ABSTRACT

Title of Dissertation: Hybrid-PGAS Memory Hierarchy
for Next Generation HPC Systems

Richard B. Johnson
Doctor of Philosophy, 2024

Demands on computational performance, power efficiency, data transfer, resource capacity, and resilience for next generation high performance computing (HPC) systems present a new host of challenges. There is a growing disparity between computational performance vs. network and storage device throughput and among the energy costs of computational, memory, and communication operations. Chapel is a powerful, high-level, parallel, PGAS language designed to streamline development by addressing code complexities and uses a shared memory model for handling large, distributed memory systems. I extended the capabilities of Chapel by providing support of persistent memory with intrinsic and programmatic features for HPC systems. In my approach I explored the efficacy of persistent memory in a hybrid-PGAS environment through latency hiding analysis via cache monitoring, identification and mitigation of performance bottlenecks via data-centric analysis, and hardware profiling to assess performance cost vs. benefits and energy footprint. To manage persistency and ensure resiliency I developed a transaction system with ACID properties that supports hybrid-PGAS virtual addressing and distributed checkpoint and recovery system.

Hybrid-PGAS Memory Hierarchy for Next Generation HPC Systems

by

Richard B. Johnson

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:

Dr. Jeffrey K. Hollingsworth, Chair and Advisor
Dr. Alan Sussman, Member
Dr. Bahar Asgari, Member
Dr. Ashok Agrawala, Member
Dr. Donald Yeung, Dean's Representative

Acknowledgments

Thank you,

Dr. Jeffrey K. Hollingsworth for advising me all of these years

Dr. Alan Sussman for your direction and support

Dr. Ilchul Yoon, Andrej Rasevic, and Dr. Hui Zhang for being good colleagues

Tracy for always being there for me

Hampton and Asher for being honorary sons

Buttercup, Zinnia, Dahlia, Daisy, Rose, Tulip, Verbena, Sunshine, and Iggy for emotional support

Mom

Table of Contents

List of Tables

List of Figures

Listings

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
ADT	Abstract Data Type
AM	Active Messages
API	Application Programming Interface
AST	Abstract Syntax Tree
Chapel	Cascade High Productivity Language
DRAM	Dynamic Random-Access Memory
eDRAM	Embedded DRAM
FAM	Fabric-Attached Memory
FLOPS	Floating Point Operations per Second
GASNet	Global-Address Space Networking
HPC	High Performance Computing
HPCS	High Productivity Computing System
HPE	Hewlett Packard Enterprise
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
NVML	Non-Volatile Memory Libraries
NVRAM	Non-Volatile Random-Access Memory
OpenMP	Open Multi-Processing
ORNL	Oak Ridge National Laboratory
PAPI	Performance API
PCRAM	Phase-Change Random-Access Memory
PE	Processing Element
PMEMoid	Persistent Memory Object Identifier
PMDK	Persistent Memory Development Kit
PGAS	Partitioned Global Address Space
ReRAM	Resistive Random-Access Memory
RMA	Remote Memory Access
SCM	Storage Class Memory
SLCNAND	Single-Level Cell NOT-AND Flash
SPMD	Single Program, Multiple Data
SLURM	Simple Linux Utility for Resource Management
SRAM	Static Random-Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
UPC	Unified Parallel C

Chapter 1: Introduction

1.1 Overview

Next generation HPC systems present a new host of challenges as critical applications drive the demand on computational performance, power efficiency, data transfer, resource capacity, and resilience. Future estimations indicate that power efficiency needs to be improved by up to three orders of magnitude and most of the aggregate system power is consumed by data motion [1]. Disparity between computational performance vs. network and storage device throughput is growing. Also there is an increasing disparity for energy costs of computational, memory, and communication operations. Several emerging technologies seek to close some of these gaps through increased parallelization via accelerators and low-energy, high-capacity persistent memory. However best practices for integrating and utilizing these technologies in a distributed computing architecture are not entirely understood by the HPC community. Simple attempts to bolt these mechanisms onto existing paradigms lead to increased code complexity.

1.2 Motiviation

For a number of years a supercomputing arms race has been raging between the United States of America, China, Switzerland, Japan, Germany, South Korea, Italy, France, Taiwan, UK,

Spain, Saudi Arabia, and India. The US Department of Energy (DOE) in collaboration with American partners lead an effort to maintain the USA as the preeminent world leader in technology and High Performance Computing (HPC). Hewlett Packard Enterprise Frontier (OLCF-5), hosted at ORNL, became operational in 2022 as the world's first exascale supercomputer, boasting a theoretical peak performance of 1.6 exaFLOPS [2] and 1.2 exaFLOPS Rmax [3]. OLCF-5 is also the top performing supercomputer in the world today according to the TOP500 [3].

In order to fulfill the demands for next generation HPC systems there are many new challenges to overcome. Emerging technologies are being developed as solutions to address some of these problems. Persistent or nonvolatile memory (NVM) provide a much larger memory capacity while requiring no energy cost to maintain the memory state unlike DRAM, which must consume energy when performing periodic refresh operations to prevent data loss.

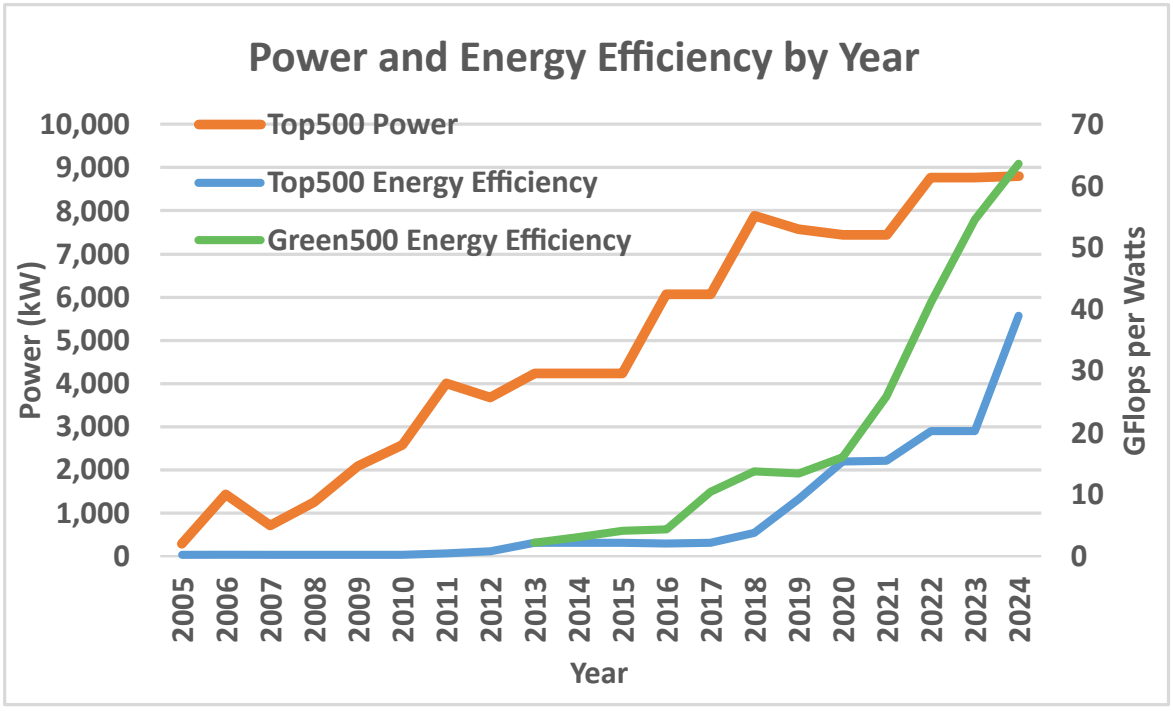

Fig. 1.1: Top500 and Green500 Median Power and Energy Efficiency of Top 10 Supercomputers by Year

Over the last twenty years, the demand for power has steadily grown. Figure 1.1 represents the median power and energy efficiency of the top 10 best performing supercomputers in the world by year, compiled from TOP500 [3] and GREEN500 [4] data. Power demand has increased linearly, while computational energy efficiency (GFlops/W) is on an exponential climb, likely due to the introduction and ever greater utilization of stream cores processing on GPU accelerators.

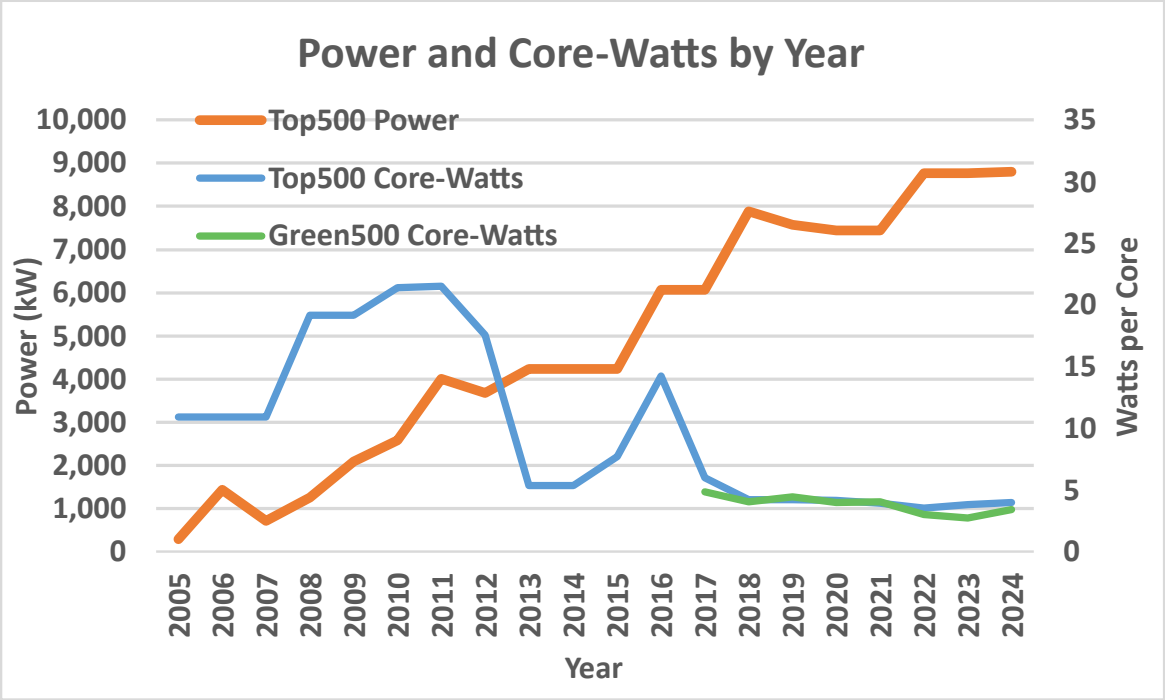

Fig. 1.2: Top500 and Green500 Median Power and Core-Watts of Top 10 Supercomputers by Year

However, a median ratio of power (W) to cores (core-watts) reveals very little has changed in recent years as illustrated in Figure 1.2. Since the increase in power demand is relatively proportional to the growth of the number of cores, per Figure 1.3, with more recent advancements, cores have higher FLOPS but still demand around the same amount of power. The proportional growth of power is not simply isolated to core count, since it is also accompanied by a need for larger memory capacity. With an ever increasing demand on power, I now seek to optimize the

energy efficiency for other parts of the architecture, such as memory.

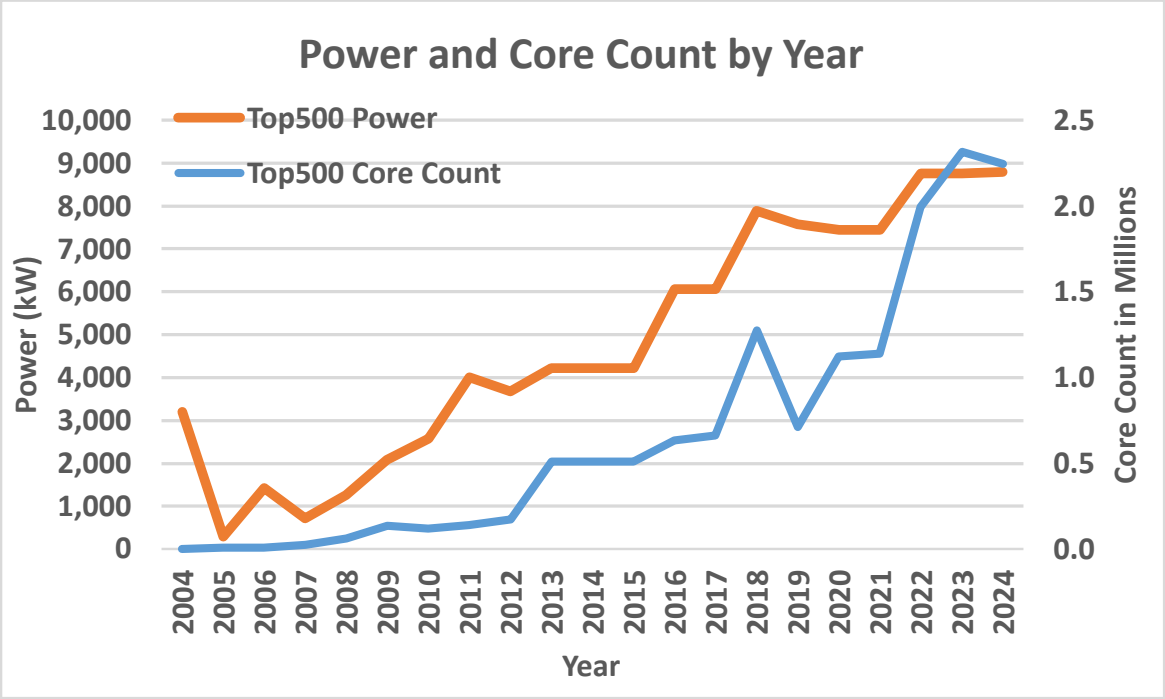

Fig. 1.3: Top500 Median Power and Core Count of Top 10 Supercomputers by Year

The introduction and increased reliance on emerging persistent memory technologies will contribute towards achieving this goal. However, researchers are struggling to integrate and utilize NVM technologies as resources in a distributed architecture. NVM support greatly expands the capacity of a distributed memory environment. Unfortunately a downside to the introduction of NVM in HPC is that it increases the burden of code complexity. In addition, NVM has higher read and write latencies and lower endurance than DRAM. More information on different NVM technologies and latency properties can be found in subsection 2.4.3. HPC systems comprised of a cluster of compute nodes with separate memory environments are easier to build. However this architecture places the burden on the programmer to manage data on each node.

A high-level parallel language with partitioned global address space (PGAS) support can

provide a foundation for addressing these challenges. Language features of high-level languages hide and manage underlying complexities while streamlining the development process. In addition, a PGAS solves the problem of programming for large distributed memory systems by providing a shared memory model. Chapel is a high-level, parallel, PGAS language distributed with an open-source compiler and runtime framework which I expanded to incorporate the utilization of persistent memory.

1.2.1 Thesis Statement

NVM technologies can offer larger capacity and lower energy consumption than traditional DRAM for HPC applications. The complexity and latencies of NVM can be minimized by using a high-level parallel PGAS language and effective use of traditional processor caches.

1.2.2 Thesis Contributions

1. I have extended the Chapel PGAS language to support DRAM-NVM hybrid-PGAS by adding a new persist type qualifier.

2. My extensions to Chapel provide transactional persistent memory with byte-addressable access to persistent objects.

3. I have added a checkpoint and recovery system for Chapel that uses my persistent memory extensions to allow effective user selection of state to checkpoint.

4. I extended my Purity data-centric profiling tool to support DRAM-NVM hybrid-PGAS profiling of memory and communication access patterns for Chapel runtime executions.

5. I conducted a case study using Purity to identify a bottleneck. The kernel was optimized based on the data from the tool, resulting in a speedup of 78x.

6. Using my cache performance profiling tool, I conducted a cache performance study.

7. A study considered architectural configurations that showed PCRAM, but not SLCNAND, could achieve performance similar to DRAM for a Chapel microbenchmark.

1.3 Structure

In Chapter 1, I introduced NVM as an emerging technology which is more energy efficient that traditional DRAM main memory and Chapel which has high-level language features that hide complex details of parallel HPC applications and can be extended to incorporate NVM. The remainder of the dissertation layout is as follows. Chapter 2 is the related work, where I compare recent publications to the research in this dissertation. Chapter 3 covers basic Chapel terminology and language features that are used throughout the dissertation. Chapter 4 demonstrates how I expanded the Chapel language to incorporate NVM in a hybrid-PGAS model by integrating persistent memory support into the Chapel compiler and runtime framework with the introduction of a new language feature and semantics, transaction system, and checkpoint and recovery. Chapter 5 introduces a data-centric profiling tool that I have developed for the Chapel language, which has been extended to support NVM profiling. In Chapter 6, serveral experiements were conducted to determine when NVM performance is similar to DRAM and to evaluate the efficacy of the checkpoint and recovery system. Chapter 7 contains the summary conclusion and Chapter 8 is the future work.

Chapter 2: Related Work

2.1 Persistent Memory

2.1.1 Disaggregated Memory

Disaggregated memory refers to a memory pool that has been decoupled from compute nodes and made accessible, expanding the memory hierachy as remote shared memory. Decoupling separates compute resources and memory, allowing for independent scaling of processing and memory capacity while reducing over provisioning of memory resources. However, the memory architecture relies on network access to transfer data to and from compute nodes, which can impact performance.

OpenFAM is an API that allows applications to utilize both volatile and persistent high-capacity fabric-attached dissaggregated memory (FAM). It can facilitate data sharing between compute nodes via shared memory. Each region represent a large heap, for which data items can be allocated and mapped into a processing element's (PE) virtual address space [5] [6] [7] [8] Access control is enforced via access permissions associated with both regions and data items. Access to specified regions and data items can be enabled for different parts of an application or other applications and are addressed via descriptors to avoid direct access to a global address space [9]. OpenFAM also handles synchonization issues for multi-threaded applications [10].

The Chapel language was extended to support FAM through the definition of a new distribution policy [11] [12]. A prototype FAM distribution module was developed to manage FAM-resident arrays which can translate high-level array operations into FAM accesses via OpenFAM APIs. It provides support for random access, bulk transfer, reduce, scan, slicing, and re-indexing operations. When a FAM array is created, the FAM allocation is partitioned and assigned to each locale and can support parallel array operations similar to a distributed array with Block distribution.

Amitha C. et al. [13] prototyped a record-based FAM pointer that utilizes OpenFAM APIs to support access to FAM-resident data objects through the Chapel language. FAMptr is an abstract language construct which hides the underlining FAM implementation. The record-based pointer maintains a data type, FAM handle / descriptor and FAM offset. It supports pointer assignment, pointer arithmetic operations, and read and write operations into FAM. The evaluation compared the performance between FAMptr and OpenFAM implementation through Chapel bindings and found that the FAMptr implementation did not contribute to additional overhead.

John Byrne et al. [14] developed a FAM dataset storage management system that utilizes OpenFAM support via a FAM distribution module [11] integrated into Arkouda to handle datasets that are too large to fit in the shared memory of computer nodes. Arkouda allows Python programmers to perform exporatory data analysis to solve HPC-scale problems by providing a Jupyter/Python frontend client with an API modeled after NumPy and an Arkouda backend server implemented in Chapel [15] [16]. It uses parallel distributed arrays or pdarrays which are composed of a series of in-memory data arrays that support filter, scatter, gather, add, subtract, and sorting operations over compute nodes. With OpenFAM support on the Arkouda server, batches of data and metadata stored in FAM are paged by a FAM dataset storage manager into pdarrays

as working data in local memory on compute nodes. Operations performed by Python programs produce derived FAM datasets which are tracked by a FAM dataset storage manager and data is sychronized between FAM, Arkouda server instances, and clients via incremental updates and an automatic update processor.

2.1.2 Distributed Memory

Command line interface plus Python (CLIPPy) [17] is a Python language interface, similar to Arkouda, which is designed to communicate with a backend execution on an HPC system with NVM. CLIPPy generates a Python API on the fly in a JSON format to run queries on big data over a cluster, which allows the backend execution to be loosely coupled and can be implemented in any language. CLIPPy uses Metall's [18] [19] persistent memory allocator based on mmap to store data in NVM. Since persistent data structures are stored in a NVM environment, expensive data format conversions between in-memory and file representations can be avoided when attaching, detaching, or reattaching NVM storage to an execution.

K.Wu and D. Li [20] developed Unimem which manages the placement of data objects in a DRAM-NVM heterogeneous memory system (HMS) for HPC over multiple nodes using MPI. Data placement is determined using profile characteristics of data objects derived from memory access patterns of execution phases, performance models, and hardware-based characteristics such as bandwidth and latency. Unimem API provides NVM allocation much like PMDK and tracks pointers for automatic selective data movement. The evaluation was conducted over 7 MPI / C benchmarks on 3 different HPC test platforms and compared Unimem with X-Mem [21]. Quartz [22] performance emulator was used to emulate NVM latencies and bandwith characteristics for basic performance tests and detailed performance analysis, while the scalability

study relied on NUMA characteristics to emulate NVM accesses. The results demonstrate that the performace gaps between DRAM and NVM can be largely mitigated through data management that limits unnecessary data movement, indicating strong viability of NVM in future HPC platforms.

2.1.3 PGAS Shared Memory

OpenSHMEM is an API for developing single program, multiple data (SPMD) applications that utilize shared memory over a distributed computing architecture via a partitioned global address space (PGAS) and a one-sided remote direct memory access (RDMA) from SHMEM [23] [24] [25]. Grodowitz et al. developed an API extension for OpenSHMEM, built on the UCX library that provides byte-level access to distributed persistent data, similar to RMA, and addresses network and storage performance issues while hiding file I/O details. The extension operates on a client-server model to establish a PGAS where PEs use a client API to connect to and access remote persistent memory stored via a symmetric file space across all PEs. Performance evaluation over a graph generation and decomposition benchmark concluded that the extension does not suffer from performance degradation [26].

2.1.4 Transactional Memory

TENET [27] is an extension of TimeStone [28], a persistent transactional memory (PTM) library, which provides support for spatial and temporal memory safety and fault tolerence. Spatial memory safety is guaranteed through Memory Protection Keys (MPK) and 8 byte canaries at the allocation boundaries. The MPK enforces read-only access to the NVM object pool outside of TENET and the canaries are checked during commit to detect when a spatial safety violation

has occurred. Temporal safety is enforced in the first reference of NVM objects through pointer tag validation by storing a tag in the upper 16-bits of each memory address to detect dangeling pointers. Data fault tolerance addresses uncorrectable media errors (UME) by replicating NVM page data.

NVL-C is a transactional programming abstraction, inspired by NV-heaps [29], that provides a small set of C language extensions that operate as first-class language constructs, type qualifiers, and runtime calls to support NVM programming while ensuring efficient, flexible, portable, and correct execution of NVL-C applications. The design goals for developing NVL-C are to provide a minimal and intuitive programming interface for accessing NVM, pointer safety for a new type of pointer class system which incorporates NVM structures, ACID transactions to avoid corruption of NVM data and ensure recovery from system failure, compiler and runtime efficiency for NVM operations, and a modular implementation for flexibility [30].

NVL-C consists of two parts. The first is a compiler supported NVL extended LLVM IR which is managed by NVL passes and benefits from additional safety constraints and diagnostics that ensure correct execution from common programming errors involving NVM. The second is a NVL runtime library which is built on top of Intel's pmemobj and libpmemobj libraries from the Persistent Memory Development Kit (PMDK), formerly known as NVML, for portability. The advantage of using the underlining library is that it supports a wide range of SCM (storage class memory) devices. Therefore no special hardware extensions need to be developed [30].

NVL-C is a programming abstraction that extends the C language to incorporate and manage persistent memory. NVL passes inside OpenARC perform instrumentation at the LLVM IR level and provide pointer safety for a new type of pointer class system that uses a virtual address space for NVM structures. NVL runtime contains a transaction system interface that enforces

ACID properties for operations performed on persistent memory and also provides undo logging for rolling back transactions [30] [31]. My approach is very similar to NVL-C in that I use wide references as a virtual address space for NVM structures and provide a transaction system with ACID properties. However, unlike NVL-C my transaction, checkpoint, and restore system must deal with parallelism over a distributed computing architecture.

OpenARC is used as the front-end compiler and translates NLV-C application source code to an extended version of LLVM. A series of NVM passes then perform optimizations on NVM operations and produces a standard LLVM which contains function calls to NVL runtime. Any LLVM back-end compiler can be employed to generate the binary for a target architecture [31].

During evaluation the authors discovered several optimizations. One of them is NVM pointer hoisting which hoists bare NVM pointers out of a loop-invariant computation to limit the number of conversions that need to be performed on the base virtual address of an NVM. Other optimizations include the aggregation of transaction data for backup clauses, avoiding unnecessary backups, and increasing the granularity of a transaction region [30].

NVL-C implements a number of transaction optimizations. Automatic canonical undo logging and automatic undo log aggregation provide the ability to roll back incomplete transactions by managing the previous states of NVM data with undo logs and can determine when only a single undo log is required. Other optimizations made are shadow updates for omitting extra synchronizations between logical and physical NVM, addressing write-first access patterns, and an abstract cost model [31].

MUMPS is a high-level, interactive, multitasking, general purpose language originally designed for the healthcare industry to address record management needs [32] [33] [34]. MUMPS stands for Massachusetts General Hospital Utility Multi-Programming System [33] and is also

referred to as M [34]. Though first developed as an interpreted language in the Octo Barnett's Laboratory [34] at Massachusetts General Hospital (MGH) in 1966 to 1967, nonmedical applications have been developed for areas such as banking, accounting, library information systems, inventory, and scheduling systems [33]. There is also a MUMPS operating system [34].

The MUMPS language provides transaction support with ACID properties through the application of commands and subscripted arrays and variables via an underlining database system. MUMPS relies on a built in general purpose database file system [32] that uses a Global Array file manager, schedule and I/O manager to provide ACID transaction access via subscripted arrays and variables [34]. Later the language was extended to incorporate PostgresSQL and MySQL as back-ends [35]. The MUMPS database organizes disk resident structures using a hierarchical model such as trees and sparse multi-dimensional arrays and provides key-value access via set, merge, and read commands [34]. All values are stored as string-oriented primitives [33] which support integer, floating point, and logical operations [34]. Global variables and arrays reside on the disk and continue to persist after the program ends [34]. Local variables on the other hand are stored in local memory and disappear upon the program's termination [34]. Originally MUMPS only supported single-line scope for variables but was later extended to facilitate variable scope within a block structure [34].

2.2 Reliability

2.2.1 Checkpoint Systems

Singh, et al. proposed JASS [36], an adaptive checkpoint system for NVM-based systems designed to reduce write amplification. Write amplification is a problem with flash memory

where unintended rewrites of user data occur as a consequence of wear leveling, reducing performance. The authors posited that high epoch size reduces write amplification and there is also an inverse relationship between checkpoint latency and write amplification. Based on these ideas they developed a checkpoint performance tuning algorithm which attempts to achieve an epoch size and checkpoint latency target while minimizing write amplification and not exceeding a checkpoint latency bound defined by the policyholder. The core of their approach relies on a coherent cache flushing scheme which was developed based on Chandy-Lamport, a snapshot algorithm for distributed systems [37] [38]. Tejas simulator was used to evaluate the performance of JASS compared with NVOverlay with 7 benchmarks resulting in from 2.3% to 96% reduction in write amplification. NVOverlay is a checkpoint system which uses multi snapshot NVM mapping and coherent snapshot tracking to account for memory changes across parallel systems while minimizing write amplification [39].

K. Kruger et. al. present DONUTS (don't pause the persistence) [40], a crash consistency mechanism that provides asynchronous checkpoints to NVM based on dynamic epochs and undo-logs. DONUTS utilizes processing-in-memory (PIM) for logging operations and to reduce data motion between the processor's memory controller and NVM. An epoch is comprised of three stages; the execution phase involving uncommited changes in volatile caches, the commit phase, and the persistence phase where all dirty cache lines and processor context are stored to NVM. DONUTS overlaps epochs to dynamically establish checkpoints by allowing an epoch in the persistent phase to be non-blocking. A modified cache LRU-R policy that evicts only clean blocks was employed to limit the epoch length while maintaining dirty blocks to avoid excessive NVM writes. An x86-64 architecture simulator was used to evaluate the performance of DONUTS over cache organization assoicative set threshold, different log row buffer sizes, and different write

buffer sizes. Results indicate a 50.6% write reduction to NVM when compared with PiCL and less than 1.8% runtime overhead.

2.2.2 Fault Tolerance

J. Ren et. al. introduced EasyCrash [41], a framework that selectively persists heap and global data objects to increase the possibility of recomputability of restarted crashed HPC applications due to faults, hardware or power failures. Application recomputability is defined as the possibility that an application recomputes the correct outcome after a crash. Spearman's rank correlation analysis was used to select critical data objects as candidates based on the correlation between data object inconsistent rate and application recomputability for a critical code region. Evaluation was performed on the entire NAS Parallel Benchmarks suite, botsspar from SPEC OMP 2012, kmeans from Rodinia and LULESH. NVCT PIN-based crash emulator was used to simulate three levels of cache, main memory, and a random crash generator. Several thousand crash and recompuation tests were conducted to reduce result variance to under 5%. To determine effectiveness of EasyCrash, application responses were categorized as successful recomputation without overhead vs with overhead, passes acceptance verification (not corrupted) but with extra iterations and failed verification. Analysis compared with vs. without EasyCrash, selected data objects vs. all data objects, selected code regions, and best recomputability. Best recomputability is defined as the persistence of data objects at each code region for each iteration of the major loop. Results demonstrate a tradeoff between relaxing crash consistency requirements and higher efficiency with a 20% improvement on average.

2.3 Profiling Systems

2.3.1 Data-Centric

A. Hassan et al. developed a profiling tool for a hybrid DRAM-NVM main memory system [42] [43]. Their tool informs developers how to optimally place their program data to achieve energy efficiency by measuring access latency to determine average memory access time as well as cache misses and write-back events that occurred for each object (i.e. program variables and memory allocations). They employed their profiler and GEM5 cycle accurate simulator to evaluate the efficacy of data placement of objects in a hybrid main memory environment using 14 different benchmarks. Their results found an energy savings of 69% to 81% with a performance degradation under 5% from their approach when comparing against RaPP [44] which saw 50% to 80% while exceeding a 10% performance penalty. The evaluation also considered placement granularity based on object size, lifetime, and number of accesses and compared the energy savings across RRAM, STT-RAM and PCM alternative NVM technologies. Finally they focused on data migration and discovered that migration is only beneficial if the object has a long allocation lifetime, is accessed frequently, and at any period of time access is concentrated to parts of the object. A. Hassan et al. also developed a hybrid memory API which uses their profile tool to gather object statistics for managing data placement in a hybrid memory architecture [45].

W. Wei et al. developed 2PP a software / hardware cooperative framework that employs an offline / online object placement scheme informed by profiler access statistics for hybrid DRAM-NVM memory [46]. The first strategy uses global access characteristics of heap objects which falls into a category of 'convergent' objects whose pages share a dominant read-write access pat-

tern to determine initial placement of data. The second strategy applies selective dynamic analysis to optimize the placement of 'divergent' objects with different page access patterns at runtime. They use energy efficiency (energy-delay product) and ratio of reads to writes (RWratios) metrics to inform data migration.

Vampir [47] [48] is a HPC I/O performance analysis and visualization tool developed and maintained by researchers at Dresden University of Technology (TUD) and later in collaboration with the NEXTGenIO project. It relies on Score-P [49] to sample, timestamp, and record application performance metrics at runtime into generated event logs. Vampir then uses these logs to construct several timeline and profile views. Since many applications rely on one or more high-level I/O library which depends on underlining I/O layers, a major goal in developing Vampir was to generate visual displays of multilayer I/O information over the entire I/O stack. These views include the Master Timeline, Process Timeline, I/O Timeline, I/O Summary, File I/O Metrics including a Performance Radar, and an I/O Filter.

2.3.2 Processor-Centric

ChplBlamer [50] is a multi-node performance profiler developed for the Chapel programming language. It uses blame analysis and propagation to designate a percentage of blame to program variables for various metrics such as CPU cycles, cache misses, and byte transfers based on variable relationships and interactions within the program's LLVM IR. It employs PAPI to sample hardware counters and libunwind to perform a stack-walk from a sampled point of execution to facilitate the blame. The analysis tool generates a load imbalance as well as both inclusive and exclusive blame views in order to identify performance bottlenecks that occurred during execution. Insights from the evaluation of three benchmarks using ChplBlamer led to globalization,

replication, and localization optimizations and a 4x speedup in LULESH, 1.05x in HPL, and 1.11x in ISx.

CUDABlamer [51] extends the use of blame analysis in ChplBlamer to target CUDA based GPGPU C applications for detecting performance bottlenecks in the GPU kernels. Since there is no GPU support for PAPI, NVIDIA CUPTI library was used to sample event counters in both the GPU hardware and CUDA drivers. CUDABlamer also relies on CUPTI's callback API for inserting analysis code before and after CUDART and CUDA driver functions to identify the data-dependency relationships between host and device memory to establish complete data-flow information while constructing the CPU-GPU calling context in order to facilitate blame. Evaluation of CUDABlamer's calling context construction over 16 GPU benchmarks from SHOC and Rodinia benchmark suites resulted in complete coverage in all but one benchmark. A case study of the Particlefilter benchmark using CUDABlamer led to the realization that if read-only arrays are stored in GPU constant memory, a 46x speedup could be achieved. Similarly with the Triad benchmark by aggregating multiple operations per thread, much overhead associated with thread creation and destruction could be eliminated leading to a 1.2x speedup.

2.4 Optimizations

2.4.1 Latency Hiding

T. Rolinger, et. al. [52] presented a compiler generated inspector-executor optimization for Chapel PGAS programs, which targets read-only irregular memory access patterns within parallel loops. To keep with the high productivity of the Chapel language, a static analyzer identifies opportunities for optimization within a set of language constraints through non-affine, interproce-

dural and alias analysis and automatically applies the code transformations. The inspector builds a communication schedule for each parallel loop over a distributed array by identifying which memory accesss are remote. The executor uses the communcation schedule to replicate remotely accessed data to eliminate redundant remote accesses to the same elements. Performance evaluation was conducted with NAS-CG which implements a sparse matrix-vector multiply (SpMV) and PageRank applications, yielding runtime speedups of 52x on Cray XC and 364x on Linux with Infiniband. Though the evolution of Rolinger's research [53] focused on manual implemention of the inspector-executor technique and [54] introduced the compiler generated optimization. [55] explored the use Chapel's Replicated distribution module to address irregular access patterns in breath-first search and PageRank applications and compared the results with a manually implemented aggregate optimization variant and MPI / OMP versions. The results concluded the optimizations led to speedups of up to 1219x for BFS and 22x for PageRank.

T. Rolinger, et. al. explored the optimization of irregular access patterns through data replication and thread migration via Emu system architecture [56]. The Emu system was designed based on Cilk to support an implicit migratory thread architecture for PGAS language environments to address communication and cache-unfriendly memory access behaviors, such as irregular accesses [57]. Each compute node is subdivided into highly multi-threaded nodelets which are managed by a migration engine. Emu also supports data replication on every nodelet to eliminate the need for thread migration for certain memory address ranges. T. Rolinger, et. al. prior work introduced a distributed block placement optimization to address read access latency by selectively replicating block data based on dynamically profiled access patterns, prioritized to achieve the largest performance gain [58]. The current work extends the authors' replication optimization strategy to support write operations as well. Evaluation was performed on SpMV

and SpGEMM kernels leading to a speedup of 4.2x and 6x respectively. T. Rolinger, et al. also presents an adaptive prefetch optimization approach for addressing fine-grain communications of irregular access patterns. The adaptive heuristic uses six adjustable parameters in the prefetch optimization. Performance evaluation over IG, SpMv, PR, and SSSP workloads across five systems demonstrated an improvement of up to 377x [59].

2.4.2 File System

EchoFS [60] [61] is a user-level file system developed by NEXTGenIO EU that allows HPC applications to benefit from an NVM storage layer transparently by hiding I/O stack complexity. A data scheduler coordinates SLURM with echoFS which supports legacy applications and provides an API as well for non-legacy applications. EchoFS constructs a collaborative burst buffer over compute nodes by joining NVM regions assigned by a batch job in SLURM. It automatically manages data location by loading input files into NVM prior to job execution and writing persistent output files generated by an application from NVM back to the parallel file system (PFS) when a job has completed. EchoFS uses pseudo-random file segment distribution to balance the workload when managing NVM I/O file operations. Since output files are staged in NVM, peak I/O operations can be absorbed to avoid PFS bottlenecks while hiding latency and reducing overhead when managing many small files.

2.4.3 Low-Level

NVM management techniques can reduce write latency and energy usage when initiating phase change memory (PCM) SET operations well in advance. A SET operation involves heating crystalline PCM cells from a high-resistance to a low-resistance state so that a write operation

can be performed. In order to split SET and write operations and hoist the SET operations to occur earlier, data is required to predict where and to what locations writes will occur [62] [63].

Wear-leveling algorithms have been developed to increase lifetime performance by using high level file abstractions from the OS and flash translation layer (FTL) in order to deduce update frequency and regency for better wear-leveling [62] [64]. Another technique balances writes by allocating a larger number of physical pages for logical pages with higher write counts for PCM [62] [65]. A third approach tracks the erase frequency of blocks and remaps logical blocks with low update regency to blocks with high regency [62] [66]. DRAM can be used as a thin layer to manage performance and wear-out limitations of emerging NVMs [62] [67]. S. Akram et al. developed a write-rationing garbage collection system to extend the life of NVM by placing write-intensive objects in DRAM and read-intensive objects in NVM [68].

Faulty block and failure detection techniques can reduce the raw bit error rate by periodically performing read operations, using ECC to correct page errors in Flash, and remapping them if necessary [62] [69].

Persistency and consistency techniques include persisting static variable declarations [62] [70] and separating volatile and non-volatile data to preserve consistency [29] [62]. Persistent memory or non-volatile memory (NVM) requires no standby power in order to retain its state unlike DRAM. The persistency of NVM is not only energy efficient, it also allows for the use of persistent data structures between runs, increasing application reliability through system-wide defensive checkpoints that reduce the latency of I/O operations by avoiding having to traverse the interconnection network in order to access a storage area network [71]. However, the typical drawback to NVM is poor write speed and write endurance. This is particularly true for NAND single-level cell (SLC) flash memory, a currently widely deployed NVM, that has a write latency

of five orders of magnitude slower than DRAM with less than a third of the endurance [62] [70]. Emerging NVMs such as phase-change memory (PCM), (magneto) resistive memory (MRAM, ReRAM), spin-torque transfer memory (STT-RAM), as well as other memory technologies such as molecular memory, ferroelectric field-effect transistor (FeFET) memory, and carbon-based nanotube memory attempt to close these gaps [62] [70] [72].

To leverage the best features of both SSD and HDD, one technique is to move only the performance-critical blocks from HDD to SSD [62] [73]. Additionally low cost, high capacity HDD and high cost, low capacity SSD can be managed at the same memory hierarchy through a file storage scheme that moves files based on their access pattern [62] [74]. Other techniques include storing read-intensive references in SSD while HDD records differences between accessed and reference blocks [62] [75], utilizing initial block allocation and migration [62] [76], using statistical modeling to reduce random writes on SSD [62] [77], and employing SSD for random reads and HDD for sequential reads for virtual memory management to achieve accelerated page swapping [62] [78].

2.4.4 Energy

A major concern for next generation exascale computing is how to best change the energy footprint to better utilize energy resources while mitigating the growing need for energy consumption. Dynamic random-access memory (DRAM) is used in virtually all modern computing architectures. While pervasive, DRAM main memory is one of the largest components of power consumption for high-performance computing (HPC) systems with an estimated upper bound of 30 to 50 percent, second only to the processor. In order to improve the ratio of computational power to power consumption, exploration into alternative energy efficient memory solutions is

essential. Unlike DRAM, persistent or non-volatile memory (NVM) requires no standby power and can increase application reliability. The target goal for next generation exascale computing is 20 MW at 1 Eflops or 50 Gflops/W, requiring an improvement of several orders of magnitude in power efficiency [71]. In 2022 the Hewlett Packard Enterprise Frontier (OLCF-5) launched at Oak Ridge National Laboratory (ORNL). Each node consists of 512GB of DDR4, 4TB of flash memory, and 128GB of GPU memory [79]. Frontier has a total of 9,408 compute nodes [79], boosts a peak performance of 1.6 exaflops [2], and has an aggregate system power of 22,786 kW [3].

NV-SCAVENGER [80] is a binary instrumentation tool that statistically analyzes the access patterns of memory objects in the heap and stack, and feds a memory power and performance simulator in real-time. Four scientific applications were chosen for the evaluation. Simulation results indicated a power savings of at least 27% while the performance loss from a memory latency increase of 20% was negligible, double latency resulted in 5% loss, and high latency had an upper bound of 25% degradation.

Chapter 3: Chapel Language

3.1 Overview

In this chapter I describe a bit about the Chapel programming language and features of the language that will be used or modified in the work described in the rest of this dissertation. I also provide an overview of the Chapel compilation strategy and runtime libraries Chapel stands for Cascade High Productivity Language and was developed by Cray, Inc as a part of DARPA's High Productivity Computing Systems (HPCS) program. It's a high-level parallel programming language and framework that operates over a shared memory environment for high-end parallel systems. Chapel provides intuitive parallel language constructs that are designed to increase the developer's productivity when writing High Performance Computing (HPC) applications while better separating algorithmic and data structure implementation [81].

