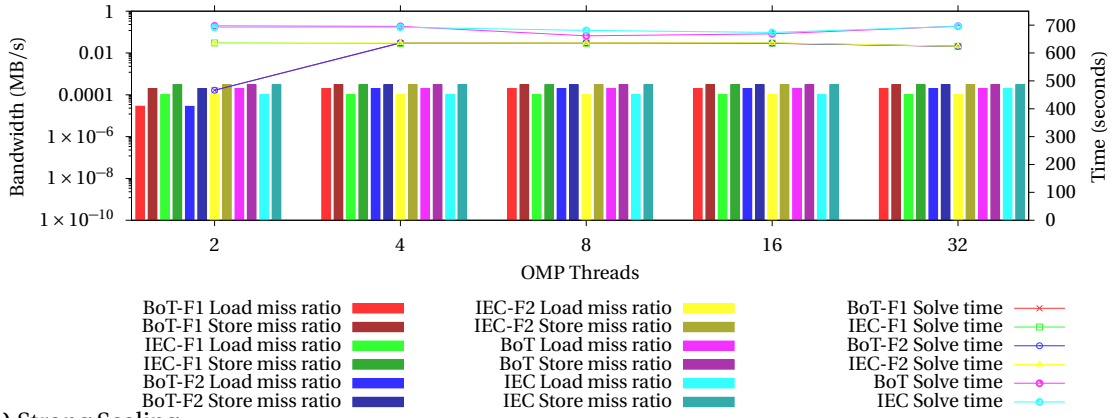


(a) Strong Scaling

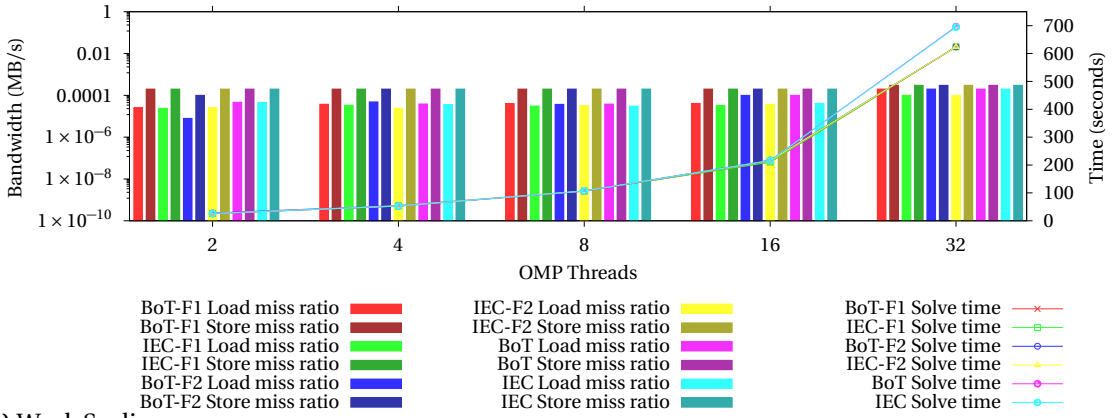


(b) Weak Scaling

Figure 15 3-pt stencil scheduling comparison Time and Cache Bandwidth

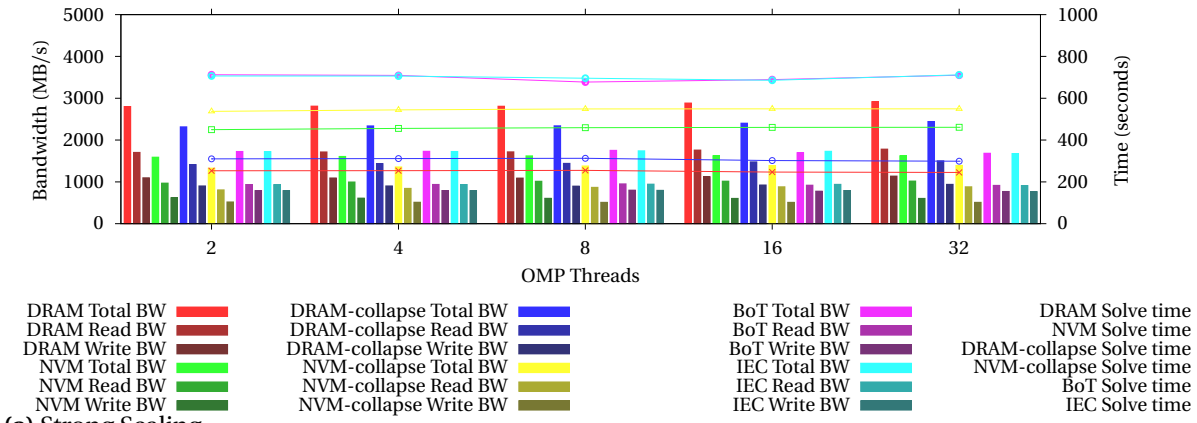


(a) Strong Scaling

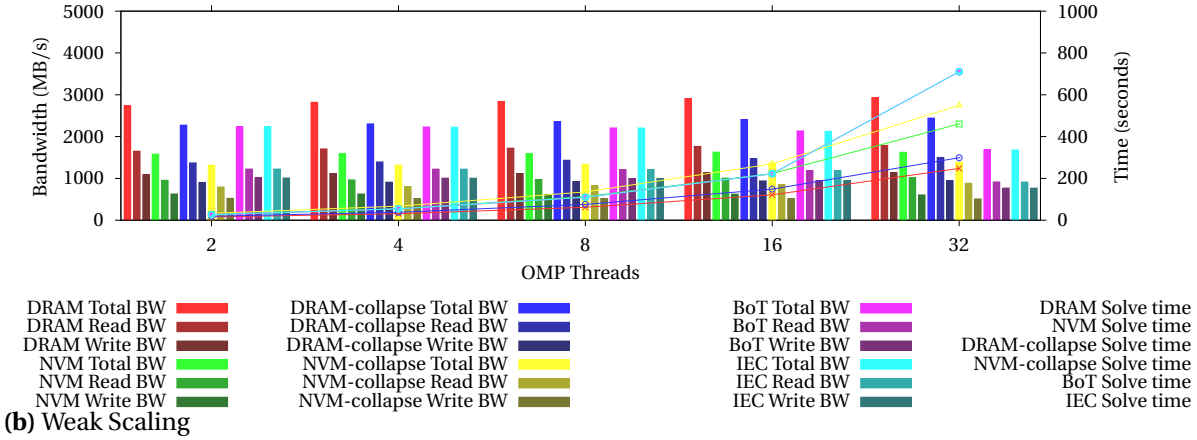


(b) Weak Scaling

Figure 16 3-pt stencil chunk size comparison Time and DTLB Load Store miss ratios

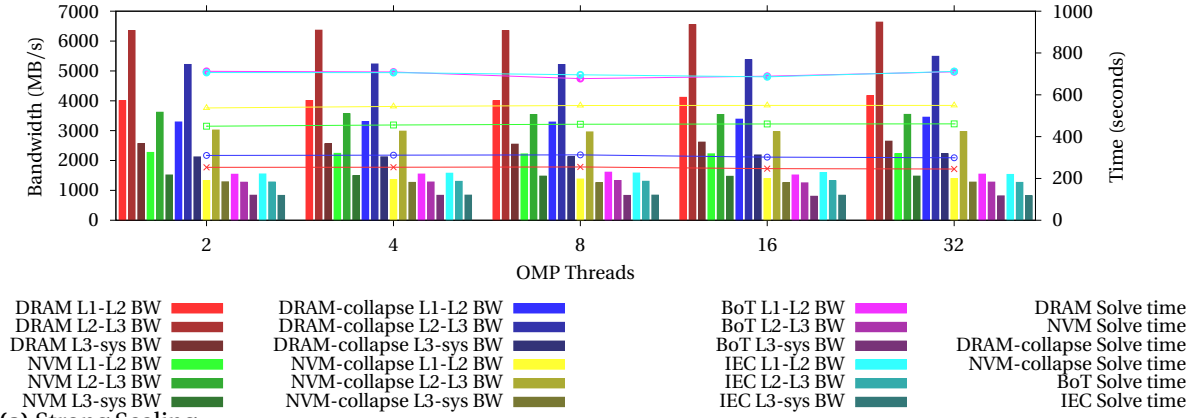


(a) Strong Scaling

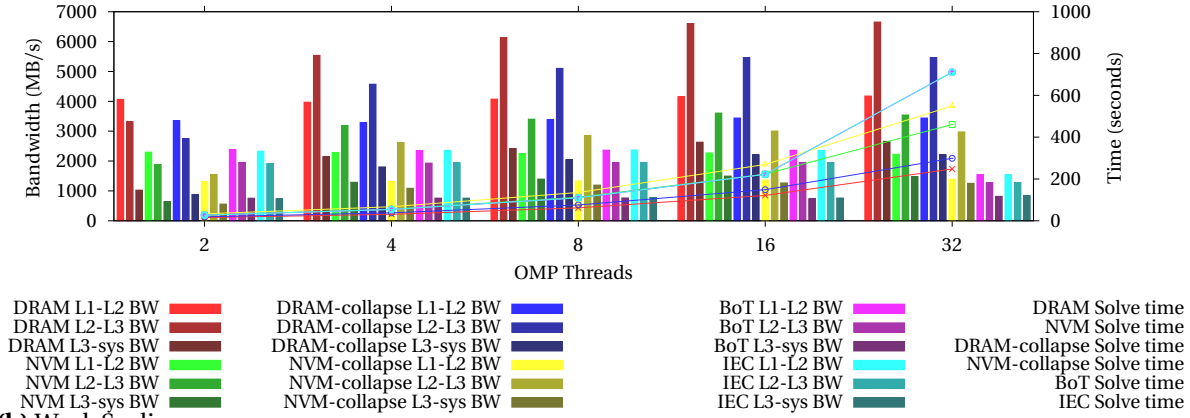


(b) Weak Scaling

Figure 17 5-pt stencil scheduling comparison Time and Bandwidth

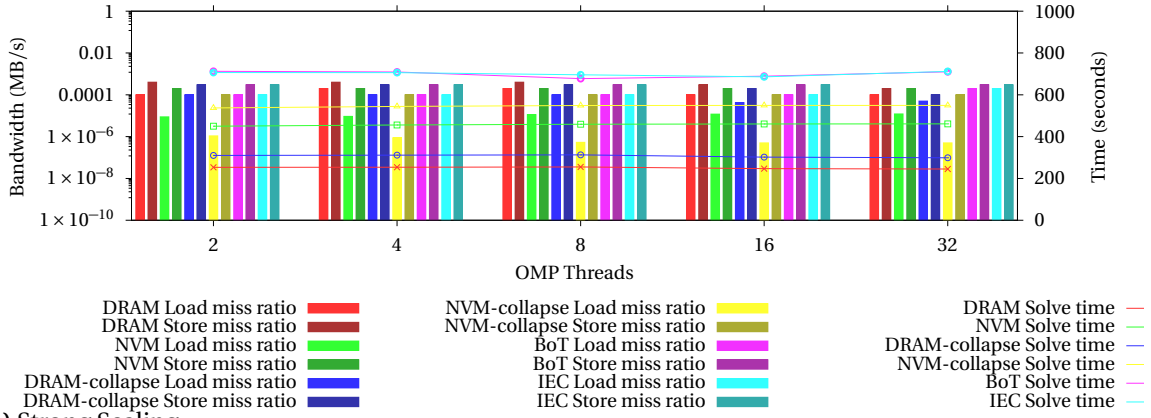


(a) Strong Scaling

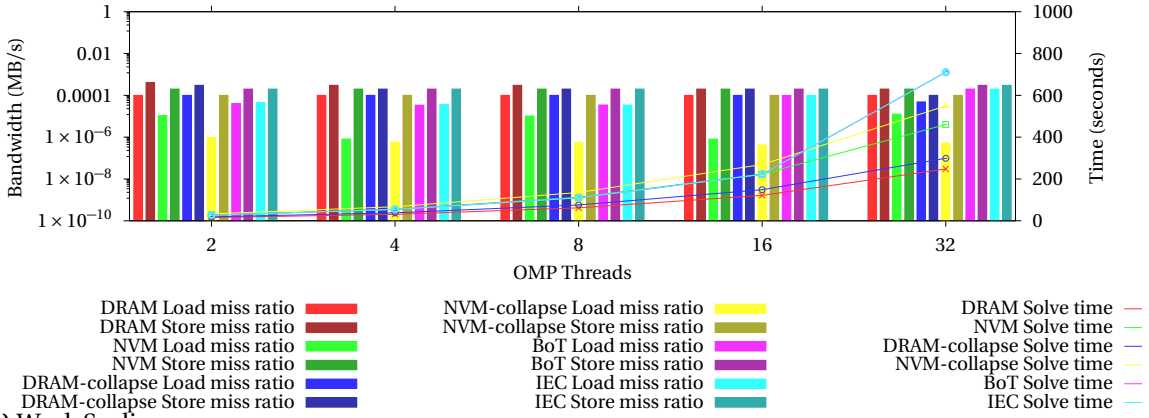


(b) Weak Scaling

Figure 18 5-pt stencil scheduling comparison Time and Cache Bandwidth

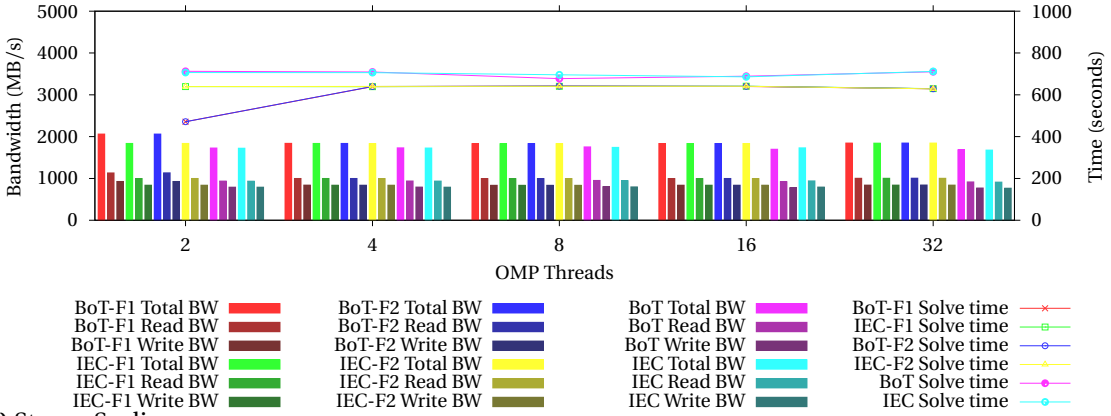


(a) Strong Scaling

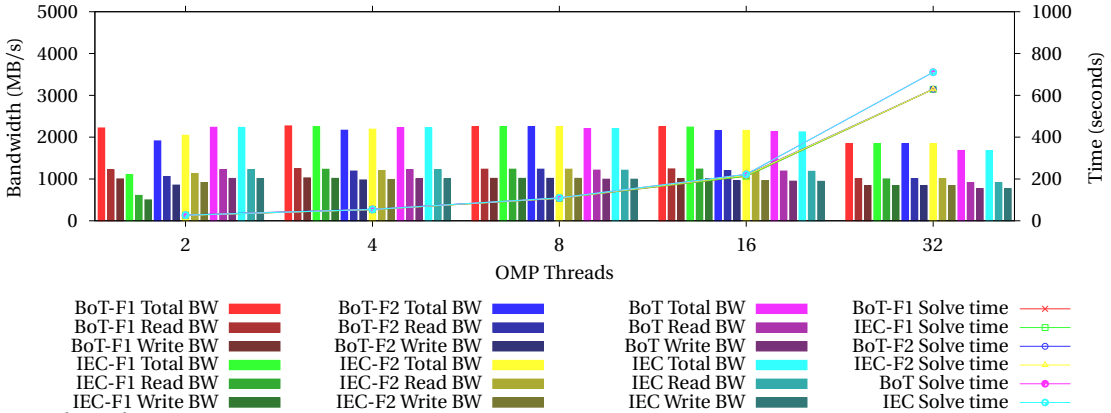


(b) Weak Scaling

Figure 19 5-pt stencil scheduling comparison Time and DTLB Load Store miss ratios

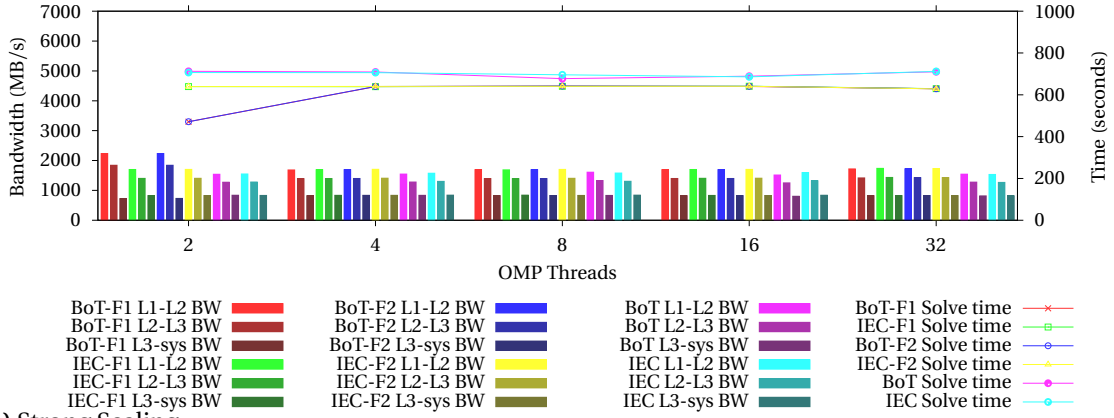


(a) Strong Scaling

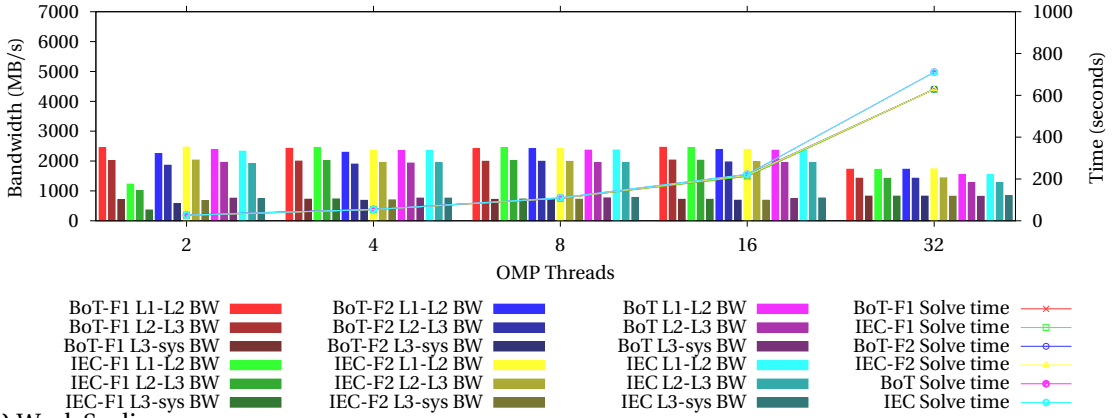


(b) Weak Scaling

Figure 20 5-pt stencil chunk size comparison Time and Bandwidth

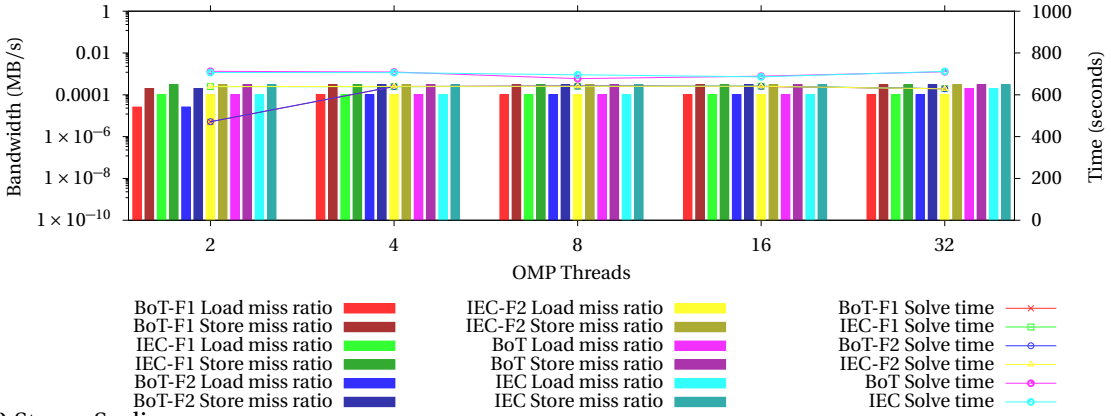


(a) Strong Scaling

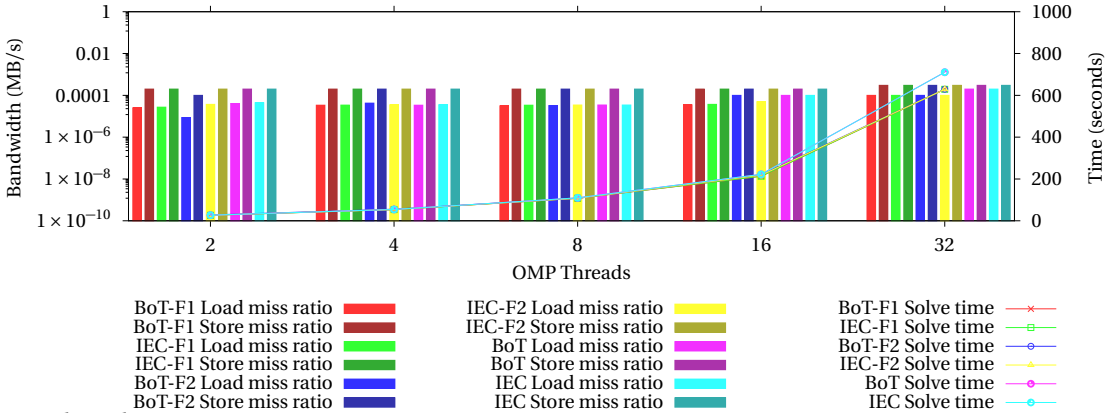


(b) Weak Scaling

Figure 21 5-pt stencil chunk size comparison Time and Cache Bandwidth

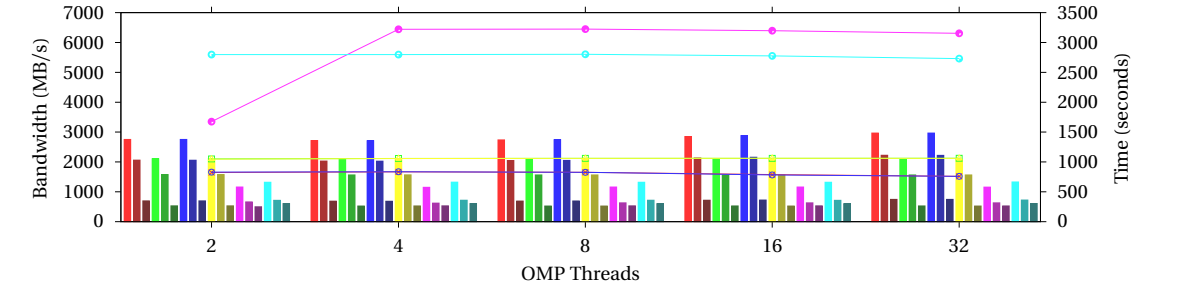


(a) Strong Scaling

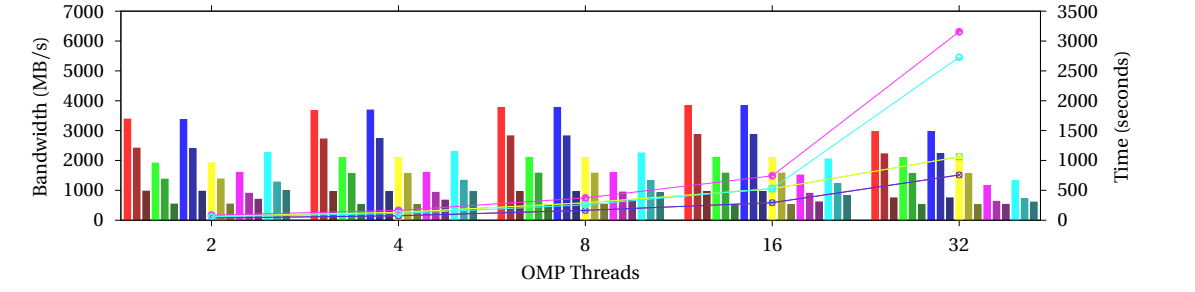


(b) Weak Scaling

Figure 22 5-pt stencil chunk size comparison Time and DTLB Load Store miss ratios

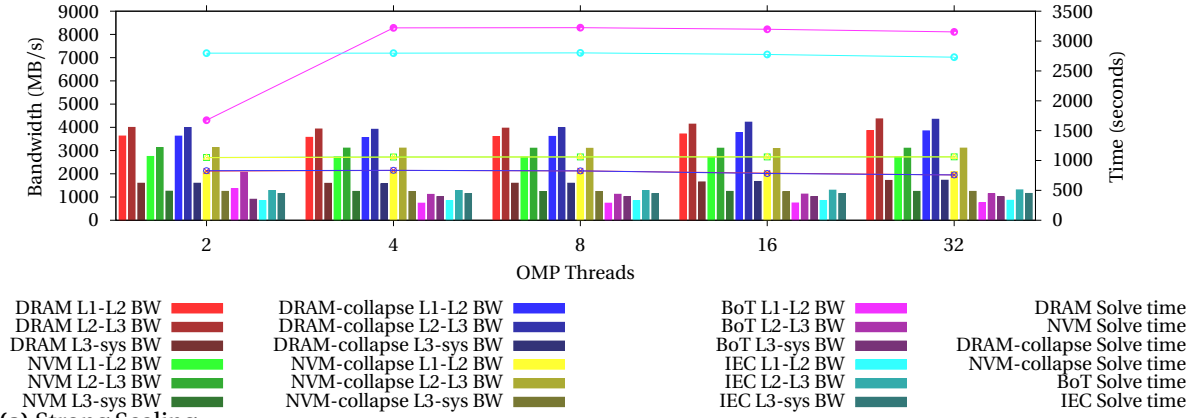


(a) Strong Scaling

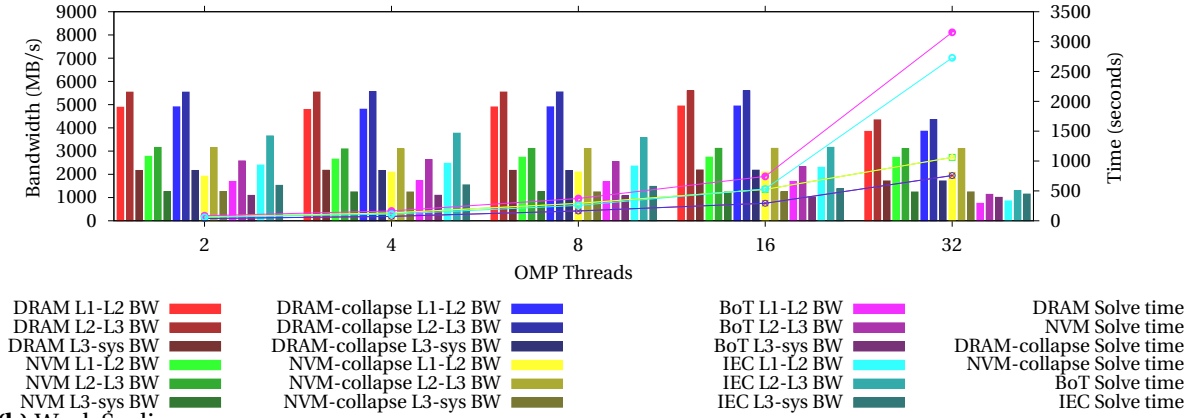


(b) Weak Scaling

Figure 23 7-pt stencil scheduling comparison Time and Bandwidth

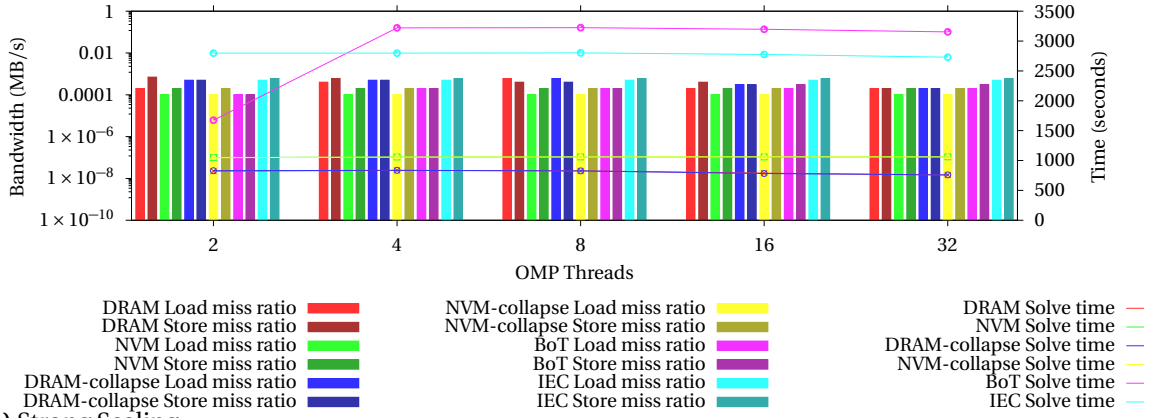


(a) Strong Scaling

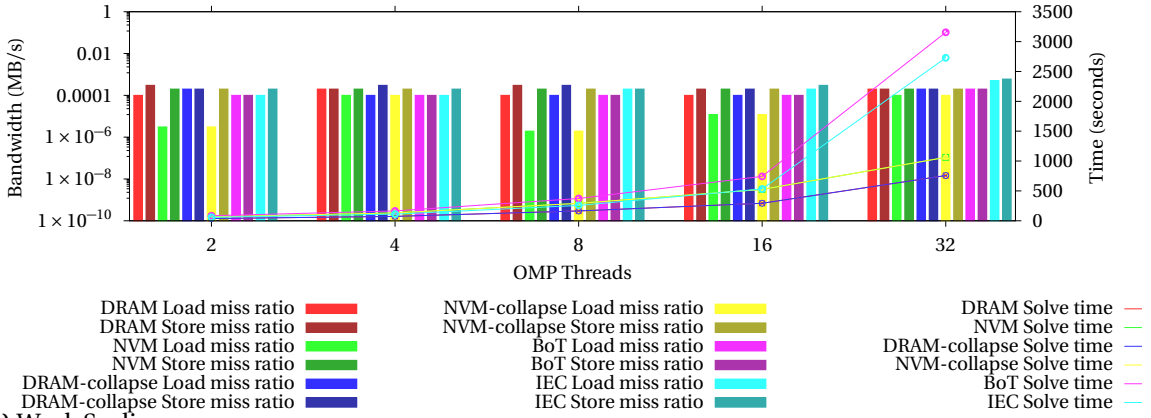


(b) Weak Scaling

Figure 24 7-pt stencil scheduling comparison Time and Cache Bandwidth

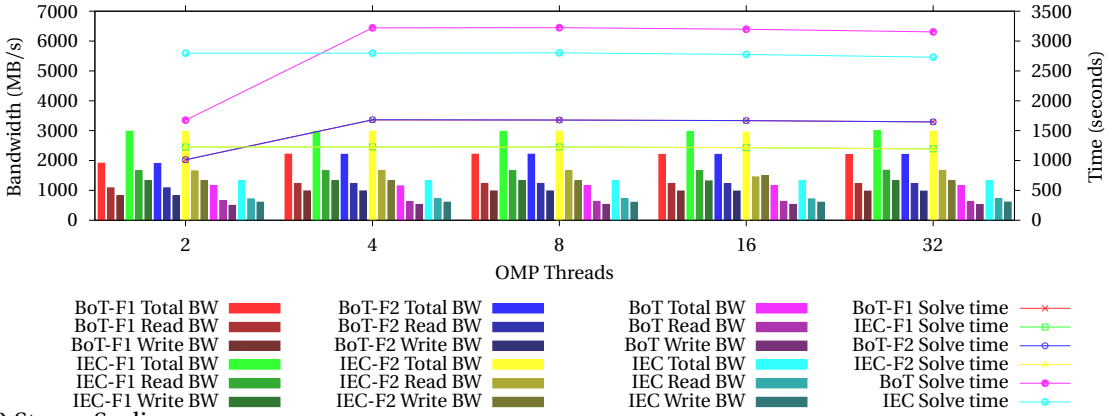


(a) Strong Scaling

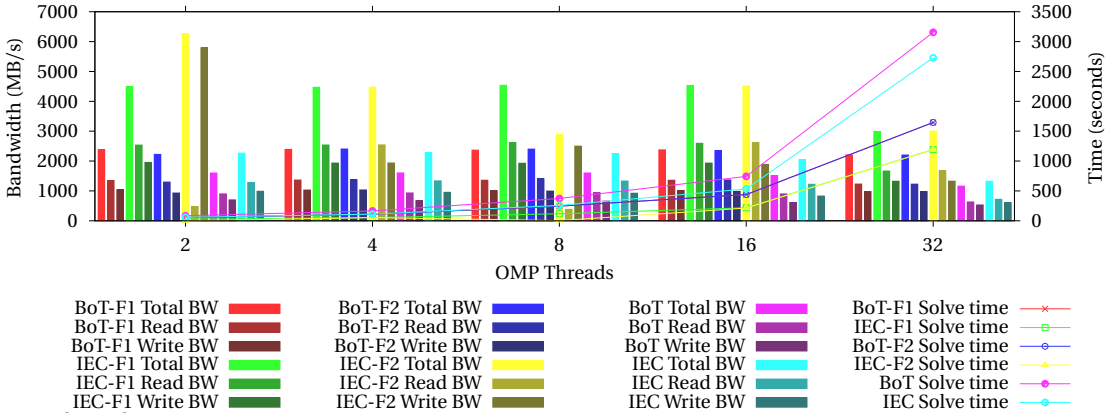


(b) Weak Scaling

Figure 25 7-pt stencil chunk size comparison Time and DTLB Load Store miss ratios