3.2 Locality

In Chapel a ***locale*** can be described as an abstract unit of parallel architecture capable of multithreaded computations and can access local memory uniformly [82]. The term ***single-locale*** refers to the use of a single shared memory compute environment where only one locale handles the entire program's execution. On the other hand, ***multi-locale*** refers to an array of locale units

connected via a cluster of network-connected nodes, usually assigned as one per compute node [83]. Different mappings are possible, such as performing a multi-locale execution using two or more locales on a single compute node. Chapel relies on a primary node to begin execution and handle the main / sequential parts of a Chapel program. Chapel provides global variables ***LocaleSpace***, ***Locales***, and ***numLocales*** for managing top-level locales.

3.3 Data Parallelism

In Chapel, data parallelism performs multithreaded, concurrent data operations in parallel on distributed logical datasets over multiple processors and compute nodes. The Chapel language provides a ***forall*** loop construct to allow developers to explicitly implement parallel iterators at a high level that utilize data level parallelism [84] [85].

3.3.1 Range and Domain

A Chapel ***range*** represents an interval for a sequential set of integer values that facilitate the iterations of for, forall, and coforall loops. A ***range*** can either be bounded or unbounded and contain properties for low bound minimum value, high bound maximum value, step or stride indicated with ***by***, and ***align*** for alignment with the base range. If the stride is positive then the alignment is based on the low bound and if negative then the high bound. A finite sequence of integers can be generated for unbound ranges by using the count operator indicated by **#**. There are also range intersection, shift, and comparison operators. Range intersection slices one range by another, producing a new range and for range shift the value will be added to the bounds of the new range [86]. Listing 3.1 provides range examples and the sequential integer sets they produce.

Listing 3.1: Examples of range and domain

```
var range1 = 0..10 by 2 align 0;  // 0, 2, 4, 6, 8, 10
var range2 = 0..10 by -2 align 1; // 9, 7, 5, 3, 1
var range3 = 3.. by 3 # 5;        // 3, 6, 9, 12, 15
var range4 = (0..10)[..6 by 2];   // 0, 2, 4, 6
var range5 = (0..10) + 3;         // 3..13
forall i in range1 { ... }

const denseDom: domain(2) = {1..n, 1..m};
var sparseDom: sparse subdomain(denseDom) = [(2,3), ..., (63, 91)];
var sparseMatrix[sparseDom];
forall x in sparseMatrix { ... }
```

The ***domain*** is an index set constructed from one or more ranges. It serves as one of the cornerstone language features of data parallelism since domains are used to define the size and shape of both local and distributed arrays, which given a domain become iterable. Chapel provides support for rectangular domains, strided domains, subdomains, sparse subdomains, associative domains, and opaque domains [81]. A sparse subdomain should be populated with values based on the parent domain's index type and is otherwise empty by default [87]. Both ranges and domains can be used to determine the characteristics when declaring an array in Chapel.

3.3.2 Distribution and Domain Map

Chapel is a partitioned global address space (PGAS) language that relies on a partitioned, shared memory model over compute nodes. Chapel developers often employ built-in distribution strategy modules for mapping the physical location and management of their data structures on the PGAS. For instance block distribution (***BlockDist***) uses a bounding box domain to partition indices into blocks for mapping onto either a specified set of target locales or otherwise all locales. Block distributions can be either 1D or multi-dimensional, in which case each index is

represented by a tuple of values. It also provides support for sparse subdomains [88]. Cyclic distribution (***CyclicDist***) uses a round-robin approach beginning with a starting index parameter value for mapping a distribution over a specified or unspecified set of locales [89]. A ***domain map*** is an iterable layout or distribution over a data structure's indices and elements that determines on which locale they reside and how they are accessed and iterated [90]. Domain map is referenced by the ***dmapped*** clause which facilitates the definition of a mapped domain type and how a domain will be distributed [90].

Listing 3.2: 2D Dense Matrix-Matrix Multiplication using Block Distribution (Chapel v1.22)

```
use BlockDist;
use Random;

config const n = 1000;
const S = {1..n, 1..n};
const D = S dmapped Block(boundingBox=S);
var A, B, C: [D] real;

fillRandom(A);
fillRandom(B);

forall (i,j) in D do
   for k in 1..n do
      C[i,j] += A[i,k] * B[k,j];
```

Listing 3.2 illustrates how to apply a domain to define 2D matrices that are block distributed over locales. 'n' is a constant that can be configured (i.e. set) at the beginning of the execution. The constant 'S' is inferred as a 2D domain literal by assignment and is composed out of two ranges from 1 to n. It is used when defining the distributed domain 'D'. 'Block' and 'boundingBox' denote how the domain is mapped over locales (e.g. compute nodes). The resulting domain instance is used to instantiate three block distributed two-dimensional matrices of a million ele-

ments each. Two-tupled indices are used to perform matrix multiplication over the domain 'D'. Azad et al. [91] derives a sparse matrix-vector multiplication (SpMV) implementation in Chapel.

3.3.3 Forall Loops

In Chapel a ***for*** loop represents a serial execution via a single task over an iterator method such as a range, domain, or array. A ***forall*** loop provides explicit support for data level parallelism, where iterations are mapped to tasks (e.g. units of work) based on its iterand expression and executed in parallel [92]. The ***zipper*** iteration feature allows support for serial and parallel processing of two or more iterators of the same size within a loop, where the schedule is determined by the first iterand [92]. Chapel supports ***reduction*** operations and user-defined class extensions to reduction to allow efficient consolidation and aggregation of values from different locales (separate memory spaces). Reduce intents refers to a reduction operation that uses any variable declared outside of the scope of a forall loop [93].

Listing 3.3: Example of forall, zipper iterator, and reduce.

```
config const n = 1000000;
var U, V: [1..n] real;
var s: real = 0;

forall (u, v) in zip(U, V) with (+ reduce s) do
   s += u * v;
```

Listing 3.3 demonstrates a dot product between two vectors, implemented via a reduction operation which explicitly uses a forall loop and zip iterator over the elements of vector U and V. For each iteration, the zip iterator provides the element values in a tuple, which are multiplied together and added to the sum. Since the forall loop is parallel, the reduce operator and variable

need to be specified to avoid data race conditions.

Listing 3.4: Implicit Parallel Operation with Promotion

```
s = (+ reduce (U * V));
```

Listing 3.4 performs the same dot product calculation as in listing 3.3. However, in this case the multiply and reduce operators use a term in Chapel's data-parallelism known as a promotion. A promotion occurs when a function or operator expecting a scalar instead receives an iterable with the same element type [84]. For this example, an implicit parallel process multiplies all of the coorsponding elements for vectors U and V and then performs a summation over the output vector.

3.4 Task Parallelism

In contrast to data parallelism, task parallelism focuses on the distribution and parallel processing of tasks over multiple processors and nodes. A task is an abstract computational unit of work that is often processed along with other tasks during parallel execution. A primary task is responsible for running the main parts of a Chapel program and exists throughout the entire duration of execution. Most tasks however are short-lived and can be nested inside other tasks. The program developer can create tasks through the use of Chapel language constructs such as a ***coforall*** loop, ***begin***, and ***cobegin*** statements. [94]

3.4.1 Begin and Cobegin

The ***begin*** statement is a prefix that allows the execution of a child statement (e.g. a block statement) by creating an unstructured task that runs along with the task that called it. The ***begin*** task is non-blocking, allowing the parent task to continue with execution. However, because of this the two tasks will not be synchronized. Chapel also supports nested tasks with the ***begin*** statement [95].

In contrast, the ***cobegin*** statement relies on concurrent, unsynchronized, heterogeneous task execution over a series of statements. For each statement within the ***cobegin*** block a child task is created to execute it's corresponding statement in parallel. Unlike ***begin***, ***cobegin*** does not allow the parent task to continue until all children tasks immediately related to the ***cobegin*** have completed. Nested ***begin*** statements embedded in a ***cobegin*** statment are not subject to this rule and therefore will execute tasks asychronously [96].

3.4.2 Coforall Loop

In a ***coforall*** statement, homogeneous tasks are created from loop iterations for parallel execution. In contrast to the ***forall*** statement which maps iterations to tasks, in ***coforall*** a unique task is created for each iteration of the loop. Each task will execute the body of the loop concurrently. Similar to ***cobegin***, the ***coforall*** statement will not allow the parent task to continue until all directly related tasks are complete [97].

Listing 3.5 is an example of a coforall statement over the Locales array. The numLocales and the Locales array are provided by internal Chapel modules. The coforall creates a separate task for each locale, specified by loc. The on clause will designate the given task to be executed

Listing 3.5: Example of coforall, on, and locales.

```
coforall loc in Locales do
   on loc do
      writeln("Node ID: ", loc.id, " out of ", numLocales);
```

on the locale provided, which in this case prints the identifier of the locale that processed the task.

3.4.3 Thread Safety

Chapel provides support for atomic variables via the ***atomic*** type qualifier which can be applied to bool, int, uint, and real primitive types. Atomics include thread-safe fetching and non fetching operations for addition, subtraction, bit-wise or, bit-wise and, and bit-wise xor as well as read, write, exchange, compare and swap, and test and set methods [98]. A sync variable denoted by a ***sync*** type qualifier, operates as a mutex that blocks the current task from proceeding if read when the current state is 'empty' or writing when already 'full'. It can be used as a lock or to create a barrier. By convention a '$' suffix should be added to the name for identifying a sync variable [99]. ***Atomic*** and ***sync*** variables will reside on one locale and are enforced when accessed remotely over the PGAS.

3.5 Framework

Compiler: The Chapel compiler is a powerful tool designed to produce scalable HPC parallel applications written in the Chapel language. Since Chapel uses high-level expressions to describe complex parallel operations, the compiler must deal with the implementation and optimization details to scale for every execution scenario involving computational resource availability. Chapel version 1.22 comes with a 42 pass compiler and Chapel version 2.0 compiler has

41 passes, as serveral passes were combined in the more recent update. The Chapel compiler provides two backend options for producing a binary, a LLVM backend compiled with clang and a C code generation which can be compiled with a C compiler such as gcc.

Modules: Chapel modules extend the Chapel language by providing additional functionality such as distributed mapping strategies and serves as a Chapel / C interopability conduit for allowing the instrumentation and execution of runtime subroutines. When a Chapel program is compiled, the Chapel source files (e.g. user modules) are combined with internal Chapel modules dependencies. In addition, Chapel provides a ***use*** clause to specify modules that the program source relies on (e.g. use BlockDist).

Runtime: The Chapel runtime is a framework that relies on internal implementations and third-party libraries when providing different options for memory, task management, and communication. It is also responsible for the Chapel program's execution via a compiler generated main function that calls user module initialization and the user main.

The Chapel runtime provides network ***cache*** support to reduce communication overhead of remote operations when accessing distributed structures over the PGAS. This option can be enabled in the Chapel compiler by using ***cache-remote***. The cache provides aggregation, write behind, and read ahead features at runtime which can cut down on the number of remote communications [100].

A ***wide reference*** is a pointer on the PGAS which is comprised of a locale identifier and memory address defined through a ***localeModel*** and can also be extended for NUMA support.

3.5.1 Task Management

Chapel provides two task layer options for managing tasks, FIFO and Qthreads. **FIFO** task management uses pthreads to execute tasks concurrently. Tasks are queued in a pool where each one is retrieved by a thread once the previous task is complete. The number of threads per locale (e.g. compute node) can be controlled explictly through environment variables. By default one thread per CPU core will be created and live for the entire duration of the program's execution. Threads that are not executing tasks will remain idle until a task becomes available [101]. FIFO tasking is fully implemented in the Chapel runtime framework, making it ideal for monitoring and performance profiling at the task or thread level.

The **Qthreads** API was developed by Sandia National Laboratories and provides a POSIX implementation for managing tasks [102]. It is included in the Chapel distribution as a third party library and utilized by Chapel's runtime task management layer. Tasks are managed by shepherd threads and delegated to worker threads. Worker threads can either be distributed over CPU cores or processing units [101].

3.5.2 Communication Layer

GASNet was developed by Lawrence Berkeley National Laboratory as a communication framework that supports remote memory access and active message for PGAS languages. GASNet extended API provides 'conduits' or device specific implementations for OpenIB and OpenFabrics for InfiniBand, Cray XC Aries, MPI, UDP, and simulated multi-locale. PGAS get and put operations are facilitated by one-sided, RMA communications. Chapel provides GASNet in its distribution and can be configured through environment variables [103] [104] [105].

Chapter 4: Persistent Memory in the Chapel Language

NVM is an emerging techology that offers persistency, lower power usage, and a much larger capacity than DRAM main memory but at a slower latency, at least for now. In this chapter, I will discuss the introduction of a new type qualifier $persist$, language semantics, and compiler passes as well as transaction and checkpoint systems that utilize Intel's Persistent Memory Development Kit (PMDK) to provide NVM support for the Chapel PGAS language.

4.1 Persistent Memory Semantics for the Chapel Language

The type qualifier $persist$ introduces an explicit programmatic approach for supporting persistent memory objects in the Chapel language. This type qualifier can be applied to Chapel records, classes, unions, tuples, ranges, domains, associative domains, sparse domains, domain maps, arrays, and primitive types. It can be used on variable or constant declarations and in conjunction with other declaration prefixes such as $config$ or $extern$ and type qualifiers like $atomic$ or $sync$. However, $persist$ must precede other type qualifiers. The persist type qualifier can also be applied to the new keyword when instantiating classes and records. Classes already provide a memory management strategy using $owned$, $shared$, $unmanaged$, and $borrowed$ qualifiers and have been extended to support persistent memory. Persistency works for both explicit and implicit type declarations. Listing 4.1 illustrates the use of $persist$ type qualifier.

Listing 4.1: Adding the Persist Type Qualifier

```
var a : persist atomic real;
var s$ : persist sync int;
var c = new persist owned MyClass();
```

During runtime, persistent memory objects will be allocated for user-defined variables that use the *persist* type qualifier and stored in a nonvolatile memory space. Through PMDK, persistent memory objects can be accessed in a byte addressable way consistent with how volatile memory is accessed in Chapel. If persistent memory has been disabled or the framework configuration does not include PMDK then persist type qualifiers in a Chapel program will simply be ignored by the Chapel compiler when generating the binary.

Listing 4.2: Partial Persistency is not Allowed

```
// Incorrect Usage
record Point {
   var x : persist real;
   var y : real;
}
var p : Point;

// Correct Usage
record Point { var x, y : real; }
var p1 : Point;
var p2 : persist Point = {0.5, 2.3};
var p3 = new persist Point(1.0, 2.0); // Implicit Persistency
var p4 : persist (real, real);
```

Since class and record instantiations can exist in volatile space, which in the unlikely event of a runtime fault will be lost, applying the persist qualifier to fields is not allowed. Instead, persistency should be applied to the entire class, record, union, or tuple, which will automatically make all fields and subfields persistent. An example has been provided in listing 4.2.

Listing 4.3: Consistency with Persistency

```
config const n : int = 100;
var D : persist domain(2) = {1..n, 1..n};
var S : persist sparse subdomain(D);
var A : persist [S] Point;
var B : persist [1..5] int = [6, 7, 2, 4, 5];
```

Chapel domains and arrays are interconnected. Domains are used to declare arrays and when the bounds of a domain are dynamically altered all associated arrays are automatically reallocated. Therefore the persist type qualifier must be consistently applied by the developer to both domains and arrays when storing array memory objects in a nonvolatile memory space. If an array is persistent then any associated domains must also use the matching persist type qualifier and vice versa. The persist keyword must precede any array brackets and will allow all array elements to take on the persistent property. Also, any sparse domain derived from a persistent domain must also consistently use the persist type qualifier. See listing 4.3 for examples.

Listing 4.4: Implicit and Explicit Persistency

```
config const n : int = 32;
config const m : int = 1024;

// Implicit
var r = 1..n;
var s = 1..m;
var A : persist [{r, s}] real;

// Explicit
var P : persist domain(1) = {1..n};
var Q : persist domain(1) = {1..m};
var R : persist domain(2) = {1..n, 1..m};
var B : persist [R] real;
var C : persist [P, Q] real;
var D : persist [P][Q] real;
```

Ranges do not need to be persistent in order to be used to declare a persistent domain. However in the event a range is used to declare a persistent array, the internally generated domain associated with that array will implicitly take on the persist type qualifier. Listing 4.4 show examples of how range can be used to build domains and arrays. Although illustrated, due to issues with internal complexity persistency in double arrays are not supported at this time.

Listing 4.5: Inheritance and Persistency

```
use BlockDist;
config const n : int = 100;
var D : persist domain(1) = {1..n};
var B : persist D dmapped Block(D);
var A : persist [B] real;
```

Finally, domain mapping strategies for distributed PGAS data structures designated to be stored in nonvolatile memory space must use persistent domains in their declarations. Memory objects associated with a distributed array on the PGAS will be stored in persistent memory on their assigned nodes consistent with the chosen mapping strategy. This assumes the architecture provides homogeneous support for NVM.

The *persist* type qualifier should only be applied to global declarations. Otherwise the entire stack frame hierarchy over all threads would need to be stored in persistent memory in order to properly map and restore local persistent allocations bounded to function calls. Therefore, there is no support for local persistent declarations at this time. However, in the future I may consider supporting local persistent declarations in a similar way to how *static* variables operate in the C language. In this scenario, a persistent allocation would be perserved beyond the scope of the invocation and accessible in subsequent calls.

4.2 Supporting Persistent Memory in Chapel

In order for developers to make the best use of NVM resources, my solution incorporates persistent memory into the Chapel PGAS language by providing an intuitive, programmatic approach centered around the ideas of portability and ease of use while encapsulating implementation details behind the Chapel runtime framework. Applications that declare user-defined variables with the $persist$ type qualifier will have byte addressable access to allocated persistent memory objects on the PGAS managed in the same way Chapel handles DRAM allocated heap objects. This methodology provides a seamless and transparent process for developers to utilize persistent memory resources.

I present Loci, a persistent memory library which utilizes PMDK for NVM support while hiding the underlining complexity and I/O NVM interface. The name 'Loci' was derived from the 'method of loci' or 'memory palace', highly regarded by ancient philosophers as a strategy for improving one's ability to recall information, particularly during the discourse of debate. Analogous to this method, my Loci library enhances the memory capability of the Chapel language by introducing NVM support to create a hybrid-PGAS memory space that can scale over a parallel, distributed computing architecture. Loci employs the libpmemobj library from PMDK to utilize NVM technology and can even emulate NVM by turning the hard drives of compute nodes on a cluster into a persistent, byte addressable, partitioned, shared memory environment. The Loci library has been integrated into the Chapel runtime framework and provides an interface which is streamlined for compiler instrumentation of various runtime functions that implement persistent memory. New passes have been introduced into the Chapel compiler that utilize an internal persistent memory module to instrument NVM support via Loci runtime subroutines.

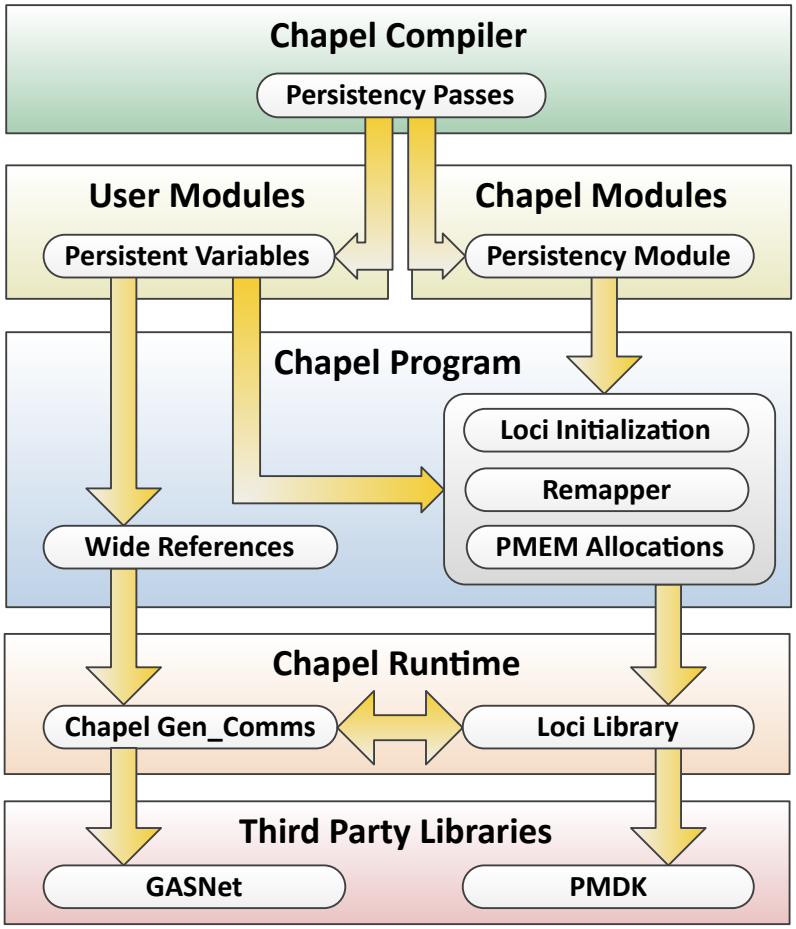

Fig. 4.1: Overview of the Chapel Framework with Persistent Memory Support

Figure 4.1 illustrates how the Loci library and persistent memory fits into the larger picture of the Chapel language. When compiling a Chapel program the Chapel compiler combines the program's source files (user modules) with relevant Chapel module dependencies. During run-time execution, all dynamic allocations for user-defined variables that are stored on the PGAS are accessible through wide references. A wide reference is composed of a locale id and a virtual memory address, which is used to access both local and remote shared memory resources. The subsequent subsections will explain how persistent memory is incorporated into this model.

4.2.1 Persistent Compiler Passes

The semantics of the language and compiler have been expanded through the introduction of a new language abstraction and type qualifier that allows a developer to easily and intuitively utilize persistent memory. By adding $persist$ type qualifier to variable declarations in the Chapel language, the compiler semantics will generate the appropriate memory access calls for the corresponding memory components on the same compute node and across compute nodes over the PGAS. This addition still supports legacy code since allocations made from declarations that do not use the $persist$ type qualifier are designated to be stored in volatile space by default. If the distributed computing architecture either does not provide resources to host NVM or Chapel support for NVM has been explicitly disabled by the Chapel pmem enable environment variable described in subsection 4.2.1, the $persist$ type qualifier is ignored and main memory is used instead. From the developer's perspective, providing the option to utilize NVM through a general Chapel type qualifier for variable declaration is sufficient enough to provide a portable and programmatic approach to allocating and utilizing different types of memory resources.

Three new Chapel compiler passes were introduced to facilitate static analysis and instrumentation of persistent memory for Chapel applications. The first pass instruments placeholder runtime calls to procedures in the persistent memory module that will be used much later in the advance stages of the compiler, after wide references have been inserted. The persistent memory module acts as a conduit between compiler generation and runtime functionality of persistent memory. These placeholders are added to the compiler generated main function immediately after its creation, to absolutely ensure these functions are not pruned by intermediate compiler passes. Also a compiler directive $no\ prune\ extern$ has been added to preserve Chapel to C

interoperability $extern$ declarations between the persistent memory module and Loci runtime structures and functions.

The primary persistency pass occurs right after the function and intents resolution passes. This persistent memory pass is responsible for building a definition list based on persistent user-defined variable declarations identified by FLAG_PERSIST (introduced by GNU Bison while parsing the Chapel source), acquiring the AST of the persistency module, adding the Loci deinitialization call to close out the NVM heap, generating NVM program root and memory context structures, and instrumenting memory context switch. Instrument context switch is responsible for finding all call expressions to Chapel runtime memory allocation functions, inserting controls for executing Loci persistent memory allocation, instrumenting root assignments to user-defined global variables inside the primary user initialization module, and adding memory context switching. Due to unforseen complexities, both top down and bottom up propagation of memory context along the call paths between assignments in user module initialization and Chapel allocation calls could not ensure that all allocations were performed in the correct designation. Therefore, memory context switching was added as a catch all solution. Once defined with a persistent heap identifier all subsequent memory will be allocated in the specified persistent memory space until the memory context is switched back to main memory.

The final persistency pass was added right after the insert wide reference pass (remotely addressable references across locales). The primary role of this pass is to resolve persistent allocation calls to Loci for wide reference types and generate remapping functions for all effected structures. It is also responsible for generating an entry point function for initializing the Loci library, which is executed very early inside the initialization of the Chapel runtime framework.

4.2.2 *Byte Addressable Persistent Transactional Memory*

The Loci library utilizes libpmemobj from PMDK to create and manage transactional persistent memory object stores on compute nodes for a DRAM-NVM hybrid-PGAS environment. PMDK's libpmemobj is a persistent memory transactional object store library implemented in C that provides functionality for dynamic allocations, transactional operations, and management using direct access (DAX) to persistent memory objects [106] [107] [108]. By employing libpmemobj functionality, Loci serves as an intermediate layer in the Chapel runtime framework for handling persistent transactions.

Listing 4.6: Loci Transactional Dynamic Allocation of Persistent Memory

```
void* loci_alloc(...
   chpl_sync_lock(&loci_heap->alloc_lock);

   TX_BEGIN(loci_heap->pool) {...
      *oidp = pmemobj_tx_alloc(len, type_num);
      addr = pmemobj_direct(*oidp);
      ...
   } TX_ONABORT {
      perror("ERROR: Transaction aborted in loci_alloc");
      fflush(stderr);
      ...
      *oidp = OID_NULL;
      addr = NULL;
   } TX_END
   ...
   chpl_sync_unlock(&loci_heap->alloc_lock);
   ...
   return addr;
}
```

Listing 4.6 is a code snippet from Loci's transactional allocation of a persistent heap object. The libpmemobj library provides access to a transactional object store via a persistent memory pool and performs all necessary mmap operations to a heap file behind the scenes [106] [109].

Code block execution is available at each stage of the transaction's life cycle [110]. Once begun, a transactional allocation to NVM returns a PMEMoid value (e.g. object handle) to a persistent object. If the transaction aborts, then the error is logged and the object handle is set to OID_NULL.

A PMEMoid is a globally unique persistent memory object identifier that represents an object that has been allocated from a persistent memory pool. Given an object handle, the pmemobj_direct() function will calculate a byte addressable pointer in virtual memory to the persistent object via DAX. When this address is returned, the program and runtime framework will handle the reference to the persistent object in the same way main memory heap objects are managed. Assigning the direct pointer to a wide reference allows for a uniform programmatic approach for accessing both DRAM and NVM objects on the PGAS. Therefore, from the perspective of the Chapel runtime framework, Chapel program, and developers, access to persistent objects will be seamless and transparent. With this approach the Chapel runtime framework is flexible enough to handle both DRAM and NVM objects when executing a hybrid PGAS application.

Listing 4.7 shows the partial implementation of Loci's transactional memmove function. All memmove calls in the Chapel gen_comm subroutines have been replaced with Loci transactional memmove invocations which handle both DRAM and NVM addresses. However, the transaction macros are only used when either the source or destination address references a persistent object. The purpose of the gen_comm functions is to provide access to objects via wide references and are instrumented throughout a Chapel program by the compiler to handle both local and remote accesses to shared memory resources. During the transactional operation in Loci, memmove is called and upon commit the destination is flushed back to memory. An explicit commit request can only be made during the work stage [110] of the current open transaction [111]

Listing 4.7: Loci Transactional Memmove Function

```
void* loci_tx_memmove(...
   ...
   if ((loci_heap->status & HEAP_USE_TX_SYSTEM) == 0)
      return memmove(dst, src, len);

   // check to see if src and / or dst are in persistent memory
   src_in_heap = loci_addr_in_heap_range(heapID, src, len);
   dst_in_heap = loci_addr_in_heap_range(heapID, dst, len);

   // perform a regular memmove if both src and dst are in DRAM
   if ((src_in_heap == 0) && (dst_in_heap == 0))
      return memmove(dst, src, len);

   TX_BEGIN(loci_heap->pool) {
      if (dst_in_heap)
         pmemobj_tx_add_range_direct(dst, len);
      ...
      ret = memmove(dst, src, len);
      ...
      if (dst_in_heap)
         pmemobj_tx_commit();
   } TX_ONCOMMIT {
      if (pmem_is_pmem(dst, len))
         pmem_persist(dst, len);
      else
         pmem_msync(dst, len);
   } TX_ONABORT {
      perror("ERROR: Transaction aborted in loci_tx_memmove");
      fflush(stderr);
      ...
      ret = NULL;
   } TX_END
   ...
   return ret;
}
```

and only the thread that began the transaction has access to it. In the event that the transaction is aborted, an error message will be generated and the function will return a NULL value. For example, if given an invalid address that references an object which has been previously freed then the transaction will abort.

To manage concurrency, Chapel provides atomic and sync language features to developers and also uses them within its internal modules and runtime framework. GASNet is a communication layer option in Chapel which also handles concurrency and prevents data race conditions for remote accesses to shared memory on the PGAS. The transactional object store support from libpmemobj ensures operations for wide references to persistent objects have ACID properties. When combined, concurrency of persistent memory can be enforced across all compute nodes. The implementation of a transactional persistent memory with ACID properties is important for the NVM-based checkpoint and recovery system in section 4.3.

4.2.3 NVM and SLURM Configuration Options

Environment variables were added to control various features of persistent memory and the checkpoint system, as illustrated in both this subsection and in subsection 4.3.2. This allows developers the ability to customize their runtime executions without needing to recompile the program. The only exception to this is the toggling of persistent memory support as it requires compiler instrumentation.

CHPL_LOCI_PMEM_ENABLE: Setting this environment variable to true enables the persistency compiler passes to instrument persistent memory support into a Chapel program. Undefined or a value of false disables NVM support and any $persist$ type qualifier that exists in the Chapel source will be ignored. Thus, all allocations will be made to main memory instead. To take effect,

enabling and disabling persistent memory requires the Chapel program to be recompiled.

CHPL_LOCI_PMEM_HEAP_SIZE: This variable sets the byte size of the NV heap file for each compute node without the need to recompile the Chapel program.

CHPL_LOCI_DEBUG: Logs additional information generated by the Loci library.

Chapel provides the ability to compile a program launcher binary that will generate and launch a SLURM job using sbatch. However, the options for configuring the sbatch file were rather limited. Out of necessity, slurm-gasnetrun_ibv launcher in Chapel has been expanded with new launch options for building sbatch files.

CHPL_LAUNCHER_NODE_ACCESS=oversubscribe: By default, Chapel generates and launches sbatch jobs with exclusive access to all resources on a compute node. However, if not all resources were intended to be used then allocated CPU hours will be unnecessarily spent rather quickly. Therefore, an option was added to allow Chapel executions to share compute node resources with other SLURM jobs.

CHPL_LAUNCHER_MEM_PER_CPU: Setting this environment variable specifies the amount of memory needed per CPU when launching a Chapel application.

CHPL_LAUNCHER_MEM_PER_LOCALE: As an alternative to specifying memory per CPU, the amount of memory needed can be specified per locale.

4.3 Checkpoint System

I have developed a checkpoint module and runtime functionality that provides an interface to checkpoint operations for Chapel developers who want to control how checkpoint performs in their program. This module serves as a Chapel / C interoperability layer for binding and

invoking the underlying runtime checkpoint routines implemented in the Loci library. It provides the developer with the means to programmatically specify where and how often a checkpoint should occur in their program. In the event of a system failure, recovery from the last restore point will automatically occur in the subsequent execution. Checkpoint should only be invoked within a sequential part of the program.

When checkpoint occurs, all persistent objects are flushed to the NVM environment and the active NV heap files for each compute node are stored at a predesignated backup location. This designation can be set before the execution begins using the shared basepath environment variable in subsection 4.3.2. Device basepath provides the option to designate active NV heaps to any desired location, not just on the compute nodes. NV heap files not only contain all persistent memory allocations, but also a map of all metadata to persistent objects from the last execution which is accessible from the persistent memory pool's root object upon loading the file. This bookkeeping is responsible for mapping both local and distributed persistent objects to their respective or new compute node resources in the subsequent execution. PMDK's libpmemobj library is used to manage the NVM structures respective to their NV heap files.

The transaction system maintains an account of persistent allocations, object handles, and their associated virtual memory references. If the execution prematurely terminates, a subsequent run will retrieve the last restore point, map and load NV heap files for each compute node and perform remapping operations over the distributed allocation map to provide the program with new virtual references based on object handles. Since PMEMoid object handles are globally unique, theoretically they can be referenced from separate heap files across compute nodes. However, in this approach their derived virtual references simply need to be updated by their respective locale.

4.3.1 How it Works

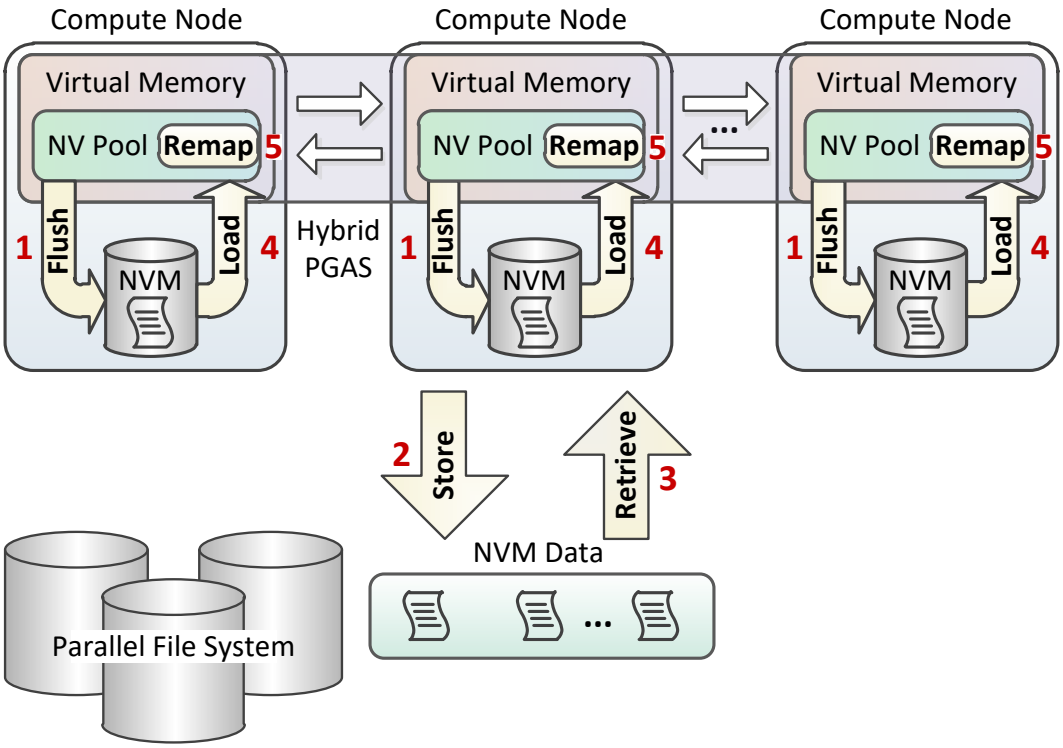

Fig. 4.2: Backup and Restore of NV Heaps

My checkpoint system was designed to handle NVM local to compute nodes. Figure 4.2 illustrates how checkpoint and recovery works in Chapel to achieve an exact state reconstruction. The numbers represent the steps performed for checkpoint and restore operations. During the checkpoint operation, all persistent memory objects in the NV pool are flushed to NVM on each compute node. Then NV heap files are stored to a predesignated shared storage environment such as a parallel file system. This is important because in the event that a compute node fails, the data needs to be recoverable. Additionally, subsequent recovery will likely be given a different map-

ping of computing resources. Thus NV heaps must be restored in a way that is consistent with the NVM-PGAS environment of previous execution mapped to the new configuration. Then the NV heap on each compute node can be loaded as an NV pool in the program's virtual memory on DRAM. Finally to complete the restoration, at the beginning of a consecutive run any references to persistent memory objects between NV pools across compute nodes will also need to be remapped before execution can commence.

Reference remapping requires a comprehensive approach involving additions to both the Chapel compiler and runtime framework. When a persistent object is created, the persistent pointer (PMEMoid) for that allocation along with it's direct pointer are recorded to the NV heap. A direct pointer can be computed from a persistent pointer using pmemobj_direct() provided by the libpmemobj library from PMDK and is a byte-addressable virtual memory reference to a persistent object that exists in an NV pool during a program's execution. The Chapel compiler generated program code uses pointers to reference all heap allocations. To minimize the number of changes to the Chapel framework, a direct pointer is returned when a persistent allocation is made. The problem comes into focus when an NV pool is loaded in a consecutive execution and the direct pointers to persistent objects from the previous execution are no longer valid. A persistent pointer for the referred persistent object is required to produce the new direct pointer address. The remapping process attempts to resolve this for all persitent objects. This involves processing all of the allocations recorded to the NV heap and following through structures and substructures of each persistent object via compiler generated mapping functions. The allocation bookkeeping allows direct pointer resolution via reverse lookup to find it's corresponding persistent pointer. Wide references that contain direct pointers to persistent objects must also be resolved across NV pools on other compute nodes.

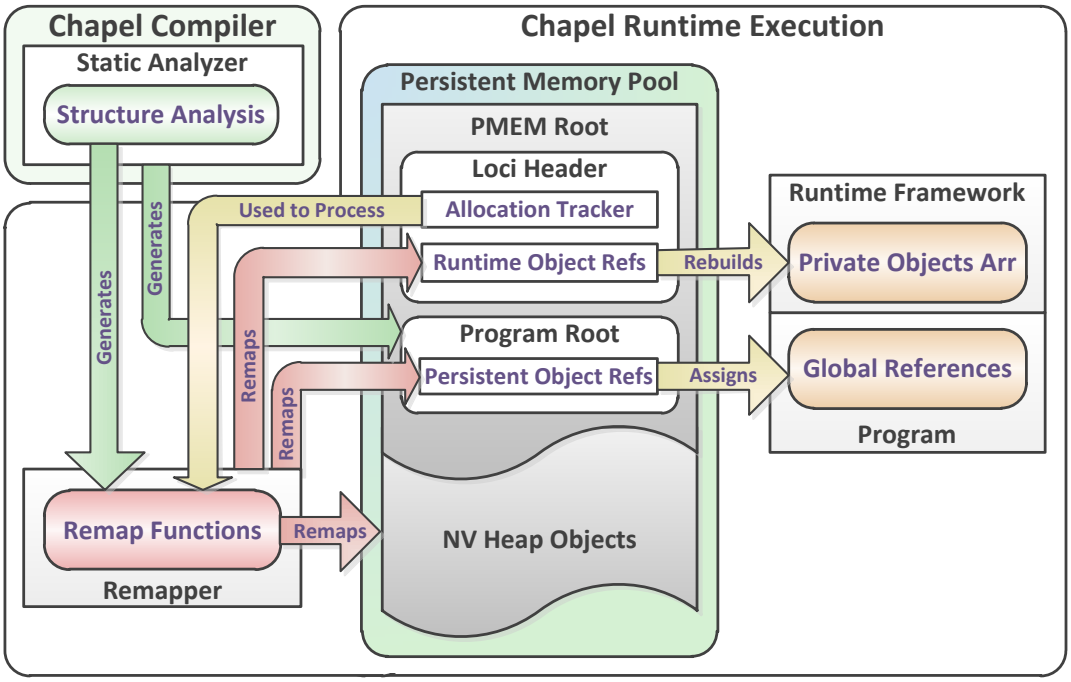

Fig. 4.3: The Remapping Process during a Subsequent Execution

Figure 4.3 details how the remapper works. A persistent pass performs static analysis to identify aggregate types that are associated with either the compiler generated program root or runtime memory allocation call expressions, to establish a list of starting points. A remap function is generated to handle the fields of each aggregate type and new aggregate types that are discovered along the way are also added to this list to be processed.

At runtime, remapping only takes place in a consecutive execution. My persistent memory library must be loaded as early as possible to handle runtime framework heap objects, such as the Chapel private objects array. However, initializing the library requires the size of the compiler generated persistent program root to be known. Therefore, support was added to the Chapel compiler via a new persistent pass that generates an entry point function into the program to initialize persistent memory, which is called inside the Chapel runtime initialization. The remapping process takes place very early in the execution. Heap allocate globals and compiler generated main are several entry points into the program from the Chapel runtime framework that are called later on. However, instead of allocating persistent objects, remapped direct pointers are assigned to the program's global references of user-defined variables by the persistent program root. The rest of the compiler and runtime then handles the byte-addressable persistent objects in the same way as volatile heap objects.

4.3.2 Checkpoint Configuration Options

CHPL_LOCI_PMEM_ENABLE_REMAPPER: Setting this environment variable to true or false will enable or disable the remapper for checkpoint and recovery. When enabled, corresponding NV heaps can be retrieved and loaded for the remapping process in a subesquent execution. Otherwise when disabled, each execution will add it's process id as a suffix to the NV heap file

naming convention. This allows for simultaneous executions of the same Chapel program on the same compute node resources without heap interference.