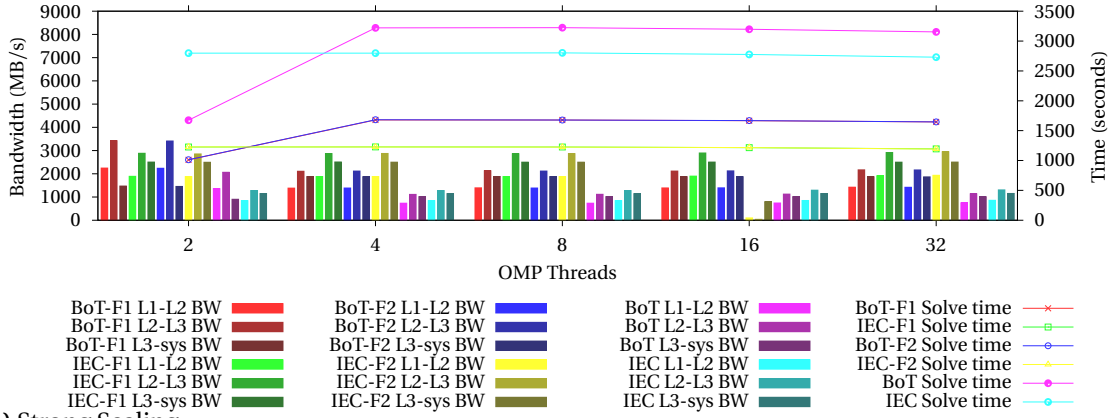


(a) Strong Scaling

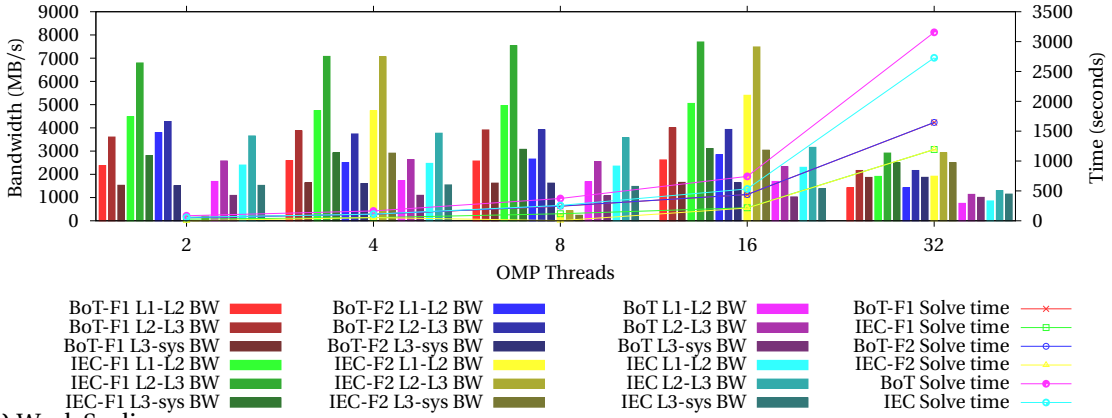


(b) Weak Scaling

Figure 26 7-pt stencil chunk size comparison Time and Bandwidth

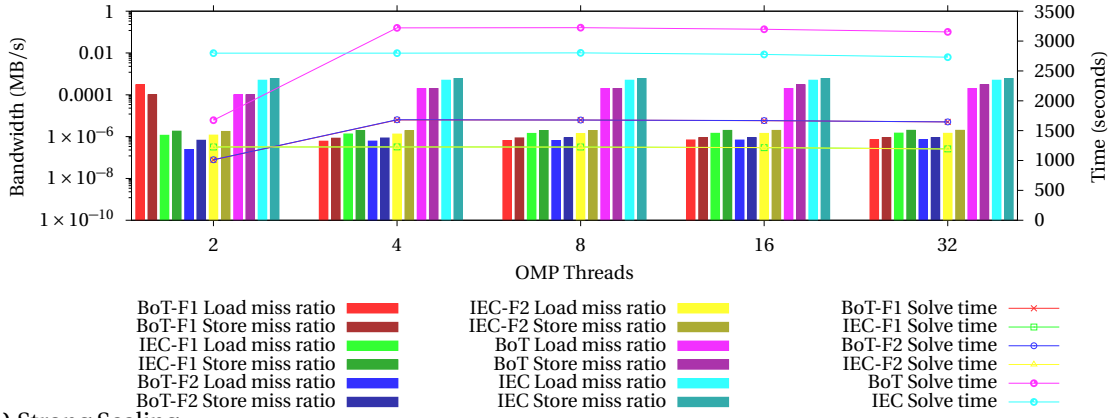


(a) Strong Scaling

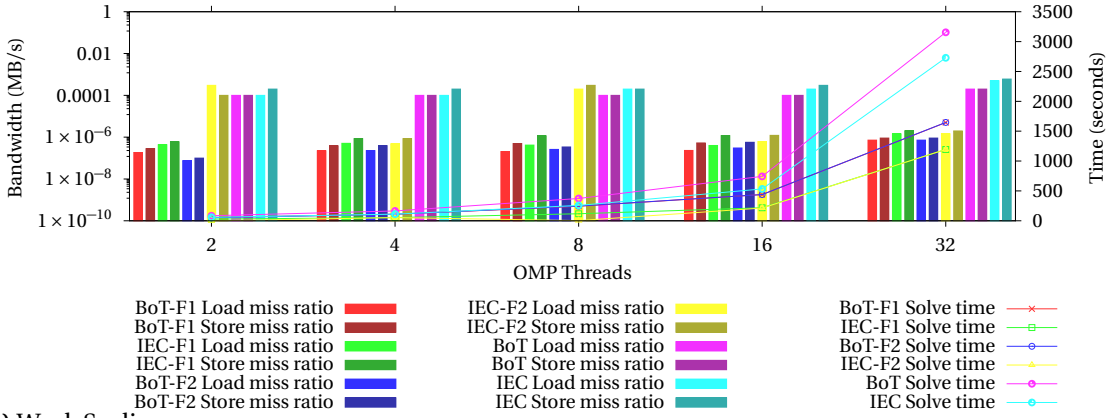


(b) Weak Scaling

Figure 27 7-pt stencil chunk size comparison Time and Cache Bandwidth

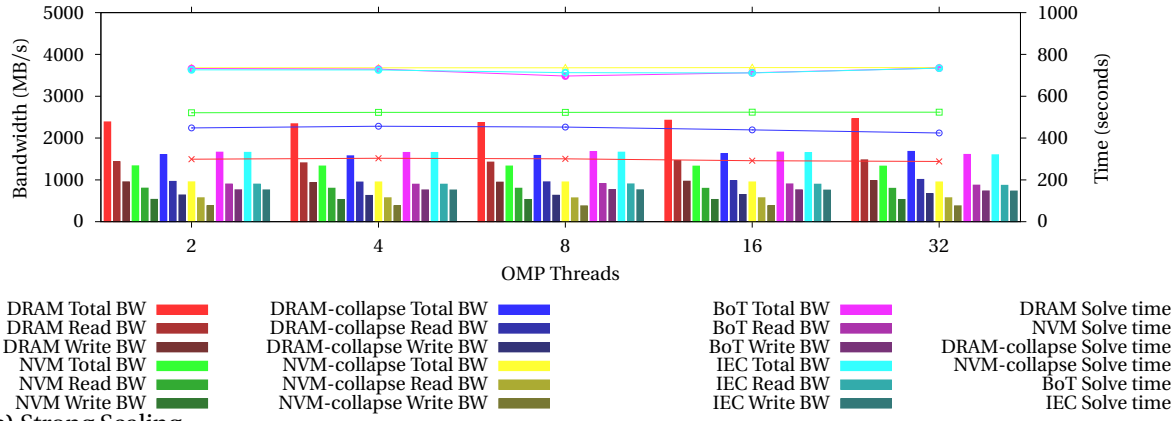


(a) Strong Scaling

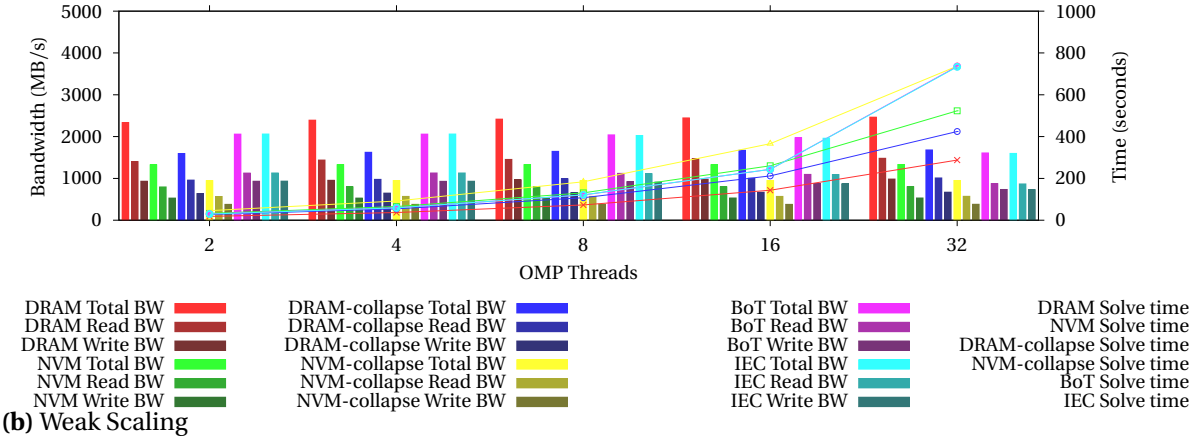


(b) Weak Scaling

Figure 28 7-pt stencil chunk size comparison Time and DTLB Load Store miss ratios

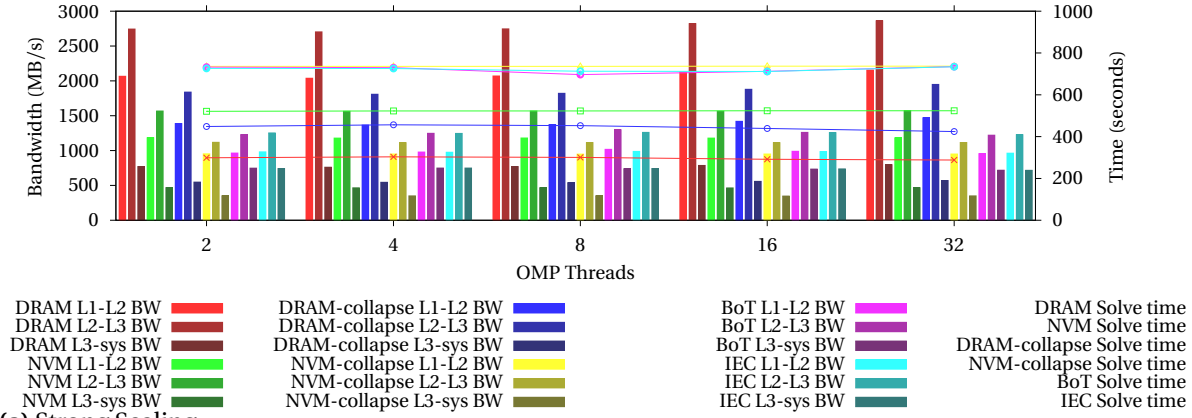


(a) Strong Scaling

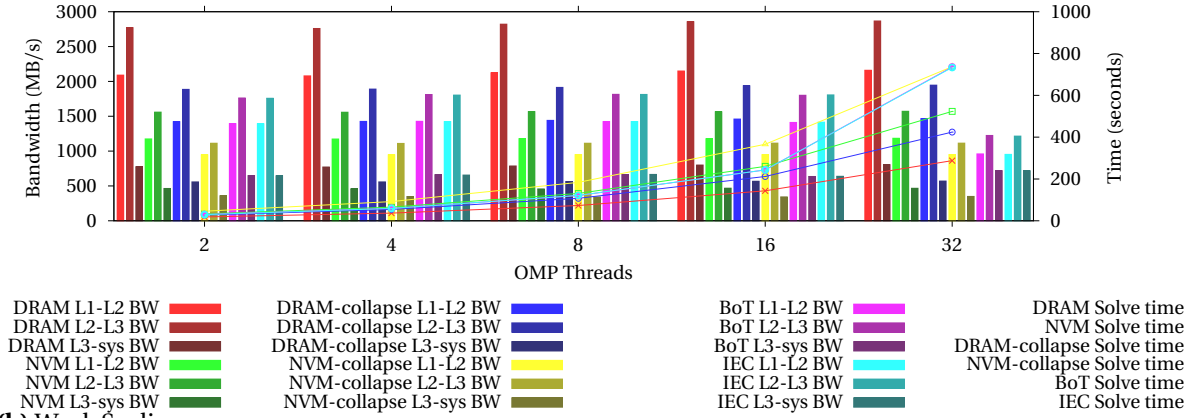


(b) Weak Scaling

Figure 29 7-pt 3 halo stencil scheduling comparison Time and Bandwidth

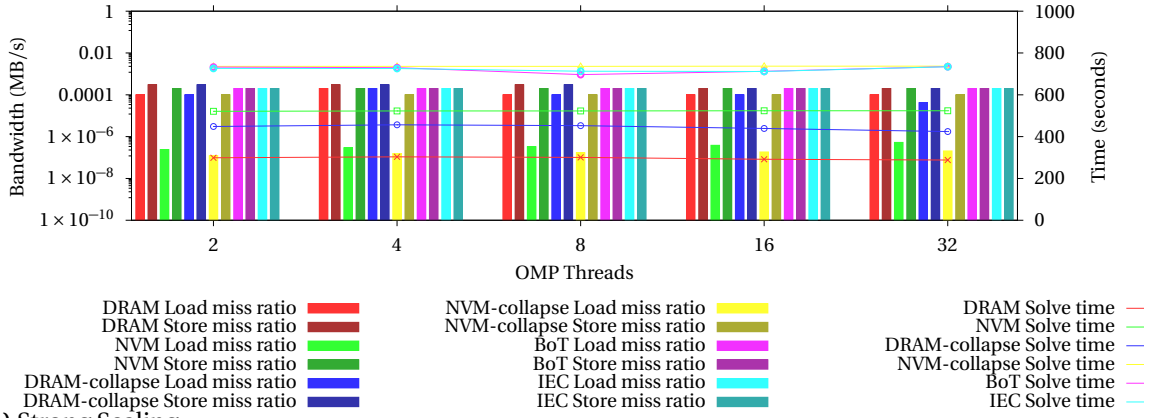


(a) Strong Scaling

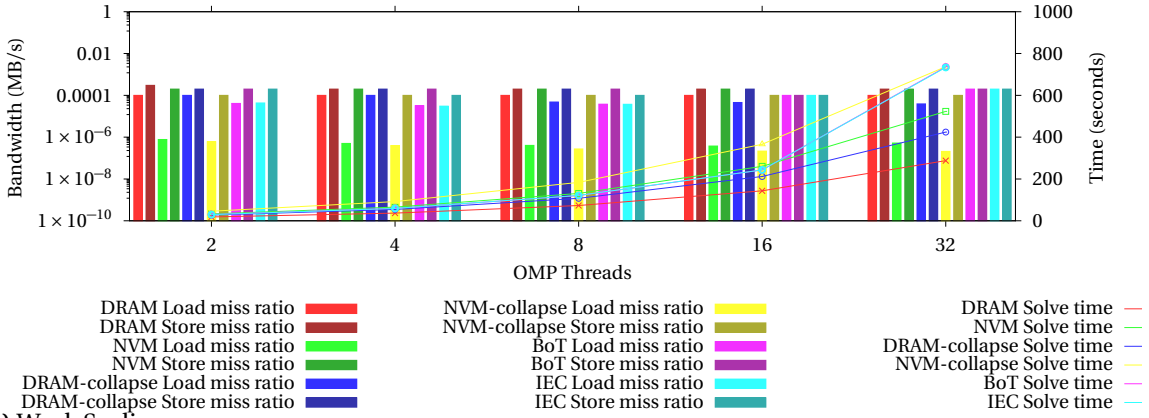


(b) Weak Scaling

Figure 30 7-pt 3 halo stencil scheduling comparison Time and Cache Bandwidth

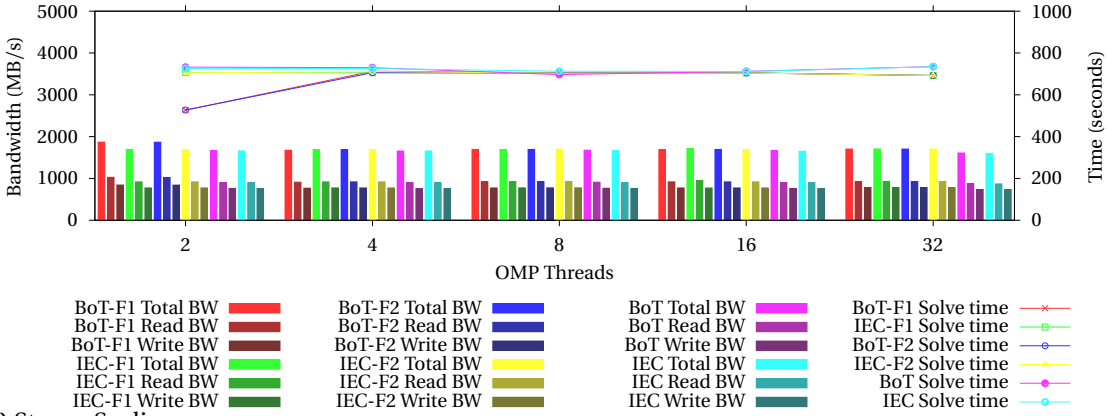


(a) Strong Scaling

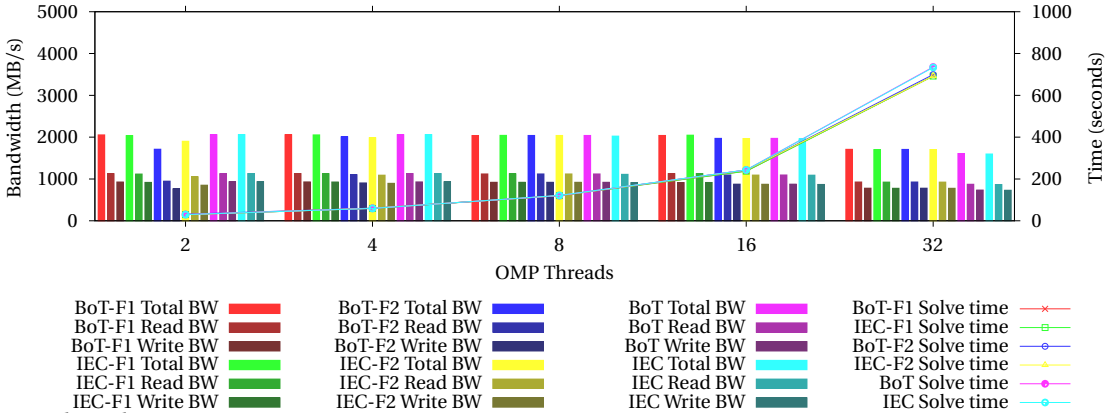


(b) Weak Scaling

Figure 31 7-pt 3-pt halo stencil scheduling Time and DTLB Load Store miss ratios

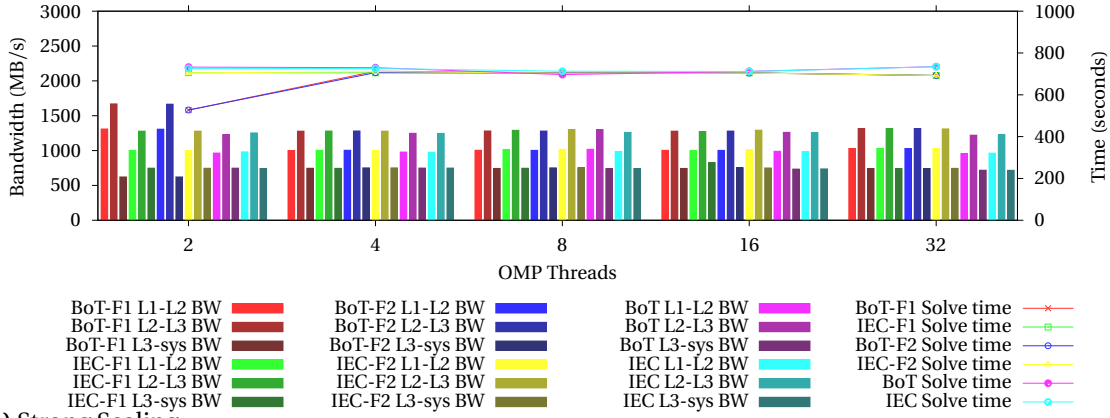


(a) Strong Scaling

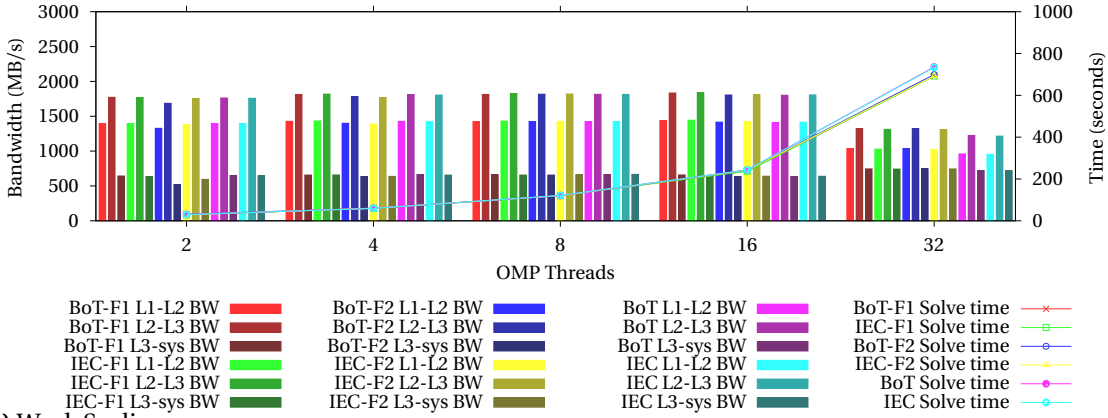


(b) Weak Scaling

Figure 32 7-pt 3 halo stencil chunk size comparison Time and Bandwidth

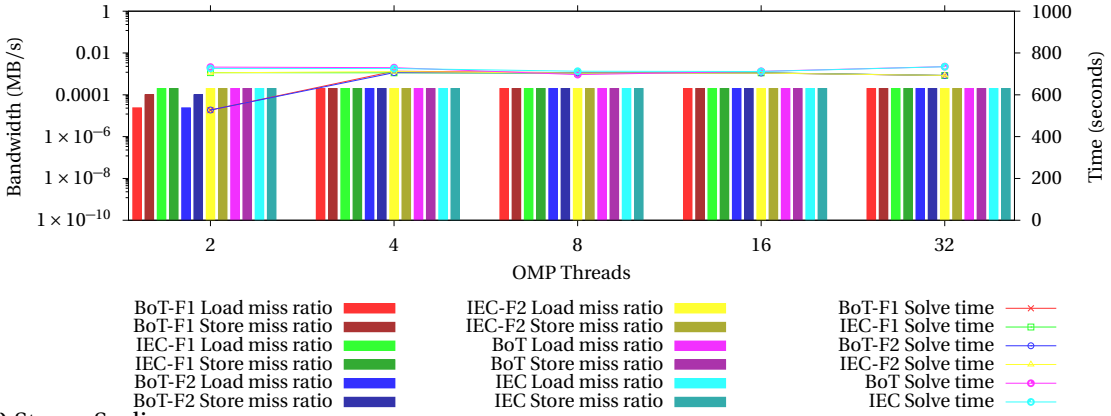


(a) Strong Scaling

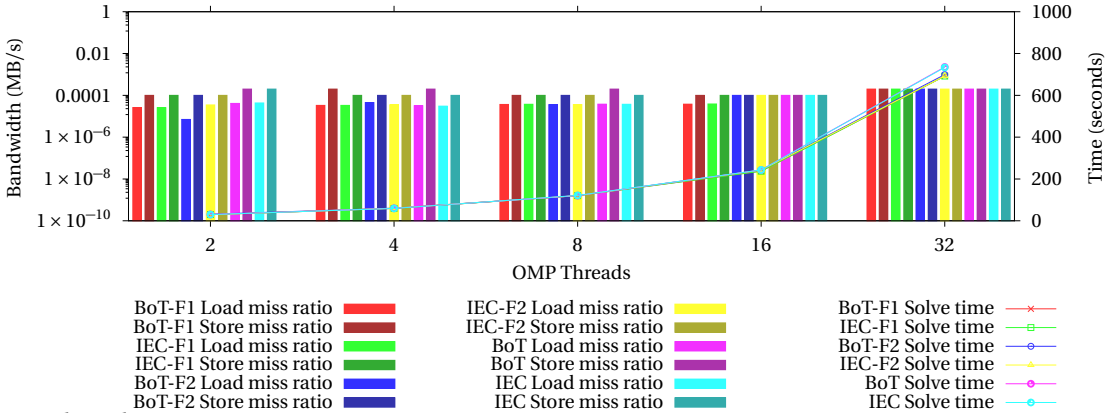


(b) Weak Scaling

Figure 33 7-pt 3 halo stencil chunk size comparison Time and Cache Bandwidth

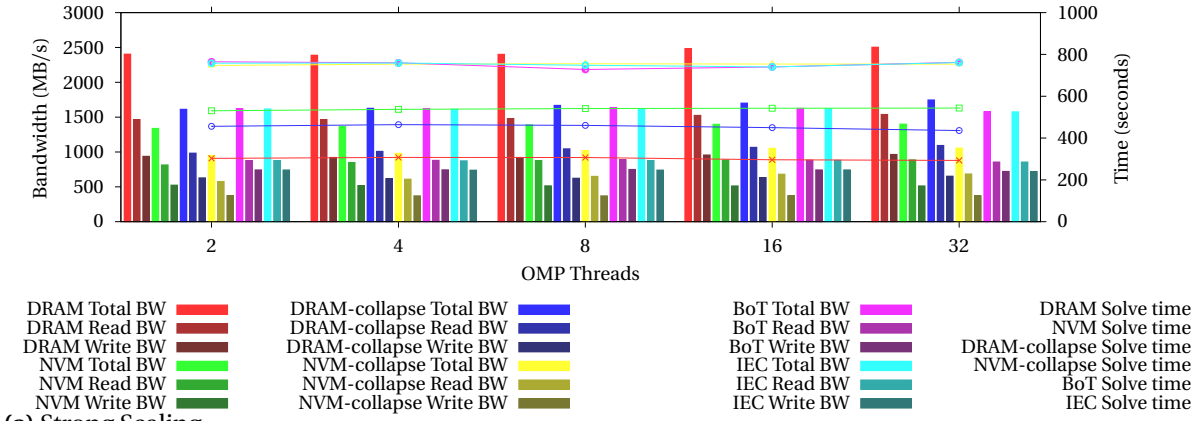


(a) Strong Scaling

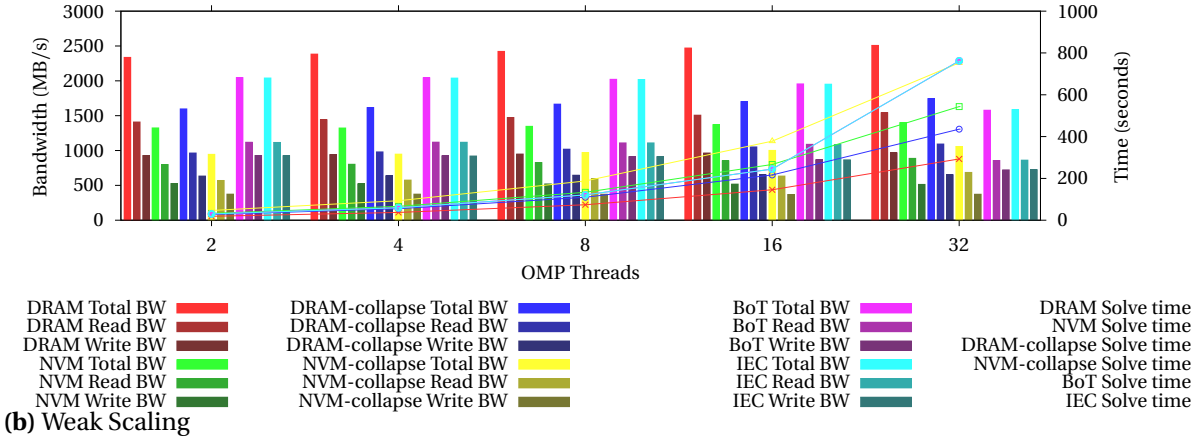


(b) Weak Scaling

Figure 34 7-pt 3-pt halo stencil chunk size comparison Time and DTLB Load Store miss ratios