CHPL_LOCI_PMEM_STORE_RUNTIME_OBJECTS: This variable informs the Loci persistent memory library whether or not to store Chapel runtime objects such as privatization lists in persistent memory for remapping in a subsequent execution. This environment variable is ignored if privatization has been disabled through the Chapel compiler.

CHPL_LOCI_PMEM_ENABLE_TIMERS: Enables logging of timing values for various operations within the checkpoint and recovery process.

CHPL_LOCI_DEVICE_BASEPATH: The device basepath environment variable specifies the path where the active NV heap file will reside for each compute node.

CHPL_LOCI_SHARED_BASEPATH: Shared basepath specifies where all NV heap files will be stored when checkpoint takes place. Although not required, storing these files in a location remote from the compute nodes provides the option for immedate recovery in the event a compute node fails and is not accessible for a period of time. Stored heap files are given a '_backup' suffix to allow the device and shared basepath to be the same.

4.4 Future considerations

Future work could involve extending support to heterogeneous architectures where only a subset of the compute nodes provide NVM support. However, new domain mapping strategies will need to be developed to store the persistent allocations of distributed user-defined NVM structures on the PGAS while ensuring persistency. This will introduce new challenges for task and data locality. Therefore, for the scope of this research I have focused on homogeneous

environments.

My current approach to transactional memmove performs a commit and flush at the end of each persistent write operation. However, according to the PMDK documentation [111], transactions can be nested inside of other transactions so long as they pertain to the same thread. This provides an opportunity to explore an optimization strategy for the transaction system. In addition to the transaction macros illustrated in listings 4.6 and 4.7, libpmemobj provides functions to begin, commit, end, and post process a transaction [111]. Although a forall loop is parallel, the Chapel compiler will generate a corresponding function that contain a sequential loop, which is executed per thread. Therefore, through careful instrumentation, transaction functions (via Loci) can be used to establish an outer transaction over a sequential loop, thereby allowing commit and aggregate flush operations to be performed at the end of that loop, but still within the same thread. This is important because the data should be maintained in a cache for as long as possible to hide the latencies of slower memory and improve the overall performance.

Chapter 5: Profiling System

As parallel programming languages become higher level and powerful language features abstract details away, the job of the developer to implement HPC applications becomes easier. However, when high level parallel programs exhibit performance issues, understanding the root of the problem becomes more difficult because of all the abstraction layers. Adding new memory types such as the persistent memory that I proposed in this dissertation only further complicates performance tuning. In this chapter I describe a data-centric profiling tool that I have developed to help identify performance problems in the Chapel language for traditional DRAM-PGAS environments and in sections 5.3 and 5.4 I explain how my tool has been extended to profile both DRAM and NVM on a hybrid-PGAS.

5.1 Single-Locale Performance Optimizations in Chapel

† The purpose of my study was to examine the runtime performance of Chapel, how it compares to competitive parallel frameworks, and find methods for speeding it up. Ultimately, I investigated optimization techniques to achieve two goals. First, I provided insight into better development practices for Chapel programmers. Second, I developed optimization strategies that can be incorporated as automated solutions in future Chapel distributions. Since the research focus is on Chapel performance over an intra-node / multi-core architecture, all of my studies

were conducted in a single-locale environment.

I decided to use parallel benchmarks because they represent real world applications for scientific computing. Furthermore, by utilizing a diverse set of benchmarks more opportunities for improvement can be found. This is generally true because various developers will employ different implementation strategies and usage of the Chapel language features. I conducted case studies on four benchmarks. Each benchmark was manually profiled to identify bottlenecks in performance. I also examined gaps in performance between Chapel benchmarks and another parallel framework. I chose OpenMP to establish a baseline of comparison since it is THE standard for parallel computing on single-node environments for scientific applications. Next I explored optimization techniques for eliminating bottlenecks and closing gaps. Finally I evaluate the performance gain over the original Chapel benchmark, competitive framework benchmark, and impact across other Chapel benchmarks. [112]

5.1.1 Benchmark Case Studies

LULESH stands for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics and was developed by the Lawrence Livermore National Laboratory. It is a widely used parallel proxy application that performs hydrodynamics calculations. In order to find performance bottlenecks the OpenMP version of LULESH and the C generated code of LULESH from the Chapel compiler were manually instrumented to determine which routines the program spends the most time in. The resulting profiles were analyzed to identify underlining common characteristics where performance of the language itself could be improved.

I discovered that the biggest performance loss I found in LULESH occurred due to the declaration of large data structures inside its subroutines. Every time a routines is called, these

† Earlier versions of these sections appear in R. Johnson et al. [112] and [100].

Listing 5.1: LULESH: CalcHourglassControlForElem()

```
proc CalcHourglassControlForElems(determ) {
    var dvdx, dvdy, dvdz, x8n, y8n, z8n: [Elems] 8*real;
    forall eli in Elems {
        ...
```

declarations request large allocations on the heap, for which every element is subsequently initialized to zero. These local memory allocations in the generated code correspond to just five lines in the LULESH Chapel source and make up 22.8% of the total wall time. In particular, the first line of code in CalcHourglassControlForElems comprises of 18.8% of the program time as a result of 6 local arrays being dynamically allocated on the heap (calloc) every time the function is called, with 1248 invocations for a problem size of 30. Since the Chapel language can allow for changes to the domain size between invocations, cases can arise where reallocation is required. However for LULESH the problem size is determined at the command line. Thus in this case the domain and memory size never change, which means optimization is possible. To explore more efficient strategies, three LULESH C generated variants were produced from the Chapel code, modified, and evaluated. Figure 5.1 shows the performance of the three variants. The y-axis represents a speedup in performance.

In order to address reoccurring allocation bottlenecks without forcing the programmer to resort to using globals, I recommend two techniques. First, automated allocation hoisting provides a table for storing and retrieving the memory of large recurring local allocations in the runtime framework. Additionally it could store all compiler generated metadata structures associated with each allocation. Second, conservative memory Initialization take a conservative approach towards initializing memory elements to zero while maintaining the semantics of the

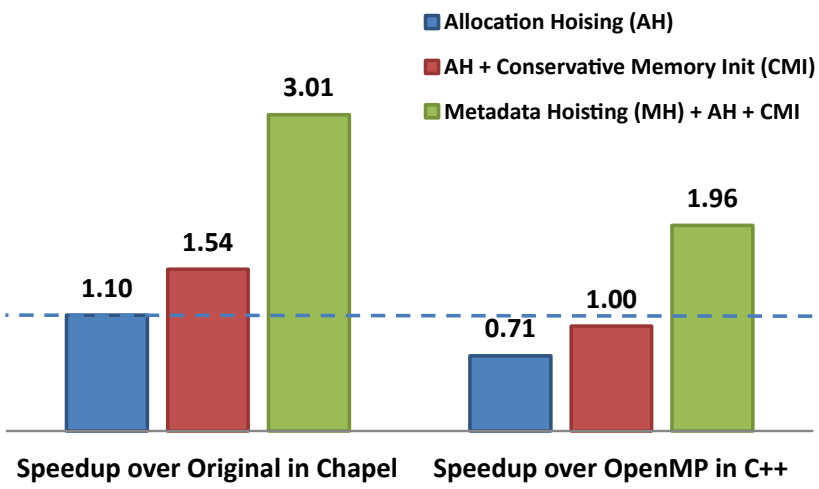

Fig. 5.1: LULESH Performance

language. For each allocation, determine in an automated way whether the programmer made an assumption about an element's initial value in the subsequent code. This can be accomplished by adding a static analyzer to the Chapel compiler. Optionally, I could provide compiler support for an explicit language feature similar to static in C with the added benefit of conservative memory initialization.

MiniMD is a parallel benchmark for computing molecular dynamics and is available in Chapel and C++ using MPI and OpenMP. The C++ version was used as a baseline comparison to identify inefficiencies in the code generated by the Chapel compiler and discover ways to close the gap.

My analysis revealed that the major performance pitfall in MiniMD is the mapping between domains in nested loops because domain mapping results in the generation of large, complex, and expensive code, which is best to be avoided. My programmatic optimizations reduced the Chapel version from being around 10 and 12 times slower than the OpenMP version to being roughly 2.7 and 2.3 times slower. I am still investigating further ways to close the gap.

5.1.2 Generalization of Optimization Techniques

To summarize, I found 9 different areas where improvements can be made. There may not be a lot of overlap between benchmarks. However, as more benchmarks are added the coverage will increase.

The purpose of this study was not to make benchmarks faster, but rather to learn of common barriers to performance in Chapel by looking at benchmarks. The following suggestions from the analysis could be used to improve the Chapel compiler and performance of Chapel applications. Manage local heap allocations with large domains with allocation hoisting. Avoiding domain remapping for iterating nested loops. Better management of parallel redundancies in nested loops. Finally, provide developers a way to manage ”thread’ level persistent heap allocations.

Degradation	**LULESH**	**MiniMD**	**SSCA#2**	**CLOMP**
Reoccurring local allocations	X			
Thread / task private allocations			X	
Adaptive memory reset	x			
Redundant memory init_elts#	x	x	x	X
Redundant autoCopy / autoDestroy	x	x		
Redundant parallelism			x	
Domain remapping overhead		X		
Application bottleneck			X	
Memory structure				X

X: Major impact, x: Minor impact

Tab. 5.1: Overlap and Impact of Bottlenecks

My Chapel-specific optimizations of LULESH, MiniMD, SSCA#2, and CLOMP microbenchmark variants led to speedups of 3.0x, 5.3x, 6.3x, and 4.8x respectively with a 2.0x for LULESH and 1.7x for CLOMP performance gain over their OMP/C counterparts.

5.2 Multi-Locale Data-Centric profiling for PGAS Languages

Based on the lessons learned from the studies in section 5.1, I developed Purity [100] which is a fine-grain, data-centric, communication profiling system for the Chapel language to identify PGAS bottlenecks. Our primary goal for the design was to develop a comprehensive tool that can analyze and profile memory and communication access patterns over a multi-locale PGAS environment. To effectively profile PGAS access patterns, communications are monitored during runtime execution and mapped back to their respective source variables. Each communication operation contains references to a local and remote memory address. These addresses can be resolved if I establish an association between allocated memory ranges and variable definition identifiers. Generating this map will involve both static and dynamic analysis.

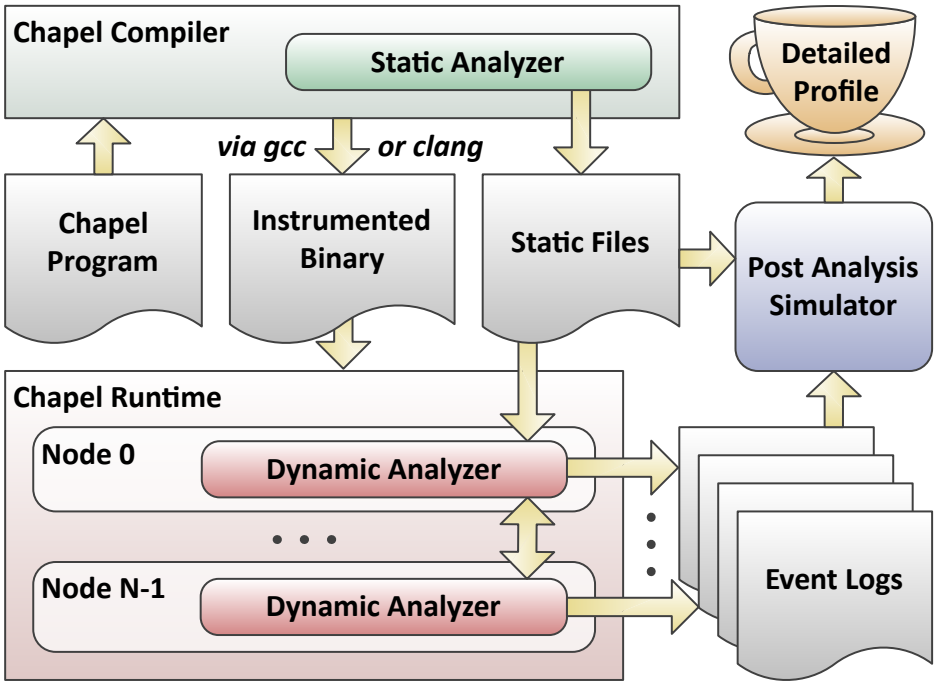

Fig. 5.2: Purity: Pipeline Overview

In Figure 5.2, Purity consists of three parts, a static analyzer, a runtime dynamic analyzer, and a post analysis simulator. The integration of Purity into Chapel involved the modification of the compiler with the introduction of five new passes for static analysis and a profile module. Incorporated into the runtime framework were the dynamic analyzer, a C implemented thread-safe ADT library, and a signal processing layer.

5.2.1 Static Analysis

The primary job of static analysis is to identify candidate source variables defined in the user modules which are relevant to remote communications by analyzing the AST structures, employ profile propagation to persist relevant information for later passes, and instrument the binary through the Chapel compiler with runtime calls after PGAS wide references have been inserted. It also generates static files detailing the program's definitions and loop hierarchy maps to be used in dynamic and post analysis.

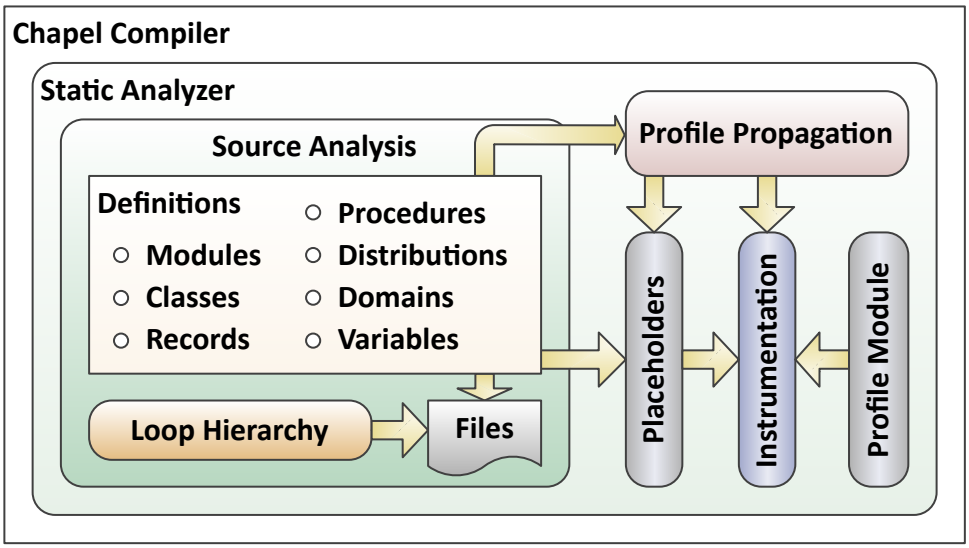

Fig. 5.3: Purity: Static Analysis

The static analyzer in Figure 5.3 is comprised of a source analysis pass, profile propagation, two placeholder passes, and two instrumentation passes. It also incorporates a profile module into the Chapel program. The profile module provides functionality for launching, updating, and finalizing dynamic analysis across the network during runtime. Through C interoperability with Chapel, the profile module also allows the compiler to link instrumented call expressions to dynamic analysis functions in the runtime framework. The profiling system supports both Chapel compiler back-ends, native C and LLVM.

Profile propagation was designed to persist profile information between compiler passes, disambiguate the different definitions of source variables when multiple definitions are generated by the compiler, and when required, spread contextual annotations down the chains of compiler generated temporary variables that originated from interactions with one or more candidate variables. The annotations of temporary variables aid in the instrumentation of candidate variable definitions and 'new' allocations in later passes so that memory addresses can be properly associated with the correct definition identifiers at runtime.

5.2.2 Dynamic Analysis

Dynamic analysis is responsible for associating memory ranges with variable definitions for use with mapping the remote addresses of communication operations during profiling. However, the prospect of building and maintaining a map of the entire partitioned global address space (PGAS) during runtime could range anywhere from cost prohibitive to downright intractable, depending on the program's needs and the number of nodes involved. Thus, in order to mitigate the impact on system and network resources that profiling may incur, dynamic analysis must be performed independently on each node. However, doing so leaves each analyzer with only a

partial view of the PGAS. A unified view is required in order to resolve the definition identifier of a remote address for the corresponding communication. Therefore, each dynamic analyzer records the memory and communication operations of its host for post processing.

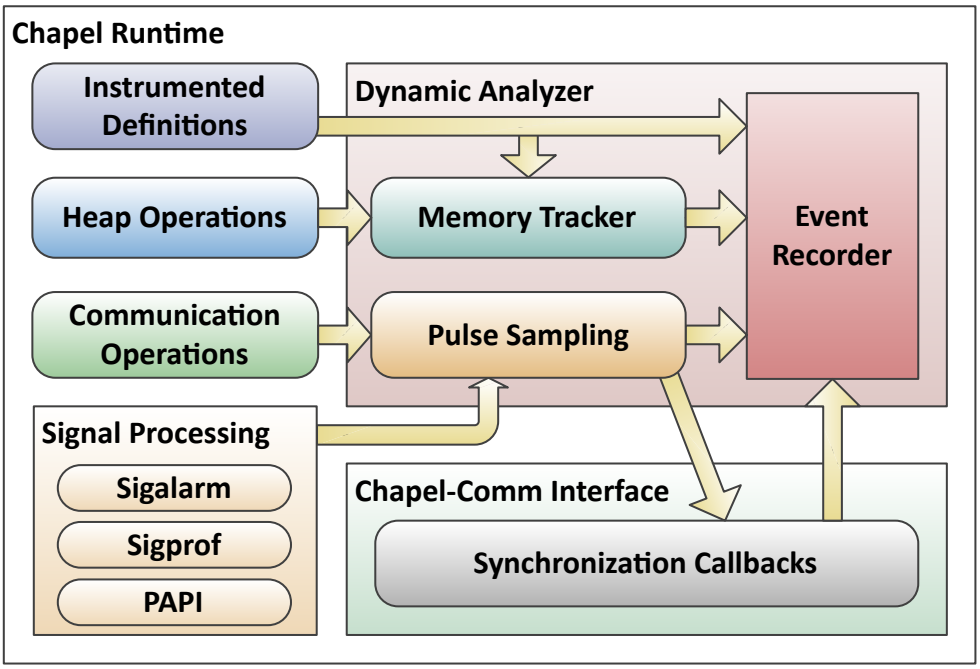

Fig. 5.4: Purity: Dynamic Analysis

The dynamic analyzer in Figure 5.4 comprises of a memory tracker, pulse sampling for scalability, a decentralized synchronization strategy, and an event recorder. Pulse sampling provide an easy way to scale back the volume and thus the recording of runtime operations. It works by turning on (pulsing) detailed data collection for short brusts of time to gather a representative sample of the data. The pulsing is controlled by a signal handler that is triggered by an alarm that goes off at a user configurable rate to allow control of the pulse width and duty cycle of the sampling. When enabled the analysis overhead and file size of generated event logs can be reduced. Synchronization allows post analysis to model the memory state of each node for resolving communications. To achieve this, a monotonically increasing synchronization identifier

is recorded with each operation and exchanged through piggy backing remote communications between nodes for synchronizing the event logs. At runtime each node employs a dynamic analyzer to monitor memory and communication access patterns for wide references on the PGAS and will produce event data based each nodes view of PGAS operations. Local, prefetch, network cache, and remote get and put operations for heap and when necessary associated stack variables are sampled during the program's execution to produce a high-level aggregate online report and event logs for post analysis.

5.2.3 Post Analysis

The post processing reads in the static files generated by the static analyzer in the compiler and the event logs produced by the dynamic analyzers on each node at runtime, performs a series of analyses, and produce a detailed profile for each execution of the instrumented Chapel program. The profile consists of a series of comma delimited files which provide a summary ranking variables by percentage of remote operations, loop analysis, request aggregation and byte throughput report at the node and flat index level, and a coverage report.

Figure 5.5 illustrates how the simulation of a profiled execution works in post analysis. The event data from each node is combined through post analysis simulation to produce a fine-grain profile of the execution. The profile consists of three parts. First it generates a summary of aggregate counts by operation category associated for each source variable which are ranked by percentage of total remote operations. Secondly, operations and source variables are broken down further through loop analysis to illustrate their operational impact on the PGAS for each loop in the program. Finally, request aggregation and byte throughput analysis generate node and flat PGAS index matrices for each source variable.

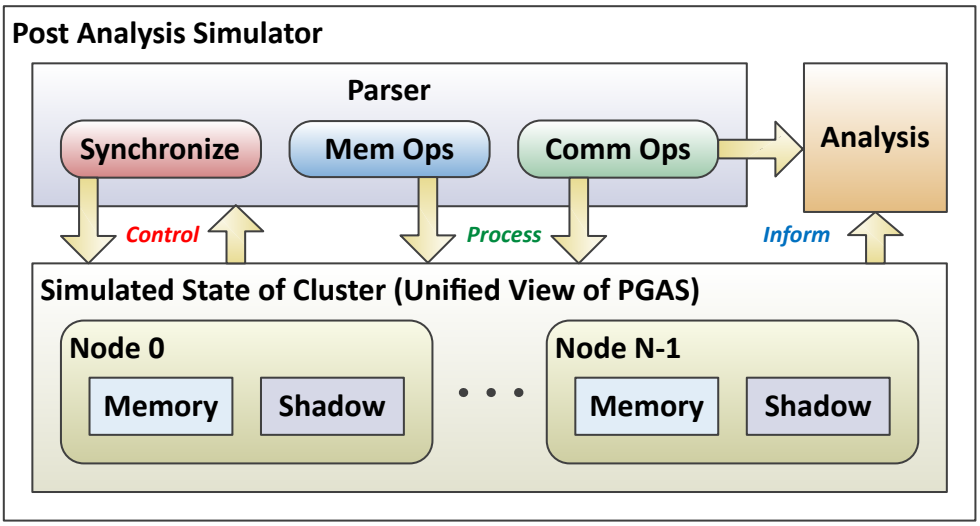

Fig. 5.5: Purity: Post Analysis

5.2.4 Evaluation

To demonstrate how the Purity profiling system works in practice, I conducted a case study. I selected SSCA#2 and employed this microbenchmark on the Deepthought2 cluster to assess the efficacy of Purity on a real system. Deepthought2 is a high-performance computing cluster hosted by the University of Maryland, College Park. The SSCA#2 microbenchmark generates and performs a series of approximate betweenness centrality (BC) computations over a weighted, directed multigraph given a set of starting vertices.

I evaluated the impact Purity had on performance for SSCA#2 on Deepthought2 with 4 nodes, 20 threads per node and an input of scale=8, low scale=5, high scale=8. When only the online reporting was enabled, the profiler added 16% to the microbenchmark's wall time. With the enabling of event log generation at a 1% sample rate monitoring the heap yielded a 62% overhead and produced 14.3 MB of event logs. Enabling tracking of both the stack and the heap added 95% to the wall time and produced 1.65 GB of event logs. A 95% overhead may seem high

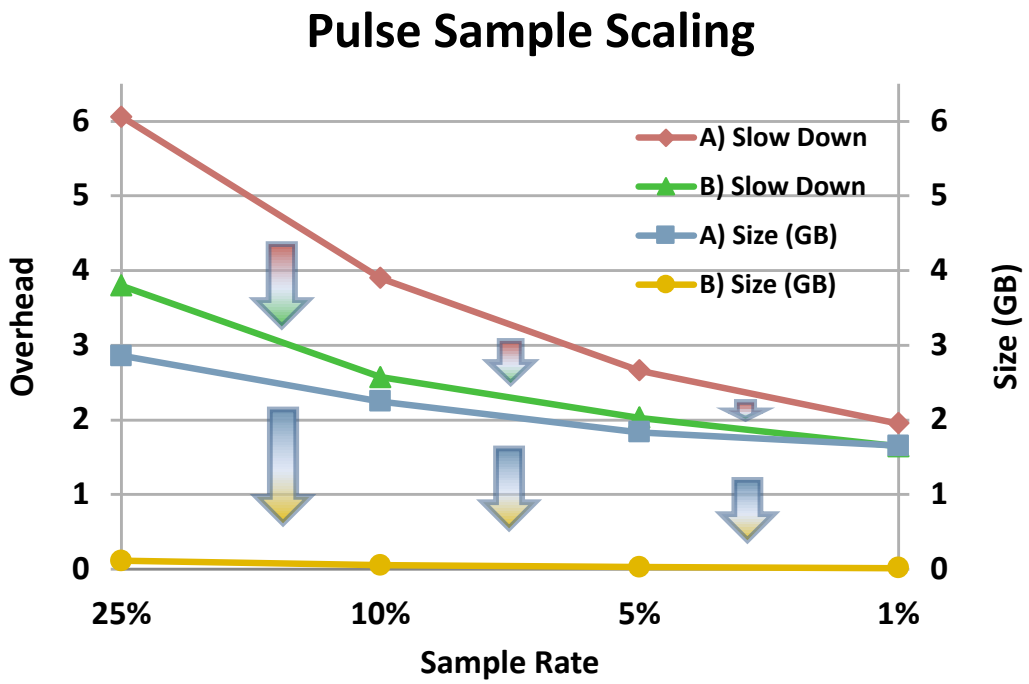

Fig. 5.6: Pulse Sample Scaling

but dealing with a less than 2x cost to tackle big performance problems with fine-grain analysis is worth it.

Since memory and communications access patterns in PGAS are determined by the distribution strategies used to map the elements of user-defined variables in the Chapel program's source, profile results should still be representative of the execution even when the overhead is large. Regarding the sample rate, like any pulse sampling approach the duty cycle of the pulse provides an accuracy vs. performance tradeoff. Approaches from prior papers that use similar sampling [113] [114] [115], can help to guide user decisions to determine appropriate sampling intervals for their applications using Purity.

The rest of the evaluation was performed on Deepthought2 under two different cluster configurations, 4 nodes with 20 threads (4N20T) each and 8 nodes with 10 threads (8N10T) each at a 5% sample rate. I compared enabling and disabling the Chapel PGAS network cache to test

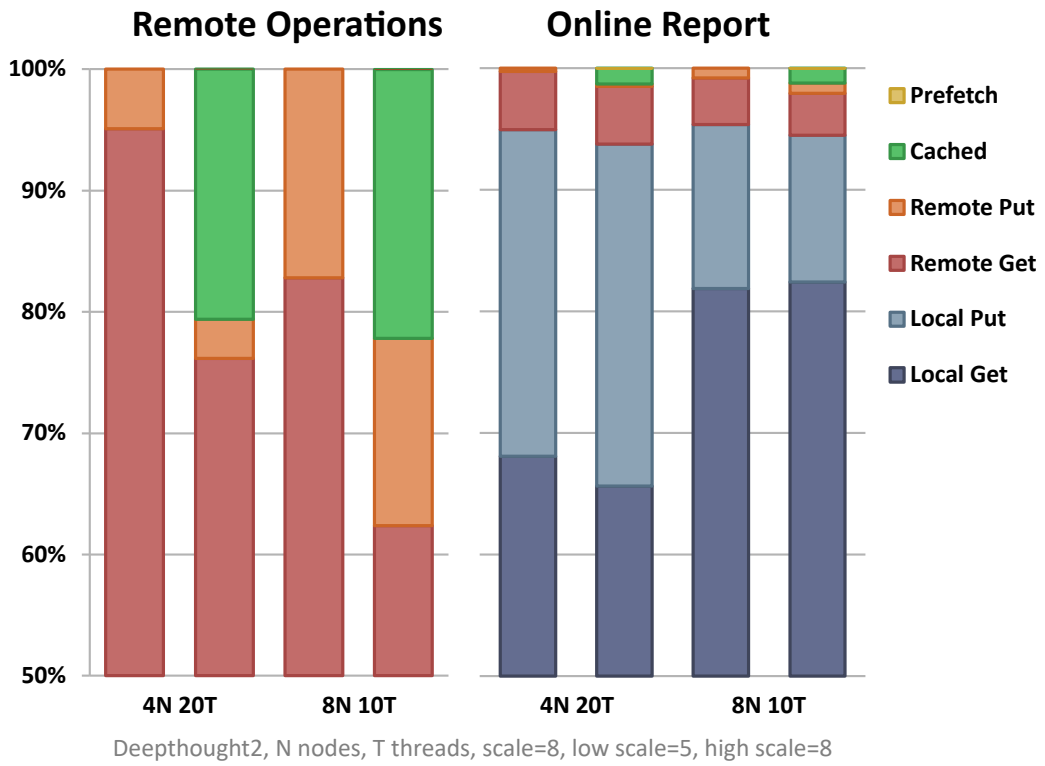

Fig. 5.7: Online Report and Remote Operations Views

the effectiveness of the automated remote cache option. This option can be enabled in the Chapel compiler by using cache-remote. The cache provides aggregation, write behind, and read ahead features at runtime which can reduce remote communications. By generating online reports for 4N20T and 8N20T in Figure 5.7 I found that the percentage of remote operations was 5.05% and 4.61% respectively for the two configurations. I also concluded that 20.6% and 22.2% of remote operations were avoided with the enabling of the network cache.

Listing 5.2: SSCA2_kernels.chpl, from Approximate Betweenness Centrality

```
279:  forall s in starting_vertices do on
            vertex_domain.dist.idxToLocale(s) {
          ...
286:      const tid = TPVM.gettid();
287:      const tpv = TPVM.getTPV(tid);
```

Through loop analysis I discovered that 42.14% (scale=5) and 80.99% (scale=8) of all remote operations occurred in the loop at SSCA2 kernels.chpl on line 582. In order to understand what this means we first need to look at how starting vertices and their corresponding tasks are delegated in the BC algorithm.

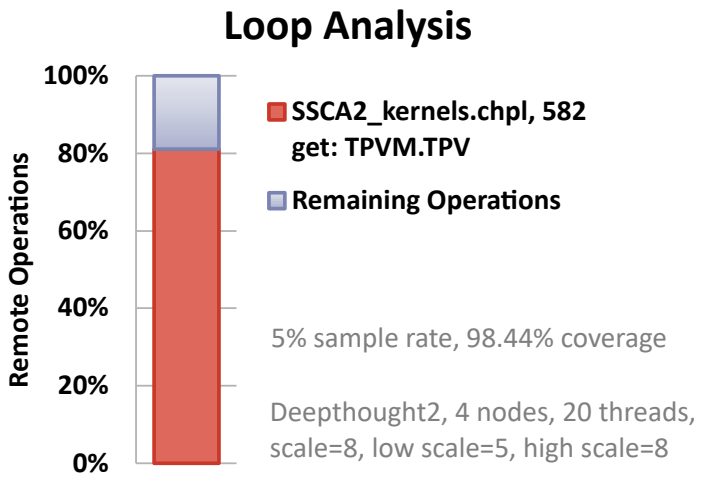

Fig. 5.8: Loop Analysis of BC

The outer most parallel loop of BC delegates tasks over starting vertices to the respective node on which that vertex is stored as illustrated in code example 3 in order to maximize task and data locality in a parallel environment. Each task requires a set of variables for processing its assigned starting vertex and these variables are managed by a taskPrivateData class. In order to reuse the instantiations of taskPrivateData for future tasks, task private variable array or TPV, a block distributed vector of taskPrivateData, is allocated and managed by TPVM. TPVM is an instantiation of TPVManager class which during the invocation of gettid() is responsible for yielding thread control back to the runtime framework until the requested element in TPV is no longer in use by another task and can be obtained and safely used by the requesting task.

Loop or Line		Remote		% of Remote
Module	**Range**	**Get**	**Put**	
TPVM.TPV, SSCA2_kernels.chpl, 577				
SSCA2_kernels.chpl	582-582	4544	0	42.14%
SSCA2_kernels.chpl	586	7	0	0.06%
SSCA2_kernels.chpl	589	5	0	0.05%
SSCA2_kernels.chpl	581	5	0	0.05%
taskPrivateData::barrier.tasksFinished, SSCA2_kernels.chpl, 600				
SSCA2_kernels.chpl	345-423	795	0	7.37%
SSCA2_kernels.chpl	456-463	192	0	1.78%
SSCA2_kernels.chpl	331-463	44	0	0.41%
taskPrivateData::BCaux, SSCA2_kernels.chpl, 569				
SSCA2_kernels.chpl	550	252	0	2.34%
Atomics.chpl	1260	0	146	1.35%
Atomics.chpl	1396	0	97	0.90%
SSCA2_kernels.chpl	331-463	87	0	0.81%
Atomics.chpl	1240	0	56	0.52%

Deepthought2, 4 nodes, 20 threads per node,
scale=5, low scale=3, high scale=4

Tab. 5.2: SSCA#2 Loop Analysis - Top Ranking Variables

The while condition of the loop in gettid() first accesses the this reference of the TPV-Manager class, followed by the class field TPV and then performs a get index operation over a distributed array to obtain a used field in taskPrivateData class and invoke a test and set atomic operation.

Receiver	Sender			
	n0	n1	n2	n3
n0	0	51256	57384	55021

Tab. 5.3: Remote Requests for TPVM.TPV

Node-level request aggregation for TPVM.TPV shows that all remote operations access the memory of the main node from other nodes, which indicates that TPVManager and the underlin-

ing Chapel framework structures that manage the TPV array were instantiated and only exist on the main node.

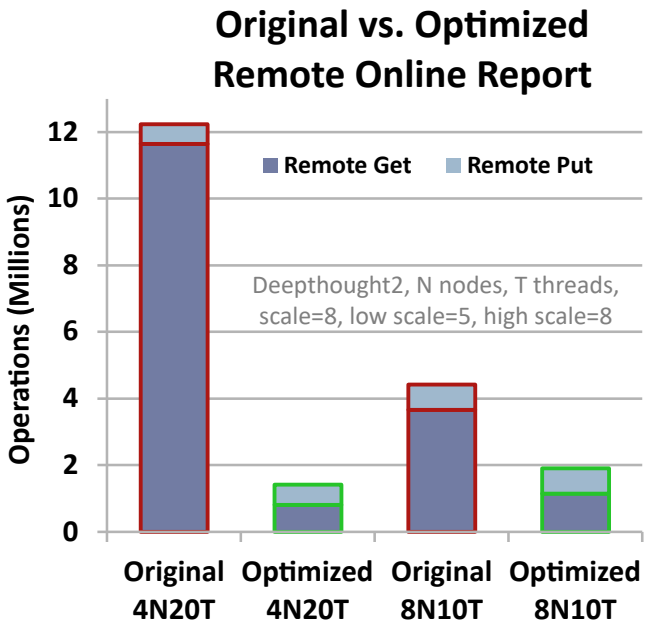

Fig. 5.9: Loop Analysis of BC

Config	Remote %	Reduction
4N 20T	0.47%	88.45%
8N 10T	1.19%	56.87%

Tab. 5.4: Remote Reduction per Configuration

With these insights in mind, a simple programmatic solution was devised in order to eliminate redundant remote communications. Code example 4 shows the original gettid() function in the SSCA#2 kernel and code example 5 illustrates the optimization that was discovered. Since the instantiation of TPVManager and the underlining structures that manage TPV reside in memory on the main node, resolving this for TPVManager, its member variable TPV, and the corresponding element at index 'tid' over the PGAS before entering the while loop will significantly reduce

Listing 5.3: Original: SSCA2_kernel.chpl

```
576:  class TPVManager {
579:      proc gettid () {
580:          const tid = this.currTPV.fetchAdd(1) // numTPVs;
581:          on this.TPV[tid] do
582:              while this.TPV[tid].used.testAndSet() do
                      chpl_task_yield();
583:          return tid;
584:      }
```

Listing 5.4: Optimized: SSCA2_kernel.chpl

```
576:  class TPVManager {
579:      proc gettid () {
580:          const tid = this.currTPV.fetchAdd(1) // numTPVs;
581:          on this.TPV[tid] do {
582:              const t = this.TPV[tid];
583:              while t.used. testAndSet () do
                      chpl_task_yield() ;
584:          }
585:          return tid;
586:      }
```

redundant remote operations generated by other nodes. Test and set operations can still be performed on the network atomic used inside the loop, which will allow gettid() to appropriate the next 'tid' to a requesting task. This optimization is a classic hoisting of redundant computation out of a loop. Also, the optimized 4N and 8N in the above chart of indicate the solution might be scalable. As the Chapel compiler matures is this the kind of implicit optimization we can expect?

Through my preliminary evaluation Purity identified a major bottleneck in SSCA#2 and provided insight into a simple programmatic solution that could be automated by the compiler. My optimization reduced 88% of remote operations on the PGAS and led to a 1.24x speedup in execution time. †

5.3 Expanding Profile Support for a DRAM-NVM Hybrid-PGAS

The Purity data-centric profiler has been extended to support DRAM-NVM hybrid-PGAS profiled executions. By using heap identifiers, the analysis can distinquish between main memory and persistent memory allocations and operations. Heap identifiers are used to determine which memory environment the operation occured (e.g. $id = 0$: DRAM main memory, $id > 0$: an NVM space). Both the introduction of persistent memory in Chapel and the Purity profiler were designed to potentially support multiple simultaneous persistent memory spaces in a hybrid-PGAS environment.

During runtime execution the dynamic analyzer records data on all allocations, reallocations, and free operations. The log stores both the definition and heap identifiers that are associated with the heap operation. Additionally, allocation stores the memory address, number of elements, byte size of each element, program time offset from the beginning of execution, and

the module file and line number in the source where the operation originated. Reallocation stores the old and new memory addresses, total bytes for the reallocation, program time, and module file and line number. In this case, program time is used to compute the lifespan of the old object that is being decommissioned and establish a birth time of the new object for the reallocation. Free stores the program time which is used to compute the object's lifespan.

Communication operations contain both local and remote accesses to memory objects in both DRAM and NVM environments. However, due to the high volume, to limit the size of the data logs, these operations are need to be sampled by the profiler. For each sampled operation the source node and thread, destination node, type of operation, local address and heap, remote address, byte size, latency (ns), and module file and line number are recorded. During a profiled execution, each locale runs the Purity dynamic analyzer which records data based on the view local to the perspective of the compute node on which it resides. To maintain low overhead, memory address resolution has been taken offline. The post analysis simulator uses the data logs generated by all compute nodes to synchronize and maintain a unified state of the entire execution across all locales. This provides an omnipotent view by which each memory address can be mapped to their corresponding node's heap, allocated heap object, definition identifier, and subsequent user-defined variable. Therefore, the heap identifiers and objects of remote addresses can be resolved through post analysis.

Heap analysis reporting contains aggregations for both local and remote operations for read and write count and byte totals categorized source node and thread to destination node and heap, which is broken down by user-defined variable. It also contains source latency totals (ns) and throughputs (Mbits/s) for remote read and write aggregates provided by the remote analysis. The total aggregate count of operations by variable determines how variables are ranked,

in descending order. Although the summary produced by dynamic analysis provides overall totals for each operation type, read to write and local to remote ratios can be determined on a per variable and memory environment basis using heap analysis. Additionally, the results from heap analysis are used to inform the hardware configuration estimations of latency, energy usage, and performance impact on execution in subsection 5.4.

Listing 5.5: Variable Assignment in Chapel

```
var a, b: real;

proc main() {
   a = b;
}
```

Listing 5.6: Compiler Generated C of User Main for the Variable Assignment Program

```
static void chpl_user_main(void) {
  _real64 tmp_chpl15;
  chpl____wide__ref__real64 chpl_macro_tmp_2209;
  chpl____wide__ref__real64 chpl_macro_tmp_2210;
  chpl_macro_tmp_2209.locale = (&b_chpl)->locale;
  chpl_macro_tmp_2209.addr = &(((&b_chpl)->addr)->value);
  chpl_gen_comm_get(((void*)(&tmp_chpl15)),
     chpl_nodeFromLocaleID(&((chpl_macro_tmp_2209).locale),
     INT64(0), INT32(0)), (chpl_macro_tmp_2209).addr,
     sizeof(_real64), COMMID(2), INT64(1), INT64(62));
  chpl_macro_tmp_2210.locale = (&a_chpl)->locale;
  chpl_macro_tmp_2210.addr = &(((&a_chpl)->addr)->value);
  chpl_gen_comm_put(((void*)(&tmp_chpl15)),
     chpl_nodeFromLocaleID(&((chpl_macro_tmp_2210).locale),
     INT64(0), INT32(0)), (chpl_macro_tmp_2210).addr,
     sizeof(_real64), COMMID(3), INT64(4), INT64(62));
  return;
}
```

Remote analysis reporting contains operation count, byte, and source latency aggregate totals categorized by operation type, source node and thread to destination node and heap, and

user-defined variable for remote operations only. When the values of elements from one distributed user-defined variable are set by another in a Chapel source file, the Chapel compiler will interpret and express the assignment by generating temporary stack variables to broker values between two communication operations. Listings 5.5 and 5.6 illustrate a simplified example of this. For distributed data, the generation of two communication operations is necessary because, depending on the compute node resources, both sides of the assignment could either be local, remote, or one local while the other is remote depending on their distributions, how they are accessed, and which locale the task is executed on. Therefore, although the source node generates a get or put request, the actual operation will always occur on the destination node and heap designation that handles that request, which is modeled by the profiler. When a remote operation occurs, both the source and destination node incur a latency penalty. Instead of measuring the round trip time, since Chapel relies on GASNet functions which block thread advancement while issuing a remote get or put operation, latency is a measure of the delay imposed on the thread of the source node that made the request. The results from this analysis are used in subsection 5.4.