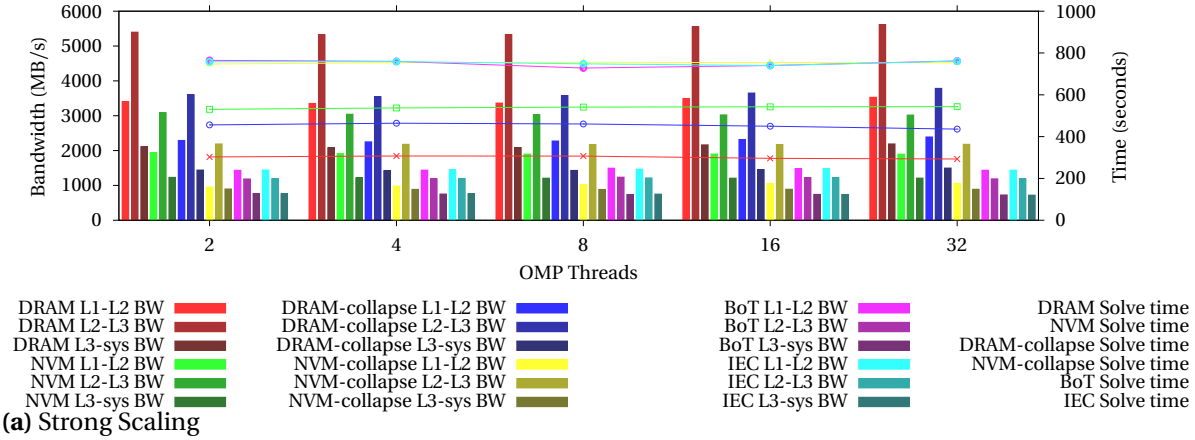


(a) Strong Scaling

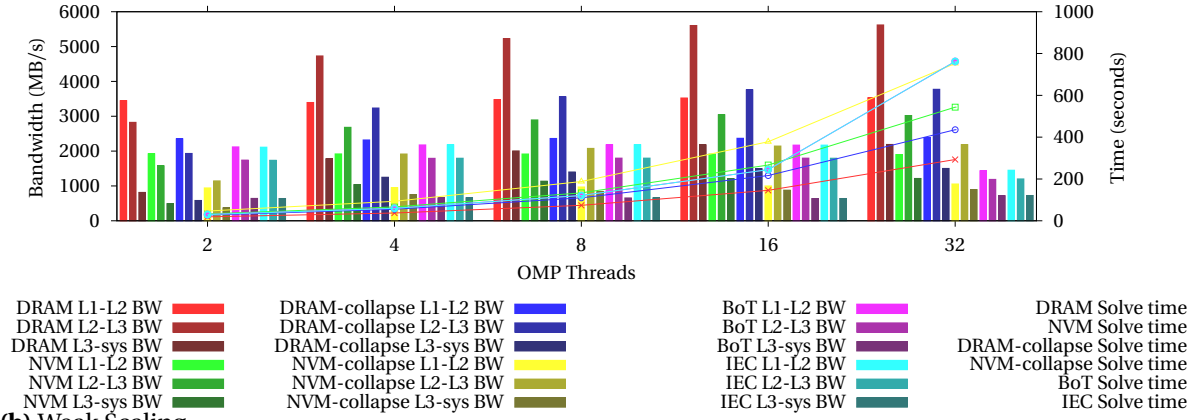


(b) Weak Scaling

Figure 35 9-pt stencil scheduling comparison Time and Bandwidth

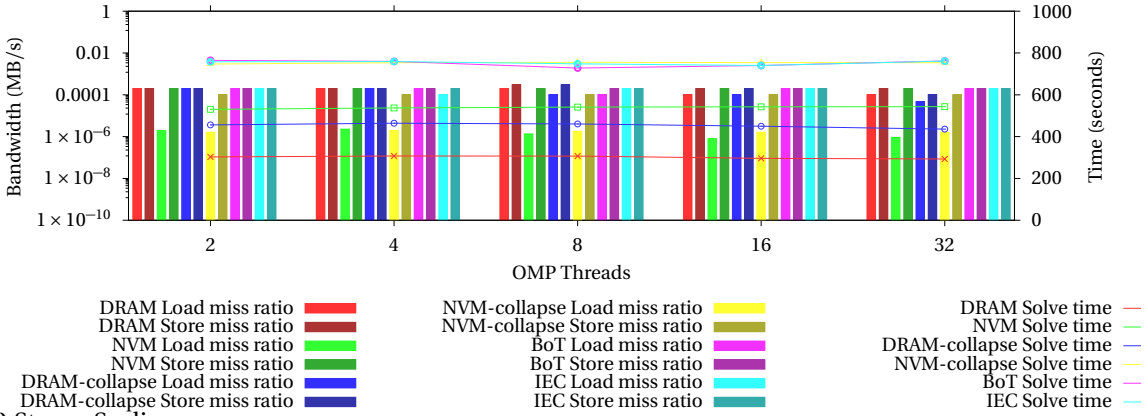


(a) Strong Scaling

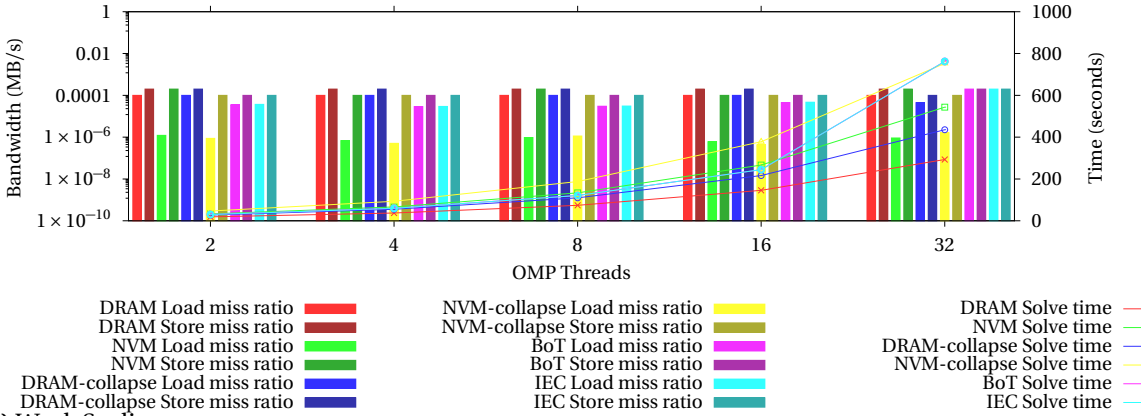


(b) Weak Scaling

Figure 36 9-pt stencil scheduling comparison Time and Cache Bandwidth

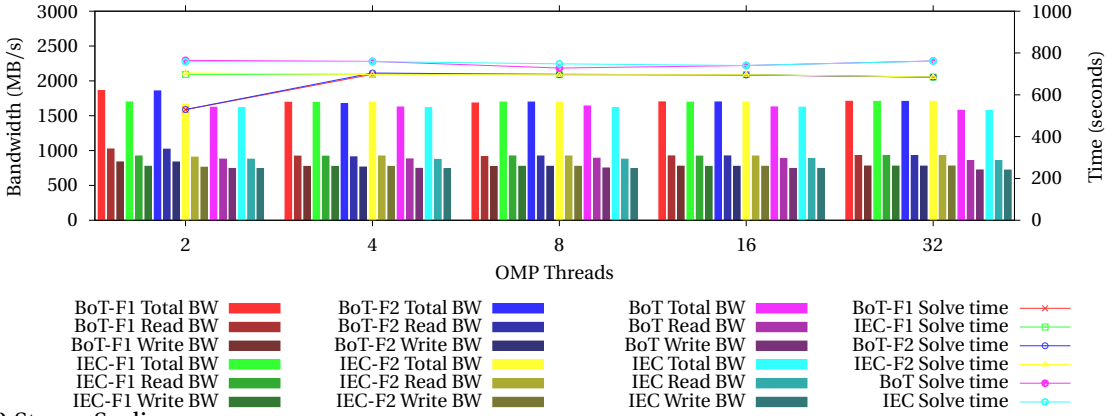


(a) Strong Scaling

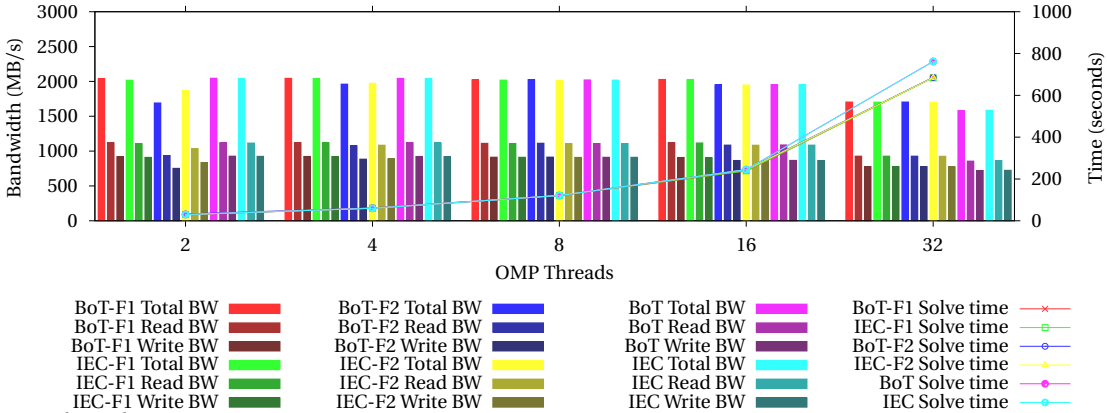


(b) Weak Scaling

Figure 37 9-pt stencil scheduling comparison Time and DTLB Load Store miss ratios

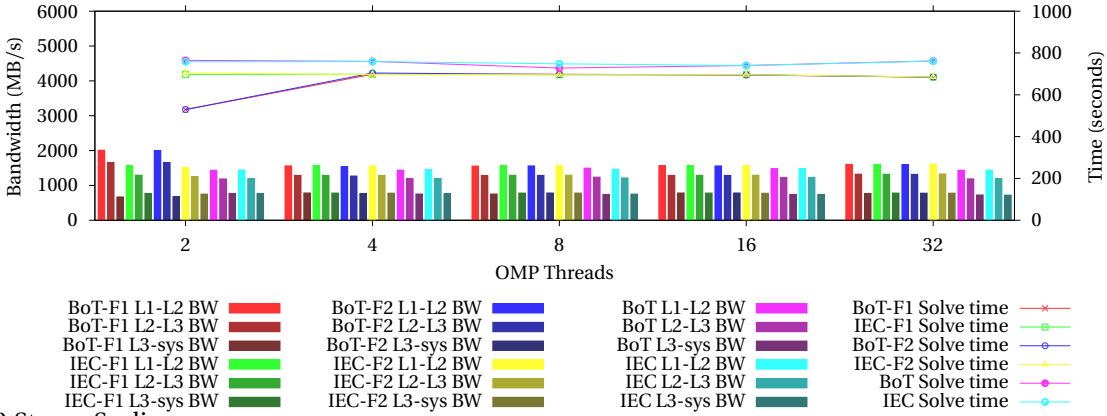


(a) Strong Scaling

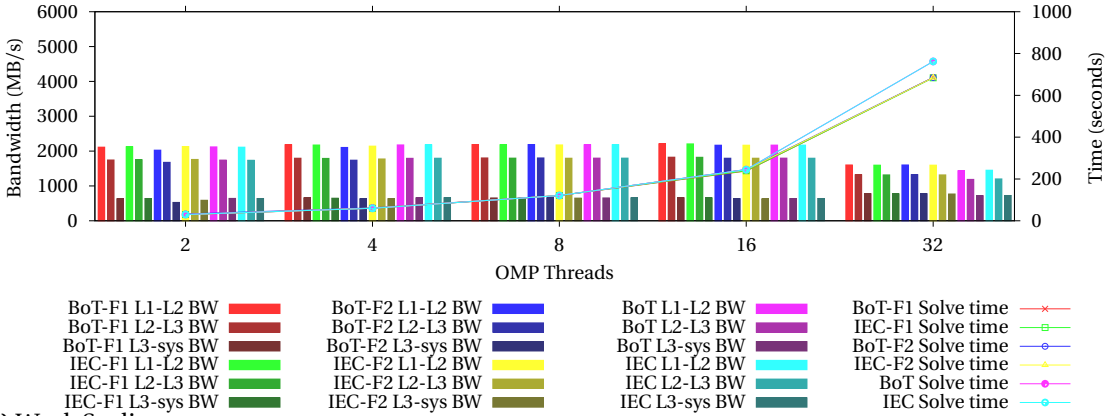


(b) Weak Scaling

Figure 38 9-pt stencil chunk size comparison Time and Bandwidth

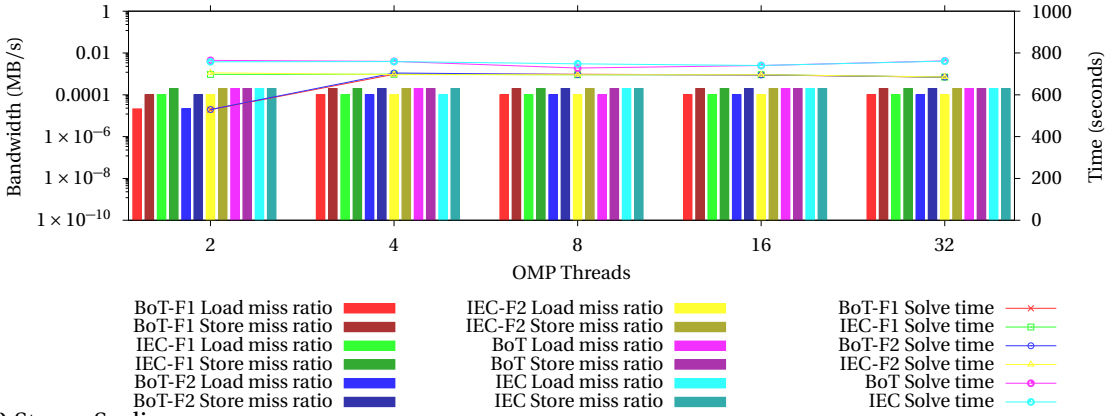


(a) Strong Scaling

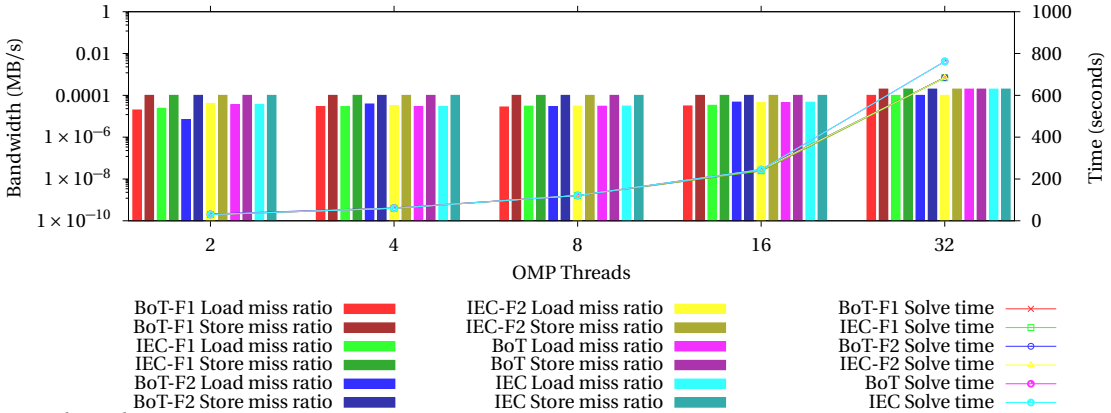


(b) Weak Scaling

Figure 39 9-pt stencil chunk size comparison Time and Cache Bandwidth

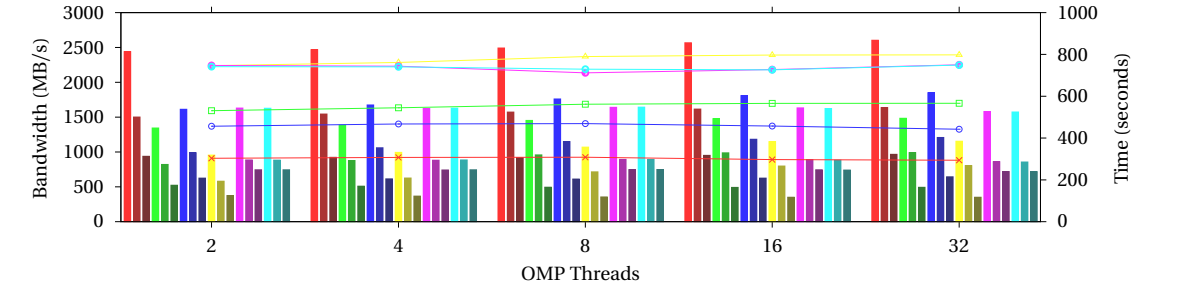


(a) Strong Scaling

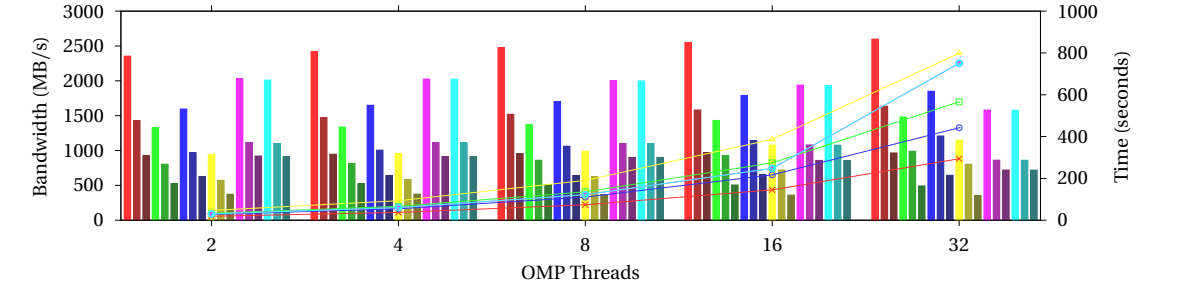


(b) Weak Scaling

Figure 40 9-pt stencil chunk size comparison Time and DTLB Load Store miss ratios

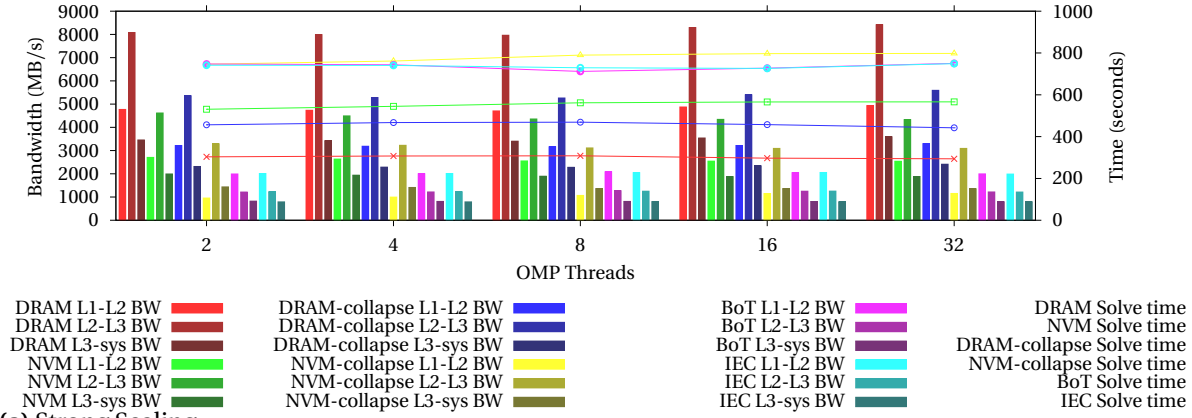


(a) Strong Scaling

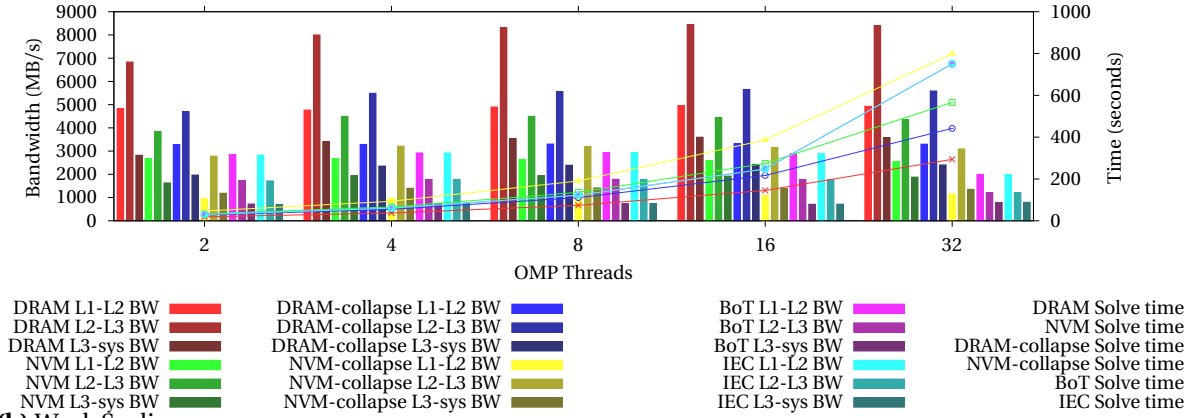


(b) Weak Scaling

Figure 41 9-pt stencil 2-pt halo scheduling comparison Time and Bandwidth

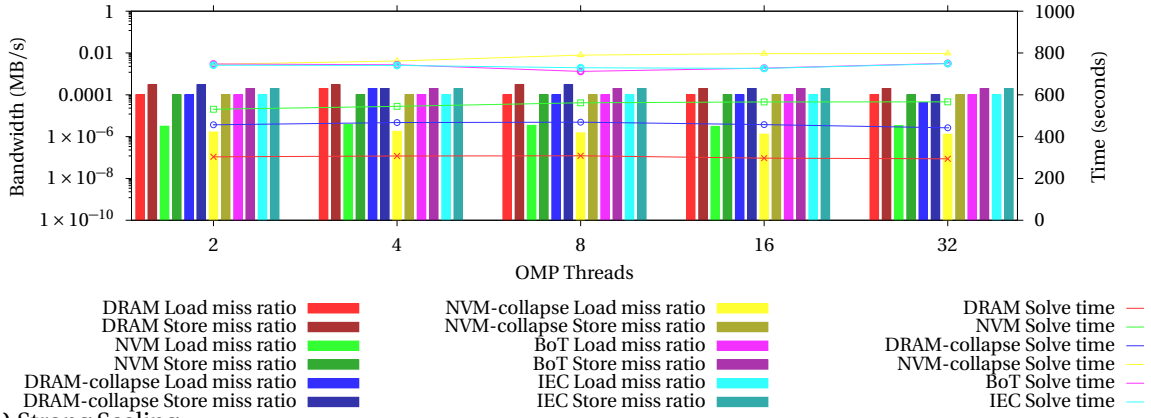


(a) Strong Scaling

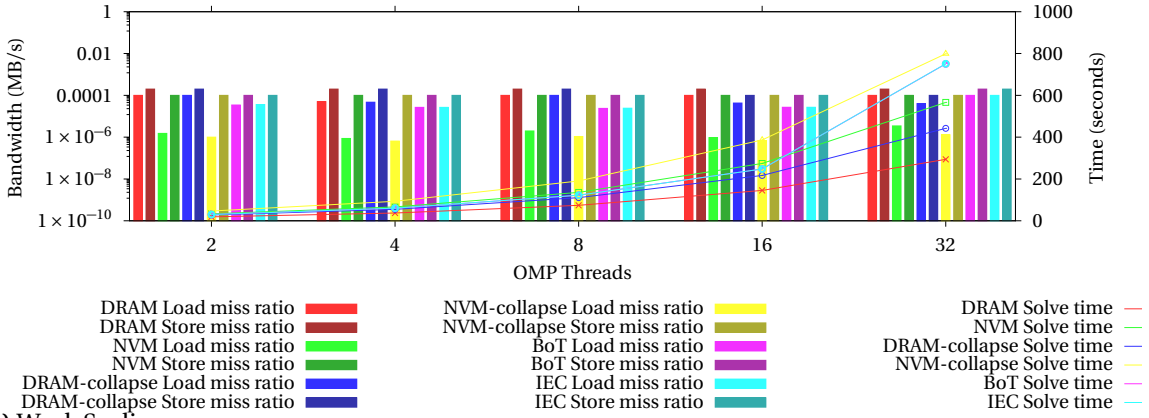


(b) Weak Scaling

Figure 42 9-pt stencil 2-pt halo scheduling comparison Time and Cache Bandwidth

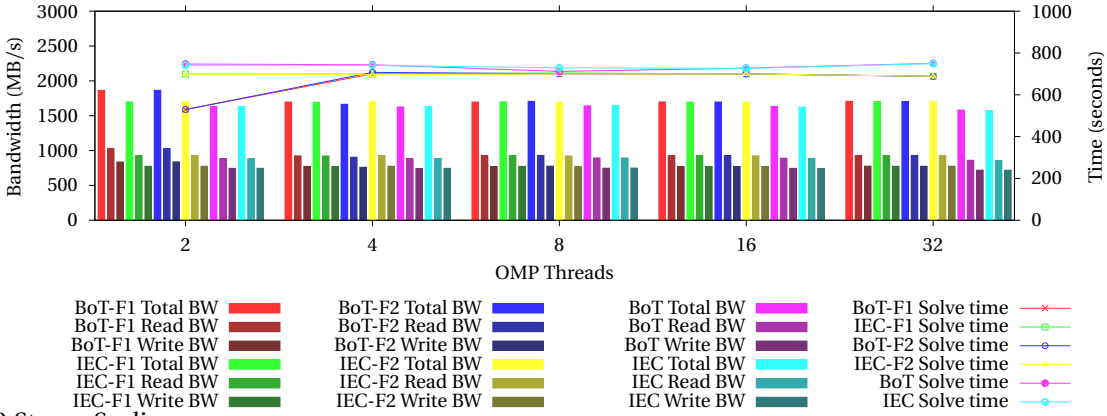


(a) Strong Scaling

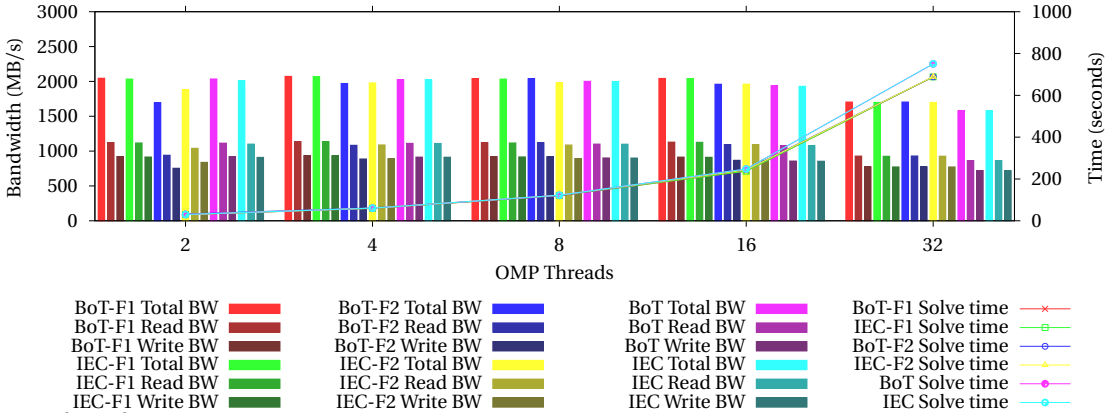


(b) Weak Scaling

Figure 43 9-pt stencil 2-pt halo scheduling comparison Time and DTLB Load Store miss ratios

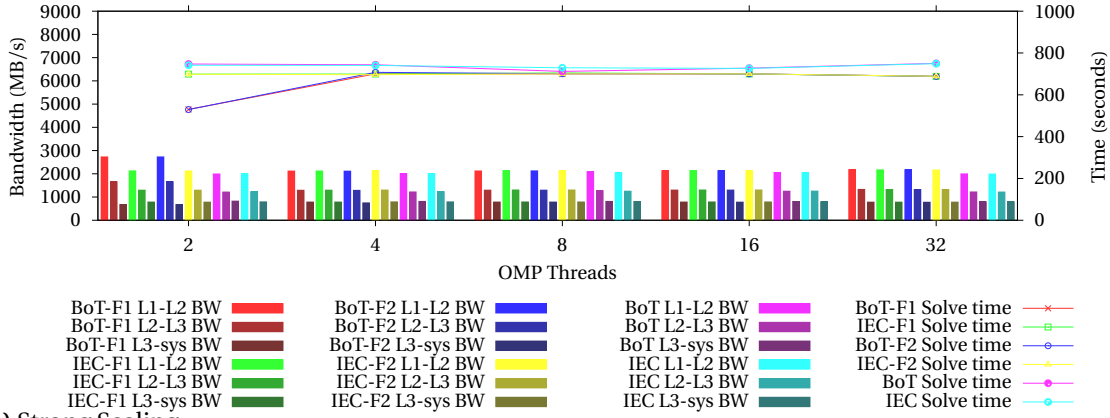


(a) Strong Scaling

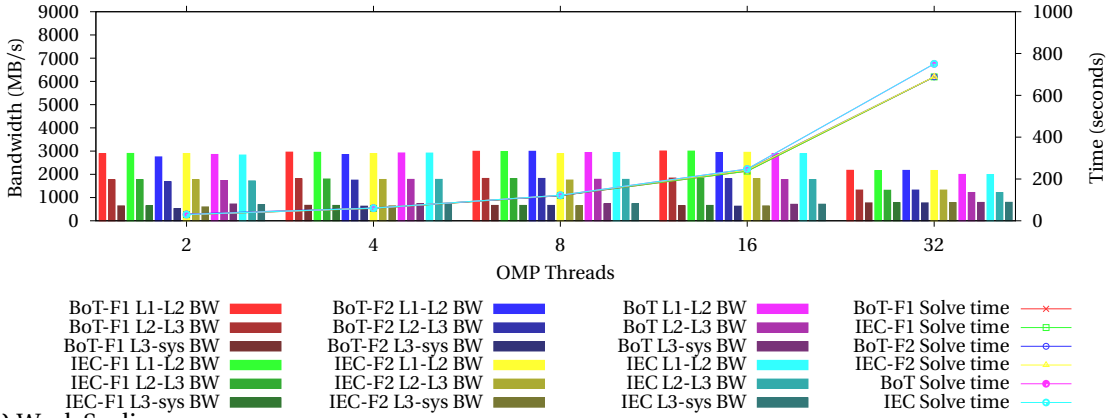


(b) Weak Scaling

Figure 44 9-pt stencil 2-pt halo chunk size comparison Time and Bandwidth

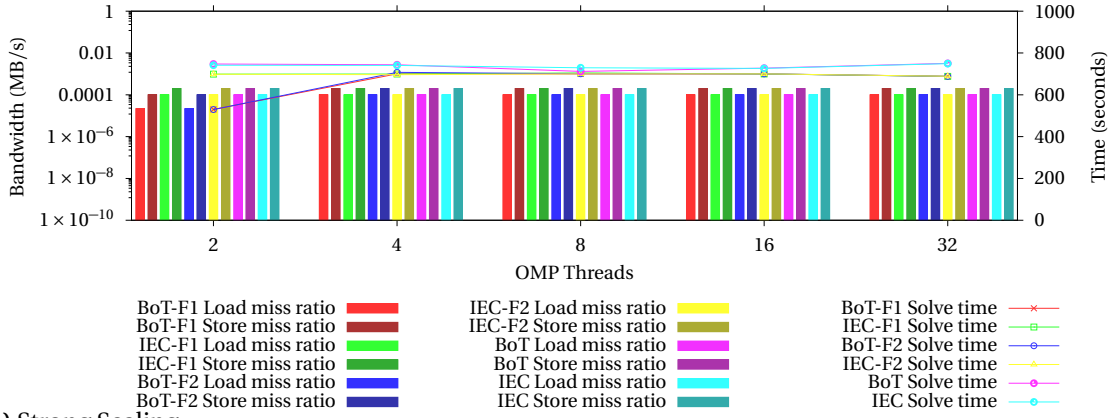


(a) Strong Scaling

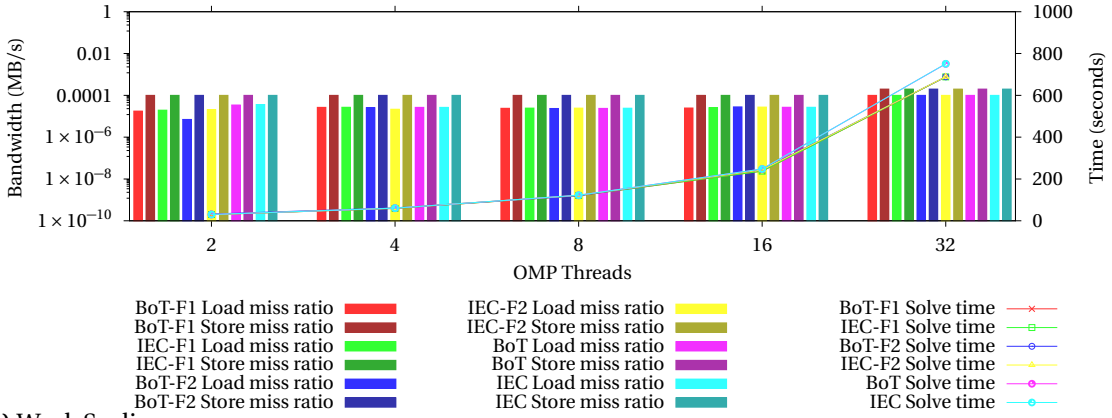


(b) Weak Scaling

Figure 45 9-pt stencil 2-pt halo chunk size comparison Time and Cache Bandwidth

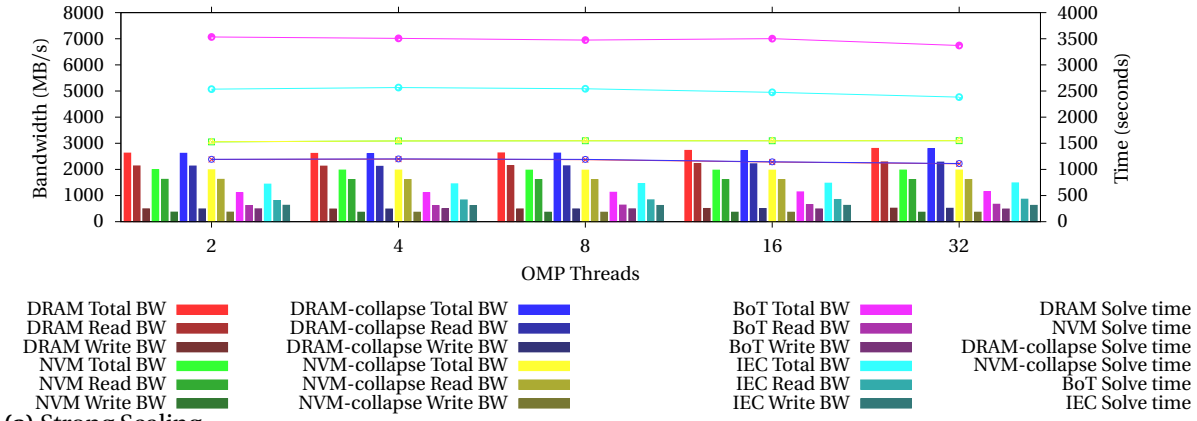


(a) Strong Scaling

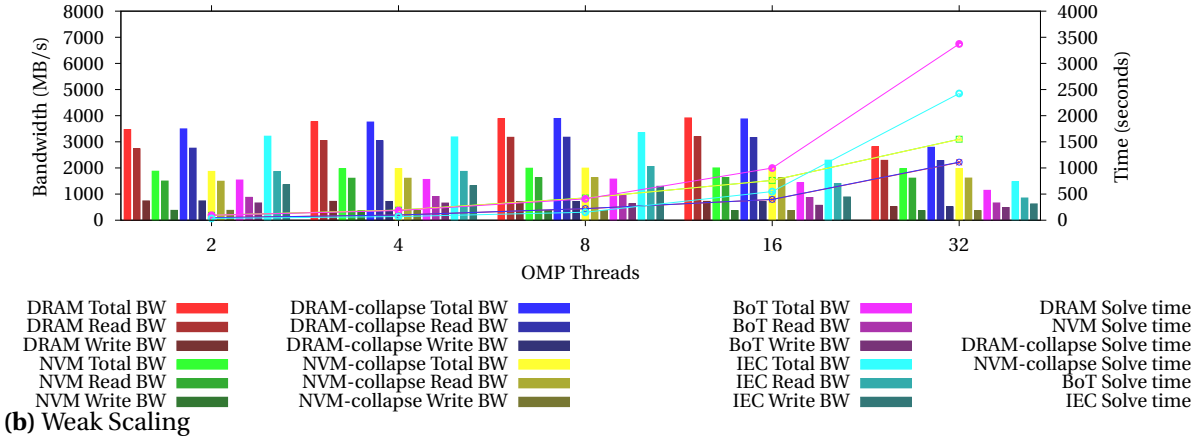


(b) Weak Scaling

Figure 46 9-pt stencil 2-pt halo chunk size comparison Time and DTLB Load Store miss ratios