The post analysis simulator uses heap operations to produce an object lifespan report which details each heap object's memory address, allocation size, node and heap it resided on, origin of allocation module and line number, allocation time from the start of execution (born time), and the object lifespan, organized by the user-defined variable it maps to. For each variable, heap objects are ranked in descending order by lifespan, byte size, and allocation time. If the heap object was allocated but never released, the object lifespan will report 'Execution' for DRAM and 'Persistent' for NVM, which means the object lived for the remainder of the execution time and still exists in the NV heap files if persistent. For objects that were allocated inside the Chapel runtime framework, the origin of the allocation will be set to internal. Currently, the report only

contains the lifespans of objects that map to user-defined variables. In the future, I would like to extend the reporting to also include unmapped heap objects so that additional analysis can be made for ancillary objects and operations.

5.4 Estimating Latency and Energy for Hardware Configurations

The Purity data-centric profiler will produce results for a profiled execution, representative of a program that was executed on a given distributed computing architecture. In contrast, the estimator can measure how a profiled execution will perform on different hardware. To do this I rely on DESTINY [116] [117] [118] to generate hardware memory profile estimations based on different SRAM, DRAM, and NVM memory models. The resulting memory profiles contain latency, energy and power consumption estimations. To better explore different architectural designs, configurations are built for all combinations of memory profiles, given several constraints. A hardware configuration is composed of a L1 data cache, L2 data cache, DRAM, and NVM. SRAM profiles are used to satisfy L1 and L2 data cache memory, where SRAM latency designated for L1 is equal or less than SRAM latency for L2. DRAM profiles are used for main memory and NVM profiles are applied as persistent memory.

Cache, memory, and communication profiles embody aggregate results from different analyses of the profiled execution, which are used by the estimator to produce a latency, energy, and impact on execution estimation model for each hardware configuration. Each model represents the estimated performance of the profiled execution respective to the hardware configuration. Additionally, 'DRAM only', 'DRAM ancillary with NVM variable store', and 'NVM only' are used as alternative scenarios to compare the estimated performance with the original model.

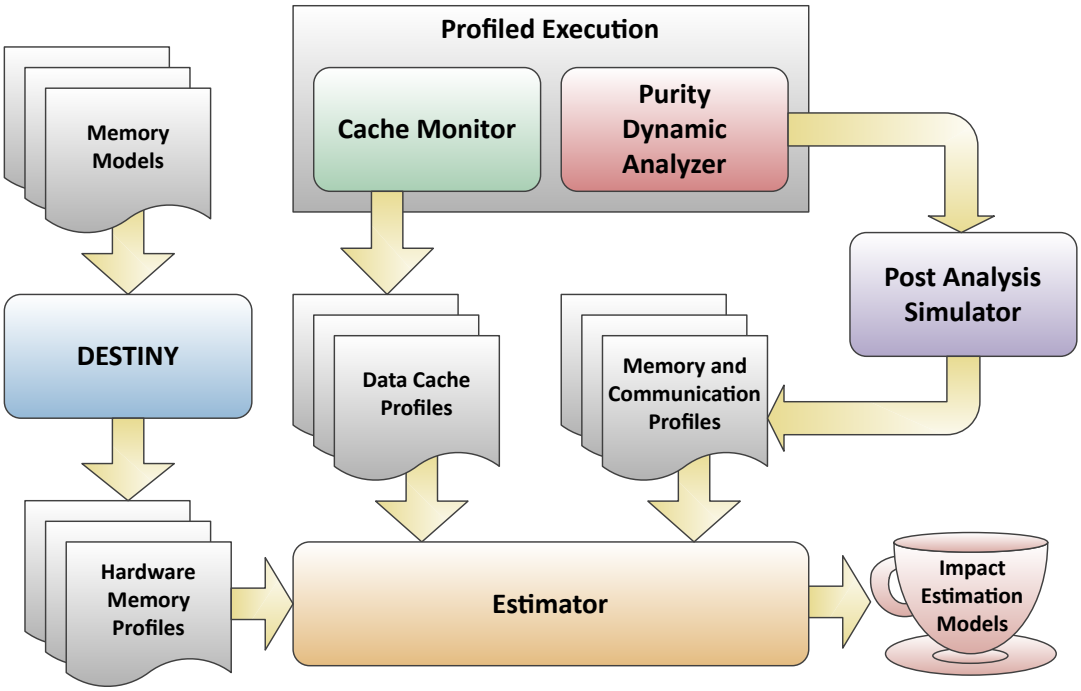

Fig. 5.10: Diagram for Estimating Latency, Energy, and Impact on Execution

Figure 5.10 illustrates how results from a profiled execution using Purity and the cache monitor, combined with hardware memory profiles are used to produce a series of impact estimation models. These models can be used to explore both the design and scenario space of different computer architectures by estimating the performance of profiled executions for these systems. They provide latency and energy estimations by user-defined variable, simplified AMAT (see parallelism in subsection 6.2.5), an aggregate view over the PGAS by memory type (L1, L2, DRAM, NVM), latency and energy by compute node, and impact on execution time, energy, and power. For each configuration, three alternative models are generated representing DRAM only, DRAM ancillary with NVM variable store, and NVM only scenarios for comparison.

Chapter 6: Evaluation

This chapter evaluates memory-based performance metrics of kernel and microbenchmark variants that utilize NVM to answer the following questions. Are there advantages to writing NVM parallel programs in Chapel vs. MPI and OpenMP? Under what circumstances is NVM performance comparable to DRAM? Can NVM be used correctly and efficiently in Chapel to recover a program's execution and data from a system failure? How can performance bottlenecks be addressed in a Chapel hybrid-PGAS application? In the analysis Chapel variants will be compared with MPI / OpenMP in C to demonstrate the efficacy of the Chapel NVM support for hybrid-PGAS memory.

6.1 Why use Chapel?

Chapel is a high-level parallel programming language that was designed for productivity, simplicity, and ease of use [81]. It is streamlined to allow a developer to quickly build parallel HPC applications that can scale in performance. Chapel relies on a PGAS memory environment and provides several distribution modules for easily mapping complex data structures over a distributed computing architecture. It supports both data and task parallelism. Recent advancements in Chapel now allow the programming of accelerators such as GPUs [119].

Listing 6.1 is an example of a Chapel implementation of 2D Jacobi stencil with a 2D de-

composition that uses block distribution to map cell elements over compute nodes. My multi-node version of Jacobi 5-point stencil was derived from a 2D Poisson's equation using the Jacobi Method in subsection 6.4.1 and inspired by a Chapel tutorial on data parallelism with Jacobi [120]. The implementation in listing 6.1 is an adaptation to Chapel's multi-locale environment and fully utilizes NVM.

Listing 6.1: 2D Jacobi Stencil in Chapel

```
1  use BlockDist;
2
3  config const n : int = 10000;
4  config const steps : int = 100;
5
6  var outer : persist = {0..(n+1), 0..(n+1)};
7  var inner : persist = {1..n, 1..n};
8  var D : persist = outer dmapped Block(boundingBox=inner);
9  var P : persist = D[inner];
10 var u : persist [D] real;
11 var v : persist [D] real;
12
13 proc main() {
14    // 2D Jacobi Stencil
15     for s in 1..steps do {
16         forall (i,j) in P do {
17             v[i,j] = (u[i-1,j] + u[i+1,j] +
18                       u[i,j-1] + u[i,j+1]) / 4.0;
19         }
20
21         u[P] = v[P];
22     }
23 }
```

To the contrary, 2D Jacobi stencil in C is not as concise. Listing 6.2 is an implementation of 2D Jacobi stencil with 1D decomposition written in C which utilizes MPI to create one process per compute node and OpenMP to parallelize over the available cores of each compute node. When compared with the Chapel 2D Jacobi stencil, the C version has 6 times more lines.

Listing 6.2: 2D Jacobi Stencil in MPI / OpenMP in C (Part 1)

```
1  int main(int argc, char *argv[]) {
2     int heap_id = 0;
3     int n = 10000;
4     int steps = 100;
5     int p, r, s, t, i, j;
6     int nodes, rank;
7     float *a, *b, *c, *x;
8     PMEMoid a_oid, b_oid, x_oid;
9     MPI_Request request;
10
11    MPI_Init(&argc, &argv);
12    MPI_Comm_size(MPI_COMM_WORLD, &nodes);
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15    if (argc > 1) sscanf(argv[1], "%d", &heap_id);
16    if (argc > 2) sscanf(argv[2], "%d", &n);
17    if (argc > 3) sscanf(argv[3], "%d", &steps);
18    n += 2;
19
20    if (heap_id > 0) { // id > 0 is PMEM else DRAM
21       if (pmem_support_open("/tmp/", argv[0], rank) == 0)
22          exit(1);
23    }
24
25    // block of array elements for each node
26    p = n * n / nodes; // elements
27    r = n / nodes;     // rows
28    printf("Rank: %d, Nodes: %d, N: %d, P: %d, Rows: %d\n\n",
         rank, nodes, n, p, r);
29
30    a = pmem_support_heap_alloc(heap_id, p * sizeof(float),
         &a_oid);
31
32    if (a == NULL) {
33       fprintf(stderr, "Node %d failed to allocate A.\n", rank);
34       fflush(stderr);
35       pmem_support_close();
36       MPI_Finalize();
37       exit(1);
38    }
```

Listing 6.3: 2D Jacobi Stencil in MPI / OpenMP in C (Part 2)

```
39
40      b = pmem_support_heap_alloc(heap_id, p * sizeof(float),
           &b_oid);
41
42      if (b == NULL) {
43         fprintf(stderr, "Node %d failed to allocate B.\n", rank);
44         fflush(stderr);
45         pmem_support_heap_free(heap_id, a, &a_oid);
46         pmem_support_close();
47         MPI_Finalize();
48         exit(1);
49      }
50
51      x = pmem_support_heap_alloc(heap_id, n * sizeof(float),
           &x_oid);
52
53      if (x == NULL) {
54         fprintf(stderr, "Node %d failed to allocate X.\n", rank);
55         fflush(stderr);
56         pmem_support_heap_free(heap_id, a, &a_oid);
57         pmem_support_heap_free(heap_id, b, &b_oid);
58         pmem_support_close();
59         MPI_Finalize();
60         exit(1);
61      }
62
63      MPI_Barrier(MPI_COMM_WORLD);
64
65      cache_monitor_papi_init_main_preconfig_omp(rank);
66      cache_monitor_papi_thread_begin_omp();
67
68      // 2D Jacobi Stencil with 1D Decomposition
69      for (s = 0; s < steps; s++) {
70         #pragma omp parallel for shared(b)
71         for (i = 1; i < r - 1; i++) {
72            int in = i * n;
```

Listing 6.4: 2D Jacobi Stencil in MPI / OpenMP in C (Part 3)

```
73            int im = (i - 1) * n;
74            int ip = (i + 1) * n;
75
76            #pragma omp parallel for shared(b)
77            for (j = 1; j < n - 1; j++) {
78               b[in + j] = (a[in + (j - 1)] +
79                            a[in + (j + 1)] +
80                            a[im + j] +
81                            a[ip + j]) / 4.0;
82            }
83         }
84
85         MPI_Barrier(MPI_COMM_WORLD);
86
87         if (rank < nodes - 1 && nodes > 1)
88            MPI_Isend(a + (p - n), n, MPI_FLOAT, rank + 1, 1,
                  MPI_COMM_WORLD, &request);
89
90         if (rank > 0) {
91            MPI_Recv(x, n, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
                  NULL);
92
93            #pragma omp parallel for shared(b)
94            for (j = 1; j < n - 1; j++) {
95               b[j] = (a[j - 1] +
96                       a[j + 1] +
97                       x[j] +
98                       a[n + j]) / 4.0;
99            }
100        }
101
102        MPI_Barrier(MPI_COMM_WORLD);
103
104        if (rank > 0)
105           MPI_Isend(a, n, MPI_FLOAT, rank - 1, 1, MPI_COMM_WORLD,
                  &request);
```

Listing 6.5: 2D Jacobi Stencil in MPI / OpenMP in C (Part 4)

```
106         if (rank < nodes - 1 && nodes > 1) {
107            int in = p - n;
108            int im = p - (2 * n);
109
110            MPI_Recv(x, n, MPI_FLOAT, rank + 1, 1, MPI_COMM_WORLD,
                  NULL);
111
112            #pragma omp parallel for shared(b)
113            for (j = 1; j < n - 1; j++) {
114               b[in + j] = (a[in + (j - 1)] +
115                            a[in + (j + 1)] +
116                            a[im + j] +
117                            x[j]) / 4.0;
118            }
119         }
120
121         MPI_Barrier(MPI_COMM_WORLD);
122
123         // swap
124         c = a;
125         a = b;
126         b = c;
127      }
128
129      MPI_Barrier(MPI_COMM_WORLD);
130
131      cache_monitor_papi_thread_end_omp();
132      cache_monitor_papi_thread_print_counts_omp();
133
134      cache_monitor_papi_print_results();
135      cache_monitor_papi_term_main();
136
137      pmem_support_close();
138
139      MPI_Finalize();
140
141      return 0;
142   }
```

6.2 DRAM vs NVM

Non-volatile random access memory (NVRAM) is an emerging technology that rivals DRAM. It is advantages include data persistency and retention, large capacity, and lower power consumption. Unlike DRAM, NVRAM does not require a periodic refresh to maintain the state of memory. In the event of power or system failure, it will retain any stored data, which can be recovered later. However, some drawbacks to this technology are higher read and write latencies than DRAM, and lower endurance.

Using Intel's Persistent Memory Development Kit, I have incorporated byte addressable NVM capability into the Chapel PGAS Language. In this section we will explore the characteristics of NVM and how they compare with DRAM. What are the performance differences of DRAM vs different NVM technologies in an HPC environment? What specific access pattern characteristics close the performance gap when taking caching mechanisms into consideration? How do performance characteristics change in a multi-locale environment when accessing NVM memory over compute nodes?

The experiment will be evaluated using a series of kernels with defined memory access patterns and microbenchmarks. To study and evaluate the performance of my implementation, I used a suite of microbenchmarks in Chapel that isolate and demonstrate certain access pattern characteristics. I then classify these microbenchmarks by their sensitivity to NVM latencies when a cache is in use (e.g. measuring latency hiding). Next, I employ hardware profiles generated from NVM modelers to compare the latency and energy use with different NVM technologies.

Application performance will be compared between variants that store all user-defined variables in DRAM vs. NVM. The baseline will be established by using NVM variants without the

benefits of latency hiding via cache verses NVM with cache support verse DRAM with support. The experiment will also involve hybrid DRAM-NVM approaches with different use cases.

6.2.1 Integrating the Cache Profiler into Chapel

As an experiment platform, I used the Zaratan system at UMD. A standard compute node on Zaratan hosts two AMD EPYC 7763 processors with 64 cores each for a total of 128 cores per node. Each AMD EPYC 7763 has a 64 KB L1 cache per core, 512 KB L2 cache per core, and a shared 256 MB L3 cache [121] and operates at a base clock rate of 2.45 GHz bursting up to 3.5 GHz turbo [122]. I used the PAPI Performance API to monitor the cache during runtime, which provides cache counters that can be used to build cache profiles on executions. The AMD EPYC 7763 processor supports cache performance counters for L1 data cache access (PAPI_L1_DCA), L1 data cache miss (PAPI_L1_DCM), L2 data cache hit (PAPI_L2_DCH), and L2 data cache miss (PAPI_L2_DCM) which can be monitored through PAPI. Unfortunately, PAPI cannot provide L3 cache performance counters for the AMD EPYC 7763.

Profiling L1 and L2 cache requires an instance of PAPI for each core. Therefore, developing a cache monitor module that can be used in Chapel programs was not enough. PAPI needed to be integrated into the Chapel runtime framework. In Chapel, workloads are divided up into tasks which are managed by the worker threads. Chapel provides two options for task management, FIFO and Qthreads. FIFO maintains a set number of threads for the lifetime of the execution. Tasks are mapped to threads via a task pool. A thread can execute on only one task at a time and must complete the task before processing another one [123]. The other option Chapel supports is Qthreads, developed by Sandia National Laboratories, that is provided as an external third library. Regardless, both options internally use POSIX threads (pthreads) to create worker threads. Since

FIFO task management is implemented in the Chapel runtime, I integrated cache profiling for the FIFO option.

Listing 6.6: runtime/main.c

```
int main(int argc, char* argv[]) {
  ///// CACHE MONITOR //////////////
  cache_monitor_init();
  ///// CACHE MONITOR //////////////

  // Initialize the runtime
  chpl_rt_init(argc, argv);

  ///// CACHE MONITOR //////////////
  cache_monitor_papi_print_info_main();
  ///// CACHE MONITOR //////////////
  ...
}
```

Listing 6.7: runtime/chplexit.c

```
static void chpl_exit_common(int status, int all) {
  ...
  ///// CACHE MONITOR //////////////
  cache_monitor_term();
  ///// CACHE MONITOR //////////////

  chpl_comm_exit(all, status);
  ...
  exit(status);
}
```

For repeatability I provided code snippets to illustrate how PAPI performance monitoring was done. I developed a cache monitor library to manage PAPI and simpilfy the interface in order to integrate cache profiling with the least number of changes to the runtime framework. Cache monitoring must be initialized before chpl_rt_init() in order to process POSIX threads as they are created by the runtime init as illustrated in listings 6.6. In listings 6.7 the cache monitor

terminates in chpl_exit_common right before chpl_comm_exit. During termination, timers are completed and all timing and cache performance counter data is printed out.

Listing 6.8: runtime/tthreads/pthreads/threads-pthreads.c

```
int chpl_thread_create(void* arg) {
  void *cm_thread_track_data = NULL;
  ...
  cm_thread_track_data = cache_monitor_papi_get_thread_data();
  ...
  if (do_pthread_create(&pthread, &thread_attributes, pthread_func,
       cm_thread_track_data ? cm_thread_track_data : arg))
  ...
}
```

The cache monitor library internally maintains it's own aggregations of cache performance counters for each CPU core as well as tracking the execution and monitor times of worker threads. In order to obtain correct counter information, an instance of PAPI must be initialized for each core that is monitored. The FIFO option will create a one worker thread for per core. Therefore, each thread must maintain it's own instance of PAPI. Thread tracking data is delegated as threads are being created as illustrated in listing 6.8 so that counter information can be accessed beyond the lifetime of the thread for reporting purposes. Since threads are mapped to a different CPUs, each thread has it's own L1 and L2 cache whose performance is tracked by PAPI. As shown in listing 6.9 the monitoring starts right before the task is executed and stops right after the task is complete. I believe this is the right granularity for monitoring the cache for relevant parts of the program on each core, since task execution is an entry point from the task management layer to the execution of the program's code.

In addition, the Chapel SLURM launcher for GASNet IBV (InfiniBand) was modified to ensure that the FIFO option produces one thread per core, except in the case where the number

Listing 6.9: runtime/tasks/fifo/tasks-fifo.c

```
static void thread_begin(void* ptask_void) {
  ...
  while (true) {
    ...
    chpl_task_do_callbacks( ... );

    ///// CACHE_MONITOR //////////////
    cache_monitor_papi_thread_begin(ptask_void);
    ///// CACHE_MONITOR //////////////

    (ptask->bundle.requested_fn)(&ptask->bundle);

    ///// CACHE_MONITOR //////////////
    cache_monitor_papi_thread_end(ptask_void);
    ///// CACHE_MONITOR //////////////

    chpl_task_do_callbacks( ... );
    ...
  }
}
```

of cores is one as FIFO requires at least one worker thread to prevent deadlocking the program. Also the SLURM option oversubscribe was added to avoid quickly burning through the allocation since Chapel assumes exclusive access to all 128 cores per node by default. Oversubscribe allows compute node resources to be shared with other jobs. When a Chapel program is executed with FIFO the primary node has a main thread and the number of cpu's per process minus one worker threads. For all other nodes, FIFO will create a worker thread for each core.

6.2.2 Hypothesis

My hypothesis is that due to the low miss rates in L1, L2, and L3 caches, the performance penalty of using NVRAM vs DRAM for program data is acceptable. Sequential and consecutive operations to the same or near-same memory locations of a persistent memory object will often

result in accesses to the same cache line. Furthermore, a write back cache policy and write buffer will hide the latency of slower memory in the hierarchy. Therefore, I expect the benefits of increased capacity and persistency of NVM outweigh performance loss as a result of longer read and write times for applications that forgo DRAM and allocate most if not all of the user-defined variables in NVM space. Subsections 6.2.3, 6.2.4, and 6.2.5 evaulate this hypothesis. If this trend holds true, I speculate that the long term implication is that NVRAM technologies could eventually replace DRAM entirely.

6.2.3 Access Patterns

The goal of this experiment is to demonstrate the impact that cache operations have towards hiding the latencies of slower memory environments. NVM operations typically incur a much larger latency penalty when compared with DRAM. My hypothesis states that sequential and consecutive operations can avoid most of the NVM penalties due to the caches. Scientific computing applications execute sophisticated algorithms which produce complex memory access patterns. However no matter how complex, algorithms in most cases use a composition of basic memory access paterns.

Therefore, a series of kernels have been developed to profile the cache for both intra-node and inter-node environments. These kernels are designed to run many trials over very large data structures to profile the designated operation. The kernel operations used are stream read, stream write, strided read, strided write, random read, random write, scatter, gather, reduce, and a stencil using the 2D Jacobi 5-point stencil. Variant kernels were developed in both Chapel and C/MPI/OpenMP, for serial and parallel, where all of the major data structures are allocated in either an NVM or DRAM environment. This way the cache profiles of NVM and

DRAM can be compared across many different types of operation. For the C implementations, MPI creates one process per node and OpenMP parallelizes over the cores for each node with OMP_PROC_BIND=true and OMP_PLACES=cores so that each thread is bound to a core.

Executions of kernel variants were performed on Zaratan for the following configurations: serial, 1 node with 8 threads, 4 nodes with 1 thread (MPI only), and 4 nodes with 8 threads each. PAPI was deployed at the thread level to record the cache performance counters on each core.

Looking at just the number of L1 data cache misses between Chapel and C/OpenMP in Figure 6.1 many of the profiled executions on the graphs appear to have similar paterns. However, there is clearly disproportionately more cache misses for the Chapel NVM implementations of stream write, strided write, random write, scatter, and gather kernel variants for 1 node with 8 threads than their DRAM variants. The reason for this disparity is because when changes from write operations are committed by the transaction system, the data is flushed back to memory, resulting in a higher cache miss count. As PMDK is used in both NVM variants, underlying activity in PMDK such as mmap operations do not seem to play a role for stream and strided write kernels in C/OpenMP. The extra cache misses only occur on the Chapel runtime framework for these two kernels during intra-node executions. Nevertheless, generally speaking the performance between Chapel kernels and C/OpenMP kernels are similar.

Figure 6.2 show the accumulated number of L1 data cache misses for kernels executed over 4 nodes each with 8 threads. Although the Chapel kernels executed two orders of magnitude less trials than the C/MPI/OpenMP variants, the number of misses appears to be higher in some of the cases. Each trial executes the kernels code. An inspection of the intermediate representation revealed the reason is because of the overhead resulting from the complexity of parallelizing a Chapel program that doesn't exist in the C kernels. Furthermore, data for random read, random

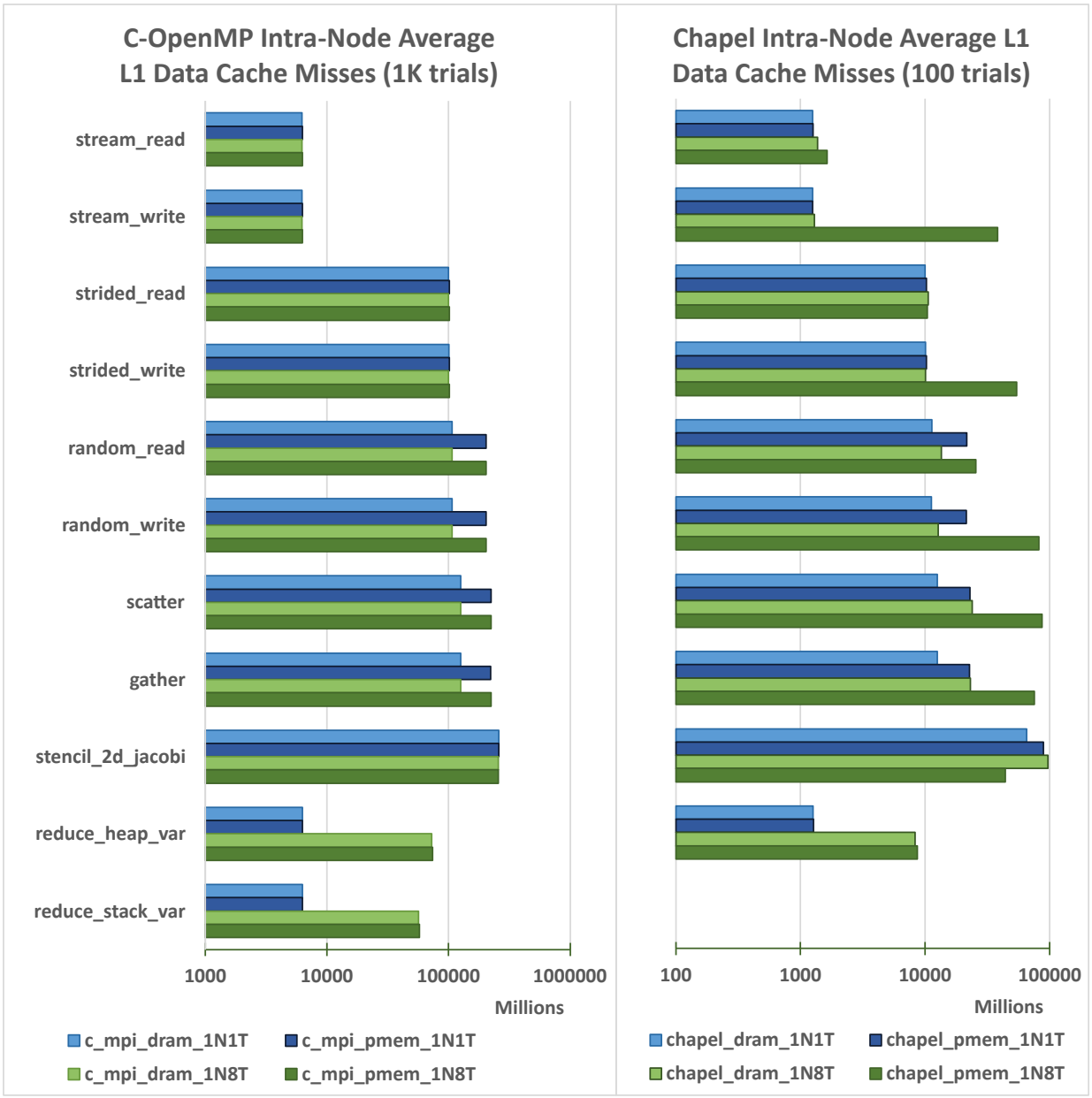

Fig. 6.1: Zaratan Average L1 Data Cache Misses in C vs. Chapel

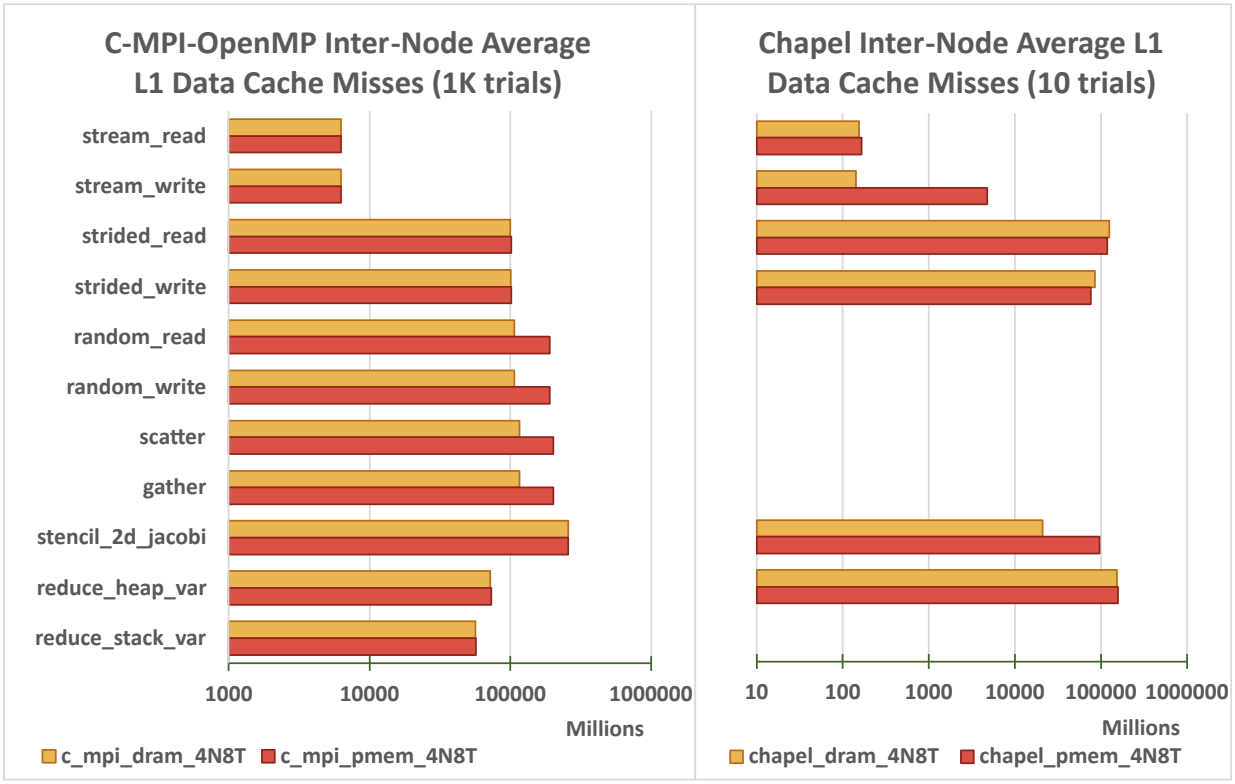

Fig. 6.2: Zaratan Average L1 Data Cache Misses in C vs. Chapel

write, scatter, and gather is not available for the Chapel versions due to timeout, as a consequence of using irregular access patterns over compute nodes. T. Rolinger, et. al. [52] addresses this problem with an inspector-executor optimization for Chapel. The overall patterns are similar except in the cases of NVM stream write and stencil which are proportionately higher than the misses in DRAM.

Next consider the miss rate for L1 and L2 cache calculated as follows. The event types are derived from PAPI event flags that are supported by the AMD EPYC 7763 chip and used to profile executions during the experiment. There is enough support for L1 and L2 data cache performance counters to compute L1 and L2 data cache miss rates in the equations below.

Event Type	**Description**
L1_DCA	L1 data cache accesses
L1_DCM	L1 data cache misses
L2_DCH	L2 data cache hits
L2_DCM	L2 data cache misses
L1_miss rate	L1 data miss rate
L2_miss rate	L2 data miss rate
L1_L2_miss rate	Data miss rate for both L1 and L2

Tab. 6.1: Cache Events

$$L1\_miss\_rate = L1\_DCM/L1\_DCA$$

$$L2\_miss\_rate = L2\_DCM/(L2\_DCH + L2\_DCM)$$

$$L1\_L2\_miss\_rate = L2\_DCM/L1\_DCA \approx L1\_miss\_rate * L2\_miss\_rate$$

If $L2\_DCH + L2\_DCM \approx L1\_DCM$ then the $L1\_L2\_miss\_rate$ holds true. $L1\_L2\_miss\_rate$ represents the percentage of data accesses that resulted in both an L1 cache miss and L2 cache miss. Please keep in mind that miss rate is represented by a value between 0.0 and 1.0. Therefore

the miss rate of L1-L2 has the potential to be extrodinarily small.

Figure 6.3 illustrates the average L1 data cache miss rate over the Chapel and C kernel variants for DRAM and NVM. Profiled executions for serial (1N1T), intra-node (1N8T), and inter-node (4N8T) are represented by blue, green, and orange respectively, where DRAM is lighter and NVM is darker. Missing bars indicate no data (e.g. the execution did not complete due to a time limit or no variant exists). Inter-node executions of Chapel kernels with irregular access patterns resulted in timeouts for random read, random write, scatter, and gather. There is only one reduce kernel implementation for Chapel.

When considering the serial execution (1N1T), strided write miss rate is nearly 100% in all variants. High miss rate is experienced with random write, scatter, and gather which all contain irregular access patterns. The Chapel NVM stencil kernel has a much lower miss rate but is much less pronounced when L2 miss rate is also considered as shown in Figure 6.4. Stream write misses about 25% of the time but the remaining Chapel kernels are under 10% miss rate for L1.

The L1 miss rates for stream read and write are lower for the intra-node and inter-node executions in Chapel, as well as intra-node runs for random read, random write, scatter and gather. The C variant executions yield similar L1 miss rates for stream read, stream write, strided read, strided right, and stencil but the NVM inter-node execution is lower. Random read, random write, scatter, and gather have similar miss rate patterns, but for the C variants miss rate is higher for NVM than DRAM where as in Chapel lower miss rates can be associated with inter-node executions. The reduce variants for both C and Chapel yield higher miss rates for intra-node and inter-node when compared with serial.

Both L1 and L2 miss rates are taken into consideration for Figure 6.4. For serial executions (1N1T) the patterns are similar between Chapel and C for the serial configuration. However, the

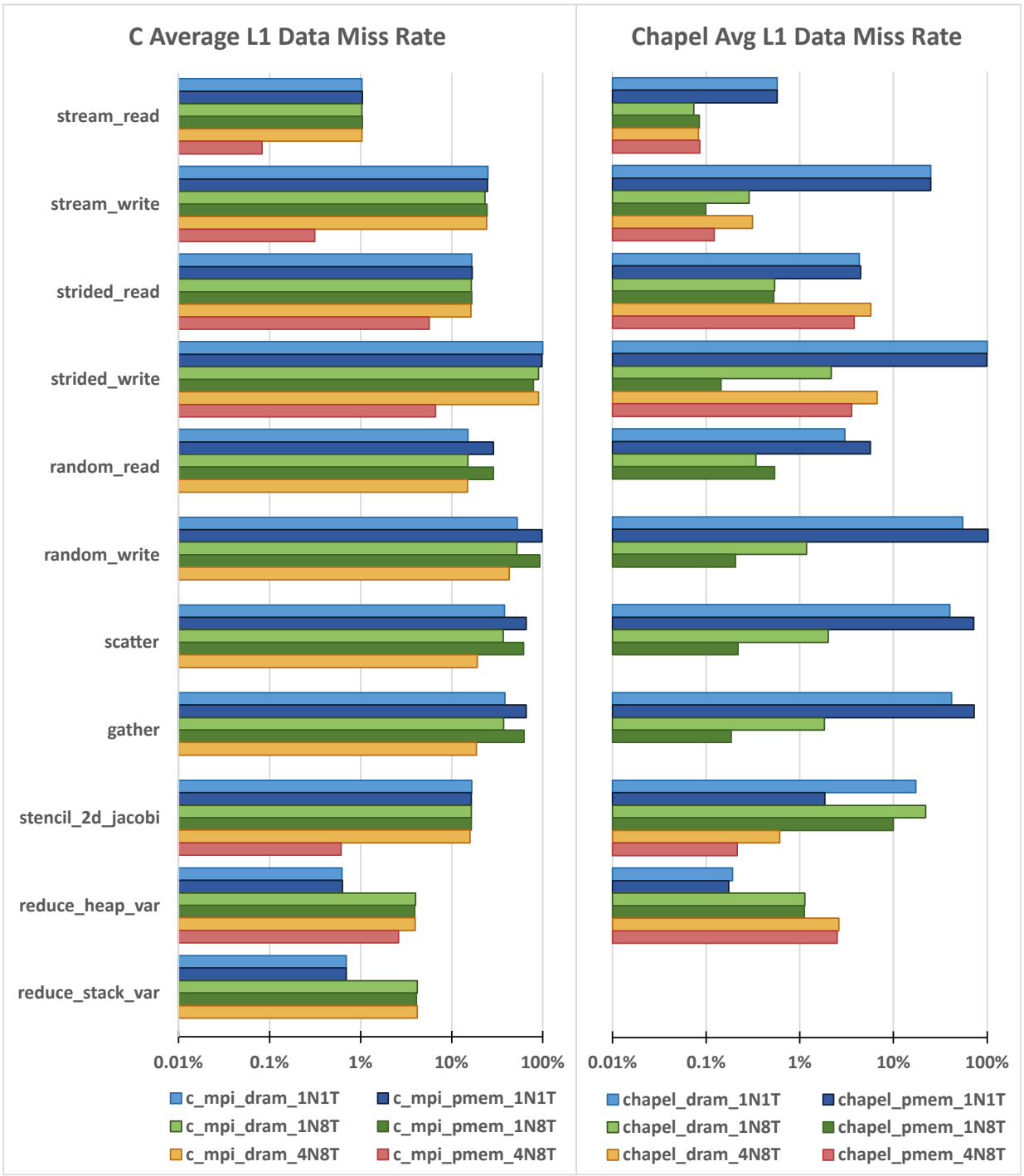

Fig. 6.3: Zaratan Average L1 Data Miss Rate, C vs. Chapel

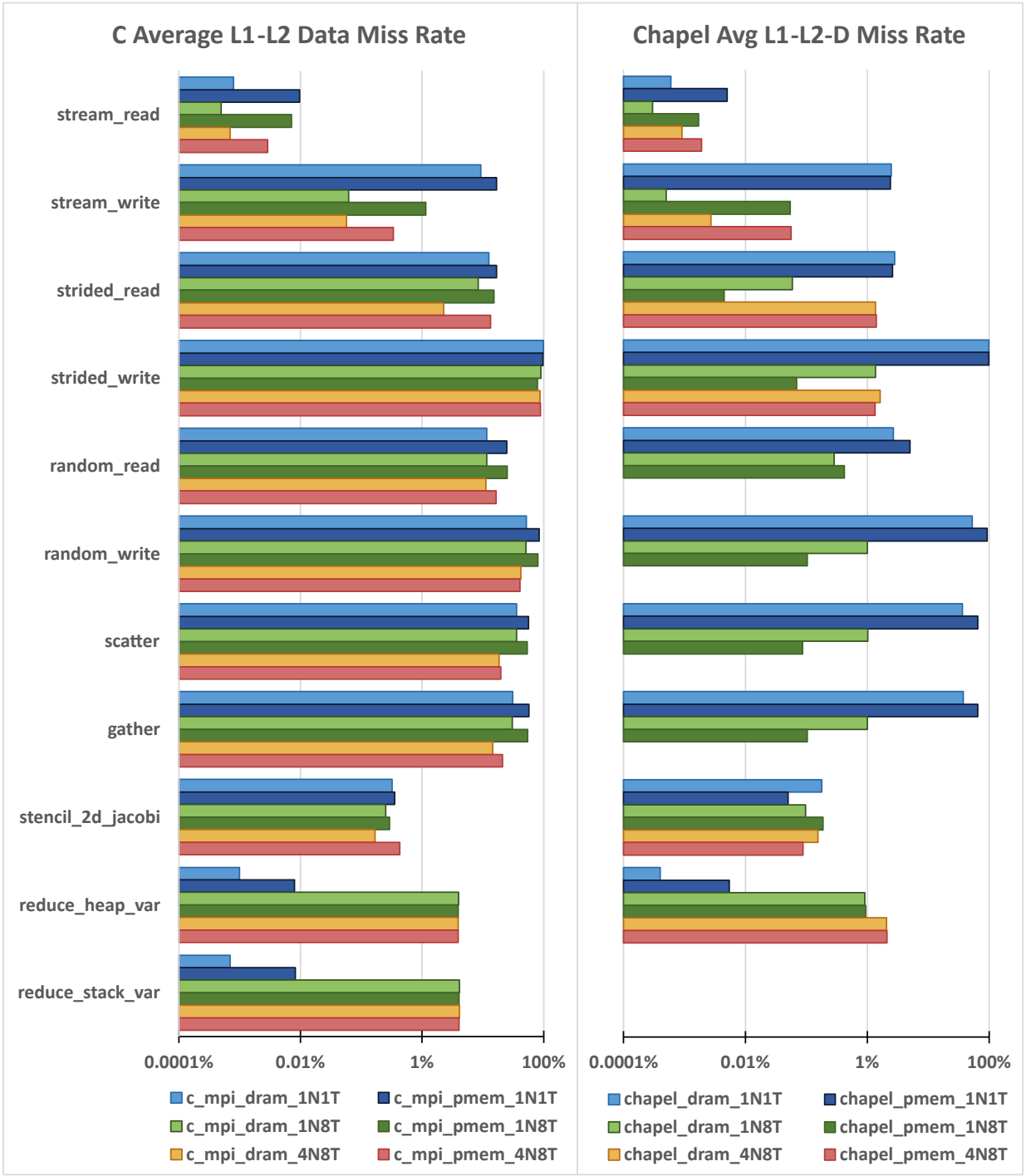

Fig. 6.4: Zaratan Average L1-L2 Data Miss Rate, C vs. Chapel

NVM stream read and reduce implementations both yield a higher miss rate than the DRAM variants. Stream write miss rate is still almost 100% and high miss rates are experienced in Chapel random write, scatter, and gather kernel variants due to irregular access patterns.