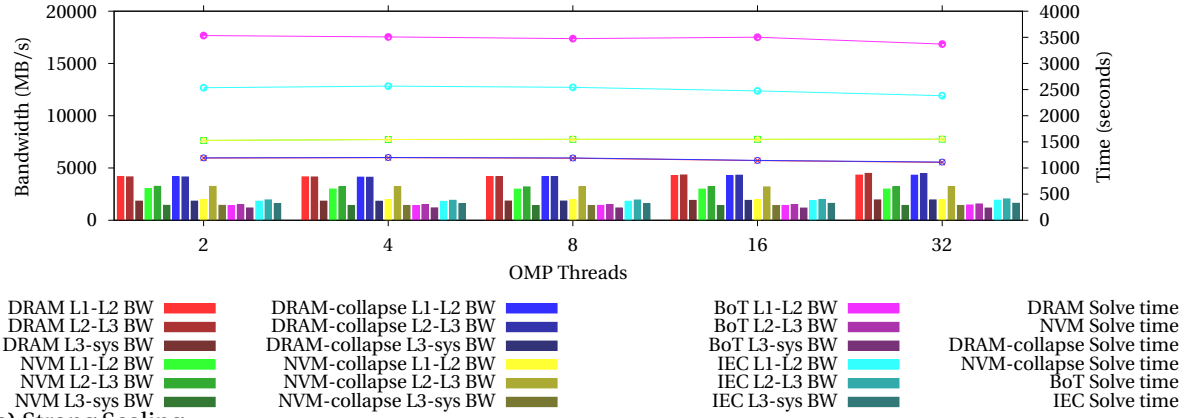


(a) Strong Scaling

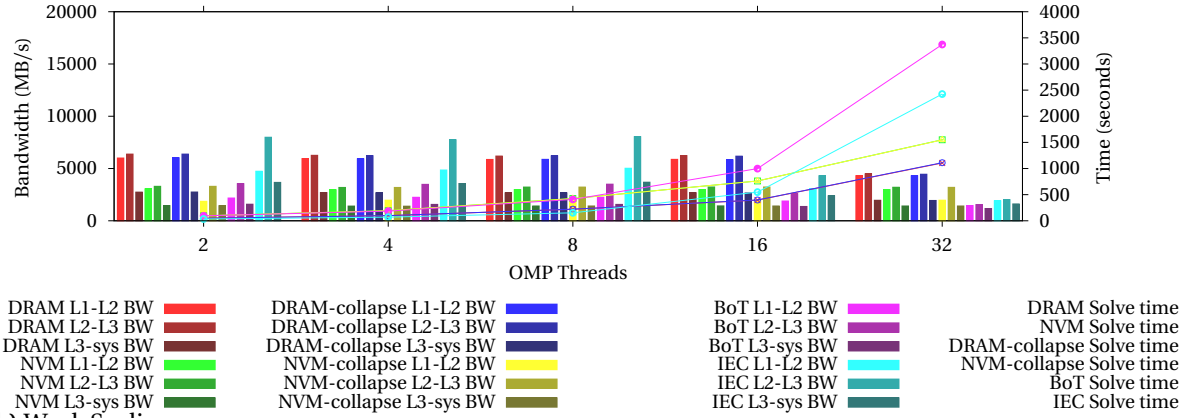


(b) Weak Scaling

Figure 47 13-pt stencil 2-pt halo scheduling comparison Time and Bandwidth

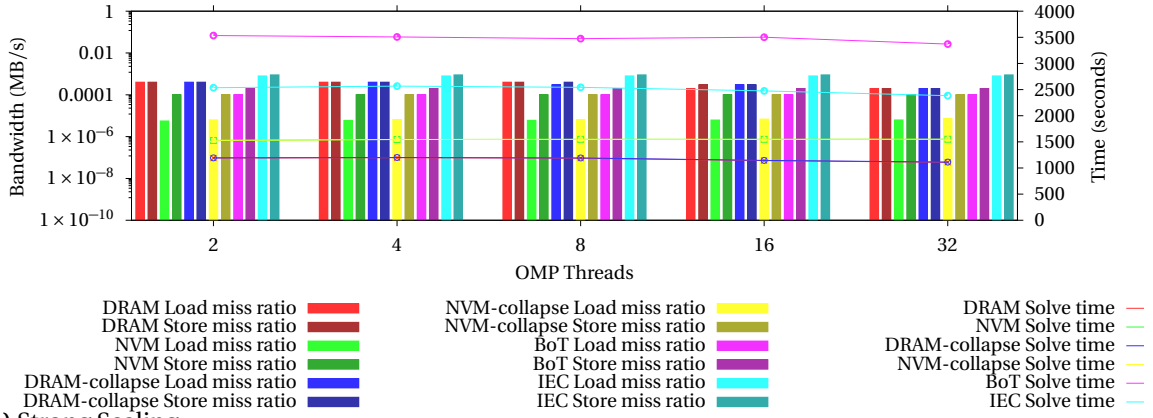


(a) Strong Scaling

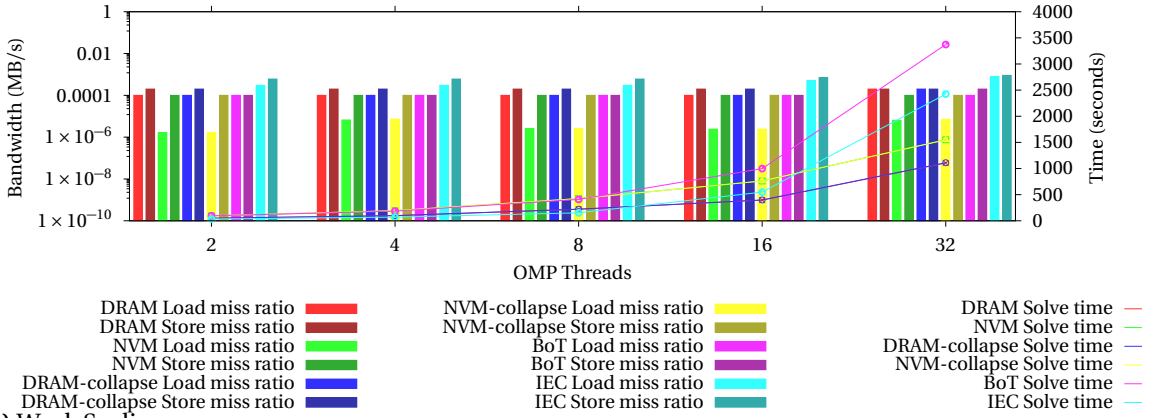


(b) Weak Scaling

Figure 48 13-pt stencil 2-pt halo scheduling comparison Time and Bandwidth

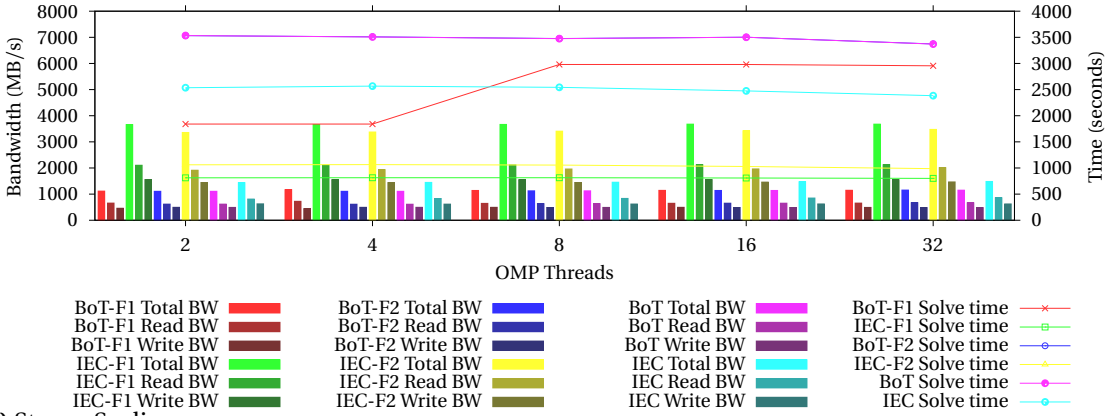


(a) Strong Scaling

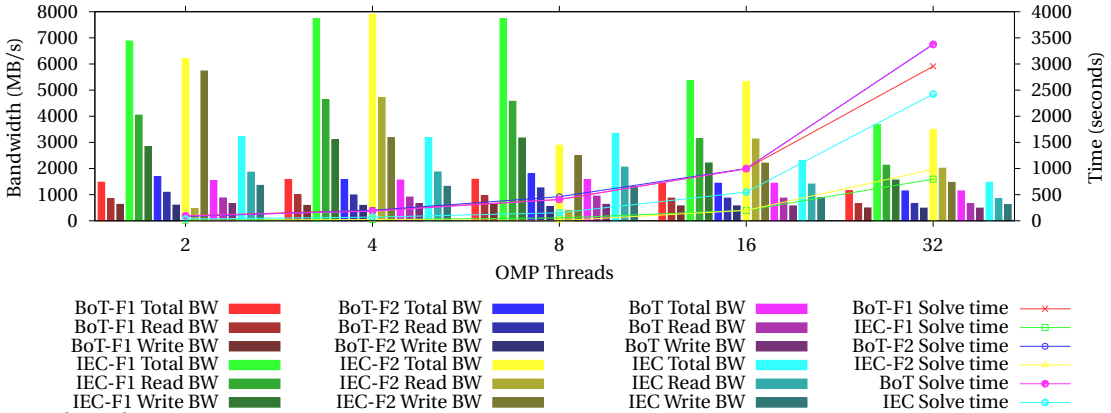


(b) Weak Scaling

Figure 49 13-pt stencil 2-pt halo scheduling comparison Time and DTLB Load Store miss ratios

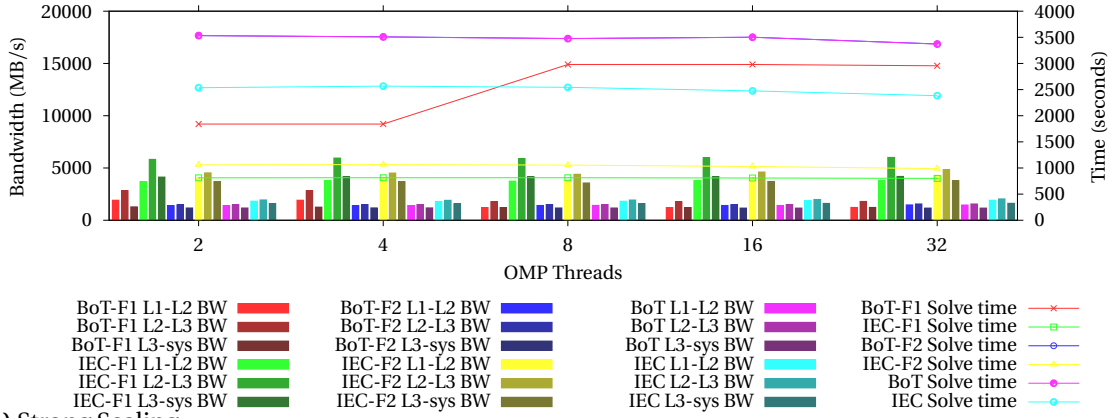


(a) Strong Scaling

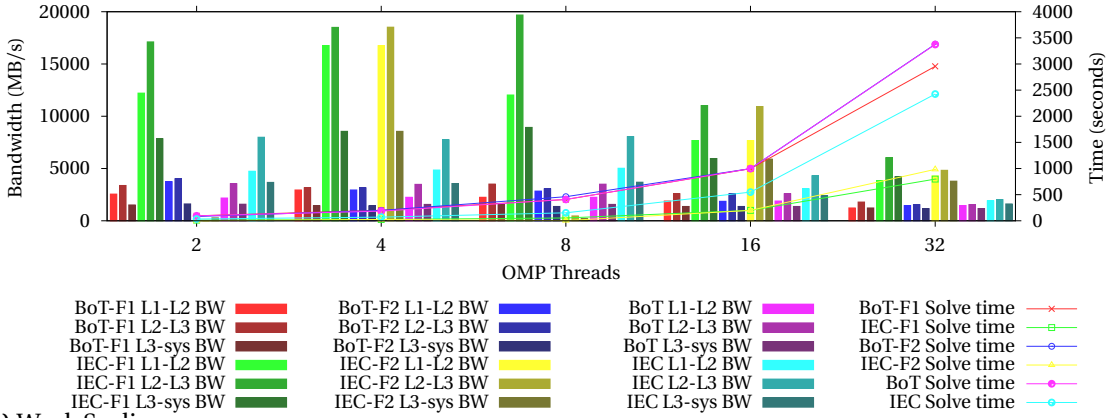


(b) Weak Scaling

Figure 50 13-pt stencil 2-pt halo chunk size comparison Time and Bandwidth

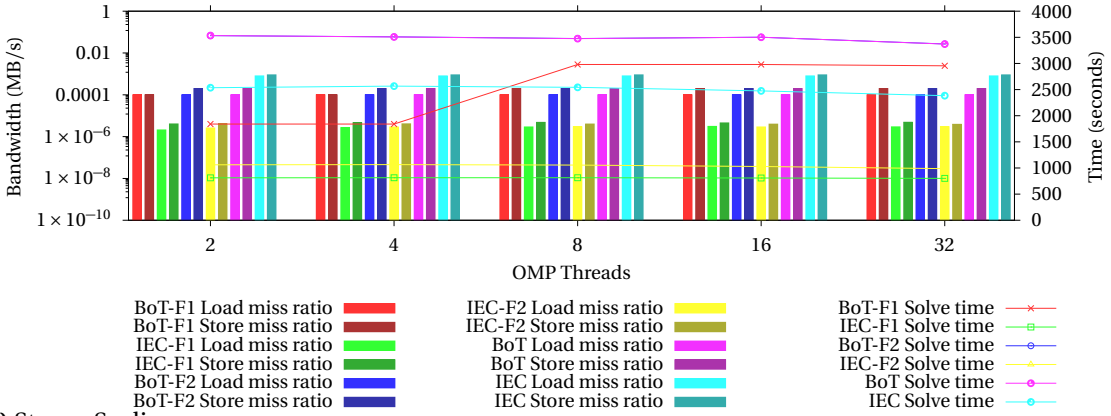


(a) Strong Scaling

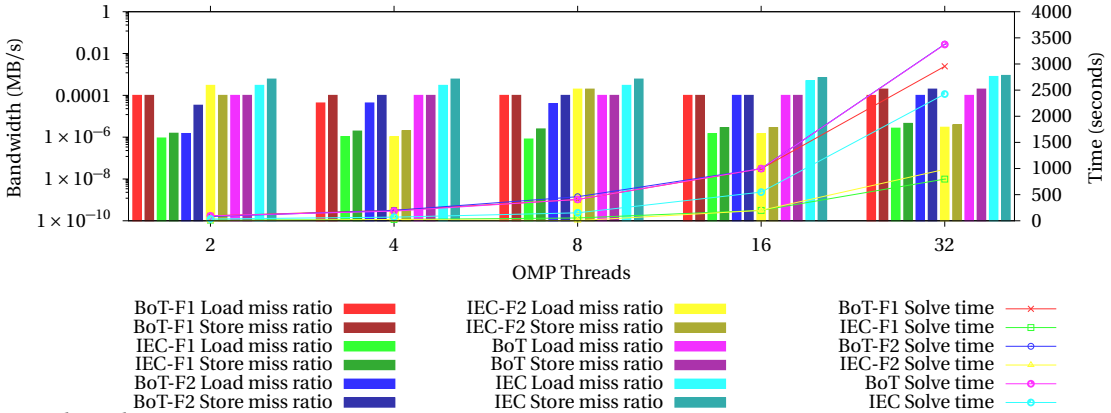


(b) Weak Scaling

Figure 51 13-pt stencil 2-pt halo chunk size comparison Time and Cache Bandwidth

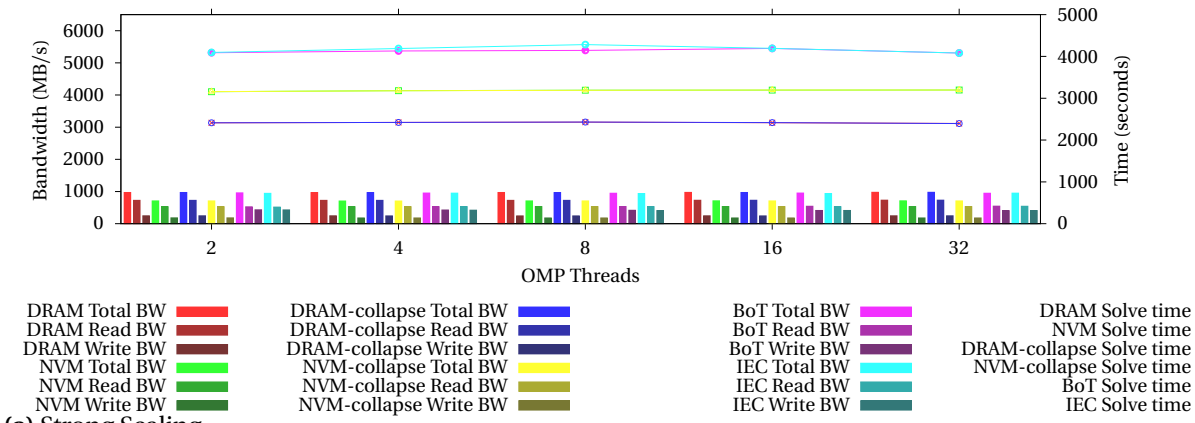


(a) Strong Scaling

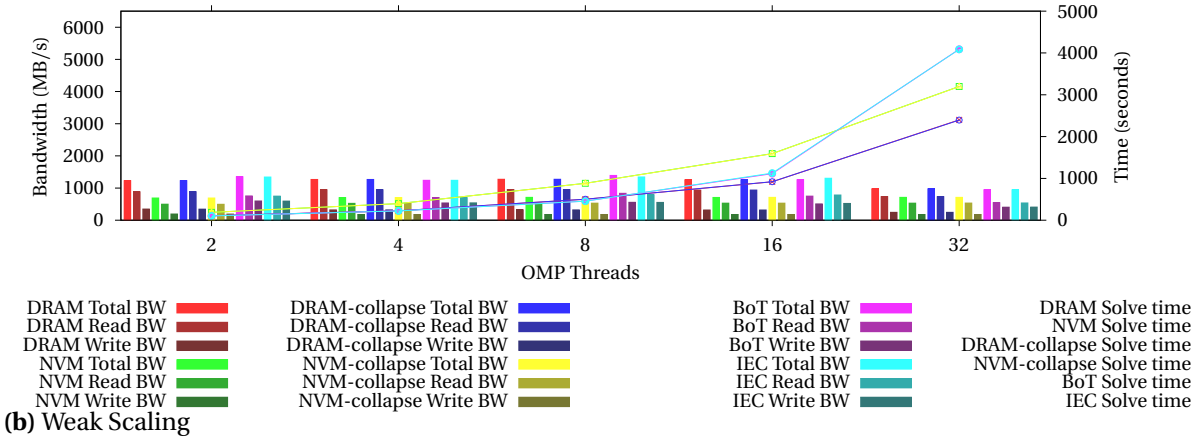


(b) Weak Scaling

Figure 52 13-pt stencil 2-pt halo chunk size comparison Time and DTLB Load Store miss ratios

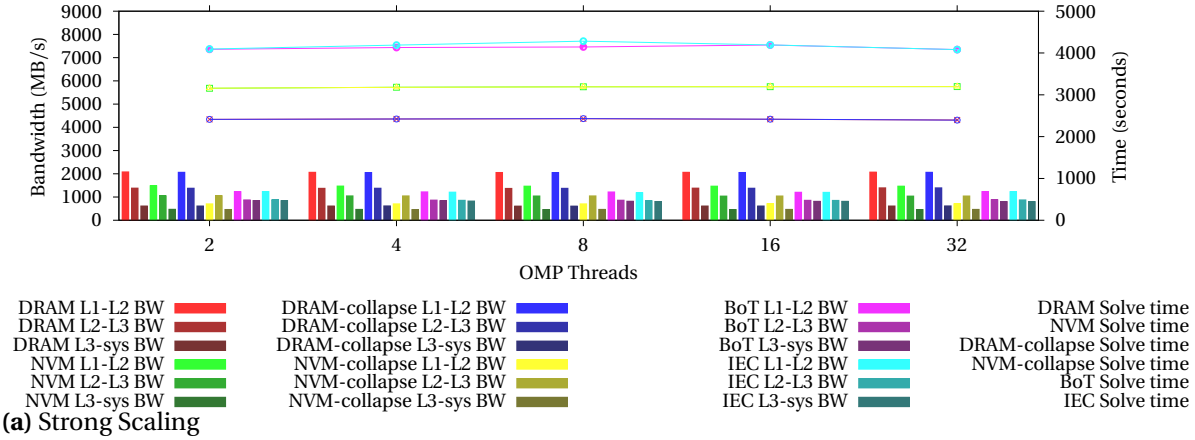


(a) Strong Scaling

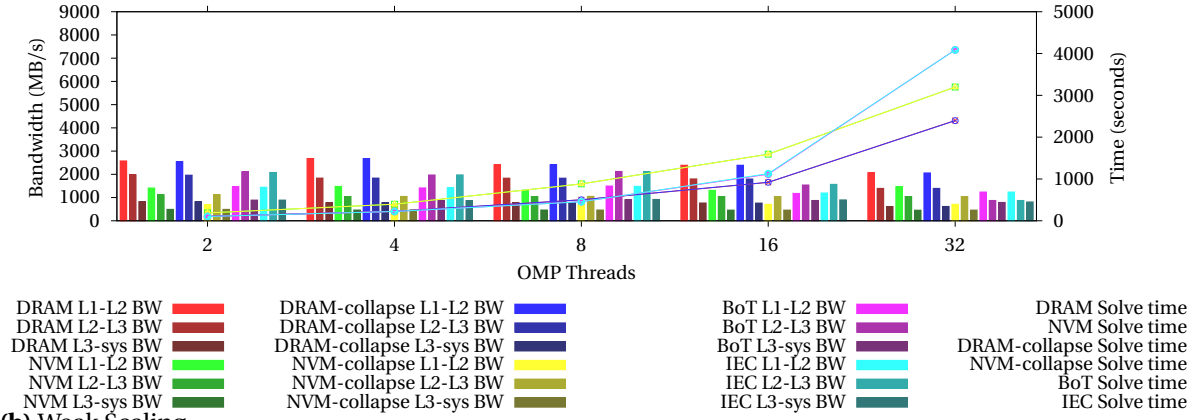


(b) Weak Scaling

Figure 53 27-pt stencil scheduling comparison Time and Bandwidth

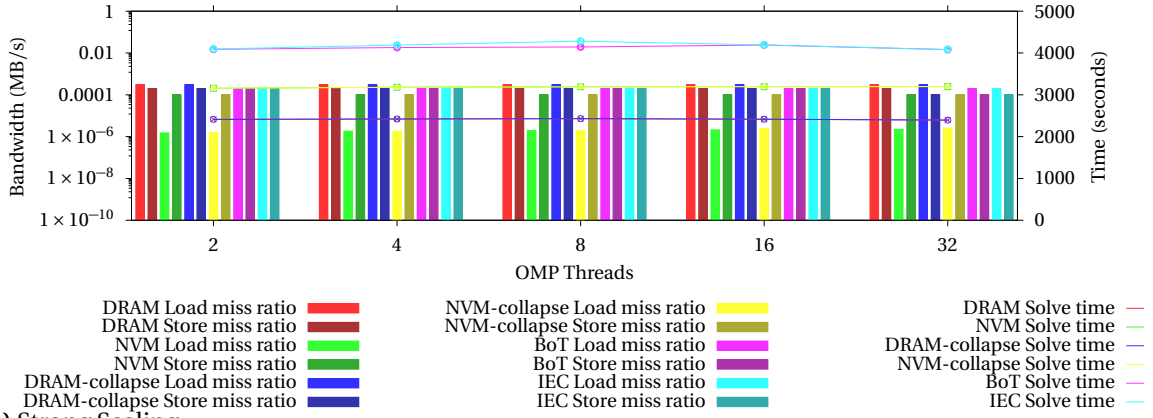


(a) Strong Scaling

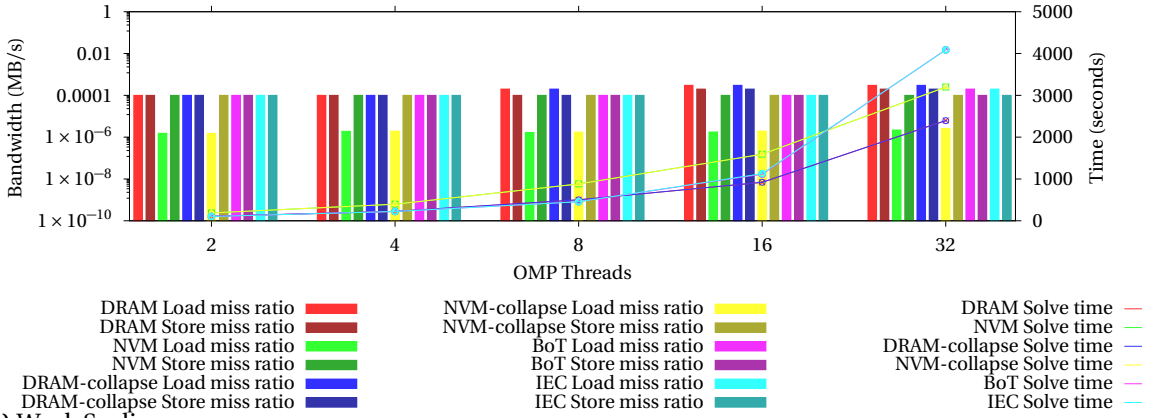


(b) Weak Scaling

Figure 54 27-pt stencil scheduling comparison Time and Cache Bandwidth

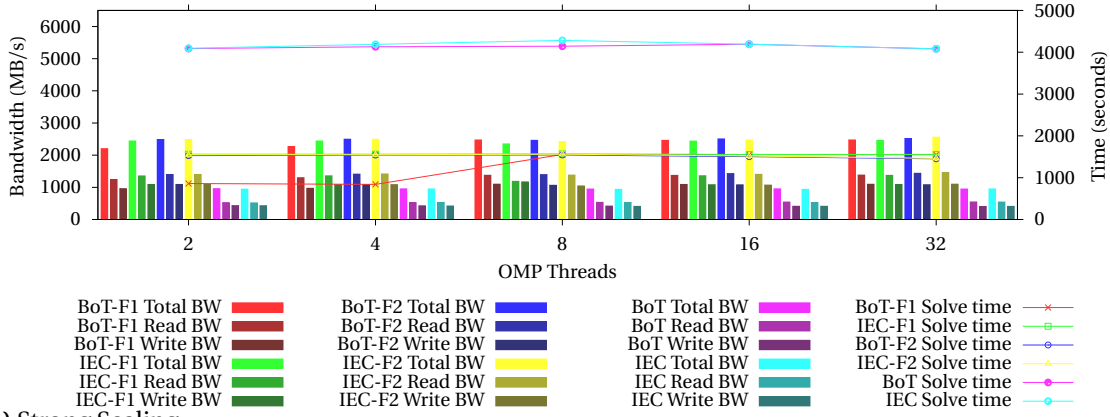


(a) Strong Scaling

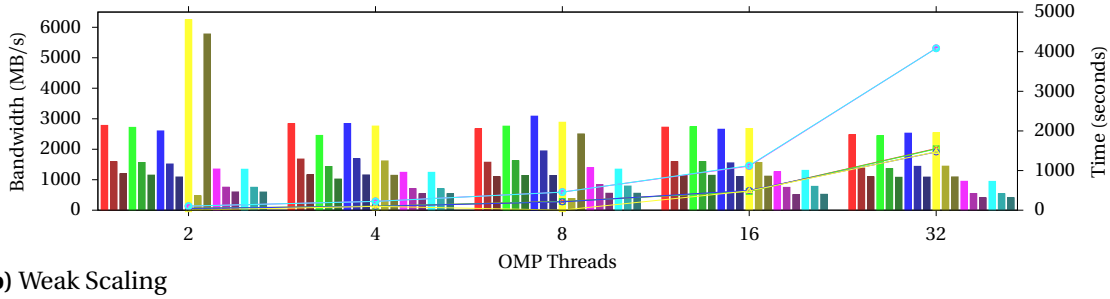


(b) Weak Scaling

Figure 55 27-pt stencil scheduling comparison Time and DTLB Load Store miss ratios

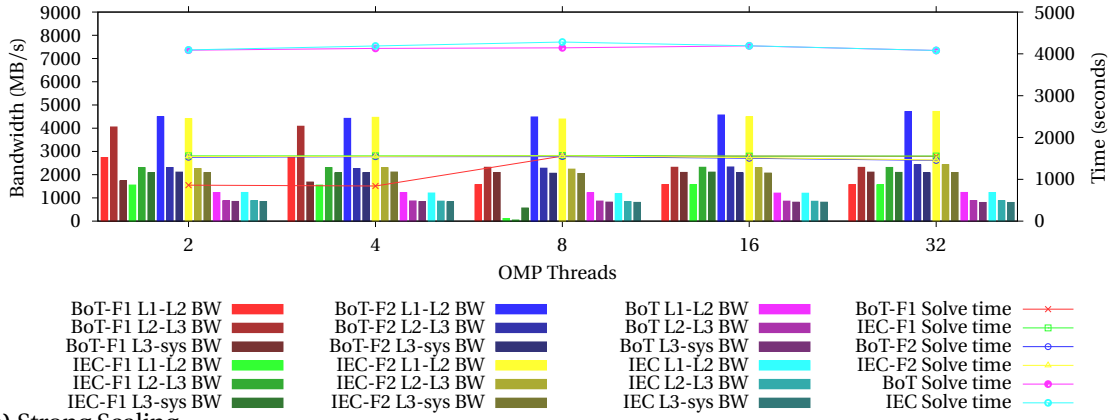


(a) Strong Scaling

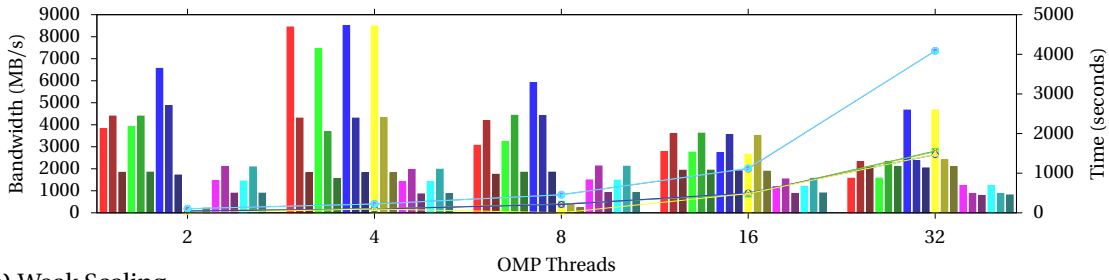


(b) Weak Scaling

Figure 56 27-pt stencil chunk size comparison Time and Bandwidth

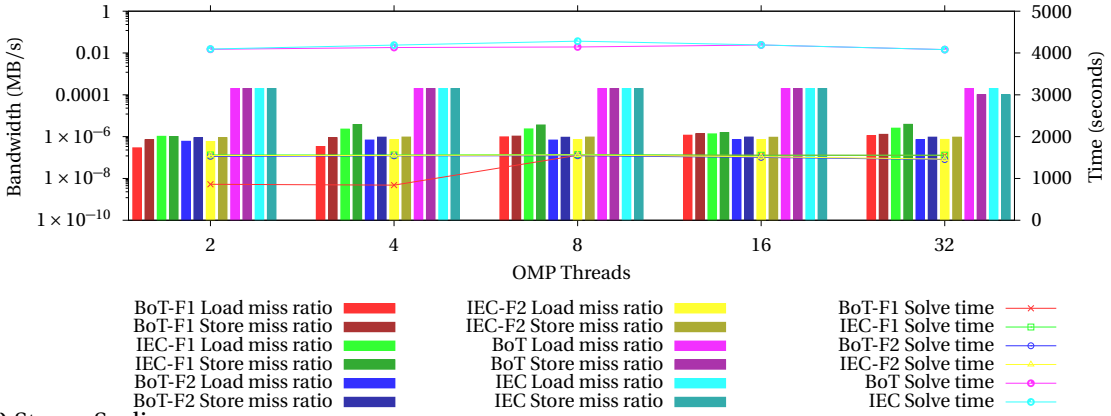


(a) Strong Scaling

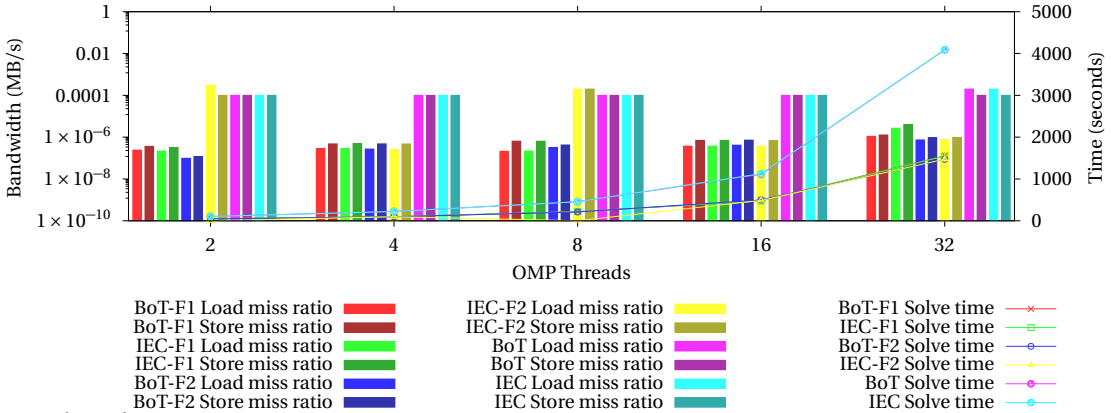


(b) Weak Scaling

Figure 57 27-pt stencil chunk size comparison Time and Cache Bandwidth



(a) Strong Scaling



(b) Weak Scaling

Figure 58 27-pt stencil chunk size comparison Time and DTLB Load Store miss ratios