Intra-node executions (1N8T) include the L1-L2 miss rate for various kernel variants on 1 node with 8 threads. In the case of strided read, strided write, random write, scatter, and gather the NVM variants appear to have a lower miss rate than DRAM. Chapel NVM and DRAM kernel variants were compiled to different binaries so the difference in miss rate could be attributed to differences in GCC compiler optimizations. The overall miss rates are much lower for all of the Chapel implementations due to overhead in the runtime framework which likely contributes to L1 data accesses. Therefore, the results of miss rate for this configuration are less compelling. The same holds true for the 4 nodes with 8 threads each. Like the intra-node, inter-node executions of the reduce kernel yielded a higher miss rate than the serial executions. Also like intra-node, stream read and write produced lower miss rates than the serial variants. Due to timeout, all variants with irregular access patterns failed to complete their executions regardless of the memory environment since most of their accesses are remote. Random access to remote memory presents performance issues in Chapel due to fine-grain communications. T. Rolinger et al. [52] used network cache to address reduntant accesses and adaptive prefetching [59] techniques to mitigate the latencies of remote irregular patterns.

6.2.4 Microbenchmarks

The previous section demonstrated the cache performance of various basic operations for different memory environments independently. In this section, the same experiement was conducted over microbenchmarks to show how a variety of different operations impact cache perfor-

mance. The microbenchmarks I will use to demonstrate this are LULESH or Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics, from HPC Challenge Codes: 1D FFT, HPL (High Performamce Linpack), PTRANS, and from the Chapel Computer Language Benchmarks Game Codes: SpectralNorm (of an infinite matrix), and Mandelbroat-fast. I chose LULESH for it's complexity and large number of block distributed variables and because it uses an outer sequential loop to simulate shock hydrodynamics. FFT was considered since it utilizes both block and cycle distribution to implement a well established algorithm in scientific computing. HPL implements a solver for a linear system of equations that uses a block-cycle dimension specifier for a 2D distribution to perform LU factorization and back substitution. PTRANS was chosen for its use of block-cycle distribution to perform a parallel matrix transpose. SpectralNorm computes the induced 2-norm of a conceptually infinite matrix. Mandabrot-fast was examined since plotting a fractal image can produce chaotic access patterns.

Figure 6.5 represents the average L1 data cache misses for serial (1N1T), intra-node (1N8T), and inter-node (4N8T) executions over each microbenchmarks implemented in Chapel. Notice the SpecralNorm microbenchmark has effectively the same L1 data cache misses regardless of how the execution was distributed over computational resources. This serves as a good baseline that demonstrates the cache profiling is correct. The same can almost be said about Mandelbroat-fast except that the NVM variant incurred more misses for the intra-node execution. LULESH, FFT, HPL, and PTRANS are similar in that the DRAM variants had fewer misses than the NVM variants for serial and inter-node runs and inter-node executions for LULESH and PTRANS yielded significantly more misses.

For Figure 6.6, the left chart illustrates the average L1 data miss rate and the right chart the average L1-L2 data miss rates. The L1 miss rates for LULESH, HPL, and Mandelbroat-fast are

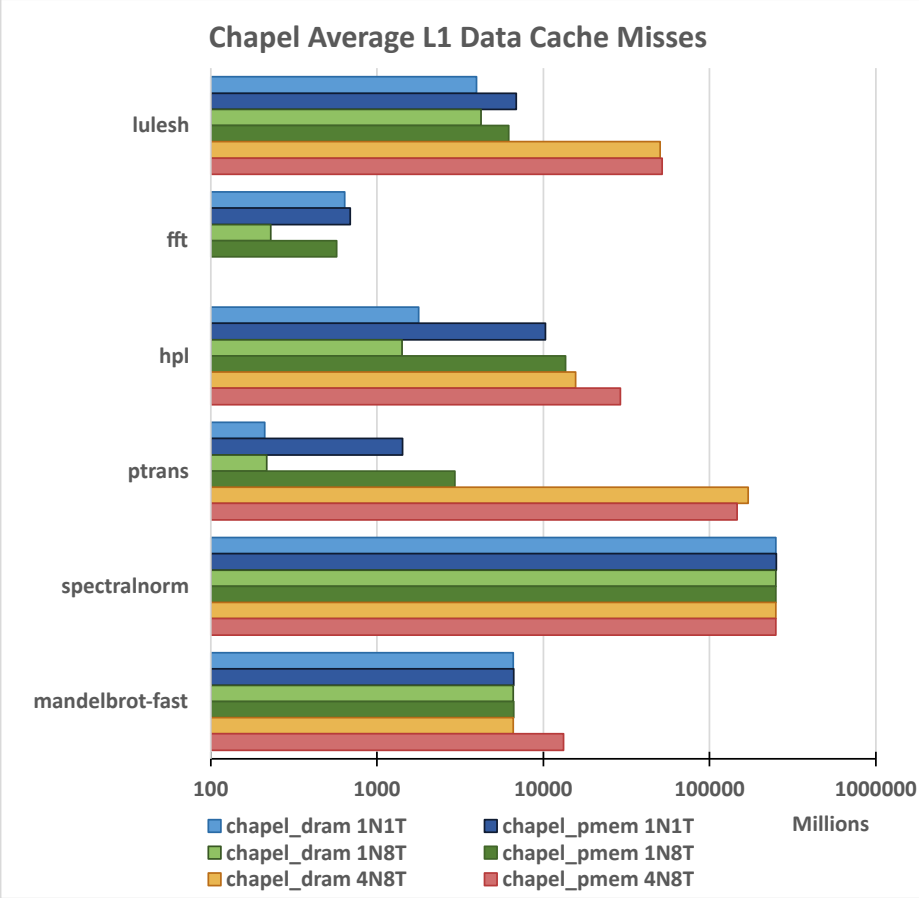

Fig. 6.5: Zaratan Average L1 Data Cache Misses for Chapel Microbenchmarks

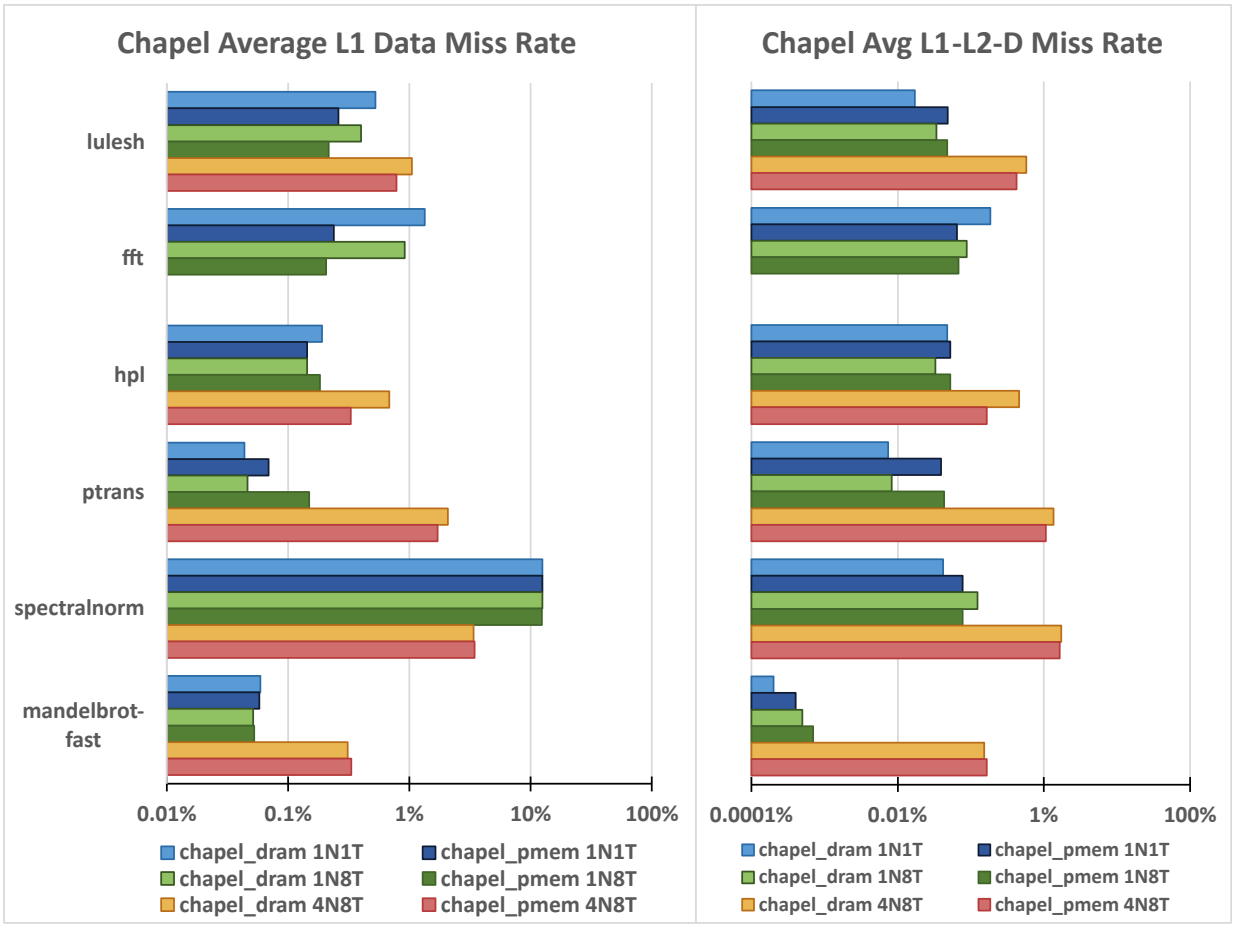

Fig. 6.6: Zaratan Average L1 and L1-L2 Data Cache Miss Rate for Chapel Microbenchmarks

around or less than 1% regardless of the execution on computational resources, with the highest average miss rate resulting from LULESH inter-node executions at 1.0526%. This means that L1 data cache is handling 99% of the memory accesses for these microbenchmarks. Also serial and intra-node miss rates are approximately 4 times higher than the inter-node (4 nodes) miss rates for SpectralNorm. The average L1-L2 data miss rates represent the percentage of accesses that are missed in both the L1 and L2 data cache. No L3 cache counters are available through PAPI for the AMD EPYC 7763.

6.2.5 Estimating Latency and Energy Usage

NVM was incorporated in the Chapel language using the Persistent Memory Development Kit developed by Intel to emulate a byte-addressable persistent memory environment. PMDK provides several pmem.io libraries that can turn even a hard drive into persistent memory. However, the CPU is unable to access NVM directly. Instead PMDK facilitates the loading and managing of data pools into virtual memory via mmap operations. Although this approach allows NVM accesses missed by the cache to potentially benefit from DRAM latencies which hide the NVM latencies of loaded data, the entire process is still constrained by the limited capacity of DRAM.

As NVM technologies continue to improve, their performance characteristics increasingly rival DRAM. For the scope of this study, lets consider a computer architecture that supports direct access to a byte-addressable NVRAM, which operates along side DRAM. On this architecture, the use of mmap is no longer necessary since NVM access does not need to be brokered through DRAM. Also there is no longer a need to remap any pmem_direct pointers contained within the structures of NVM heap objects in subsequent executions. NVM access is no longer shadowed

by DRAM latencies. Instead the architecture recognizes persistent memory in the same way DRAM is utilized, with full access to the large capacity of NVM. How might NVM compare in terms of latency and energy consumption when standing on the same footing as DRAM? In this section, we will explore the performance characteristics of different memory configurations under the proposed architecture.

To estimate the latency and energy impact on performance, hardware profilers were employed to model the latency and energy characteristics associated with the operations for various NVM, DRAM, and SRAM memory technologies. The two hardware profilers I used were NVSim and DESTINY. NVSim [124] [125] [126] is a circuit-level NVM performance, energy and area estimation modeler that supports STT-RAM, PCRAM, ReRAM, and NAND Flash. DESTINY [116] [117] [118] is a microarchitecture tool, built on NVSim, which support both 2D and 3D SRAM cache designs, eDRAM, STT-RAM, PCRAM, and ReRAM. Models and profiles can be found in sections C.1 and C.2. Hardware profile characteristics have been verified in [116] [117] [127] [20].

The Purity data-centric profiler was employed to profile the execution of a LULESH Chapel variant with a problem size of elemsPerEdge=48. LULESH is composed of both global and function scope user-defined variables which allocate objects on the heap. For this experiment all global user-defined variables were defined as persistent and allocated in NVM instead. All function scope user-defined variables which resulted in allocations remained in DRAM. For simplicity, the run was performed on Zaratan with 1 compute node over 8 threads.

Purity provides several options for data sampling, based on signal processing implemented via sigalarm, sigprof, and PAPI. A signal handler is used to toggle between 'on' and 'off' phase intervals during runtime execution. Sampling takes place during the 'on' phase, at which time

detailed information about local and remote data accesses are logged for post analysis. By adjusting the sample intervals, log file size and analysis time can be greatly reduced. In addition, Purity maintains a basic summary count of operations which are aggregated over compute nodes and reported at the end of execution.

Operations	Reads	Writes	Total	Percent
DRAM	1,201,740	220,423	1,422,163	19.54%
NVM	1,397,431	385,691	1,783,122	24.49%
Ancillary	2,056,482	2,018,046	4,074,528	55.97%
Sampled	4,655,653	2,624,160	7,279,813	0.005%
Actual	103,482,618,945	40,975,255,632	144,457,874,577	

Tab. 6.2: Data Profile Summary for LULESH elemsPerEdge=48

For the LULESH profiling, sample intervals of 0.01 seconds for the 'on' phase and 9.99 seconds for the 'off' phase were used to sample data accesses via sigprof through Purity. However, this does not necessarily translate into 0.1% of the data being sampled. According to table 6.2, only about 0.005% of the data was sampled during the execution or 20 times smaller than the expected 0.1%. Data sampling uses sigprof which relies on a timer that is not perfectly accurate and asynchronous signals which may not immediately call the handler. This can lead to delays and shorter record times when sampling. Actually, I consider the inconsistencies of sigprof to be an advantage as I had considered periodically adding small random values to the intervals to avoid a potentially rare case of the sample rate synchronizing with the programs's execution. Hypothetically, perfect timing consistently recording from the same parts of a loop during the 'on' phase while consistently not recording from others parts of the same loop due to always being executed during the 'off' phase would produce a profile that is not representative of the entire execution. Since sigprof is not perfectly accurate, this case can be avoided when sampling data.

Table 6.2 gives an overview of total operations broken down by different categories. Sampling comprises of DRAM, NVM, and ancillary operations. DRAM refers to the number of sampled operations that could be mapped to heap objects associated with user-defined variables that were allocated in DRAM, and the same for NVM.

Listing 6.10: chpl_localeID_to_locale in Chapel

```
proc chpl_localeID_to_locale(id : chpl_localeID_t) : locale {
  if rootLocale._instance != nil then
    return (rootLocale._instance:borrowed
       AbstractRootLocale?)!.localeIDtoLocale(id);
  else {
    ...
```

Ancillary operations may not directly relate to user-defined variables but instead help facilitate various aspects of the runtime environment. Listings 6.10 and 6.11 show an example of high frequency ancillary operations stemming from the chpl_gen_comm_get() calls inside chpl_localeID_to_locale(). Instances of locale objects are used to facilitate forall loops and resolving their wide references using locale identifiers often generates local accesses and sometimes remote communications. Purity maps all ancillary operations to a file and line number and upon further inspection nearly 85% of these operations are either attempting to resolve a locale or facilitate the task management layer.

Actual operations refer to the total operations over the entire execution that were processed through compiler generated communication functions. These functions are used to perform read and write operations to both local and remote objects on the PGAS, which makes them ideal for profiling. Only a small subset of operations are logged by the profiler. Since only 0.005% of the accesses were sampled, the number of read and write operations for each user-defined variable

Listing 6.11: chpl_localeID_to_locale in compiler generated C

```
/* ChapelLocale.chpl:797 */
static void chpl_localeID_to_locale(...) {
  ...
  if (...) {
    ...
    chpl_macro_tmp_2754.locale = (coerce_tmp_chpl5).locale;
    chpl_macro_tmp_2754.addr =
       &(((object_chpl)((coerce_tmp_chpl5).addr))->chpl__cid);
    chpl_gen_comm_get(((void*)(&chpl_macro_tmp_2755)),
       chpl_nodeFromLocaleID(&((chpl_macro_tmp_2754).locale),
       INT64(0), INT32(0)), (chpl_macro_tmp_2754).addr,
       sizeof(int32_t), COMMID(27), INT64(799), INT64(24));
    ...
    chpl_macro_tmp_2759.locale = (&ret_chpl2)->locale;
    chpl_macro_tmp_2759.addr =
       &(((object_chpl)((&ret_chpl2)->addr))->chpl__cid);
    chpl_gen_comm_get(((void*)(&chpl_macro_tmp_2760)),
       chpl_nodeFromLocaleID(&((chpl_macro_tmp_2759).locale),
       INT64(0), INT32(0)), (chpl_macro_tmp_2759).addr,
       sizeof(int32_t), COMMID(28), INT64(799), INT64(24));
    ...
    goto _end_chpl_localeID_to_locale_chpl;
  } else {
    ...
```

were scaled accordingly so that the estimations would represent the entire execution.

Depending on the number of nodes and amount of data that is sampled, the post analysis simulation can take a while to complete. For instance, LULESH profiling of 1 node with 8 CPU cores with a problem size of 48 elements per edge executed with dynamic analysis in 4,834 seconds and generated a 730 MB operation log file with 11,987,056 lines. However, the post analysis simulation took 159,447 seconds or 44 hours, 17 minutes, and 27 seconds to complete. The completion time for post analysis profiling is effected by the number of nodes, CPUs, user-defined variables, sample rate, and duration of execution. Performing the more expense computations in post analysis allows Purity's dynamic analyzer to take most operations offline instead of generating memory access and communication traffic that could impact the data-centric profile. As long as the sample rate is small, dynamic analysis can remain very light weight. In addition, the post analysis simulator generates all of the underlying results needed that when combined with hardware profiles can make quick estimations on how the execution may look on a cluster with different memory hardware configurations.

A standard compute node on Zaratan uses two AMD EPYC 7763 chips. Each chip contains 64 CPU cores with a 64 KB L1 and 512 KB L2 cache per core, and a 256 MB L3 shared cache [121]. When profiling the cache, counters were available for L1 and L2 data cache but not L3. Cache profiles of the LULESH hybrid variant yielded an average L1 miss rate of 0.2149% and L2 miss rate of 22.0272% over all of the CPU cores. For better representation, averages over each CPU core were used to calculate the read and write latency and energy aggregates. The averages are based on cache profiles from five separate executions of LULESH on Zaratan using 8 CPU cores on 1 compute node. Figure 6.7 uses a box and whisker's plot to illustrate statistics for each CPU's L1 and L2 data cache miss rate. The average is indicated by 'x' connected by

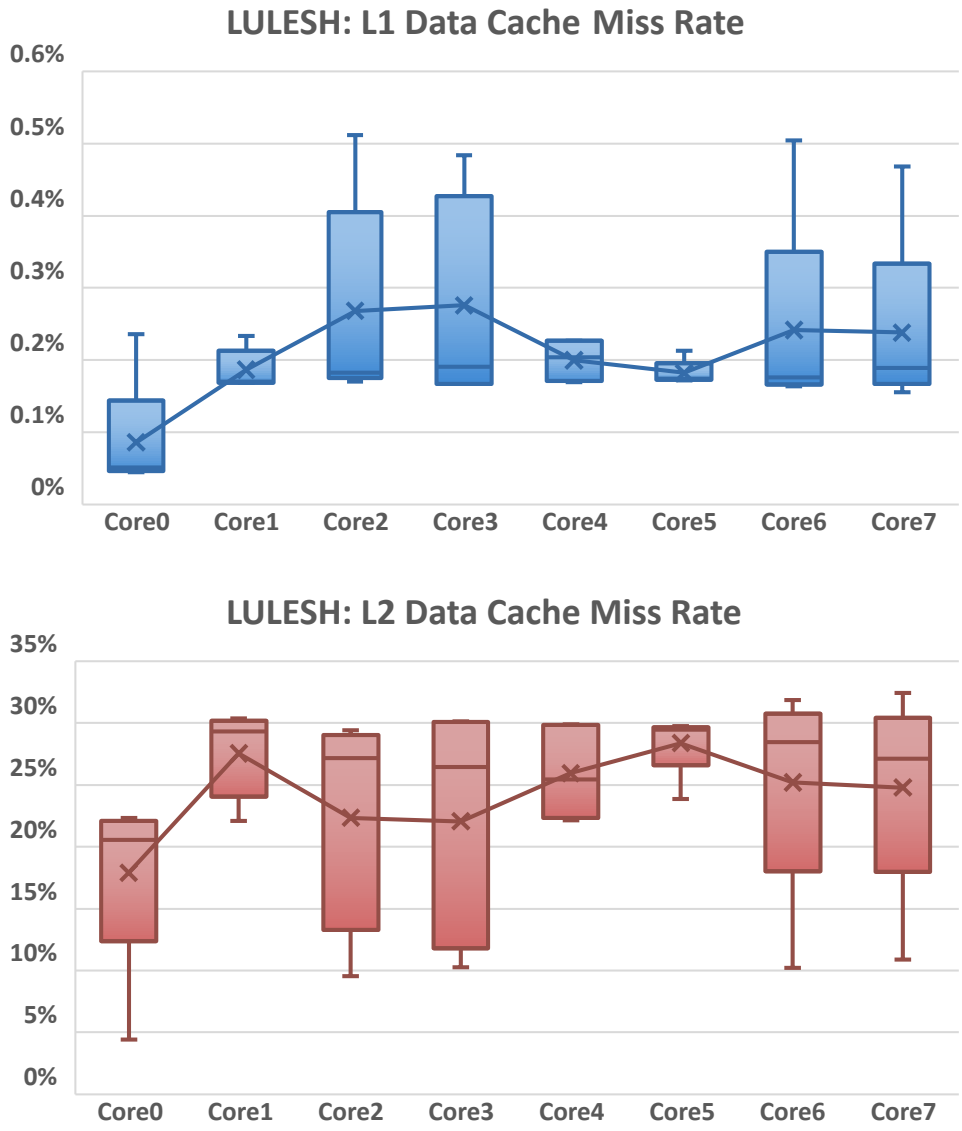

Fig. 6.7: LULESH: L1 and L2 Data Cache Miss Rate by CPU Core

a line between each adjacent CPU, which is used in the subsequent analysis. The whisker's represent the minimum and maximum thresholds. Any data points beyond these thresholds are considered outliers. Finally, the boxes indicate the first and third quartiles with the median line runing through them,

The analysis combines data-centric profiling of local and remote memory accesses, cache profiling of CPU cores, and hardware profiling of various memory components to make projections on latency and energy usage as well as estimations of memory impact on execution time and energy footprint for different memory configurations. Impact on execution is an estimations of time, energy, and power for profiled parallel execution running on an architecture with a given memory configuration. This analysis is used to answer three questions: When is NVM performance comparible to DRAM? When is it not? And how does NVM impact on the execution?

When does NVM perform like DRAM?

A configuration represents choices of CPU cache and memory hardware for a standard compute node on a cluster. Each configuration is composed of a set of hardware memory profiles assigned to L1 and L2 data cache, DRAM, and NVM. Using a profiled execution of a benchmark or application, estimations can be made on how the application might perform on new hardware. Consider the following configuration.

Configuration		Latency (ns)			Energy (nJ)			Power (W)
Type	Memory	Read	Write	Refresh	Read	Write	Refresh	Leakage
L1	SRAM	0.849	0.781		0.0111	0.0069		0.00321
L2	SRAM	2.220	2.220		0.0198	0.0053		0.01186
DRAM	DDR4	12.570	12.570	451.282	0.0982	0.0996	94.932	5.41500
NVM	PCRAM	33.808	183.400		0.2355	26.0350		2.78700

Tab. 6.3: Memory Configuration with Phase-Change Memory

The latency and energy metrics in table 6.3 were generated from hardware profiles run through the DESTINY hardware simulator. This configuration uses a PCRAM with a read latency of 2.69x and write latency of 14.59x higher than DRAM. Accesses to PCRAM consume 2.4x more read and 261.4x more write energy than on DRAM. However, DRAM requires a periodic memory refresh to maintain the data in memory and while not active, incurs a power leakage nearly twice (1.94x) as high as PCRAM. Therefore, PCRAM is slower and consumes more energy per operation but will still consume less energy over the course of execution than DRAM.

DRAM	Op Read Latency (s)		Op Write Latency (s)	
Variable	With Cache	No Cache	With Cache	No Cache
y8n	2.084	30.426	0.073	1.162
x8n	2.083	30.425	0.073	1.159
z8n	2.081	30.383	0.073	1.157
dvdx	2.050	29.936	0.074	1.171
dvdy	2.047	29.899	0.074	1.167
...	...	...	...	...

	Op Read Energy (J)		Op Write Energy (J)	
Variable	With Cache	No Cache	With Cache	No Cache
y8n	13.93	121.67	0.326	4.70
x8n	13.93	121.75	0.327	4.70
z8n	13.91	121.57	0.326	4.69
dvdx	13.70	119.71	0.329	4.73
dvdy	13.69	119.59	0.328	4.72
...	...	...	...	...

Tab. 6.4: DRAM Latency and Energy, Top Variables: Cache vs No Cache

Table 6.4 refers to the top five user-defined variables in DRAM, ranked based on the number of local and remote operations performed on their associated heap objects on the PGAS. The latency and energy aggregates are computed from the L1 (SRAM), L2 (SRAM), and DRAM properties using the average miss rates from the cache profiles and compared with a no cache advantage. The aggregate values represent estimations over all compute nodes, CPU cores, memory

environments, and over the duration of execution.

PCRAM	Op Read Latency (s)		Op Write Latency (s)	
Variable	With Cache	No Cache	With Cache	No Cache
elemToNode	6.00	232.38	0.000	0.00
x	3.09	119.51	0.287	59.16
y	3.08	119.32	0.286	59.05
z	3.08	119.12	0.287	59.19
xd	2.18	84.50	0.653	134.55
...	...	...	...	...

	Op Read Energy (J)		Op Write Energy (J)	
Variable	With Cache	No Cache	With Cache	No Cache
elemToNode	39.84	828.71	0.000	0.00
x	2.56	53.28	0.444	537.46
y	2.56	53.19	0.443	536.47
z	2.55	53.10	0.444	537.79
xd	1.81	37.67	1.012	1,222.38
...	...	...	...	...

Tab. 6.5: NVM Latency and Energy, Top Variables: Cache vs No Cache

Similarly, table 6.5 illustrates the estimated aggregates for the top five user-defined variables in NVM. The L1, L2, and PCRAM properties are used along with the aforementioned cache miss rates to perform the calculations.

Totals	Operation Latency (s)				Operation Energy (J)			
	Read		Write		Read		Write	
	With Cache	No Cache	With Cache	No Cache	With Cache	No Cache	With Cache	No Cache
DRAM	20.53	299.74	3.47	54.94	89.65	783.29	3.65	52.59
Ancillary	+30.73	+448.84	+27.79	+440.45	+25.69	+224.51	+15.52	+223.38
PCRAM	24.20	937.50	6.81	1,403.65	54.99	1,143.87	10.55	12,759.62
Total	75.46	1,686.07	38.07	1,899.05	170.33	2,151.66	29.73	13,035.59

Tab. 6.6: Latency and Energy Totals by Memory Component using PCRAM

In table 6.6, the previous two tables are totalled by their respective memory component.

DRAM refers to operations performed on heap objects associated with user-defined variables. All other operations in the DRAM environment are considered ancillary. Config represents the estimated total local and remote operation latency and energy usage of scaled operations over all user-defined variables, compute nodes, CPU cores, memory environments, and ancillary accesses during the entire execution.

Totals	Operation Latency (s)			Operation Energy (J)		
	With Cache	No Cache	Benefit	With Cache	No Cache	Benefit
DRAM	23.99	354.68	93.24%	93.30	835.88	88.84%
Ancillary	+58.53	+889.29	93.42%	+41.21	+447.88	90.80%
PCRAM	31.01	2,341.15	98.68%	65.54	13,903.49	99.53%
Total	113.53	3,585.12	96.83%	200.06	15,187.25	98.68%

Tab. 6.7: Latency and Energy Totals, Cache Benefit using PCRAM

Table 6.7 illustrates the latency and energy benefits from using a cache. The aggregated totals estimate the resources spent on each memory component. The configuration latency total (with cache) could be thought of as the combined latency of all operations over all nodes and CPU cores categorized by each memory component for the given memory hardware configuration.

The total number of sampled operations that were directly mapped to heap objects of user-defined variables are 1,422,163 for DRAM and 1,783,122 for PCRAM, according to table 6.2. There are almost twice (1.254x) as many PCRAM operations than DRAM for the profiled execution. Therefore, to make a fair comparision, the total latencies of both environments need to be normalized by the number of operations. Dividing the total latency of PCRAM with cache of 31.01 by 1.254 yields 24.73 which is close to the total latency of 23.99 for DRAM by 1.031x. With a difference of only 3.1% impact on execution time, PCRAM performs similarly to DRAM and validates the hypothesis.

	Estimated Latency (ns)	
Memory	Read	Write
DRAM	0.854792	0.787373
PCRAM	0.860737	0.835381
	1.007x	1.061x

Tab. 6.8: Estimated Read and Write Latencies based on Simplified AMAT

The performance of DRAM and PCRAM can also be compared by calculating the estimated average read and write latencies of accesses to each memory environment while benefiting from using cache. Table 6.8 illustrates a weighted average over CPU cores of read and write latencies by memory environment. Although the latency profile for PCRAM is 2.69 and 14.59 times the read and write latencies of DRAM according to table 6.3, the findings indicate that the read latencies for DRAM and PCRAM are almost identical (1.007x) and the PCRAM write latency is only 1.061 times higher than DRAM. The use of cache hides the latencies of both memory environments since operations are performed at lower latency. However, PCRAM benefits more from using the cache than DRAM, due to it's higher latency. Since most of the operations are cached due to the low L1 data cache miss rates, the resulting latency differences between DRAM and PCRAM are negligible. Therefore, the estimation of NVM performance is similar to DRAM performance for this configuration, which supports the hypothesis in subsection 6.2.2.

Since PAPI does not support L3 cache counters for the AMD EPYC 7763 chip, L3 cache was not considered. However, had the cache profile included L3 then further latency hiding of DRAM and NVM operations could be modeled, which would further support the conclusion.

How does PCRAM perform vs. DRAM when applying parallelism?

The types of parallelism modeled in the analysis are compute nodes, CPU cores, and the

number of memory channels. The end goal is to use the aggregate latency and energy of local and remote memory accesses to estimate the impact of memory access on execution time and energy footprint of a benchmark or application for a given configuration. The computer architecture considered supports both DRAM and PCRAM each having 2 memory channels. Although high performance systems may support up to 8 memory channels, only 2 were considered for the analysis.

The modeling only regards a subset of an actual computer architecture since CPU clock rate, instruction cache, L3 shared cache, cache block size, cache organization, cache write polices, interface bandwidth between different levels of cache, and memory bus bandwidth are not taken into consideration. Latency calculations are therefore based on a simplified average memory access time (AMAT) that uses the SRAM latencies for cache-level hit times rather than the number of clock cycles * clock cycle time. See section B.1 for details on the AMAT calculation. A more accurate calculation would require processor characteristics (e.g. for AMD EPYC 7763) which are not publicly available or a generated profile of a hypothetical processor. For the scope of this analysis, lets assume the architecture handles both memory environments the same way. Support for AMAT calculations based on detailed processor profiles can be added to future work.

In this section, I will compare the estimated parallel performance of DRAM exclusive, PCRAM hybrid, PCRAM mapped, and PCRAM exclusive architectural models. For DRAM exclusive, no NVM hardware exists and all user-defined variables reside in DRAM. In PCRAM hybrid, variables exist in both environments and DRAM handles all ancillary allocations. The PCRAM mapped configuration hosts all variables in NVM, but the ancillary allocations still reside in DRAM. Finally, for PCRAM exclusive, no DRAM hardware exists and all variables are stored in NVM. Consequently, PCRAM exclusive must handle all ancillary allocations and

operations as well. The analysis will compare estimated impact on execution time, energy, and power. In order to better understand the comparison, I will use data from the PCRAM hybrid execution to illustrate how these metrics are derived.

	Time (s)	Energy (J)			
Memory	Elapsed	Operation	Leakage	Refresh	Total
L1_SRAM	16.574	190.780	0.099		190.88
L2_SRAM	0.123	0.722	1.672		2.39
DRAM	0.100	0.472	190.687	3.725	194.88
Ancillary	+0.251	+0.252	-2.714		-2.46
PCRAM	0.659	7.832	95.025		102.86
Total	17.707	200.059	284.769	3.725	488.55

Tab. 6.9: PGAS Elapsed Time and Energy Summary for PCRAM Hybrid Execution

Table 6.9 represents a view of elapsed time and aggregate energy by memory environment over the PGAS for the PCRAM hybrid execution. When producing elapsed time, L1 and L2 data cache operations were separated from the DRAM and PCRAM. Some DRAM and PCRAM operations still occur as a result of cache misses. Although the elapsed time for PCRAM is an order of magnitude larger than DRAM, both have a marginal impact on the overall elapsed time when compared with L1.

For PCRAM hybrid, all ancillary allocations exist in DRAM. Since leakage is a loss of energy when a memory component is not actively in use, the time spent handling ancillary operations effects the overall energy leakage calculation. Aside from leakage, when just considering operation and refresh energy, PCRAM appears to have a smaller energy footprint than DRAM. This is also the case when adding refresh energy to total operation energy from table 6.7. Once the estimated impact on execution time in table 6.11 is known, the total refresh energy for DRAM can be calculated using the profiled component's refresh latency and energy. Refresh latency

refers to how often the memory component needs to be refreshed to retain its state and refresh energy is the energy consumption required. Total refresh energy is scaled by the estimated impact time and added to the energy consumption for DRAM.

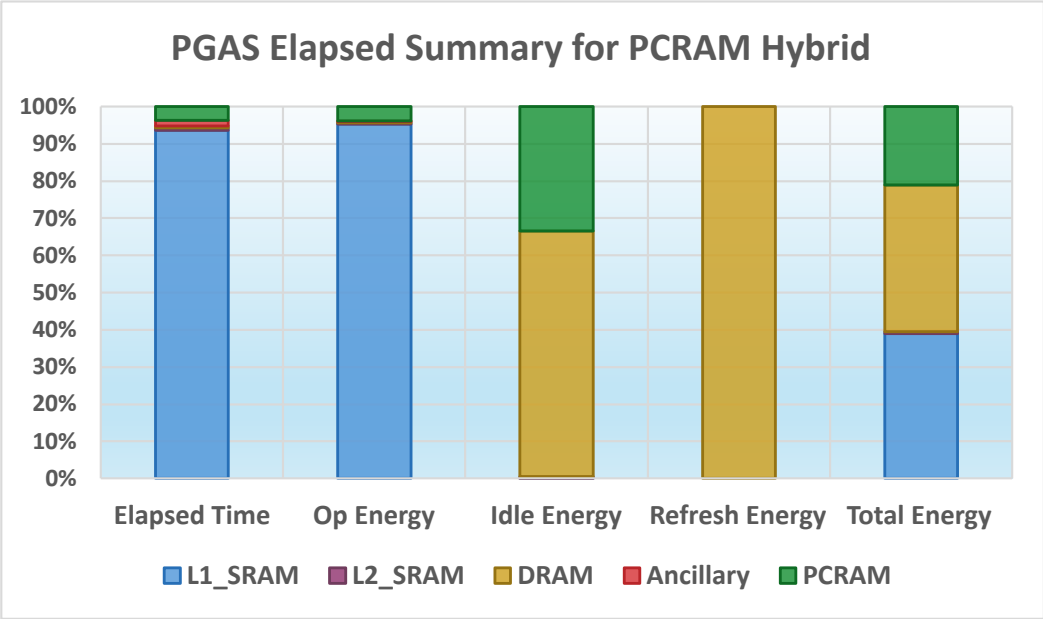

Fig. 6.8: PGAS Elapsed Time and Energy Consumption by Percentage for PCRAM hybrid

Figure 6.8 illustrates each memory environment by percentage. Given the low miss rates of L1 from the cache profiling, most of the elapsed time is spent in L1 data cache. However, since cache profiling relies on cache counters, the mapping between each operation and their respective memory environment is unknown. Consequently, miss rates are applied evenly over DRAM, NVM, and ancillary operations.

Elapsed time and energy consumption for L1 and L2 are calculated for each CPU of a compute node. To achieve this, the associated thread identifier of each sampled operation was recorded during Purity's dynamic analysis. Since Chapel's FIFO task management layer maps each thread to a CPU core for the duration of execution, latency and energy can be aggregated by

thread / core.

		Elapsed (s)			Energy (J)			
Node	CPUs	Local	Remote	Total	Operation	Leakage	Refresh	Total
0	8	17.71	0.00	17.71	200.06	284.77	3.72	488.55

Tab. 6.10: Cluster Elapsed Time and Energy Summary for PCRAM Hybrid Execution

Table 6.10 represents the elasped time and energy footprint impact by compute node over the cluster. To calculate the elapsed time per node, the thread with the maximum elapsed time over all memory environments is selected. The thread on the compute node with the maximum elapsed time is broken down by memory environment to produce the time values for table 6.9. The impact on execution time in table 6.11 can be found by taking the maximum elapsed time over each compute node. Total energy is broken down by operation, leakage and refresh energy. Leakage and refresh energy are computed once the execution impact time is known.

The estimated impact on execution time represents the memory access impact on execution based on latencies and parallelism. The memory energy footprint represents the total energy consumed over all relevant CPU cores and memory environments. Since the microbenchmark execution shared compute node resources with other applications, the cores that were not used during execution were not considered in the energy calculation. Finally, the power consumption in watts (joules / second) can be determined by dividing the total energy by the impact on execution time.

Table 6.11 and Figure 6.9 show the estimated impact on execution time, energy, and power for all four models. When observing impact time, DRAM exclusive is only 1.16x faster than PCRAM exclusive, 1.05x faster than PCRAM mapped, and 1.03x faster than PCRAM hybrid, which supports the hypothesis in subsection 6.2.2. This is due to the benefits that PCRAM gains

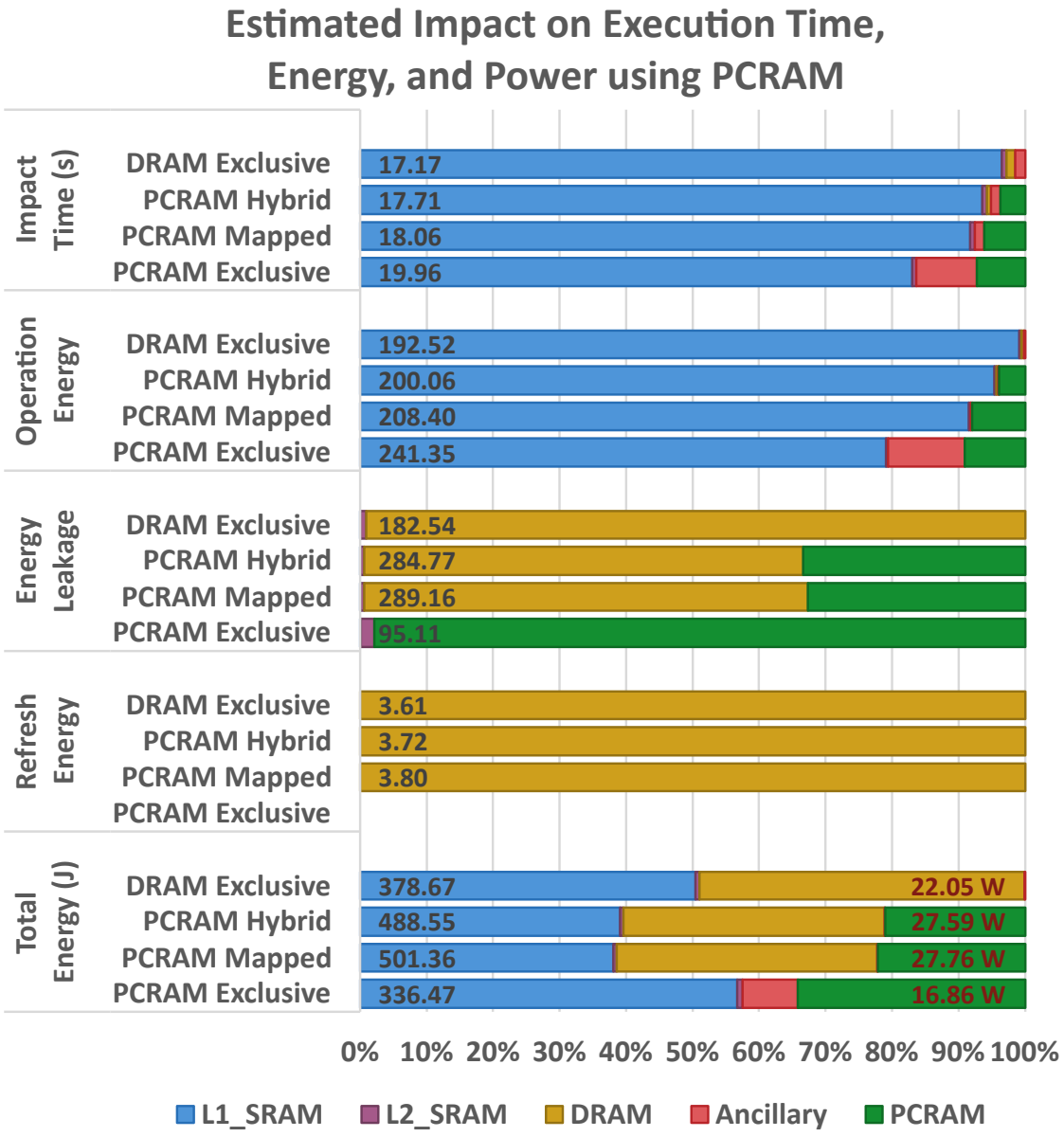

Fig. 6.9: Estimated Impact on Execution Time, Energy, and Power using PCRAM

Configuration	Variables	Ancillary	Estimated Impact on Execution		
			Time (s)	Energy (J)	Power (W)
DRAM Exclusive	DRAM	DRAM	17.17	378.67	22.05
PCRAM Hybrid	Hybrid	DRAM	17.71	488.55	27.59
PCRAM Mapped	NVM	DRAM	18.06	501.36	27.76
PCRAM Exclusive	NVM	NVM	19.96	336.47	16.86

Tab. 6.11: PCRAM: Estimated Impact of Memory on Execution Time and Energy Footprint

from low L1 and L2 data cache miss rates from Figure 6.7, which hide the higher latency of PCRAM and the fact that the remaining accesses to PCRAM only impact the execution time by at most an estimated 2.79 seconds. Concerning energy consumption, PCRAM exclusive is 11% more energy efficient than DRAM exclusive and 23.5% more power efficient. PCRAM hybrid and PCRAM mapped consume more energy because their architectures must power both DRAM and NVM. Storing all user-defined variables in NVM, as opposed to hybrid, slightly increased the estimated energy consumption over a longer execution impact time, which resulted in a slightly lower power usage. Keep in mind that results only considered the energy consumption of the memory system and not the network. CPU energy only accounts for L1 and L2 data cache for the cores that were used.

Figure 6.10 illustrates a sweep over L1 data cache miss rates to estimate the impact on execution time for all models. The L2 miss rates from Figure 6.7 are still used when producing the estimations. In all four cases, the impact time grows linearly when L1 miss rate is increased. Although the figure only shows a sweep up to 1% miss rate, higher miss rates yield the same gradients. DRAM exclusive, PCRAM hybrid, PCRAM mapped, and PCRAM exclusive increase at a rate of 2.57, 4.94, 6.52, 15 seconds per L1 data cache miss rate percent respectively, and diverge from DRAM exclusive at a rate of 2.37, 3.95, and 12.43. There is a tradeoff between

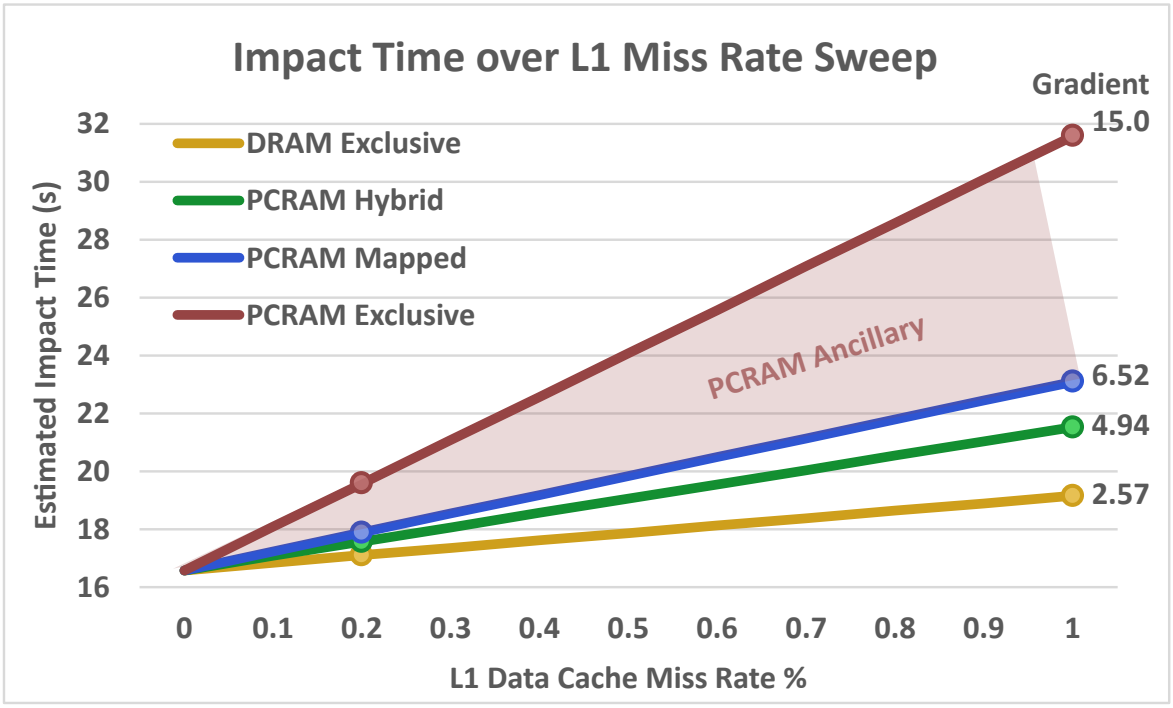

Fig. 6.10: Estimated Impact Time for L1 Data Cache Miss Rate Sweep by Model

impact on execution time and increased capacity, persistency, energy efficiency when storing some or all user-defined variables in NVM. However, PCRAM hybrid is only 1.12x slower than DRAM exclusive at 1% L1 miss rate, 1.56x at 10%, and 1.87x at 100%, while PCRAM mapped is 1.21x higher at 1%. Recalling the average L1 miss rate of 0.2149% for the profiled execution, the differences for 0.2% miss rate are 1.03x, 1.05x, and 1.15x when compared with DRAM exclusive. This indicates that when L1 data cache miss rate is low, the performance for PCRAM is similar to DRAM.

When does NVM not perform like DRAM?

The last configuration illustrated how DRAM and NVM can perform similarly when using the profile properties of PCRAM. However, similarity in performance still depend largely on how many orders of magnitude slower the latency for NVM is than DRAM.

Configuration		Latency (ns)			Energy (nJ)			Power (W)
Type	Memory	Read	Write	Refresh	Read	Write	Refresh	Leakage
L1	SRAM	0.849	0.781		0.0111	0.0069		0.00321
L2	SRAM	2.220	2.220		0.0198	0.0053		0.01186
DRAM	DDR4	12.570	12.570	451.28	0.0982	0.0996	94.932	5.41500
NVM	SLCNAND	16,256	200,440		696.41	1,782		0.00074

Tab. 6.12: Memory Configuration with Single-Level Cell NAND Flash

The configuration in table 6.12 uses the same L1, L2, and DRAM memory component profile properties from the previous configuration. However, Single-Level Cell NAND Flash is now used for NVM with orders of magnitude greater read and write latency and energy characteristics than PCRAM. The profile read and write latencies for SLCNAND are 1,293 and 15,946 times that of DRAM. Also notice the power leakage of SLCNAND is very small when compared with the leakage of PCRAM from the previous configuration. Both were generated by the DESTINY hardware memory profiler. See the Appendix for memory models and profiles.

Totals	Latency (s)			Energy (J)		
	With Cache	No Cache	Benefit	With Cache	No Cache	Benefit
DRAM	24	355	93.2%	93	836	88.8%
Ancillary	+59	+889	93.4%	+41	+448	90.8%
SLCNAND	1,148	1,984,849	99.9%	2,455	4,256,176	99.9%
Total	1,231	1,986,093	99.9%	2,590	4,257,459	99.9%

Tab. 6.13: Latency and Energy Totals: Cache Benefit

When considering SLCNAND hybrid, as table 6.13 illustrates, even through SLCNAND benefits greatly from using cache, the total latency is significantly high than DRAM. Since the total latency with cache is over 51 times higher (51.3x) than DRAM, normalizing by the number of operations will not close the gap. The resulting value of 915.47 for SLCNAND is more than 38 times (38.2x) the total latency of 24 seconds for DRAM. The higher cache benefit cannot

make up the difference. Even with low L1 data cache miss rates, the aggregated total latency for SLCNAND flash comprises of 93.3% of the total latency for the entire configuration, which will inevitably impact the execution.

	Estimated Latency (ns)	
Memory	Read	Write
DRAM	0.854792	0.787373
SLCNAND	5.416083	57.096267
	6.336x	72.515x

Tab. 6.14: Memory Configuration with Single-Level Cell NAND Flash based on Simplified AMAT

Furthermore, when considering the estimated weighted average read and write latencies for DRAM and SLCNAND from table 6.14 , the latencies for SLCNAND are 6.3 and 72.5 times that of DRAM. Although this is still a dramatic improvement from 1,293x and 15,946x due to latency hiding as a result of using the cache, SLCNAND still performs much slower than DRAM. Therefore, given the profile characteristics for the memory configuration in table 6.12, SLCNAND flash will likely never perform as well as DRAM.

	Time (s)	Energy (J)			
Memory	Elapsed	Operation	Leakage	Refresh	Total
L1_SRAM	16.57	190.78	14.44		205.22
L2_SRAM	0.12	0.72	54.65		55.37
DRAM	0.10	0.47	6,239.21	121.21	6,360.89
Ancillary	+0.25	+0.25	-2.71		-2.46
SLCNAND	559.16	2,397.78	0.03		2,397.81
Total	576.20	2,590.01	6,305.61	121.21	9,016.83

Tab. 6.15: PGAS Elapsed Time and Energy Summary for SLCNAND Hybrid Execution

When taking parallelism into consideration, according to Figure 6.11 and table 6.15, even though most of the operations benefit from cache, the latencies from SLCNAND flash still dominate the overall elapsed time. However, 70.5% of the energy consumption can be attributed to

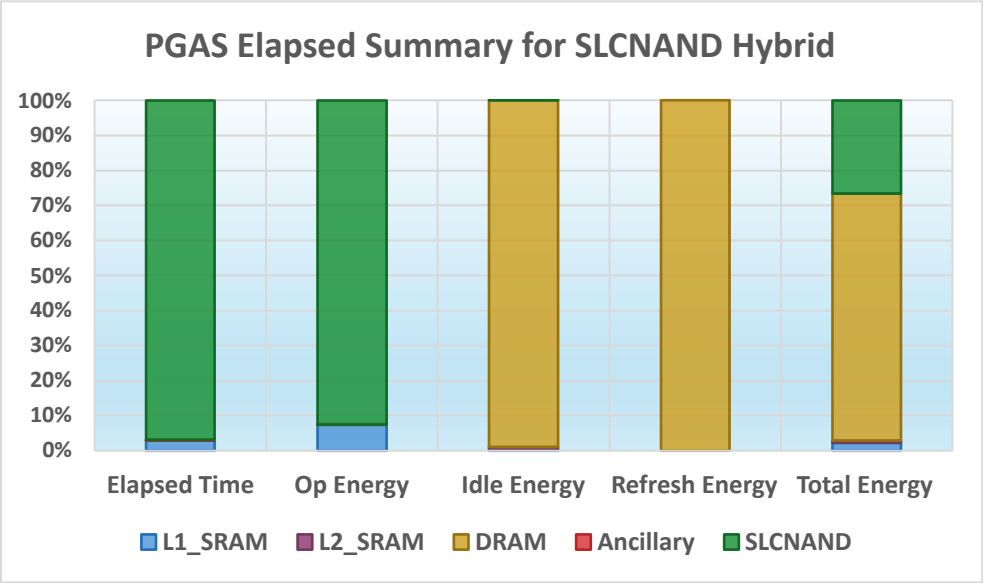

Fig. 6.11: PGAS Elapsed Time and Energy Consumption by Percentage for SLCNAND Hybrid

DRAM.

Configuration	Variables	Ancillary	Estimated Impact on Execution Time (s)	Energy (J)	Power (W)
DRAM Exclusive	DRAM	DRAM	17.17	378.67	22.05
SLCNAND Hybrid	Hybrid	DRAM	576.20	9,016.83	15.65
SLCNAND Mapped	NVM	DRAM	932.25	16,658.54	17.87
SLCNAND Exclusive	NVM	NVM	3,074.59	9,775.80	3.18

Tab. 6.16: SLCNAND: Estimated Lower-bound Execution Time Impact and Memory Energy Footprint

Table 6.16 and Figure 6.12 compare SLCNAND hybrid, mapped, and exclusive with DRAM exclusive. As the program relies more on SLCNAND to host some, all user-defined variables, and even ancillary allocations, the impact on execution time diverges greatly in comparison to DRAM exclusive. The energy consumption for SLCNAND hybrid and mapped are high due to DRAM's leakage and refresh energy requirements. SLCNAND exclusive consumes very little power due to a very low power leakage in it's hardware profile, no refresh energy requirement, and the fact that DRAM does not exist in this model. The energy consumption estimates only

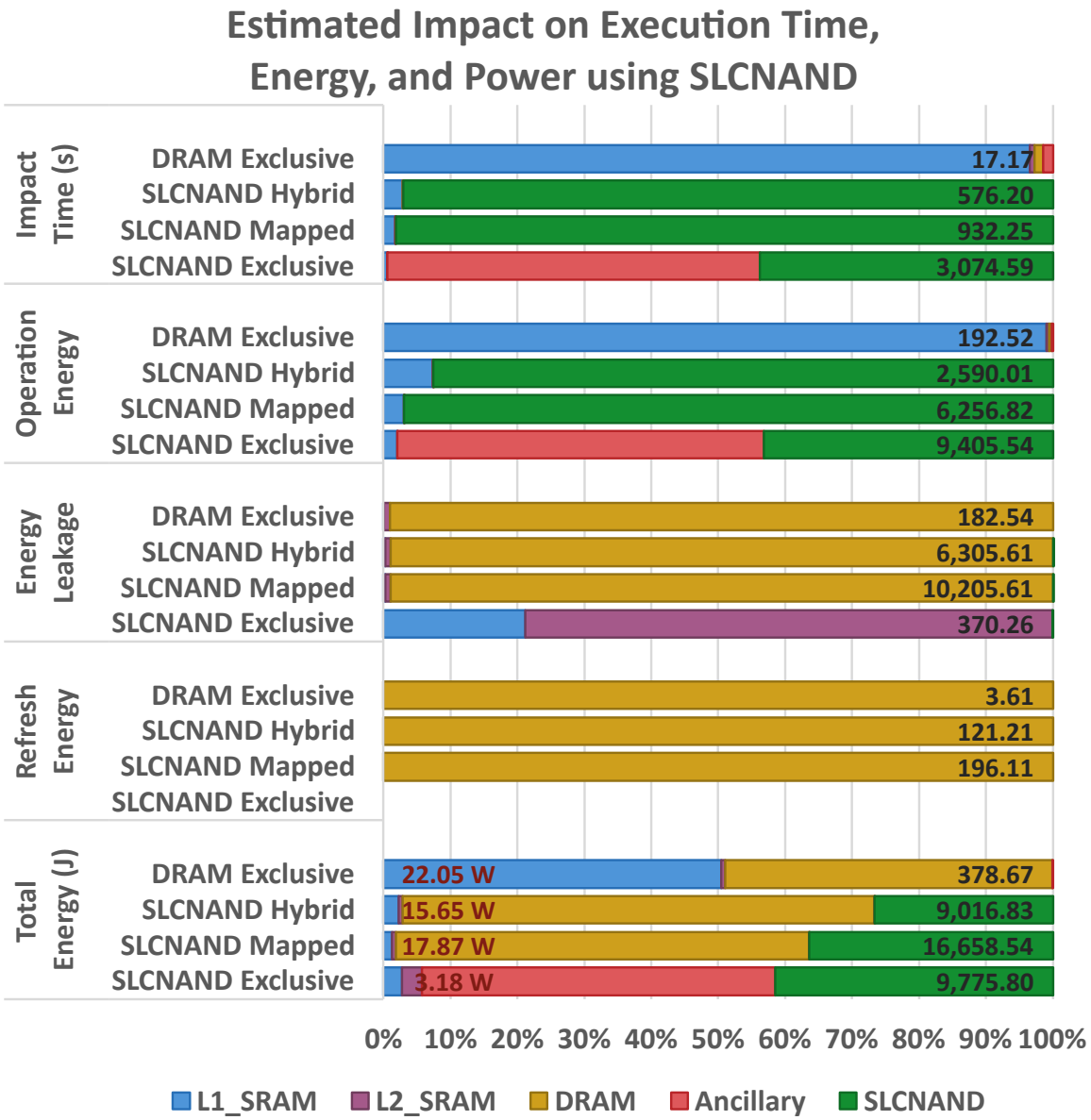

Fig. 6.12: Estimated Impact on Execution Time, Energy, and Power using SLCNAND

apply to the memory system. Network and CPU energy are not considered beyond L1 and L2 data cache on the cores used in the profiled execution.

For a second comparison, Zaratan compute node SATA SSD latencies were estimated and modeled to produce a NVM hardware profile. Since NVM was emulated during the microbenchmark's profiled execution, lets consider a configuration using the existing L1, L2, and DRAM profile latencies with the performance of a standard compute node's SATA SSD on Zaratan for the NVM environment. To measure the latencies for read and write operations on the SSD, I ran the 'iostat xd 1' command for 5 minutes on a compute node and parsed all of the nonzero r_await and w_await values for sda. The r_await and w_await metrics represent the average read and write times in milliseconds, including request time spent in the queue [128]. The minimum r_await (read) and w_await (write) are 0.08ms and 0.05ms which were applied to Zaratan SSD as NVM latencies.

Read and write dynamic energy of 0.48mJ and 0.325mJ for the SSD was calculated from figures (6W read, 6.5W write) derived from an article [129] on SATA SSD power consumption as well as an idle power (as opposed to leakage) of 1.125W. The difference is idle power is consumed to maintain power to a SSD device while idle and power leakage refers a rate at which a memory cell losses energy when not actively being accessed. High leakage can result in problems with data retention. However, SSD devices do not have this problem.

The SSDs on Zaratan's standard compute nodes composed the actual emulated hybrid-PGAS NVM environment that the profiled execution ran on. Therefore, I modeled and developed an NVM profile based on it. However, since the NVM profile is based on an SSD dual memory channels were not considered. To model the SSD as memory, I took the latency performance characteristics and energy estimations and applied them as a hypothetical byte-addressable NVRAM.

	Latency (ns)			
	Hardware Profile		Estimated AMAT	
NVM	Read	Write	Read	Write
PCRAM	33.8	183.4	0.861	0.835
SLCNAND	16,256.0	200,440.0	5.416	57.096
Zaratan SSD	80,000.0	50,000.0	45.781	28.878

Tab. 6.17: Comparing Zaratan SATA SSD await times with estimated latencies

The same calculations were then performed on the new configuration to produce the latency totals and estimated AMAT. Table 6.17 contains Zaratan SATA SSD read and write latencies derived from r_await and w_await as well as the cooresponding estimated AMAT. PCRAM and SLC-NAND flash were also added for comparison. The result is a really slow NVM environment that performs worse than SLCNAND flash.

			Estimated Impact on Execution		
Configuration	Variables	Ancillary	Time (s)	Energy (J)	Power (W)
DRAM Exclusive	DRAM	DRAM	17.17	378.67	22.05
Zaratan SSD Hybrid	Hybrid	DRAM	1,482.11	25,647.19	17.30
Zaratan SSD Mapped	NVM	DRAM	2,681.82	46,297.83	17.26
Zaratan SSD Exclusive	NVM	NVM	5,278.75	33,081.54	6.27

Tab. 6.18: Zaratan SSD: Estimated Lower-bound Execution Time Impact and Memory Energy Footprint

According to table 6.18, predictably the Zaratan SSD estimated impact on execution times diverge even more than SLCNAND flash. As a consequence of slower execution, Zaratan SSD hybrid, mapped, and exclusive also consume far more energy than the PCRAM and SLCNAND models, since the impact time is longer. As I have established, there are cases when NVM latency is so slow that NVM will never perform like DRAM.

6.2.6 Conclusion

In conclusion, cache performance profiling was studied and applied to establish estimations of latency hiding of slower memory such as NVM. Kernel variants in both Chapel and C/MPI/OpenMP for DRAM vs NVM over different compute node resources were cache profiled to model and compare their performance with various access patterns for L1 and L2 data cache. We learned that data cache miss rates on L1 for stream read and reduce are low, while strided write and kernels that use irregular access patterns yield very high miss rates. Microbenchmarks were profiled to illustrate how levels of cache can hide the latencies of slower memory environments lower in the hierarchy. The cache profiles of all microbenchmarks, except SpectralNorm and ptrans 4N8T, resulted in L1 data cache miss rates of around or below 1% and when combined with L2, without exception all microbenchmarks L1-L2 data cache miss rates were around or below 1%. I then used the cache profile of LULESH to estimate the impact on execution time, energy, and power for DRAM exclusive, hybrid, NVM mapped, and NVM exclusive cases for various configurations to determine when NVM performance is similar to DRAM and when it is not. PCRAM hybrid performed similarly to DRAM exclusive at 1.03x when using the cache profile and even showed robustness when L1 data miss rates were high, with 1.84x DRAM impact time for 100% miss rate. PCRAM exclusive diverged, indicating that ancillary objects should be handled by DRAM. However, SLCNAND flash and Zaratan SSD models diverged greatly from DRAM exclusive due to their high profile latencies. Therefore, it is important that the NVM technology has latencies that operate within 15x of DRAM, as is the case for PCRAM.

6.2.7 *Threats to Validity*

The follow items enumerate the assumptions and shortcomings that pose as threats to the validity of the studies in this section. Althrough the findings in the conclusion can serve as guidelines, a more expansive and deeper analysis may be necessary to determine the accuracy of the results and further identify nuances of NVM performance under different conditions.

1. The memory profile characteristics for PCRAM were derived from DESTINY version 1.0, released in 2016 and are based on properties adapted from models packaged with DESTINY and NVSim, included in sections C.1 and C.2. In one study, recent advancements in PCM technology yielded a latency of up to 20ns [130]. Although this is a vast improvement, it also means the PCRAM latencies used in the study maybe considered out of date. Therefore, estimations made in the analysis may underestimate the performance capabilities of real PCM. Furthermore, the study only considered SET and not RESET+SET for write operations. However, followup estimations indicated the difference is minimal.

2. Persistent memory offers opportunities for up to two orders of magnitude larger capacity than traditional memory. With larger memory utilization, further analysis is required to determine whether or not programming models and use patterns will change and how this will effect cache performance, memory and communcation access patterns on the PGAS.

3. Since the performance results were based on a relatively low number of threads per node, more analysis is needed to determine how increasing the number of threads on each node will impact level 3 shared cache and memory performance due to interleaving. This also requires hardware that supports L3 cache performance counters through an API like PAPI.

6.3 Checkpoint and Recovery

6.3.1 Architectures

Emerging NVM in HPC environment will likely see NVM support on each compute node. For instance Summit at ORNL provides 1600 GB of NVM per node [131] and Aurora at ALF uses Intel Optane DCPMM [132]. However a team at Hewlett-Packard is developing Chapel NVM support using Fabric Attached Memory (FAM) [11] [13]. The advantage of native compute node NVM resources is that it allows local utilization of NVM with lower latencies than FAM over an interconnect. However if a compute node fails then the NV heap on that node will become inaccessable. Therefore, NV heaps need to be backed up to a shared storage environment.

6.3.2 Hypothesis

The performance characteristics for recovery will be exactly the same in a consecutive execution since all loaded persistent memory objects will need to be remapped to the new virtual address space. Changing problem sizes and distributed mapping may have a marginal if not negligible impact on application performance for checkpoint operations. Since cached data is implicitly flushed back to NVM environements, explicit flush operations will be small and using checkpoint more often means the file operations for copying NV heap files will have the most impact on performance. Also, file operation times will grow linearly with NV heap file sizes. Finally, operating on remote heaps will degrade the overall performance of an application.

6.3.3 Evaluation

Checkpoint operations should be executed in a sequential section of the program. A scientific computing application will typically contain an outermost sequential loop that either uses a time step iteration, such as for a simulator, or converges with an acceptable error threshold when approximating a solution. Ideally a checkpoint would be added at the end of a sequential program loop so that in the event of system failure, a consecutive execution can continue from the last iteration. Listing 6.12 illustrates an example of this. Checkpoints can also be added between different phases of a program.

I provide a Checkpoint module within the Chapel framework that allows for explicit request to checkpoint data from Chapel source code. Checkpoint.backup() will perform a coforall over locales to flush all persistent objects back to NVM and then backup the NV heap file associated with that program for each compute node. Recovery is an implicit process that I incorporated in the runtime environment that occurs before the user main is executed.

For the evaluation of checkpoint in Chapel, analysis was conducted to answer a few questions. What is the performance degredation and risk vs. reliability when checkpoint operations are performed at different frequencies? When backup occurs, the compute node's NV heap will be stored in a remote location. In the event a node fails, all NV heaps will still be accessible remotely and can be mapped to new computation resources to continue the execution. How might application performance be effected if each compute node's active heap was remote?

The LULESH microbenchmark was used for the analysis. The Chapel implementation of LULESH is a good candidate since the main function contains a sequential outer loop. The main loop has two sentinel cases, stop time and maximum cycles. Similarly to listing 6.12 checkpoint

Listing 6.12: Checkpoint General Idea

```
use Checkpoint;
...
config const n = 100000;

// these variables need to be persistent so that
// restore can continue from the previous execution
var s : persist int = 1;
var e : persist int = n;

// all data stores that need to be
// recovered should also be persistent
var ddom : persist = ...
var data : persist = ...

proc main() {
   // set the range for this execution
   var r = s..e;
   ...
   // perform initialization only on the first run
   if Checkpoint.initial_execution() then performInit();
   ...
   // sequential outer loop
   for i in r do {
      // parallel region(s)
      forall ...
         ...
      ...
      // update the start to the next iteration
      // for recovery and consecutive execution
      s = i + 1;
      // flush and backup the heaps in a
      // sequential part of the program
      Checkpoint.backup()
   }
}
```

was added at the end of the loop. Executions were performed on UMD's Zaratan Cluster. During execution, an active NV heap was located on each compute node's SSD and also stored remotely on Zaratan's shared BeeGFS parallel file system during each checkpoint operation.

How do different intervals of checkpoint operations effect the application?

LULESH was executed with an input size of 52 elements per edge over various resource allocations and heap sizes per node, totaling 96GB. The analysis compares the execution times of runs with checkpoint disabled against runs that checkpoint 3 times and 9 times during the program's execution to evaluate the impact of checkpoint operations on execution. Quarterly backups perform 3 checkpoint operations at or around 25%, 50%, and 75% of the execution's progress respectively and 10 percent executions perform 9 backups since there is no checkpoint operation at the beginning or end of execution.

Figure 6.13 contains two charts detailing execution performance and risk analysis for the checkpoint system. The analysis compares no checkpoint vs. checkpoint scheduled for every 25% and 10% of the time. The total percentage of execution time spent in checkpoint was 1.2%, to 5.4% depending on the compute node resource allocation for quarterly and 6.7% to 11.6% for 9 checkpoints, resulting in a 2.1x to 6.7x increase vs 3 checkpoints. A checkpoint backup operation first performs flush operations over all NVM allocated objects, which has been recorded to total in 10's of microseconds, and then stores the NV heap file of each compute node to a remote location. The store operation time scales linearly with the heap size and comprises of the majority of the checkpoint backup time as outlined in table 6.19. The median risk time was computed from the elapsed times between checkpoint operations. In the event of a failure, risk time represents the maximun elapsed time at risk for a given checkpoint frequency, as a consecutive execution

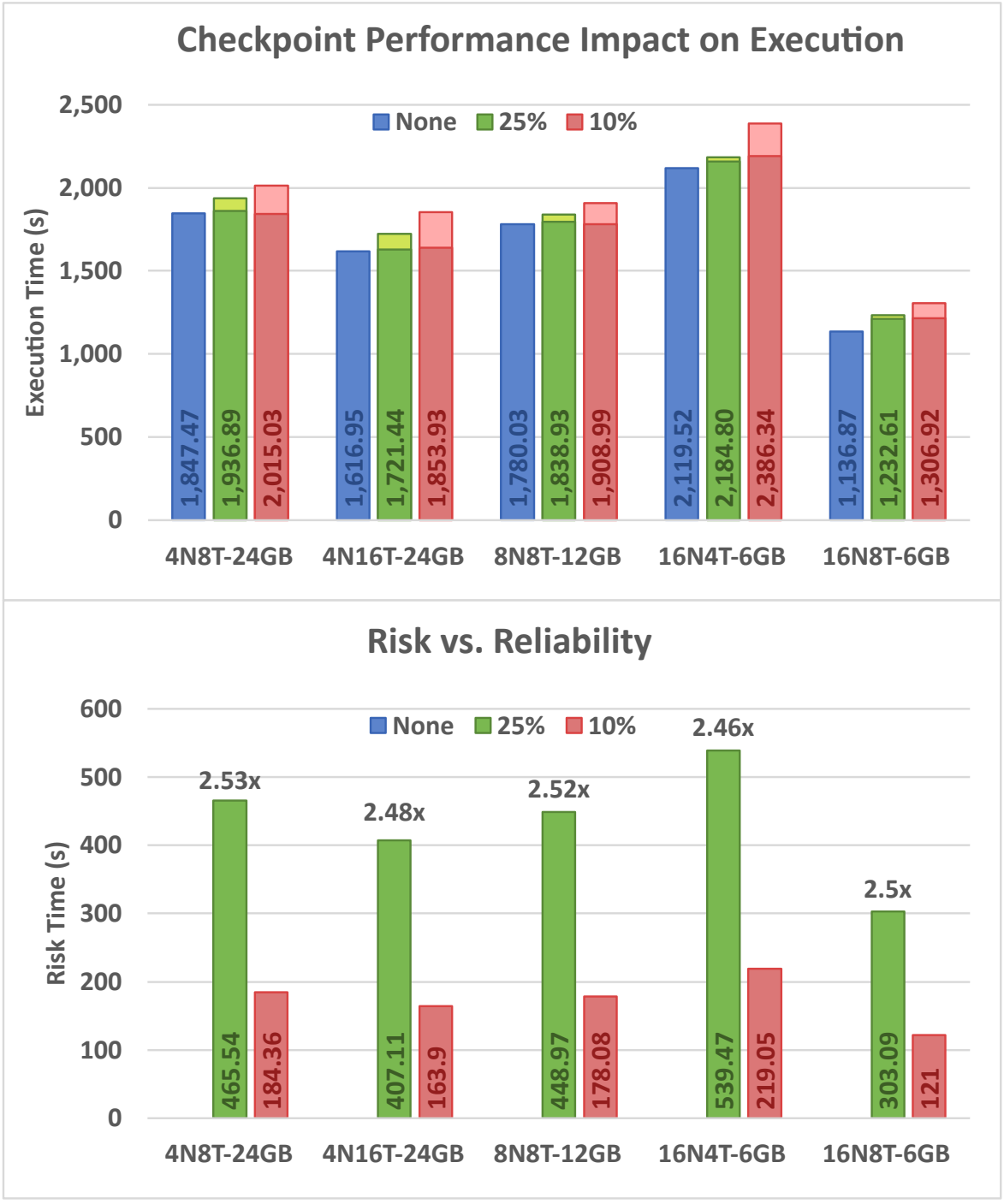

Fig. 6.13: Performance and Risk vs. Reliability: LULESH, elemsPerEdge=52

will use NV heap files and restart from the last checkpoint operation. Risk time is based on the division of execution time by checkpoint operations. When comparing quarterly and 10%, the risk time is around 2.5x higher for the quarterly regardless of the compute node and CPU resource allocation.

What is the performance impact of using a local vs. remote active heaps?

Lets consider two different computer architectures that use NVM technologies. For the first one, each compute node hosts onboard support for a local NVM and backups are stored at a remote location, such as a shared parallel file system. When an application is executed, if they exist, remotely stored heaps are mapped and transfered to their respective compute nodes and become local active heaps for the duration of the execution. This architectural approach to backup and restore will be referred to as local heap. In the second case all persistent memory is remote, for example fabric attached memory. Both active and backup heaps of each compute node are stored remotely, which will be referred as remote heap.

When an application initially executes, each compute node creates a NV heap file. During checkpoint, a flush and store of each NV heap file to a remote destination is performed. In the event of a failure, the state of the NVM-PGAS can be recovered from the last checkpoint and reconstructed for new computational resources in a consecutive run. Recovery performs a restore operation, which is responsible for retrieving the NV heaps, loading persistent memory pools, and remapping the direct references of persistent objects.

For the experiment, LULESH was executed with an input size of 16 elements per edge on Zaratan with different heap sizes to compare the performance characteristics of local vs. remote active persistent heaps for create, backup, and restore operations, to determine the advantages and

Local Heap

	Heap Size	1GB	2GB	4GB	8GB	16GB	32GB
Create	Create Time (s)	0.02	0.02	0.03	0.03	0.04	0.06
Backup	Flush Time (ms)	0.02	0.03	0.04	0.02	0.02	0.01
	Store Time (s)	0.77	1.20	3.10	5.44	13.34	27.52
	Store Rate (GB/s)	1.29	1.67	1.29	1.47	1.20	1.16
Restore	Retrieve Time (s)	0.47	1.15	2.26	4.89	9.77	20.35
	Retrieve Rate (GB/s)	2.11	1.74	1.77	1.64	1.64	1.57
	Load Time (s)	0.02	0.02	0.02	0.02	0.02	0.02
	Remap Time (s)	0.01	0.01	0.01	0.01	0.01	0.04

Remote Heap

	Heap Size	1GB	2GB	4GB	8GB	16GB	32GB
Create	Create Time (s)	9.63	18.73	36.54	71.25	148.39	298.20
	Create Rate (GB/s)	0.10	0.11	0.11	0.11	0.11	0.11
Backup	Flush Time (ms)	0.02	0.02	0.02	0.02	0.02	0.02
	Store Time (s)	0.51	0.75	1.82	3.48	9.01	16.64
Restore	Retrieve Time (s)	1.27	2.41	4.53	10.74	19.78	41.74
	Load Time (s)	0.11	0.05	0.08	0.04	0.02	0.04
	Remap Time (s)	0.04	0.05	0.05	0.03	0.03	0.05

Tab. 6.19: Create, Backup, and Restore Times for Local and Remote Heaps on Zaratan

disadvantages of each backup and restore strategy. Table 6.19 contains the times and throughput rates of each operation for local and remote heaps from 1GB to 32GB size. All times are the median values derived from three to five executions. One of the most noticible differences is the heap creation times for local vs. remote. The create and load times for local heaps are roughly the same. However, remote heap create times are from 467x to 4982x greater than local heap creation. In both cases, the time for backup and restore operations is dominated by their store and retrieve operations respectively, which grows linearly with the NV heap file size. Retrieve times are roughly double for the remote case. Flush, load, and remap times are marginal for local and remote regardless of heap size, with flush operations having the smallest time values by several

orders of magnitude.

Figure 6.14 is a graphical representation of the timing values from table 6.19, divided into the three categories of create, backup, and restore over different heap sizes, and charted on a binary logorithmic scale. The totals for each category are given in the bar charts. The operations with the smallest time values were prioritized to the lowest parts of the stacked logorithmic chart to illustrate their impact on the overall operation category. If you look very carefully, there is a faint line at the bottom of the bars for backup that represents the flush operations. Load and remap have roughly similar time values per table 6.19, but on a log scale, load operations appear much larger. The create time for remote active heaps, which is incurred at the beginning of the initial execution dominates all other operational time values. Using PMDK to create a heap remotely significantly degrades performance and will undoubtedly impact the execution time.

Problem Size		8	16	24	32	40	48	56	64
Flush (ms)	Local	0.020	0.022	0.025	0.037	0.043	0.047	0.048	0.052
	Remote	0.012	0.011	0.018	0.033	0.043	0.047	0.055	0.045
Remap (s)	Local	0.013	0.013	0.013	0.013	0.014	0.011	0.011	0.010
	Remote	0.014	0.014	0.014	0.014	0.013	0.015	0.015	0.016

Tab. 6.20: LULESH: Local and Remote Heap Flush and Remap Times by Problem Size

Table 6.20 contains results from an additional study which was conducted to determine whether flush and remap operation times grow with the problem size. When comparing problem sizes for 8 with 64, flush operation times increased by 2.6x and 3.75x for local and remote heaps. However, flush times remain in the tens of microseconds. Remap operations did not change significantly with different problem sizes and there is no noticible difference between local and remote heaps. Any changes in time for both of these operations can be almost considered neglible.

For the next part of the analysis, stream and strided read and write kernels were used to

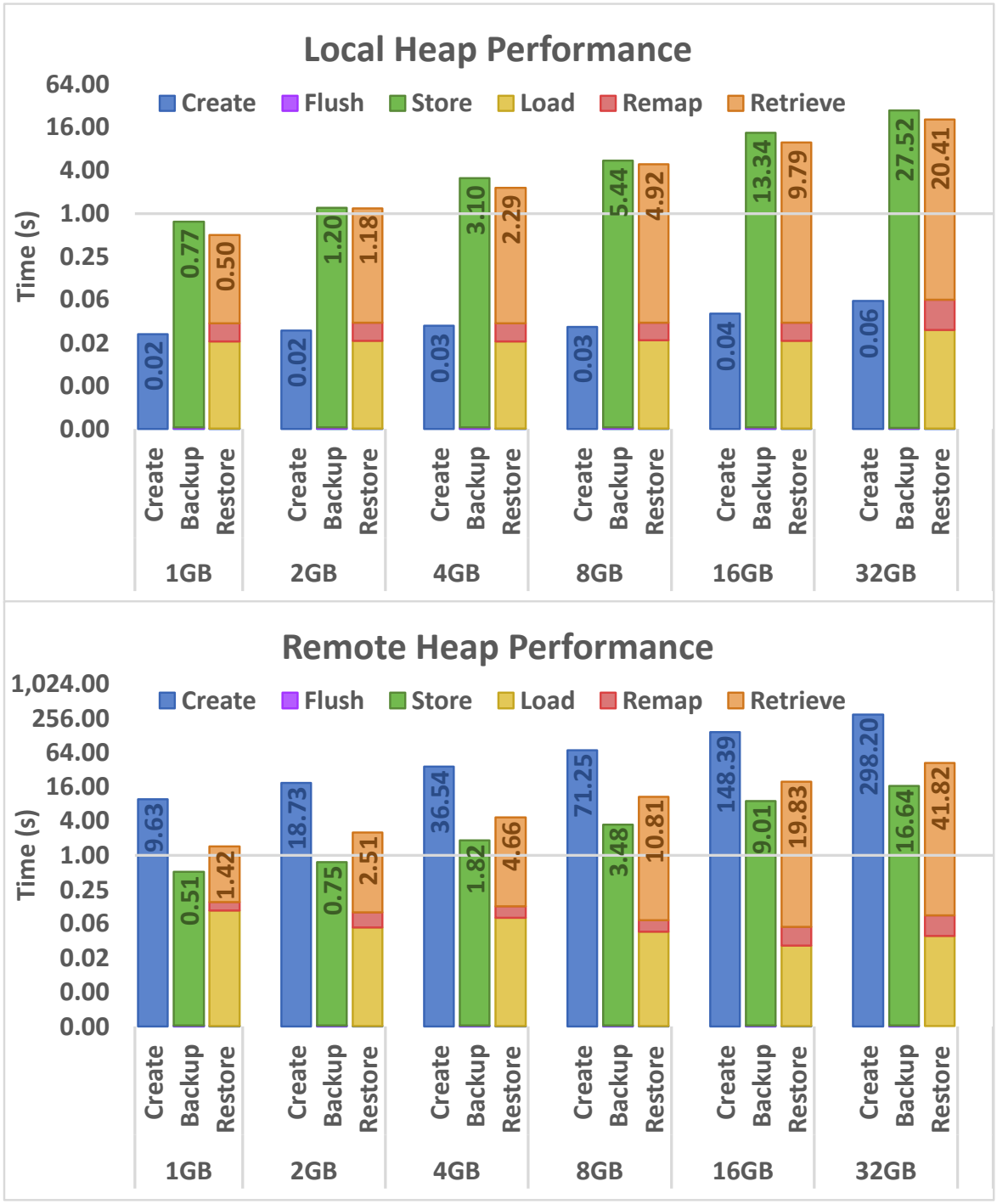

Fig. 6.14: Local and Remote Heap Performance Characteristics for Create, Backup, and Restore

determine how read and write operations will perform on local and remote heaps. Recall from subsection 6.2.3 on access patterns, stream read operations yielded 0.57% and 1.04%, and strided write operations produced 99.62% and 98.15% average L1 data cache miss rates for Chapel and C serial executions. When misses occur on all levels of cache, addtional penalties are incurred since the block must be loaded from an environment lower in the memory hiearchy. In addition, accessing persistent memory through PMDK may also involve mmap operations. Active heaps residing in a remote location only further contributes to slower access times.

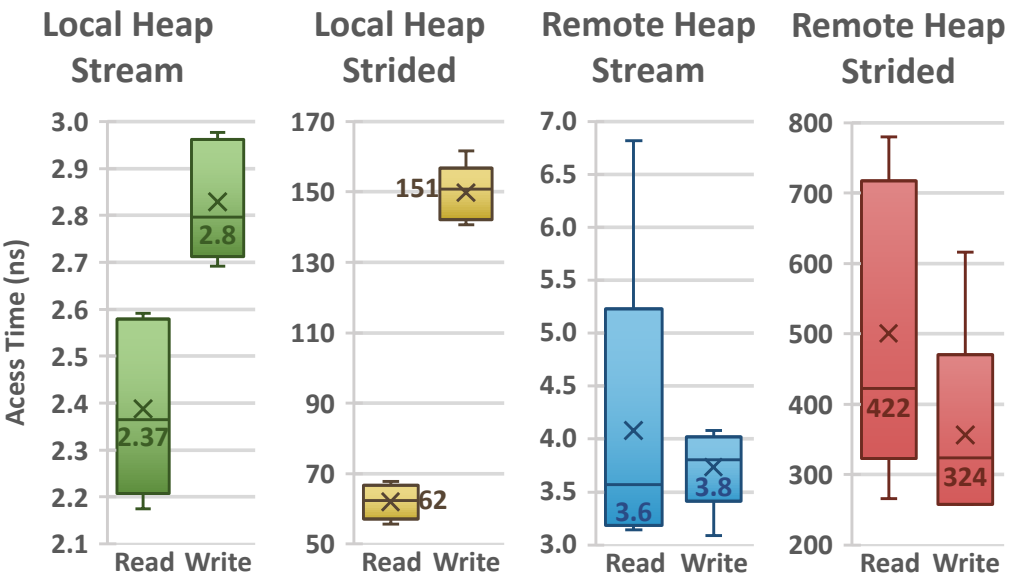

Fig. 6.15: Local vs. Remote Heap: Best vs. Worst Case Access Times

Figure 6.15 illustrates the read and write access times for stream and strided operations performed on local and remote active heaps. Median access time values have been included in the box and whisker charts. As demonstrated, remote heap read and write accesses are 1.5x to 6.8x and 1.4x to 2.2x higher respectively than local heap accesses. Therefore, accessing active persistent heaps from a remote location will inevitably yield much higher access times, which

will significantly degrade the performance of an application.

6.3.4 Conclusion

Checkpoint overhead is 1.2% to 5.4% and 6.7% to 11.6% of the execution time for 25% and 10% frequencies respectively. Increasing checkpoint frequency from 25% to 10% yields from 2.1x to 6.7x more overhead, while the quarterly checkpoint option risk is around 2.5x the at risk time of the 10% checkpoint. Most of the backup and restore operation time is spent performing file copy and transfer which increases linearly with the heap size, according to table 6.19. Regardless of whether restore is performed during the current execution or via recovery in a consecutive execution, retrieve, load, and remap operations are still required. Therefore, the restore times will be similar. Performing operations on remote active heaps generally yielded better store times but worse retrieve and abysmally longer creation times than local active heap designations when using SATA SSDs on compute nodes and the BeeGFS on Zaratan to simulate NVM. Larger problem sizes increased flush operation times by 2.6x and 3.75x for local and remote heaps and had virtually no effect on remap operation time. However, read and write access times were 1.5x to 6.8x and 1.4x to 2.2x higher on remote heaps.

6.4 Finding Performance Bottlenecks

Performance is important for any HPC scientific computing application. When performance problems occur they may be difficult to find or understand and will inevitably effect both the development and turnaround time of an application, which is only magnified by a potentially slower NVM environment that creates a new source of potential bottlenecks. In this section, I

conduct a case study to derive and tune a 2D Poisson equation solver and 5-point stencil kernel Chapel implementations that use NVM. I employed the Purity data-centric profiler to identify and fix performance bottlenecks that occur.

6.4.1 Derivation and Implementation

Equation A.3 from section A.1 in Appendix A was used to develop the iterative solver. Listing 6.13 is the implementation of a 2D Poisson's equation solver using the Jacobi method, which was developed in Chapel based on derivations [133] [134] [135] [136] and a Chapel tutorial on data parallelism with Jacobi [120] [137]. All relevant user-defined variables are stored in persistent memory indicated by the $persist$ type qualifier in the declaration.

Listing 6.14 is a simplified and unoptimized 5-point stencil version of listing 6.1, which was derived from listing 6.13. A data performance bottleneck was introduced into this kernel intentionally so that the Purity Data-Centric Profiler can demonstrate how performance bottlenecks are identified during execution. The kernel performs only the stencil calculation of the iterative equation solver over a two-dimensional matrix which is defined in the listing by the configuration constant n. Generally, a convergence test in a sequential outer loop would be used to determine when the condition has been meet, e.g. the approximation is sufficiently close as indicated by $residual$ being within the $epsilon$ convergence threshold in listing 6.13. However, the kernel uses $step$ to control the number of iterations in the outer loop, instead of a convergence test which relies on the input data to determine iterations. The variable $outer$ represents a two-dimensional outer interval range based on the input from n and $inner$ is an inner interval range, both of which are used to define the domain D. Block distribution with bounding box is used for mapping domain D over an arbitrary number of locales.

Listing 6.13: 2D Poisson's Equation Solver using the Jacobi Method in Chapel with NVM

```
use BlockDist;

config const n : int = 2500;
config const epsilon : real = 1.0e-5;
const dx : real = 1.0 / (n + 1.0);
const dx_sq : real = dx * dx; // n x n

var outer : persist = {0..(n+1), 0..(n+1)};
var inner : persist = {1..n, 1..n};
var D : persist = outer dmapped Block(boundingBox=inner);
var P : persist = D[inner];
var u, v, f : persist [D] real;

proc main() {
    generate_data(); // store data in u[P]
    initialize();    // compute f(x,y)
    solve_eq();      // approximate solution
}

proc initialize() {
    forall (i,j) in P do {
        f[i,j] = (u[i-1,j] + u[i+1,j] +
                  u[i,j-1] + u[i,j+1] -
                  4.0 * u[i,j]) / dx_sq;
    }
}

proc solve_eq() {
    u = 0.0; // set all cell values of u to zero

    do {
        v = (-dx_sq / 4.0) * f;

        forall (i,j) in P do {
            v[i,j] += (u[i-1,j] + u[i+1,j] +
                       u[i,j-1] + u[i,j+1]) / 4.0;
        }

        const residual = max reduce abs(v[P] - u[P]);
        u[P] = v[P];
    } while (residual > epsilon); // convergence test
}
```

Listing 6.14: Unoptimized 5-Point Stencil Kernel from 2D Poisson's Eq. Solver using Jacobi

```
use BlockDist;

config const n : int = 2500;
config const steps : int = 100;

var outer : persist = {0..(n+1), 0..(n+1)};
var inner : persist = {1..n, 1..n};
var D : persist = outer dmapped Block(boundingBox=inner);
var P : persist = outer[inner];
var u : persist [D] real;
var v : persist [D] real;

proc main() {
    for s in 1..steps do {
        forall (i,j) in P do {
            v[i,j] = (u[i-1,j] + u[i+1,j] +
                      u[i,j-1] + u[i,j+1]) / 4.0;
        }

        u[P] = v[P];
    }
}
```

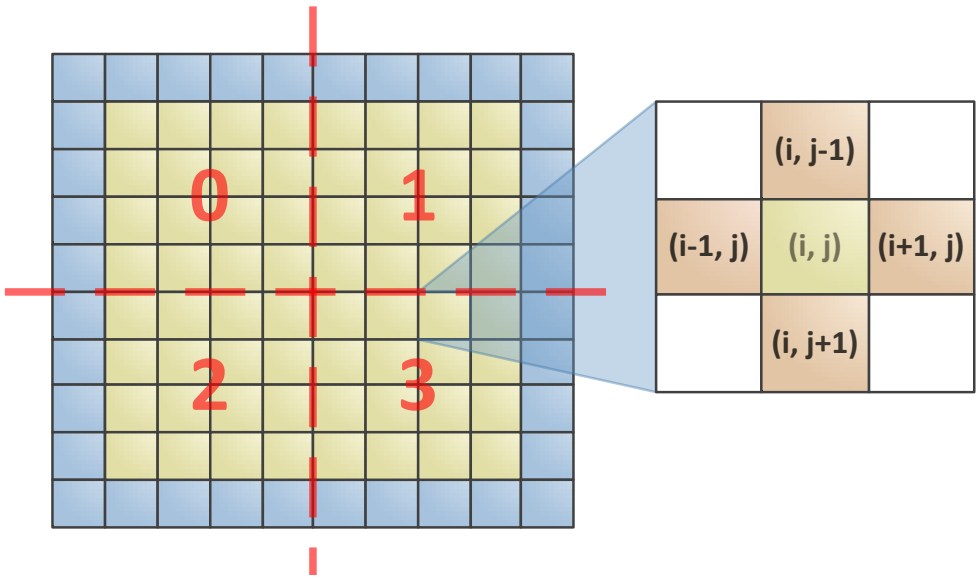

Fig. 6.16: Data Boundries, Distribution, and 5-Point Stencil for Iterative Solver and Kernel

Figure 6.16 illustrates how a 2D matrix is distributed over four locales. The blue cells represent the outer boundary defined by an outer range and the inner area in yellow is defined by an inner range. The domain D determines how matrices u and v are distributed. The kernel performs a 5-point stencil calculation with the adjacent cells of $u_{i,j}$ which is divided by 4 and stored in $v_{i,j}$. Cell values are accessed via 2-tuple indices provided by a forall loop that performs data parallel iterations over the inner index range P, defined by a range slice of $outer$ by $inner$. Remote communications are generated when locales require cell values stored at other locales, marked by the red boundries.

6.4.2 Evaluation

The experiment was conducted on persistent memory variants of the Jacobi stencil kernel implemented in Chapel. Purity was employed to develop a detailed profile of executions performed on 4 nodes with 8 threads per node, 1 thread per cpu, at a sample rate of 0.029%. The problem size was defined as 2500 x 2500 matrix size (n=2500) with 100 steps where each step performed stencil calculations over matrices.

The execution time for the unoptimized version is 4,419 seconds. Given the input size and compute node resources the resulting time value should be regarded as extremely slow. For comparison an MPI / OpenMP in C implementation of the Jacobi kernel with persistent memory can complete an execution with an input size of n=10000, 1000 trials, 10 steps per trial in 116.5 seconds with the same resources. How can the profiler be used to find performance bottlenecks and identify what to address in the Chapel variant? Purity provides results from a number of different analyses for determining why a Chapel program is not running efficiently. Highlights from heap-level, node-level, and loop-level analysis have been provided as well as a summary of

operations for both unoptimized and optimized versions of the Jacobi stencil kernel.

Heap-Level Analysis

Persistent	Node			Read		Write	
Variable	Src	Dst	Thread	Count	Bytes	Count	Bytes
D	0	0	0	1,422,443	36	22,758,998	672
D	0	0	-2	1,019,820	20	16,316,912	400
D	0	0	-3	1,093,771	0	17,500,336	0
D	0	0	-4	1,071,513	0	17,144,208	0
D	0	0	-5	1,064,834	0	17,037,344	0
D	0	0	-6	1,088,002	0	17,408,032	0
D	0	0	-7	1,088,539	0	17,416,624	0
D	0	0	-8	1,032,078	0	16,513,248	0
...	...	...	...	...	...	...	...

Tab. 6.21: Top Number of Heap Operations for User-Defined Variable D

Table 6.21 represents the top number of operations for the domain D. Heap-level analysis provides aggregate totals for operations, bytes, estimated latencies, and throughput per user-defined variable broken out by source and destination nodes and threads for each memory environment. Variables are ranked by the total number of operations for each heap. For the case study, D was the top ranking variable with almost all accesses being local and occuring on the main locale, represented by node 0. The access count for D is 16x and 61x larger than operations on u and v and the local counts are 15,365x higher than all other accesses for D combined.

Node-level analysis provides an access matrix between source and destination nodes for each user-defined variable. A source node can be described by the locale that is making a request and the destination node is the locale that services that request. These requests translate into get and put operations between compute nodes on the PGAS. Diagonal counts represent local access while the upper and lower triangle parts of the matrix are remote accesses. In perfect data parellism, all matrices would have evenly distributed values along the diagnonal and relatively

small remote values on the upper and lower regions.

Node-Level Analysis

	Destination			
D	Node 0	Node 1	Node 2	Node 3
Node 0	8,881,056	0	0	0
Node 1	2	220	0	0
Node 2	2	0	164	0
Node 3	0	26	0	164

u	Node 0	Node 1	Node 2	Node 3
Node 0	108,780	162,218	136,938	148,087
Node 1	0	5	0	0
Node 2	0	0	0	0
Node 3	0	0	0	0

v	Node 0	Node 1	Node 2	Node 3
Node 0	27,372	43,325	36,398	39,378
Node 1	0	2	0	0
Node 2	0	0	0	0
Node 3	0	0	0	0

Tab. 6.22: Top Operations by User-Defined Variable over the PGAS

Table 6.22 contains the access matrices for D, u, and v. Variables are ranked by total number of operations. Almost all of the operations for D are local accesses on node 0. Also node 0 is making all of the requests for accesses to u and v. Given these facts, it appears that the stencil calculations are being performed by node 0. But how can this be the case when D, u, and v are distributed and the calculations are using a forall loop? Lets investigate further and find out.

Loop-level analysis produces a view on data accesses by loops and locations in both the user source and Chapel modules, broken down by variables and a category for unmapped activity, otherwise known as ancillary operations. Table 6.23 shows a large number of local reads for D and ancillary in contrast to the activity of u and v over the $step$ for loop in the user main. Nearly half of the operations for D are coming from line 583 in DefaultRectangular.chpl which refers to a

Loop-Level Analysis

Location	User-Defined Variable	Local Reads	Local Writes	Remote Reads	Remote Writes
DefaultRectangular.chpl:583	D	4,435,862	0	1	0
ChapelArray.chpl:1362	D	4,445,064	0	0	0
BlockDist.chpl:1049	Unmapped	2,174,161	0	0	0
BlockDist.chpl:1060	Unmapped	2,156,469	0	0	0
BlockDist.chpl:1061	Unmapped	2,139,880	0	0	0
BlockDist.chpl:1062	Unmapped	2,123,599	0	0	0
poisson..._unopt.chpl:14-20	u	447,243	0	108,780	0
poisson..._unopt.chpl:14-20	v	1	119,101	0	27,372
...	...	...	...	...	...

Tab. 6.23: Top Locations in the User-Source and Modules with the Highest Number of Operations

procedure called dsiDim() that is responsible for resolving range information from a domain and the other half from line 1,362 in ChapelArray.chpl which calls dsiDim(). Lines 1,049,1,060-1,062 in BlockDist.chpl exist within a procedure called nonLocalAccess() and produce a large number of local ancillary operations when attempting to access an internal structure $RAD(rlocIdx)$. The nonLocalAccess() procedure is called by dsiAccess() for a block array, which is responsible for resolving both local and remote indexed values. However, the concern is the number of local ancillary operations generated by the calls to nonLocalAccess(). This would indicate that D is not the primary culprit, but that both D and ancillary are symptoms of an underlying problem.

The true source of the problem is what is driving the forall loop. The inner index range P was defined as a range slice of $outer$ by $inner$. However, these closed-interval ranges only exist on node 0 and thus P as well. A forall loop over P does in fact perform data parallelism, but only over the threads on node 0. As a consequence, node 0 must perform all of the stencil calculations. However, the 2-tuple indices from P do not map to D across compute nodes. Therefore, a large number of operations are generated when resolving indexed matrix values.

Listing 6.15: Optimization: Change *outer* to D on line 9

```
var P : persist = D[inner];
```

Now that the performance bottleneck has been identified and fully understood, the change that is required to reach an optimized solution is quite small. Listing 6.15 provides the solution by switching the interval range *outer* for the distributed domain D. Since the new P is now a domain slice of D, P inherits the block distributed mapping of D. This time when P used in the forall loop, the data parallelism will scale over all compute node resources and allow a workload distribution on mapped data that maximizes task and data affinity. As a result, the number of remote operations is reduced to the data boundries between compute nodes and most of the D and ancillary operations never occur.

Summary Reports for Both Executions

Operation	Unoptimized	Optimized	
Local Reads	60,321,664,800	2,200,393,998	3.65%
Local Writes	255,691,911	1,162,631,689	454.70%
Remote Reads	1,874,979,497	1,000,822	0.05%
Remote Writes	468,599,193	15	$\sim$ 0.00%
Total	62,920,935,401	3,364,026,524	5.35%
Remote	3.72%	0.03%	
Execution(s)	4,419.34	56.627	78x

Tab. 6.24: Summary Report: Unoptimized vs. Optimized Jacobi Stencil

Purity's dynamic analyzer produces a summary report at the end of a profiled execution detailing the total number of accesses that were detected. The optimized Jacobi stencil kernel was also profiled for comparison with the unoptimized version. According to table 6.24, the Jacobi optimization yielded a speedup of 78x, finishing in only 56.6 seconds in contrast to the unopti-

mized version which completed in 4,419 seconds. Mitigating the performance bottleneck reduced the total number of operations by 18.7x. Also the percentage of remote operations dropped from 3.72% to 0.03% even though the overall total was reduced. However the number of local write for the optimized kernel is 454.7% of the unoptimized.

6.4.3 Conclusion

In conclusion, I derived the fomula for implementing a 2D Poisson's equation solver using the Jacobi method in Chapel with NVM and produced both unoptimized and optimized 5-point stencil Jacobi kernel variants. The execution of the unoptimized kernel was profiled using the Purity Data-Centric Profiler to identify performance bottlenecks. An investigation involved heap, node, and loop-level analysis to discover and mitigate the underlying problem, resulting in a speedup of 78x.

Chapter 7: Conclusion

7.1 Summary

I extended the Chapel language with intrinsic and programmatic features that incorporate NVM in a hybrid-PGAS model. A transaction system was developed for the Loci library which was integrated into the Chapel runtime framework, that when combined with Chapel concurrency control language features and GASNet, enforces ACID properties across all locales. I implemented a distributed checkpoint and recovery system for Chapel applications that use persistent memory. An NVM extension was added to the Purity data-centric profiler to support profiled executions with DRAM-NVM hybrid-PGAS environments. In addition, Purity reporting was expanded to include heap and remote operation analysis, object lifespan, and an impact on execution estimator for different memory profiles. I also illustrated why developing in Chapel is faster and easier than C by comparing the implementation of a Jacobi stencil in Chapel vs. MPI / OpenMP in C (nearly 6x more lines).

7.1.1 When does DRAM perform like NVM?

I performed an experiment to evaluate memory-based performance metrics of kernel and microbenchmark variants that utilize NVM and compared NVM performance with DRAM. The cache monitor profiling system I developed was applied to understand how levels of cache hide

the latencies of slower memory environments lower in the hierarchy. Chapel and MPI / OpenMP C kernel variants for DRAM and NVM were executed over different compute node resources to model the L1 and L2 data caches and compare the performance for kernels that represent distinct access patterns. My findings indicate that the L1 data cache miss rates for stream read and reduce kernels are low regardless of which memory environment variables are stored and kernels with irregular access patterns, (random read, random write, scatter, and gather) yield very high miss rates. As expected all strided write kernel executions exhibited near 100% miss rates on L1. The cache of LULESH, FFT, HPL, ptrans, Mandabrot-fast, and SpectralNorm executions were also profiled to establish estimations of latency hiding of slower memory in microbenchmarks. My results conclude that all microbenchmarks, except SpectralNorm and ptrans (for 4N8T), yielded L1 data cache miss rates of around or below 1%. This means that L1 data cache handles nearly 99% of all accesses to slower memory during the execution for most of these microbenchmarks. When examining the percentage of accesses that were missed by both L1 and L2, without exception all microbenchmarks L1-L2 data cache miss rates were around or below 1%. Therefore, around 99% of all accesses to slower memory are handled by either L1 and L2 data cache for all of the microbenchmarks in the study.

Next I conducted a study to explore and evaluate different NVM technologies and how they perform when compared with DRAM. Using the LULESH microbenchmark I employed the Purity data-centric profiler with NVM support, cache monitor profiling, and the DESTINY simulator for memory profiles of SRAM, DRAM, and different NVM technologies. I explored different architectural configuration options using these memory types to estimate total latency, total energy, simplified AMAT, and the impact on execution time, energy, and power with different cache characteristics for DRAM exclusive, DRAM-NVM hybrid variable store (hybrid),

NVM variable store (mapped), and NVM exclusive cases to determine when and how well NVM performs like DRAM and when it does not.

When observing impact on execution time, PCRAM hybrid performed similarly to DRAM exclusive with a 3% difference in performance (1.03x) when cache profiles were employed. Similarly, the normalized total latencies for PCRAM and DRAM yielded a 3.1% difference (1.031x) and the estimated simplified AMAT showed differences in read (1.007x) and write (1.061x) times for DRAM and PCRAM to be very small. DRAM exclusive was only 1.16x faster than PCRAM exclusive and 1.05x faster than PCRAM mapped. PCRAM exclusive was 11% more energy efficient than DRAM exclusive and 23.5% more power efficient. The results from the L1 data cache miss rate sweep indicate that PCRAM hybrid was 1.12x DRAM exclusive at 1% L1 miss rate, 1.56x at 10%, and 1.87x at 100%, demonstrating a robustness with high L1 miss rates. The outcome shows that when L1 data cache miss rate is low, the performance for PCRAM is similar to DRAM. However, since the PCRAM exclusive case diverged, ancillary objects and operations should be managed by DRAM main memory. As expected the SLCNAND flash and Zaratan SSD models diverged greatly from DRAM exclusive due to their high profile latencies. 70.5% of the energy consumption can be attributed to DRAM for the SLCNAND hybrid model.

Therefore, the finding from this study confirm the hypothesis. We learned that due to the latency hiding benefits gained from low L1 and L2 data cache miss rates, certain NVM technologies can perform similarly to DRAM so long as their latencies operate within 15x of DRAM, as is the case for PCRAM. Although differences do exist between DRAM and NVM, there is a trade-off between impact on execution time and increased capacity, persistency, and energy efficiency when storing some or all user-defined variables in NVM.

7.1.2 Checkpoint and Recovery

An experiment was conducted using the outer sequential loop of LULESH to examine the efficacy of the checkpoint and recovery system. The study compared 25% checkpoint frequency with 10% frequency, using no checkpoint executions as the baseline for various compute node resource allocations. Performance impact analysis showed checkpoint overhead was between 1.2% to 5.4% and 6.7% to 11.6% of the execution time for 25% and 10% frequencies respectively. I established that the tradeoff between checkpoint overhead and risk time is proportional with 2.1x to 6.7x more overhead when transitioning from 25% to 10% and 2.5x less risk to loss of computation due to a failure.

Next I studied the performance differences between compute node NVM support and remote shared NVM (like fabric attached memory) by simulating these persistent memory environments on compute node SATA SSDs and Zaratan's BeeGFS respectively, for local and remote designations of active heaps during execution. The findings indicate that backup and restore operations performed on remote active heaps generally yielded better store times but worse retrieve and extremely longer creation times when compared with local active heap designations. Read and write access times were 1.5x to 6.8x and 1.4x to 2.2x higher on remote heaps. Most of the backup and restore operation time was spent performing file copy and transfer, which linearly increased with heap size. However, the restore time will be similar to the recovery time since retrieve, load, and remap operations are required for both. Larger problem sizes increased flush operation times by 2.6x and 3.75x for local and remote heaps and had virtually no effect on remap operation time.

7.1.3 Finding Performance Bottlenecks

I derived the fomula for a 2D Poisson's solver for a linear system of equations using the Jacobi method and implemented it in Chapel for an unoptimized and optimized 5-point stencil Jacobi kernel that uses NVM. I used the Purity data-centric profiler with NVM support to identify performance bottlenecks in the profiled execution of the unoptimized kernel. My investigation involved heap, node, and loop-level analysis to discover and mitigate the underlying problem, resulting in a speedup of 78x.

Chapter 8: Future Work

8.1 Optimizing the Persistent Transaction System in Chapel

I would like to explore an optimization strategy for the NVM based transaction system that I developed for Chapel, to improve the overall performance of NVM based Chapel applications. Currently, my approach performs a commit and flush at the end of each persistent write operation. However, to hide the latencies of slower memory the data needs to be maintained in a cache for as long as possible. To maximize the use of caches, transactions should be established over loops. According to the PMDK documentation [111], a currently opened transaction is only accessible by the thread that began it. However, transactions can be nested inside of other transactions so long as they pertain to the same thread. This means the loops that are transacted must be sequential, but can contain inner transactions for handling inner loops and changes in data. The libpmemobj library provides begin, commit, end, and post process functions to manage transactions [111]. Performing large transactions over a parallel forall loop in Chapel is possible since the Chapel compiler translates forall loops into corresponding functions that contains a sequential for loop, to be executed by a thread. Therefore, through careful instrumentation, transaction functions (via Loci) can be used to establish an outer transaction over a sequential loop, thereby allowing commit and aggregate flush operations to be performed at the end of that loop, but still within the same thread.

8.2 Expand Latency and Energy Estimation Modeling

The hardware estimator can be an excellent tool for measuring how a profiled execution might behave on different hardware. However, the current version only considers a subset of the computer architecture, namely L1 and L2 data cache, DRAM, and NVM metrics. As a consequence latency calculations are produced by a simplified AMAT computation that uses the SRAM latencies for cache-level hit times rather than hit cycles x clock cycle time, etc. A more accurate calculation would require additional processor and system characteristics such as CPU clock rate, L3 shared cache, block size, cache organization, write polices, interface bandwidth between different levels of cache, channel support, memory bus bandwidth, GPU accelerators, and emerging technologies.

I would like to extend the estimator to support more comprehensive and detailed hardware configurations that represent existing or emerging computer architectures and provide an analysis centered around data motion and impact on execution time and energy. A broad data motion analysis would incorporate both intra-node and inter-node accesses and operations across various interfaces and memory environments to identify both software and hardware design flaws and bottlenecks. The hardware estimator can use profiled execution data generated from the Purity data-centric profiler to explore the design space of future systems and with this extension will become more accurate.

8.3 Adding Cache Profile Support to Purity

Using PAPI, I developed a cache monitor library that works with both Chapel and C applications. I would like to extend the Purity data-centric profiler to incorporate the cache monitoring system. Since cache monitor has already been integrated into the Chapel runtime framework, the transition should be easy. However, care will need to be taken in how this may impact the way cache monitor interfaces with runtime subroutines. Cache profiling currently supports the FIFO task management option in Chapel but I would like to extended it to also support QThreads. To access it I will either add to the CHPL_PROFILE environment variable a 'dccache' option for exclusive profiling or add a new CHPL_PROFILE_RECORD_CACHE environment variable to provide the option to profile cache along side memory and communication.

8.4 Extending Hybrid-PGAS in Chapel to Heterogeneous Environments

I am interested in understanding how a parallel HPC framework like Chapel can be extended to support heterogeneous architectures. To start with, we could consider a cluster where only a subset of the compute nodes provide an emerging technology, such as NVM or accelerators. Heterogeneous computing will introduce new challenges for task and data locality. Work distribution models that are flexible enough to handle a variety of asymmetic architectural configurations will need to be explored. New domain mapping strategies will need to be developed that can store distributed persistent allocations of user-defined variables on the hybrid-PGAS. Since Chapel already supports GPUs, I may also consider heterogeneous architectures where some of the compute nodes support accelerators.

APPENDIX

A.1 Derivation of 2D Poisson Equation Solver

To better understand how a 2D Poisson equation solver is developed, lets first look at the 2D Poisson PDE A.1 with Dirichlet boundary conditions [133] [135]. Let x and y be coordinates on a finite two-dimensional domain where $0 \leq x \leq N$ and $0 \leq y \leq M$ [133] [135].

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{A.1}$$

The 2D Poisson formula, A.1, can be discretized by applying a finite difference approximation to derive a 5-point stencil equation, A.2, below where u will be used to approximate the exact solution [133] [135]. Let $dx = \frac{N}{n+1}$ and $dy = \frac{M}{m+1}$ on a uniform grid [133] . For simplicity lets consider the grid to be an NxM matrix where $N = M$ and therefore $dx = dy$.

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{dx^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{dy^2}, \tag{A.2}$$

$$\text{where } dx = dy$$

By rewriting equation A.2, the Jacobi Iteration Method can be applied to produce equation A.3, where k represents the current iteration and $k + 1$ is the next iteration [133] [136]. For simplicity, the derivation uses a relaxation factor of 1.0 and therefore that term does not appear in the equation [133].

$$u_{i,j}^{k+1} = \frac{u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k}}{4} - \frac{dx^2}{4} f_{i,j} \tag{A.3}$$

Equation A.3 was used to develop the iterative solver in subsection 6.4.1.

B.1 Average Memory Access Time

The simplified AMAT calculation described in subsection 6.2.5 that was used to estimate the latency and energy of profiled executions is defined as follows. $L1$ and $L2$ are data caches, MEM is the underlining memory being accessed which can be either DRAM or NVM, MR represents the miss rate, and MP is the miss penalty.

$$AMAT = L1_{HIT} + L1_{MR} * L1_{MP}$$

$$L1_{MP} = L2_{HIT} + L2_{MR} * L2_{MP}$$

$$L2_{MP} = MEM_{HIT}$$

$$AMAT = L1_{HIT} + L1_{MR} * (L2_{HIT} + L2_{MR} * MEM_{HIT})$$

$$AMAT = L1_{HIT} + (L1_{MR} * L2_{HIT}) + (L1_{MR} * L2_{MR} * MEM_{HIT})$$

Since there is no profile data on L3, it was not included in the AMAT calculation. Cache block sizes, organization, write policies, and interfaces between different levels of cache and memory bus are not considered in this calculation. Since the processor's clock cycle time is unknown, the latencies are applied directly.

$$L1_{HIT} = L1_{latency}$$

$$L2_{HIT} = L2_{latency}$$

$$MEM_{HIT} = MEM_{latency}$$

To calculate the true AMAT and perform data motion analysis, support for processor and system profile characteristics will need to be added. See section 8.2 in future work for details.

C.1 Memory Model Configurations

For reproducability, I have included the memory models that were used by DESTINY to generate a series of memory hardware profiles used in subsection 6.2.5. The models are an adaptation from the configurations provided by NVSim and DESTINY distributions and their derived profile characteristics have been verified in [116] [117] [127] [20].

Listing 1: SRAM Cell

```
-MemCellType: SRAM
-CellArea (F^2): 146
-SRAMCellNMOSWidth (F): 2.08
-SRAMCellPMOSWidth (F): 1.23
-AccessCMOSWidth (F): 1.31
-AccessType: CMOS
-MinSenseVoltage (mV): 80
-CellAspectRatio: 1.46
-ReadVoltage (V): 1.1
-Stitching: 16
```

Listing 2: SRAM (L1) Configuration

```
-DesignTarget: cache
-CacheAccessMode: Normal
-Associativity (for cache only): 8
-ProcessNode: 22
-Capacity (KB): 128
-WordWidth (bit): 512
-DeviceRoadmap: LOP
-LocalWireType: LocalAggressive
-LocalWireRepeaterType: RepeatedNone
-LocalWireUseLowSwing: No
-GlobalWireType: LocalConservative
-GlobalWireRepeaterType: RepeatedNone
```

```
-GlobalWireUseLowSwing: No
-Routing: H-tree
-InternalSensing: true
-MemoryCellInputFile: showcase_SRAM.cell
-Temperature (K): 350
-OptimizationTarget: WriteEDP
-EnablePruning: Yes
-BufferDesignOptimization: latency
-StackedDieCount: 2
-PartitionGranularity: 0
-LocalTSVProjection: 0
-GlobalTSVProjection: 0
-TSVRedundancy: 1.0
```

Listing 3: SRAM (L2) Configuration

```
-DesignTarget: cache
-CacheAccessMode: Normal
-Associativity (for cache only): 8
-ProcessNode: 22
-Capacity (KB): 512
-WordWidth (bit): 512
-DeviceRoadmap: LOP
-LocalWireType: LocalAggressive
-LocalWireRepeaterType: RepeatedNone
-LocalWireUseLowSwing: No
-GlobalWireType: LocalConservative
-GlobalWireRepeaterType: RepeatedNone
-GlobalWireUseLowSwing: No
-Routing: H-tree
-InternalSensing: true
-MemoryCellInputFile: showcase_SRAM.cell
-Temperature (K): 350
-OptimizationTarget: WriteEDP
-EnablePruning: Yes
-BufferDesignOptimization: latency
-StackedDieCount: 2
-PartitionGranularity: 0
-LocalTSVProjection: 0
```

```
-GlobalTSVProjection: 0
-TSVRedundancy: 1.0
```

Listing 4: 3D DRAM Cell

```
-MemCellType: DRAM
-CellArea (F^2): 38.1
-CellAspectRatio: 2.39
-ReadMode: voltage
-AccessType: CMOS
-AccessCMOSWidth (F): 1.31
-DRAMCellCapacitance (F): 15e-15
-ResetVoltage (V): vdd
-SetVoltage (V): vdd
-MinSenseVoltage (mV): 10
```

Listing 5: 3D DRAM Configuration

```
-DesignTarget: RAM
-ProcessNode: 22
-Capacity (MB): 8
-WordWidth (bit): 256
-DeviceRoadmap: HP
-LocalWireType: LocalConservative
-LocalWireRepeaterType: RepeatedNone
-LocalWireUseLowSwing: No
-GlobalWireType: LocalConservative
-GlobalWireRepeaterType: RepeatedNone
-GlobalWireUseLowSwing: No
-Routing: Non-H-tree
-InternalSensing: true
-MemoryCellInputFile: showcase_3D_DRAM.cell
-Temperature (K): 380
-RetentionTime (us): 40
-OptimizationTarget: WriteEDP
-EnablePruning: Yes
-BufferDesignOptimization: latency
-StackedDieCount: 2
```

Listing 6: PCRAM Cell

```
-MemCellType: PCRAM
-ProcessNode: 22
-CellArea (F^2): 4
-CellAspectRatio: 0.5
-ResistanceOn (ohm): 1000
-ResistanceOff (ohm): 1000000
-ReadMode: voltage
-ReadCurrent (uA): 40
-ReadEnergy (pJ): 20
-ResetMode: current
-ResetCurrent (uA): 300
-ResetPulse (ns): 40
-SetMode: current
-SetCurrent (uA): 150
-SetPulse (ns): 150
-AccessType: CMOS
-VoltageDropAccessDevice (V): 0.3
-AccessCMOSWidth (F): 2
```

Listing 7: PCRAM Configuration

```
-DesignTarget: RAM
-ProcessNode: 22
-Capacity (MB): 64
-WordWidth (bit): 256
-DeviceRoadmap: HP
-LocalWireType: LocalConservative
-LocalWireRepeaterType: RepeatedNone
-LocalWireUseLowSwing: No
-GlobalWireType: LocalConservative
-GlobalWireRepeaterType: RepeatedNone
-GlobalWireUseLowSwing: No
-Routing: Non-H-tree
-InternalSensing: true
-MemoryCellInputFile: showcase_PCRAM.cell
-Temperature (K): 350
-EnablePruning: Yes
```

```
-OptimizationTarget: WriteEDP
-BufferDesignOptimization: latency
-StackedDieCount: 1
```

Listing 8: SLCNAND Cell

```
-MemCellType: SLCNAND
-CellArea (F^2): 4
-CellAspectRatio: 1
-GateCouplingRatio: 0.7
-FlashEraseTime (ms): 1.25
-FlashProgramTime (us): 200
-FlashEraseVoltage (V): 16
-FlashProgramVoltage (V): 6
-FlashPassVoltage (V): 3.8
-ReadMode: voltage
-ReadVoltage (V): 0.5
```

Listing 9: SLCNAND Configuration

```
-DesignTarget: RAM
-OptimizationTarget: Area
-ProcessNode: 22
-Capacity (MB): 256
-WordWidth (bit): 64
-FlashPageSize (Byte): 2048
-FlashBlockSize (KB): 128
-DeviceRoadmap: LOP
-LocalWireType: LocalConservative
-LocalWireRepeaterType: RepeatedNone
-LocalWireUseLowSwing: No
-GlobalWireType: LocalConservative
-GlobalWireRepeaterType: RepeatedNone
-GlobalWireUseLowSwing: No
-Routing: H-tree
-InternalSensing: true
-MemoryCellInputFile: showcase_SLCNAND.cell
-Temperature (K): 380
-BufferDesignOptimization: area
```

```
-ForceBank (Total AxB, Active CxD): 1x1, 1x1
-ForceMat (Total AxB, Active CxD): 1x1, 1x1
```

C.2 Memory Hardware Profiles

Also included are the profiles generated from memory models provided to DESTINY. The only the exception is the Zaratan SATA SSD profile which was derived from r_await and w_await values [128], an article on SSDs [129], and the Zaratan documentation [122]. For the estimations, the PCRAM write latency and energy in table 6.3 were defined by the SET characteristics. Post estimates with RESET+SET yielded a difference of only 1% when compared with table 6.11. Also, SLCNAND write in table 6.12 uses programming latency and energy and does not consider the erase.

Listing 10: 3D DRAM Profile

```
User-defined configuration file (./showcase_3D_DRAM.cfg) is loaded


====================
DESIGN SPECIFICATION
====================
Design Target: Random Access Memory
Capacity   : 8MB
Data Width : 256Bits (32Bytes)

Searching for the best solution that is optimized for write
   energy-delay-product ...
Using cell file: showcase_3D_eDRAM.cell
numSolutions = 187798 / numDesigns = 13977846

=============
   SUMMARY
=============
Optimized for: Write Energy-Delay-Product
```

```
Memory Cell: DRAM
Cell Area (F^2)    : 38.100 (9.542Fx3.993F)
Cell Aspect Ratio  : 2.390

=============
CONFIGURATION
=============
Bank Organization: 32 x 16 x 2
 - Row Activation   : 1 / 32 x 1
 - Column Activation: 1 / 16 x 1
Mat Organization: 1 x 1
 - Row Activation   : 1 / 1
 - Column Activation: 1 / 1
 - Subarray Size    : 64 Rows x 1024 Columns
Mux Level:
 - Senseamp Mux      : 1
 - Output Level-1 Mux: 1
 - Output Level-2 Mux: 4
Local Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Global Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Buffer Design Style: Latency-Optimized
=============
   RESULT
=============
Area:
 - Total Area = 552.794um x 1.633mm = 2.176mm^2
 |--- Mat Area      = 17.275um x 102.069um = 1763.230um^2
    (137.079%)
 |--- Subarray Area = 17.275um x 97.758um = 1688.749um^2
    (143.125%)
 |--- TSV Area      = 1600.000um^2
 - Area Efficiency = 56.861%
Timing:
 -  Read Latency = 12.570ns
 |--- TSV Latency    = 0.030ps
 |--- Non-H-Tree Latency = 12.095ns
 |--- Mat Latency    = 475.230ps
    |--- Predecoder Latency = 44.543ps
    |--- Subarray Latency   = 430.687ps
       |--- Row Decoder Latency = 264.211ps
       |--- Bitline Latency     = 154.364ps
```

```
      |--- Senseamp Latency    = 1.779ps
      |--- Mux Latency         = 2.489ps
      |--- Precharge Latency   = 49.689ps
 - Write Latency = 12.570ns
 |--- TSV Latency    = 0.015ps
 |--- Non-H-Tree Latency = 12.095ns
 |--- Mat Latency    = 475.230ps
    |--- Predecoder Latency = 44.543ps
    |--- Subarray Latency   = 430.687ps
       |--- Row Decoder Latency = 264.211ps
       |--- Charge Latency      = 15.106ps
 - Refresh Latency = 451.282ns
 |--- TSV Latency    = 0.015ps
 |--- Non-H-Tree Latency = 423.833ns
 |--- Mat Latency    = 27.449ns
    |--- Predecoder Latency = 44.543ps
    |--- Subarray Latency   = 27.405ns
 - Read Bandwidth  = 148.035GB/s
 - Write Bandwidth = 74.300GB/s
Power:
 -  Read Dynamic Energy = 98.242pJ
 |--- TSV Dynamic Energy    = 59.337pJ
 |--- Non-H-Tree Dynamic Energy = 35.778pJ
 |--- Mat Dynamic Energy    = 3.127pJ per mat
    |--- Predecoder Dynamic Energy = 0.042pJ
    |--- Subarray Dynamic Energy   = 3.085pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.056pJ
       |--- Mux Decoder Dynamic Energy = 0.176pJ
       |--- Senseamp Dynamic Energy    = 0.750pJ
       |--- Mux Dynamic Energy         = 0.055pJ
       |--- Precharge Dynamic Energy   = 0.401pJ
 - Write Dynamic Energy = 99.609pJ
 |--- TSV Dynamic Energy    = 59.337pJ
 |--- Non-H-Tree Dynamic Energy = 35.778pJ
 |--- Mat Dynamic Energy    = 4.494pJ per mat
    |--- Predecoder Dynamic Energy = 0.042pJ
    |--- Subarray Dynamic Energy   = 4.452pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.056pJ
       |--- Mux Decoder Dynamic Energy = 0.176pJ
       |--- Mux Dynamic Energy         = 0.055pJ
 - Refresh Dynamic Energy = 94.932nJ
 |--- TSV Dynamic Energy    = 4.033pJ
 |--- Non-H-Tree Dynamic Energy = 94.742nJ
 |--- Mat Dynamic Energy    = 185.336pJ per mat
    |--- Predecoder Dynamic Energy = 2.653pJ
    |--- Subarray Dynamic Energy   = 182.683pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.000pJ
```

```
       |--- Senseamp Dynamic Energy    = 0.750pJ
       |--- Precharge Dynamic Energy   = 0.401pJ
 - Leakage Power = 5.415W
 |--- TSV Leakage              = 0.000pW
 |--- Non-H-Tree Leakage Power = 0.000pW
 |--- Mat Leakage Power        = 5.289mW per mat
 - Refresh Power = 0.000pW

Finished!
```

Listing 11: PCRAM Profile

```
User-defined configuration file (./showcase_PCRAM.cfg) is loaded


====================
DESIGN SPECIFICATION
====================
Design Target: Random Access Memory
Capacity   : 64MB
Data Width : 256Bits (32Bytes)

Searching for the best solution that is optimized for write
   energy-delay-product ...
Using cell file: showcase_PCRAM.cell
numSolutions = 623674 / numDesigns = 13977846

=============
   SUMMARY
=============
Optimized for: Write Energy-Delay-Product
Memory Cell: PCRAM (Phase-Change)
Cell Area (F^2)    : 4.000 (1.414Fx2.828F)
Cell Aspect Ratio  : 0.500
Cell Turned-On Resistance : 1.000Kohm
Cell Turned-Off Resistance: 1.000Mohm
Read Mode: Voltage-Sensing
  - Read Current: 40.000uA
Reset Mode: Current
  - Reset Current: 300.000uA
  - Reset Pulse: 40.000ns
Set Mode: Current
  - Set Current: 150.000uA
  - Set Pulse: 150.000ns
Access Type: CMOS
```

```
=============
CONFIGURATION
=============
Bank Organization: 64 x 4
 - Row Activation   : 4 / 64
 - Column Activation: 1 / 4
Mat Organization: 2 x 2
 - Row Activation   : 2 / 2
 - Column Activation: 2 / 2
 - Subarray Size    : 128 Rows x 4096 Columns
Mux Level:
 - Senseamp Mux      : 128
 - Output Level-1 Mux: 1
 - Output Level-2 Mux: 2
Local Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Global Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Buffer Design Style: Latency-Optimized
=============
   RESULT
=============
Area:
 - Total Area = 863.212um x 2.489mm = 2.148mm^2
 |--- Mat Area      = 13.488um x 622.172um = 8391.669um^2
    (48.382%)
 |--- Subarray Area = 6.744um x 307.580um = 2074.273um^2   (48.934%)
 - Area Efficiency = 48.382%
Timing:
 -  Read Latency = 33.808ns
 |--- Non-H-Tree Latency = 26.916ns
 |--- Mat Latency    = 6.891ns
    |--- Predecoder Latency = 84.766ps
    |--- Subarray Latency   = 6.807ns
       |--- Row Decoder Latency = 2.376ns
       |--- Bitline Latency     = 370.012ps
       |--- Senseamp Latency    = 2.656ps
       |--- Mux Latency         = 37.430ps
       |--- Precharge Latency   = 40.016ps
 - RESET Latency = 73.400ns
 |--- Non-H-Tree Latency = 26.916ns
 |--- Mat Latency    = 46.484ns
    |--- Predecoder Latency = 84.766ps
```

```
    |--- Subarray Latency   = 46.399ns
       |--- RESET Pulse Duration = 40.000ns
       |--- Row Decoder Latency  = 2.376ns
       |--- Charge Latency   = 2.271ps
 - SET Latency   = 183.400ns
 |--- Non-H-Tree Latency = 26.916ns
 |--- Mat Latency    = 156.484ns
    |--- Predecoder Latency = 84.766ps
    |--- Subarray Latency   = 156.399ns
       |--- SET Pulse Duration   = 150.000ns
       |--- Row Decoder Latency  = 2.376ns
       |--- Charger Latency      = 2.271ps
 - Read Bandwidth  = 7.157GB/s
 - Write Bandwidth = 204.605MB/s
Power:
 -  Read Dynamic Energy = 235.484pJ
 |--- Non-H-Tree Dynamic Energy = 65.306pJ
 |--- Mat Dynamic Energy    = 42.544pJ per mat
    |--- Predecoder Dynamic Energy = 0.179pJ
    |--- Subarray Dynamic Energy   = 10.591pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.211pJ
       |--- Mux Decoder Dynamic Energy = 1.673pJ
       |--- Bitline & Cell Read Energy = 0.003pJ
       |--- Senseamp Dynamic Energy    = 0.013pJ
       |--- Mux Dynamic Energy         = 1.168pJ
       |--- Precharge Dynamic Energy   = 1.544pJ
 - RESET Dynamic Energy = 26.081nJ
 |--- H-Tree Dynamic Energy = 65.306pJ
 |--- Mat Dynamic Energy    = 6.504nJ per mat
    |--- Predecoder Dynamic Energy = 0.179pJ
    |--- Subarray Dynamic Energy   = 1.623nJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.211pJ
       |--- Mux Decoder Dynamic Energy = 1.673pJ
       |--- Mux Dynamic Energy         = 1.168pJ
       |--- Cell RESET Dynamic Energy  = 864.248pJ
 - SET Dynamic Energy = 26.035nJ
 |--- Non-H-Tree Dynamic Energy = 65.306pJ
 |--- Mat Dynamic Energy    = 6.493nJ per mat
    |--- Predecoder Dynamic Energy = 0.179pJ
    |--- Subarray Dynamic Energy   = 1.623nJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.211pJ
       |--- Mux Decoder Dynamic Energy = 1.673pJ
       |--- Mux Dynamic Energy         = 1.168pJ
       |--- Cell SET Dynamic Energy    = 1.620nJ
 - Leakage Power = 2.787W
 |--- Non-H-Tree Leakage Power = 0.000pW
 |--- Mat Leakage Power        = 10.887mW per mat
```

```
Finished!
```

Listing 12: SLCNAND Profile

```
User-defined configuration file (./showcase_SLCNAND.cfg) is loaded


====================
DESIGN SPECIFICATION
====================
Design Target: Random Access Memory
Capacity   : 256MB
Data Width : 64Bits (8Bytes)
Page Size  : 2048Bytes
Block Size : 128KB

Searching for the best solution that is optimized for area ...
Using cell file: showcase_SLCNAND.cell
numSolutions = 6 / numDesigns = 3645

=============
   SUMMARY
=============
Optimized for: Area
Memory Cell: Single-Level Cell NAND Flash
Cell Area (F^2)    : 4.000 (2.000Fx2.000F)
Cell Aspect Ratio  : 1.000
Pass Voltage       : 3.800V
Programming Voltage: 6.000V
Erase Voltage      : 16.000V
Programming Time   : 200.000us
Erase Time         : 1.250ms
Gate Coupling Ratio: 0.700

=============
CONFIGURATION
=============
Bank Organization: 1 x 1
 - Row Activation   : 1 / 1
 - Column Activation: 1 / 1
Mat Organization: 1 x 1
 - Row Activation   : 1 / 1
 - Column Activation: 1 / 1
 - Subarray Size    : 32768 Rows x 65536 Columns
Mux Level:
```

```
 - Senseamp Mux      : 4
 - Output Level-1 Mux: 1
 - Output Level-2 Mux: 1
Local Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Global Wire:
 - Wire Type : Local Conservative
 - Repeater Type: No Repeaters
 - Low Swing : No
Buffer Design Style: Area-Optimized
=============
   RESULT
=============
Area:
 - Total Area = 1.500mm x 2.918mm = 4.378mm^2
 |--- Mat Area      = 1.500mm x 2.918mm = 4.378mm^2   (94.964%)
 |--- Subarray Area = 1.500mm x 2.886mm = 4.329mm^2   (96.029%)
 - Area Efficiency = 94.964%
Timing:
 -  Read Latency = 16.256us
 |--- H-Tree Latency = 0.000ps
 |--- Mat Latency    = 16.256us
    |--- Predecoder Latency = 441.315ps
    |--- Subarray Latency   = 16.255us
       |--- Row Decoder Latency = 202.809ns
       |--- Bitline Latency     = 15.845us
       |--- Senseamp Latency    = 2.266ps
       |--- Mux Latency         = 14.155ps
       |--- Precharge Latency   = 33.881ns
 - Erase Latency = 1.250ms
 |--- H-Tree Latency = 0.000ps
 |--- Mat Latency    = 1.250ms
 - Programming Latency   = 200.440us
 |--- H-Tree Latency = 0.000ps
 |--- Mat Latency    = 200.440us
 - Read Bandwidth  = 127.313MB/s
 - Write Bandwidth = 10.218MB/s
Power:
 -  Read Dynamic Energy = 696.412nJ
 |--- H-Tree Dynamic Energy = 0.000pJ
 |--- Mat Dynamic Energy    = 696.412nJ per mat
    |--- Predecoder Dynamic Energy = 0.349pJ
    |--- Subarray Dynamic Energy   = 696.412nJ per active subarray
       |--- Row Decoder Dynamic Energy = 693.677nJ
       |--- Mux Decoder Dynamic Energy = 2.153pJ
```

```
       |--- Senseamp Dynamic Energy    = 2.241pJ
       |--- Mux Dynamic Energy         = 2.043pJ
       |--- Precharge Dynamic Energy   = 7.675pJ
 - Erase Dynamic Energy = 16.532uJ per block
 |--- H-Tree Dynamic Energy = 0.000pJ
 |--- Mat Dynamic Energy    = 16.532uJ per mat
    |--- Predecoder Dynamic Energy = 0.349pJ
    |--- Subarray Dynamic Energy   = 1.006uJ per active subarray
       |--- Row Decoder Dynamic Energy = 408.239nJ
       |--- Mux Decoder Dynamic Energy = 2.153pJ
       |--- Mux Dynamic Energy         = 2.043pJ
 - Programming Dynamic Energy = 1.782uJ per page
 |--- H-Tree Dynamic Energy = 0.000pJ
 |--- Mat Dynamic Energy    = 1.782uJ per mat
    |--- Predecoder Dynamic Energy = 0.349pJ
    |--- Subarray Dynamic Energy   = 1.006uJ per active subarray
       |--- Row Decoder Dynamic Energy = 408.239nJ
       |--- Mux Decoder Dynamic Energy = 2.153pJ
       |--- Mux Dynamic Energy         = 2.043pJ
 - Leakage Power = 735.622uW
 |--- H-Tree Leakage Power     = 0.000pW
 |--- Mat Leakage Power        = 735.622uW per mat

Finished!
```

Listing 13: SRAM L1 Profile

```
User-defined configuration file (./showcase_SRAM_L1.cfg) is loaded


====================
DESIGN SPECIFICATION
====================
Design Target: Cache
Capacity   : 128KB
Cache Line Size: 64Bytes
Cache Associativity: 8 Ways

Searching for the best solution that is optimized for write
   energy-delay-product ...
Using cell file: sample_SRAM.cell
numSolutions = 4347 / numDesigns = 15604974

======================
CACHE DESIGN -- SUMMARY
======================
```

```
Access Mode: Normal
Area:
 - Total Area = 0.065mm^2
 |--- Data Array Area = 173.014um x 250.272um = 0.046mm^2
 |--- Tag Array Area  = 197.752um x 97.768um = 0.020mm^2
Timing:
 - Cache Hit Latency   = 1.095ns
 - Cache Miss Latency  = 0.099ns
 - Cache Write Latency = 0.781ns
Power:
 - Cache Hit Dynamic Energy   = 0.013nJ per access
 - Cache Miss Dynamic Energy  = 0.013nJ per access
 - Cache Write Dynamic Energy = 0.009nJ per access
 - Cache Total Leakage Power  = 4.107mW
 |--- Cache Data Array Leakage Power = 3.209mW
 |--- Cache Tag Array Leakage Power  = 0.898mW

CACHE DATA ARRAY DETAILS
    =============
       SUMMARY
    =============
    Optimized for: Write Energy-Delay-Product
    Memory Cell: SRAM
    Cell Area (F^2)    : 146.000 (14.600Fx10.000F)
    Cell Aspect Ratio  : 1.460
    SRAM Cell Access Transistor Width: 1.310F
    SRAM Cell NMOS Width: 2.080F
    SRAM Cell PMOS Width: 1.230F

    =============
    CONFIGURATION
    =============
    Bank Organization: 2 x 1 x 2
     - Row Activation   : 2 / 2 x 1
     - Column Activation: 1 / 1 x 1
    Mat Organization: 2 x 2
     - Row Activation   : 2 / 2
     - Column Activation: 2 / 2
     - Subarray Size    : 128 Rows x 512 Columns
    Mux Level:
     - Senseamp Mux      : 1
     - Output Level-1 Mux: 1
     - Output Level-2 Mux: 1
     - One set is partitioned into 1 rows
    Local Wire:
     - Wire Type :     Local Aggressive
     - Repeater Type:     No Repeaters
```

```
 - Low Swing :     No
Global Wire:
 - Wire Type :     Local Conservative
 - Repeater Type:     No Repeaters
 - Low Swing :     No
Buffer Design Style:     Latency-Optimized
=============
   RESULT
=============
Area:
 - Total Area = 173.014um x 250.272um = 45624.777um^2
 |--- Mat Area      = 86.507um x 250.272um = 21650.263um^2
    (171.122%)
 |--- Subarray Area = 43.254um x 123.522um = 5342.780um^2
    (173.357%)
 |--- TSV Area      = 2.250um^2
 - Area Efficiency = 162.404%
Timing:
 -  Read Latency = 848.611ps
 |--- TSV Latency    = 0.018ps
 |--- H-Tree Latency = 135.162ps
 |--- Mat Latency    = 713.431ps
    |--- Predecoder Latency = 89.120ps
    |--- Subarray Latency   = 624.311ps
       |--- Row Decoder Latency = 309.200ps
       |--- Bitline Latency     = 292.772ps
       |--- Senseamp Latency    = 2.558ps
       |--- Mux Latency         = 19.781ps
       |--- Precharge Latency   = 210.253ps
 - Write Latency = 781.021ps
 |--- TSV Latency    = 0.009ps
 |--- H-Tree Latency = 67.581ps
 |--- Mat Latency    = 713.431ps
    |--- Predecoder Latency = 89.120ps
    |--- Subarray Latency   = 624.311ps
       |--- Row Decoder Latency = 309.200ps
       |--- Charge Latency      = 206.327ps
 - Read Bandwidth  = 121.820GB/s
 - Write Bandwidth = 102.513GB/s
Power:
 -  Read Dynamic Energy = 11.143pJ
 |--- TSV Dynamic Energy    = 4.238pJ
 |--- H-Tree Dynamic Energy = 1.741pJ
 |--- Mat Dynamic Energy    = 2.582pJ per mat
    |--- Predecoder Dynamic Energy = 0.029pJ
    |--- Subarray Dynamic Energy   = 0.638pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.019pJ
```

```
            |--- Mux Decoder Dynamic Energy = 0.045pJ
            |--- Senseamp Dynamic Energy    = 0.035pJ
            |--- Mux Dynamic Energy         = 0.032pJ
            |--- Precharge Dynamic Energy   = 0.070pJ
      - Write Dynamic Energy = 6.853pJ
      |--- TSV Dynamic Energy    = 4.238pJ
      |--- H-Tree Dynamic Energy = 1.741pJ
      |--- Mat Dynamic Energy    = 0.437pJ per mat
         |--- Predecoder Dynamic Energy = 0.029pJ
         |--- Subarray Dynamic Energy   = 0.102pJ per active subarray
            |--- Row Decoder Dynamic Energy = 0.019pJ
            |--- Mux Decoder Dynamic Energy = 0.045pJ
            |--- Mux Dynamic Energy         = 0.032pJ
      - Leakage Power = 3.209mW
      |--- TSV Leakage              = 0.000pW
      |--- H-Tree Leakage Power     = 0.000pW
      |--- Mat Leakage Power        = 802.263uW per mat
```

Listing 14: SRAM L2 Profile

```
User-defined configuration file (./showcase_SRAM_L2.cfg) is loaded


====================
DESIGN SPECIFICATION
====================
Design Target: Cache
Capacity   : 512KB
Cache Line Size: 64Bytes
Cache Associativity: 8 Ways

Searching for the best solution that is optimized for write
   energy-delay-product ...
Using cell file: sample_SRAM.cell
numSolutions = 17283 / numDesigns = 15604974

======================
CACHE DESIGN -- SUMMARY
======================
Access Mode: Normal
Area:
 - Total Area = 0.187mm^2
 |--- Data Array Area = 333.189um x 488.378um = 0.165mm^2
 |--- Tag Array Area  = 152.646um x 141.960um = 0.022mm^2
Timing:
 - Cache Hit Latency   = 2.979ns
```

```
 - Cache Miss Latency  = 0.218ns
 - Cache Write Latency = 2.220ns
Power:
 - Cache Hit Dynamic Energy   = 0.021nJ per access
 - Cache Miss Dynamic Energy  = 0.021nJ per access
 - Cache Write Dynamic Energy = 0.006nJ per access
 - Cache Total Leakage Power  = 13.226mW
 |--- Cache Data Array Leakage Power = 11.858mW
 |--- Cache Tag Array Leakage Power  = 1.368mW

CACHE DATA ARRAY DETAILS
    =============
       SUMMARY
    =============
    Optimized for: Write Energy-Delay-Product
    Memory Cell: SRAM
    Cell Area (F^2)    : 146.000 (14.600Fx10.000F)
    Cell Aspect Ratio  : 1.460
    SRAM Cell Access Transistor Width: 1.310F
    SRAM Cell NMOS Width: 2.080F
    SRAM Cell PMOS Width: 1.230F

    =============
    CONFIGURATION
    =============
    Bank Organization: 1 x 1 x 2
     - Row Activation   : 1 / 1 x 1
     - Column Activation: 1 / 1 x 1
    Mat Organization: 2 x 2
     - Row Activation   : 2 / 2
     - Column Activation: 2 / 2
     - Subarray Size    : 512 Rows x 1024 Columns
    Mux Level:
     - Senseamp Mux      : 1
     - Output Level-1 Mux: 1
     - Output Level-2 Mux: 1
     - One set is partitioned into 1 rows
    Local Wire:
     - Wire Type :     Local Aggressive
     - Repeater Type:     No Repeaters
     - Low Swing :     No
    Global Wire:
     - Wire Type :     Local Conservative
     - Repeater Type:     No Repeaters
     - Low Swing :     No
    Buffer Design Style:     Latency-Optimized
    =============
```

```
   RESULT
=============
Area:
 - Total Area = 333.189um x 488.378um = 165050.780um^2
 |--- Mat Area      = 333.189um x 488.378um = 162722.030um^2
    (182.143%)
 |--- Subarray Area = 166.594um x 241.442um = 40222.941um^2
    (184.215%)
 |--- TSV Area      = 2.250um^2
 - Area Efficiency = 179.573%
Timing:
 -  Read Latency = 2.220ns
 |--- TSV Latency    = 0.018ps
 |--- H-Tree Latency = 0.000ps
 |--- Mat Latency    = 2.220ns
    |--- Predecoder Latency = 124.209ps
    |--- Subarray Latency   = 2.095ns
       |--- Row Decoder Latency = 930.241ps
       |--- Bitline Latency     = 1.143ns
       |--- Senseamp Latency    = 2.558ps
       |--- Mux Latency         = 19.781ps
       |--- Precharge Latency   = 997.782ps
 - Write Latency = 2.220ns
 |--- TSV Latency    = 0.009ps
 |--- H-Tree Latency = 0.000ps
 |--- Mat Latency    = 2.220ns
    |--- Predecoder Latency = 124.209ps
    |--- Subarray Latency   = 2.095ns
       |--- Row Decoder Latency = 930.241ps
       |--- Charge Latency      = 1.000ns
 - Read Bandwidth  = 29.589GB/s
 - Write Bandwidth = 30.543GB/s
Power:
 -  Read Dynamic Energy = 19.813pJ
 |--- TSV Dynamic Energy    = 4.254pJ
 |--- H-Tree Dynamic Energy = 0.000pJ
 |--- Mat Dynamic Energy    = 15.559pJ per mat
    |--- Predecoder Dynamic Energy = 0.065pJ
    |--- Subarray Dynamic Energy   = 3.873pJ per active subarray
       |--- Row Decoder Dynamic Energy = 0.034pJ
       |--- Mux Decoder Dynamic Energy = 0.084pJ
       |--- Senseamp Dynamic Energy    = 0.069pJ
       |--- Mux Dynamic Energy         = 0.064pJ
       |--- Precharge Dynamic Energy   = 0.136pJ
 - Write Dynamic Energy = 5.263pJ
 |--- TSV Dynamic Energy    = 4.254pJ
 |--- H-Tree Dynamic Energy = 0.000pJ
```

```
    |--- Mat Dynamic Energy    = 1.008pJ per mat
       |--- Predecoder Dynamic Energy = 0.065pJ
       |--- Subarray Dynamic Energy   = 0.236pJ per active subarray
          |--- Row Decoder Dynamic Energy = 0.034pJ
          |--- Mux Decoder Dynamic Energy = 0.084pJ
          |--- Mux Dynamic Energy         = 0.064pJ
    - Leakage Power = 11.858mW
    |--- TSV Leakage              = 0.000pW
    |--- H-Tree Leakage Power     = 0.000pW
    |--- Mat Leakage Power        = 5.929mW per mat
```

Listing 15: Zaratan SATA SSD Profile

```
Memory Cell: SATA SSD
RESULT
- Read Latency = 0.08ms
- Write Latency = 0.05ms
- Read Dynamic Energy = 0.48mJ
- Write Dynamic Energy = 0.325mJ
- Idle Power = 1.125W
```

Bibliography

[1] S. Hemmert, J. Ang, P. Chiang, B. Carnes, D. Doerfler, M. Leininger, S. Dosanjh, P. Fields, K. Koch, J. Laros *et al.*, "Exascale Hardware Architectures Working Group," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2011.

[2] Frontier at Oak Ridge National Laboratory. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[3] Top500 Lists. [Online]. Available: https://www.top500.org/lists/top500/

[4] Green500 Lists. [Online]. Available: https://www.top500.org/lists/green500/

[5] S. Singhal, C. R. Crasta, M. Abdulla, F. Barmawer, D. Emberson, R. Ahobala, G. Bhat, R. k. K. Rajak, and P. Soumya, "OpenFAM: A Library for Programming Disaggregated Memory," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2021, pp. 21–38.

[6] K. Keeton, S. Singhal, and M. Raymond, "The OpenFAM API: a programming model for disaggregated persistent memory," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity: 5th Workshop, OpenSHMEM 2018, Baltimore, MD, USA, August 21–23, 2018, Revised Selected Papers 5*. Springer, 2019, pp. 70–89.

[7] ——. [Online]. Available: https://pdfs.semanticscholar.org/5a86/76979f8eab2e1c219bea740eb7b190b2bcf3.pdf

[8] OpenFAM Reference Implementation. [Online]. Available: https://openfam.github.io/index.html

[9] S. Singhal, C. R. Crasta, M. Abdulla K, F. Barmawer, G. Bhat, R. A. Rao, S. PN, and R. K. K. Rajak, "OpenFAM: Programming Disaggregated Memory," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 28, p. e7910, 2023.

[10] OpenFAM: Programming Fabric-Attached-Memory. [Online]. Available: https://openfam.org/

[11] A. Chandrachar, B. Chamberlain, S. Singhal, and C. Crasta, "Extending Chapel to Support Fabric Attached Memory."

[12] ——. Enabling FAM Access in Chapel. [Online]. Available: https://chapel-lang.org/CHIUW/2022/CSlides.pdf

[13] A. Chandrachar, C. Crasta, B. Chamberlain, S. Singhal, P. Shome, and D. Emberson, "A Record-Based Pointer to Fabric Attached Memory."

[14] J. Byrne, H. Kuno, C. Ghosh, P. Shome, C. Amitha, S. Singhal, C. Riana, and D. Emberson, "Coupling chapel-powered hpc workflows for python," 2023.

[15] M. Merrill, W. Reus, and T. Neumann, "Arkouda: interactive data exploration backed by chapel," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, 2019, pp. 28–28.

[16] B. Chamberlain. Arkouda:Data Science at Massive Scales and Interactive Rates slides. [Online]. Available: https://chapel-lang.org/presentations/ArkoudaForPuPPy-presented.pdf

[17] P. Pirkelbauer, S. Bromberger, K. Iwabuchi, and R. Pearce, "Towards Scalable Data Processing in Python with CLIPPy," in *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2021, pp. 43–52.

[18] K. Iwabuchi, L. Lebanoff, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator enabling graph processing," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2019, pp. 39–44.

[19] K. Iwabuchi, K. Youssef, K. Velusamy, M. Gokhale, and R. Pearce, "Metall: A persistent memory allocator for data-centric analytics," *Parallel Computing*, vol. 111, p. 102905, 2022.

[20] K. Wu and D. Li, "Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory for High Performance Computing," *Journal of Computer Science and Technology*, vol. 36, pp. 90–109, 2021.

[21] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[22] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proceedings of the 16th Annual Middleware Conference*, 2015, pp. 37–49.

[23] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[24] *OpenSHMEM: Application Programming Interface, Version 1.4*, Dec. 2017. [Online]. Available: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[25] OpenSHMEM Documentation. [Online]. Available: https://docs.open-mpi.org/en/v5.0.3/man-openshmem/man3/OpenSHMEM.3.html

[26] M. Grodowitz, P. Shamis, and S. Poole, "OpenSHMEM I/O extensions for fine-grained access to persistent memory storage," in *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26-28, 2020, Revised Selected Papers 17*. Springer, 2020, pp. 318–333.

[27] R. M. Krishnan, D. Zhou, W.-H. Kim, S. Kannan, S. Kashyap, and C. Min, "TENET: Memory Safe and Fault Tolerant Persistent Transactional Memory," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 247–264.

[28] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[29] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *Comput. Archit. News*, vol. 39, no. 1, pp. 105–118, March 2011.

[30] J. E. Denny, S. Lee, and J. S. Vetter, "NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA: ACM, May 2016.

[31] ——, "Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1163–1173.

[32] J. T. O'Neill, *MUMPS Language Standard*. U.S. Department of Commerce, National Bureau of Standards, March 1976, vol. 118.

[33] J. Lewkowicz, *The complete MUMPS: an introduction and reference manual for the MUMPS programming language*. New York State College of Veterinary Medicine, Cornell University, 1989.

[34] K. C. O'Kane, *Introduction to the Mumps Language, A Quick Introduction to the Mumps Programming Language*. University of Northern Iowa, November 2017.

[35] ——, *Mumps Programming Language Interpreter, Compiler, and C++ Class Library User's Guide Including PostgreSQL and MySQL, Global Array Database Storage Facility, Version 17*, May 2018.

[36] A. Singh and S. R. Sarangi, "JASS: A Flexible Checkpointing System for NVM-based Systems," *arXiv preprint arXiv:2301.11511*, 2023.

[37] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[38] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[39] Z. Wang, C.-H. Choo, M. A. Kozuch, T. C. Mowry, G. Pekhimenko, V. Seshadri, and D. Skarlatos, "Nvoverlay: enabling efficient and scalable high-frequency snapshotting to nvm," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 498–511.

[40] K. Kruger, R. Pannain, and R. Azevedo, "DONUTS: An efficient method for checkpointing in non-volatile memories," *Concurrency and Computation: Practice and Experience*, p. e7574, 2023.

[41] J. Ren, K. Wu, and D. Li, "Easycrash: Exploring non-volatility of non-volatile memory for high performance computing under failures," *arXiv preprint arXiv:1906.10081*, 2019.

[42] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Software-managed energy-efficient hybrid DRAM/NVM main memory," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, May 2015.

[43] ——, "Energy-Efficient Hybrid DRAM/NVM Main Memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 492–493.

[44] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*. New York, NY, USA: ACM, May 2011.

[45] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, "Energy-efficient in-memory data stores on hybrid memory hierarchies," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*. New York, NY, USA: ACM, May 2015.

[46] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting Program Semantics to Place Data in Hybrid Memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 163–173.

[47] H. Mix, C. Herold, and M. Weber, "Visualization of multi-layer i/o performance in vampir," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 387–394.

[48] R. Tschüter, C. Herold, B. Wesarg, and M. Weber, "A Methodology for Performance Analysis of Applications Using Multi-layer I/O," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 16–30.

[49] D. A. Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-p: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010: Proceedings of an International Conference on Competence in High Performance Computing, June 2010, Schloss Schwetzingen, Germany*. Springer, 2012, pp. 85–97.

[50] H. Zhang and J. K. Hollingsworth, "ChplBlamer: A Data-centric and Code-centric Combined Profiler for Multi-locale Chapel Programs," in *Proceedings of the 2018 International Conference on Supercomputing*. New York, NY, USA: ACM, June 2018.

[51] H. Zhang, "Data-Centric Performance Measurement and Mapping for Highly Parallel Programming Models," Ph.D. dissertation, 2018.

[52] T. B. Rolinger, C. D. Krieger, and A. Sussman, "Compiler Optimization for Irregular Memory Access Patterns in PGAS Programs," *arXiv preprint arXiv:2303.13954*, 2023.

[53] ——, "Runtime Optimizations for Irregular Applications in Chapel," in *The 8th Annual Chapel Implementers and Users Workshop (CHIUW)*, 2021.

[54] T. B. Rolinger and A. Sussman, "Compiler optimization for irregular memory accesses in chapel," in *The 9th Annual Chapel Implementers and Users Workshop (CHIUW)*, 2022.

[55] T. B. Rolinger, J. Craft, C. D. Krieger, and A. Sussman, "Towards High Productivity and Performance for Irregular Applications in Chapel," in *2021 SC Workshops Supplementary Proceedings (SCWS)*. IEEE, 2021, pp. 1–11.

[56] T. B. Rolinger, C. D. Krieger, and A. Sussman, "Optimizing Memory-Compute Colocation for Irregular Applications on a Migratory Thread Architecture," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 58–67.

[57] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin *et al.*, "Highly scalable near memory processing with migrating threads on the emu system architecture," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. IEEE, 2016, pp. 2–9.

[58] T. Rolinger, C. Krieger, and A. Sussman, "Optimizing data layouts for irregular applications on a migratory thread architecture," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 7–15.

[59] T. B. Rolinger and A. Sussman, "Adaptive Prefetching for Fine-grain Communication in PGAS Programs."

[60] A. Miranda, R. Nou, and T. Cortes, "ECHOFS: A Scheduler-Guided Temporary Filesystem to Leverage Node-Local NVMS," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 225–228.

[61] A. Miranda. ECHOFS: Enabling Transparent Access to Node-local NVM Burst Buffers for Legacy Applications. [Online]. Available: http://www.nextgenio.eu/sites/default/files/documents/publications/17202.AlbertoMiranda.Slides.pdf

[62] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[63] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "PreSET: improving performance of phase change memories by exploiting asymmetry in write times," *Comput. Archit. News*, vol. 40, no. 3, pp. 380–391, Sep. 2012.

[64] C. Wang and W.-F. Wong, "SAW: system-assisted wear leveling on the write endurance of NAND flash devices," in *Proceedings of the 50th Annual Design Automation Conference*. New York, NY, USA: ACM, May 2013.

[65] S. Im and D. Shin, "Differentiated space allocation for wear leveling on phase-change memory-based storage device," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 1, pp. 45–51, 2014.

[66] L.-P. Chang and L.-C. Huang, "A low-cost wear-leveling algorithm for block-mapping solid-state disks," *SIGPLAN Not.*, vol. 46, no. 5, pp. 31–40, Apr. 2011.

[67] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. New York, NY, USA: ACM, Nov. 2013.

[68] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-rationing garbage collection for hybrid memories," *SIGPLAN Not.*, vol. 53, no. 4, pp. 62–77, Dec. 2018.

[69] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai, "Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, 2012, pp. 94–101.

[70] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, Mar. 2011.

[71] J. S. Vetter and S. Mittal, "Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing," *Computing in Science and Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[72] M. Mohseni and A. H. Novin, "A survey on techniques for improving phase change memory (pcm) lifetime," *Journal of Systems Architecture*, p. 103008, 2023.

[73] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proceedings of the international conference on Supercomputing*. New York, NY, USA: ACM, May 2011.

[74] H. Payer, M. Sanvido, Z. Z. Bandic, and C. M. Kirsch, "Combo drive: Optimizing cost and performance in a heterogeneous storage device," in *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, vol. 1, no. 1, 2009, pp. 1–8.

[75] Q. Yang and J. Ren, "I-CASH: Intelligently Coupled Array of SSD and HDD," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 278–289.

[76] X. Wu and A. N. Reddy, "Exploiting Concurrency to Improve Latency and throughput in a Hybrid Storage System," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010, pp. 14–23.

[77] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 227–236.

[78] K. Liu, X. Zhang, K. Davis, and S. Jiang, "Synergistic coupling of ssd and hard disk for qos-aware virtual memory," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 24–33.

[79] Frontier User Guide. [Online]. Available: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[80] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 945–956.

[81] B. L. Chamberlain, S. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, and G. Titus, "The State of the Chapel Union," *CUG 2013*, May 2013.

[82] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," in *International Journal of High Performance Computing Applications*, vol. 21, August 2007, pp. 291–312.

[83] Chapel Documentation: Locales. [Online]. Available: https://chapel-lang.org/docs/1.22/builtins/ChapelLocale.html

[84] Chapel Documentation: Data Parallelism. [Online]. Available: https://chapel-lang.org/docs/1.22/language/spec/data-parallelism.html

[85] Chapel Documentation: Parallel Iterators. [Online]. Available: https://chapel-lang.org/docs/1.22/language/spec/iterators.html#parallel-iterators

[86] Chapel Documentation: Ranges. [Online]. Available: https://chapel-lang.org/docs/1.22/builtins/ChapelRange.html

[87] Chapel Documentation: Domains. [Online]. Available: https://chapel-lang.org/docs/1.22/language/spec/domains.html

[88] Chapel Documentation: BlockDist. [Online]. Available: https://chapel-lang.org/docs/1.22/modules/dists/BlockDist.html

[89] Chapel Documentation: CyclicDist. [Online]. Available: https://chapel-lang.org/docs/1.22/modules/dists/CyclicDist.html

[90] Chapel Documentation: Domain Maps. [Online]. Available: https://chapel-lang.org/docs/1.22/language/spec/domain-maps.html

[91] A. Azad and A. Buluc, "Towards a GraphBLAS library in Chapel," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1095–1104.

[92] Chapel Documentation: forall-loops, data-parallel loops. [Online]. Available: https://chapel-lang.org/docs/1.22/users-guide/datapar/forall.html

[93] Chapel Documentation: Reduce Intent. [Online]. Available: https://chapel-lang.org/docs/1.22/technotes/reduceIntents.html

[94] Chapel Documentation: Task Parallelism Overview. [Online]. Available: https://chapel-lang.org/docs/1.22/users-guide/taskpar/taskParallelismOverview.html

[95] Chapel Documentation: begin. [Online]. Available: https://chapel-lang.org/docs/1.22/users-guide/taskpar/begin.html

[96] Chapel Documentation: cobegin. [Online]. Available: https://chapel-lang.org/docs/1.22/users-guide/taskpar/cobegin.html

[97] Chapel Documentation: coforall-loops, loop-based tasking. [Online]. Available: https://chapel-lang.org/docs/1.22/users-guide/taskpar/coforall.html

[98] Chapel Documentation: atomics. [Online]. Available: https://chapel-lang.org/docs/1.22/primers/atomics.html?highlight=atomic

[99] Chapel Documentation: sync and singles. [Online]. Available: https://chapel-lang.org/docs/1.22/primers/syncsingle.html?highlight=sync

[100] R. B. Johnson and J. K. Hollingsworth, "Purity: An Integrated, Fine-Grain, Data-Centric, Communication Profiler for the Chapel Language," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 934–942.

[101] Chapel Documentation: Chapel Tasks. [Online]. Available: https://chapel-lang.org/docs/1.22/usingchapel/tasks.html

[102] The Qthread library. [Online]. Available: https://www.sandia.gov/qthreads/

[103] Chapel Documentation: Multilocale Chapel Execution. [Online]. Available: https://chapel-lang.org/docs/1.22/usingchapel/multilocale.html

[104] GASNet, Berkeley Lab. [Online]. Available: https://gasnet.lbl.gov/

[105] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2018, pp. 138–158.

[106] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[107] Installing PMDK using Linux Packages. [Online]. Available: https://docs.pmem.io/persistent-memory/getting-started-guide/installing-pmdk/installing-pmdk-using-linux-packages

[108] Persistent Memory Development Kit on Github. [Online]. Available: https://github.com/pmem/pmdk

[109] P. Balcer. (2015) An Introduction to pmemobj (Part 1) - Accessing The Persistent Memory. [Online]. Available: https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-1-accessing-the-persistent-memory/

[110] ——. (2015) An Introduction to pmemobj (Part 2) - Transactions. [Online]. Available: https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-2-transactions/

[111] pmemobj API version 2.3. [Online]. Available: https://pmem.io/pmdk/manpages/linux/v1.7/libpmemobj/pmemobj_tx_begin.3/

[112] R. Johnson and J. K. Hollingsworth, "Optimizing Chapel for Single-Node Environments," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, pp. 1558–1567.

[113] M. Shin, M. Kim, G. Park, and A. Abraham, "Adaptive variable sampling model for performance analysis in high cache-performance computing environments," *Heliyon*, vol. 9, no. 6, 2023.

[114] C. Woralert, J. Bruska, C. Liu, and L. Yan, "High frequency performance monitoring via architectural event measurement," in *2020 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2020, pp. 114–122.

[115] V. M. Weaver *et al.*, "Advanced hardware profiling and sampling (pebs, ibs, etc.): creating a new papi sampling interface," *Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, Tech. Rep.*, 2016.

[116] S. Mittal, R. Wang, and J. Vetter, "Destiny: A comprehensive tool with 3d and multi-level cell memory modeling capability," *Journal of Low Power Electronics and Applications*, vol. 7, no. 3, p. 23, 2017.

[117] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1543–1546.

[118] DESTINY : 3D Design-Space Exploration Tool for SRAM, eDRAM and Non-volatile memory. [Online]. Available: https://code.ornl.gov/3d_cache_modeling_tool/destiny

[119] An Introduction to Chapel. [Online]. Available: https://chapel-lang.org/

[120] M. Ferguson and L. Duncan. Productive Programming in Chapel: A Computation-Driven Introduction, Data Parallelism with Jacobi. [Online]. Available: https://chapel-lang.org/tutorials/SC15/Chapel-SC15-6-DataPar-Jacobi.pdf

[121] Tech Power Up: AMD EPYC 7763 Specs. [Online]. Available: https://www.techpowerup.com/cpu-specs/epyc-7763.c2373

[122] The Zaratan HPC cluster: Hardware. [Online]. Available: https://hpcc.umd.edu/hpcc/zaratan.html

[123] Chapel Tasks: FIFO. [Online]. Available: https://chapel-lang.org/docs/usingchapel/tasks.html#chpl-tasks-fifo

[124] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[125] S. Li, L. Liu, P. Gu, C. Xu, and Y. Xie, "NVSim-CAM: A circuit-level simulator for emerging nonvolatile memory based Content-Addressable Memory," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–7.

[126] NVSim - A performance, energy and area estimation tool for non-volatile memory (NVM). [Online]. Available: https://github.com/SEAL-UCSB/NVSim

[127] M. M. S. Aly, T. F. Wu, A. Bartolo, Y. H. Malviya, W. Hwang, G. Hills, I. Markov, M. Wootters, M. M. Shulaker, H.-S. P. Wong *et al.*, "The N3XT Approach to Energy-Efficient Abundant-Data Computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 19–48, 2018.

[128] iostat(1) - Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man1/iostat.1.html

[129] SuperSSD: SSD Power Consumption: What to Expect. [Online]. Available: https://www.superssd.com/kb/ssd-power-consumption/

[130] R. Liu, A. Li, Z. Xu, Y. Yuan, J. Zhai, S. Song, Z. Song, X. Zhou, H. Zhang, J. Song *et al.*, "Ultrafast phase change speed and high thermal stability of scandium doped snsb4 thin film for pcram applications," *Journal of Non-Crystalline Solids*, vol. 613, p. 122395, 2023.

[131] Summit Features. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2018/06/Summit_bythenumbers_FIN-1.pdf

[132] Y. Fridman, Y. Snir, H. Levin, D. Hendler, H. Attiya, and G. Oren, "Recovery of Distributed Iterative Solvers for Linear Systems Using Non-Volatile RAM," in *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2022, pp. 11–23.

[133] Q. Xu and W. Wang, "A New Parallel Iterative Algorithm for Solving 2D Poisson Equation," *Numerical Methods for Partial Differential Equations*, vol. 27, no. 4, pp. 829–853, 2011.

[134] J. Burkardt, "Jacobi iterative solution of poisson's equation in 1d," 2011.

[135] Jacobi Solver with HIP and OpenMP offloading. [Online]. Available: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-jacobi-readme/

[136] Q. Wang and J. K. White. Numerical Methods for PDE, Lecture 19. [Online]. Available: https://learning-modules.mit.edu/materials/index.html?uuid=/course/16/fa16/16.920#materials

[137] Jacobi Method Example. [Online]. Available: https://github.com/deniskin82/chapel/blob/master/examples/programs/jacobi.chpl