Exploring the Future of Out-Of-Core Computing with Compute-Local Non-Volatile Memory

Myoungsoo Jung[1,3], Ellis H. Wilson III[2], Wonil Choi[1,2], John Shalf[3,4],
Hasan Metin Aktulga[3], Chao Yang[3], Erik Saule[5], Umit V. Catalyurek[5,6], and Mahmut Kandemir[2]
[1] Department of Electrical Engineering, The University of Texas at Dallas
[2] Department of Computer Science and Engineering, The Pennsylvania State University
[3] Computational Research Division, Lawrence Berkeley National Laboratory
[4] National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory
[5] Biomedical Informatics, [6] Electrical and Computer Engineering, The Ohio State University
jung@utdallas.edu, {ellis, wuc138}@cse.psu.edu, {jshalf, hmaktulga, cyang}@lbl.gov, {esaule, umit}@bmi.osu.edu, kandemir@cse.psu.edu}

Abstract

Drawing parallels to the rise of general purpose graphical processing units (GPGPUs) as accelerators for specific high-performance computing (HPC) workloads, there is a rise in the use of non-volatile memory (NVM) as accelerators for I/O-intensive scientific applications. However, existing works have explored use of NVM within dedicated I/O nodes, which are distant from the compute nodes that actually need such acceleration. As NVM bandwidth begins to out-pace point-to-point network capacity, we argue for the need to break from the archetype of completely separated storage.

Therefore, in this work we investigate co-location of NVM and compute by varying I/O interfaces, file systems, types of NVM, and both current and future SSD architectures, uncovering numerous bottlenecks implicit in these various levels in the I/O stack. We present novel hardware and software solutions, including the new Unified File System (UFS), to enable fuller utilization of the new compute-local NVM storage. Our experimental evaluation, which employs a real-world Out-of-Core (OoC) HPC application, demonstrates throughput increases in excess of an order of magnitude over current approaches.

1. INTRODUCTION

Purpose-built computing for acceleration of scientific applications is gaining traction in clusters small and large across the globe, with general-purpose graphic processing units (GPGPUs) leading the charge. However, for out-of-core (OoC) scientific algorithms [23, 34, 44, 47] such as solvers for large systems of linear equations as commonly employed in nuclear physics, the bottleneck continues to be the speed at which one can access the large dataset the computation relies upon. Improving the speed of computation at the compute node in the cluster will not accelerate these problems – the central processing units (CPUs) and/or GPGPUs simply idle waiting for the data from remote, slow magnetic disk.

SC '13, November 17 - 21 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503261.

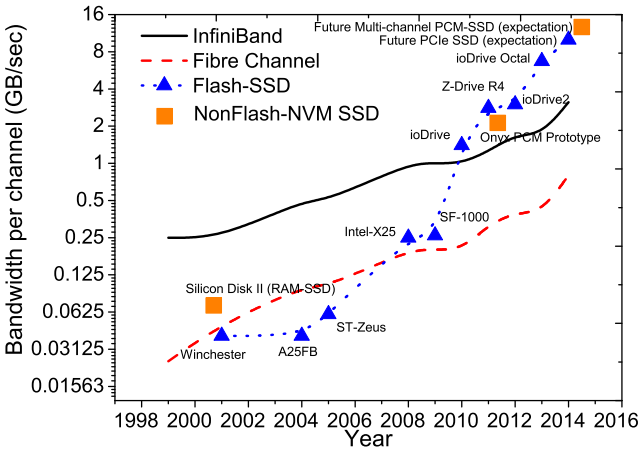
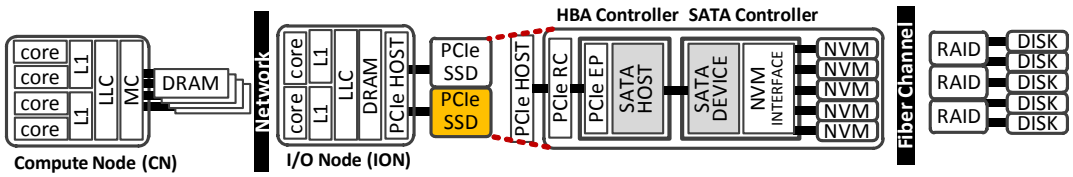


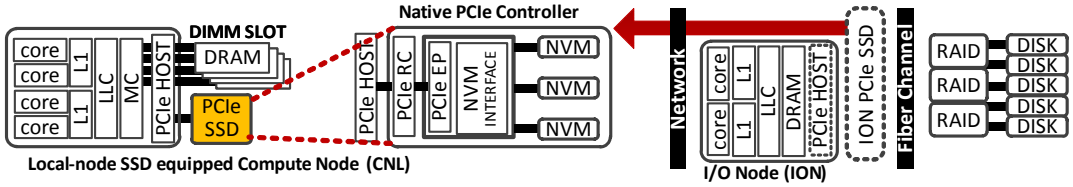
Figure 1: Trend of bandwidth over time for *real-world* high-performance networks versus various NVM storage solutions. These results indicate that even state-of-the-art network solutions are falling behind NVM bandwidth.

The traditional solution to this problem is to utilize shared, distributed memories across the cluster to prevent costly disk accesses during computation. This approach requires a cluster with an aggregate amount of memory large enough to bring the entire dataset in at the start of the algorithm. Further, high-performance networks such as Infiniband are required to enable low enough latency and high enough bandwidth to make remote accesses in this distributed memory efficient. These requirements represent very tangible costs to the system builder and maintaining organization in terms of initial capital investment for the memory and network and high energy use of both over time.

To cope with these issues, recent works [35,36] have demonstrated that utilizing non-volatile memory (NVM), specifically in the form of flash-based solid-state drives (SSDs), as storage accelerators can compete with and even out-perform large distributed memory. This finding is compelling, as the properties of modern SSDs firmly occupy the previously sprawling no man's land between main memory and disk latency, which without SSDs spans three orders of magnitude. By employing these SSDs alongside traditional magnetic storage on the I/O nodes (IONs) in the cluster as shown in Figure 2a, these works demonstrate that only fractions of



(a) ION-Local NVM-based HPC Architecture.



(b) Compute-Local NVM-based HPC Architecture.

Figure 2: The previously proposed SSD-accelerated OoC architecture (top), and our proposed migration to compute-local PCIe NVM (bottom).

the large dataset need be kept in compute node memory at any one time; new chunks of the dataset can be brought in over the network from the SSDs on the ION on an as-needed basis and without much delay to the algorithm.

These efforts embody a very appreciable step towards bridging the I/O gap between CPU and magnetic disk latencies and bandwidth and reducing overall cluster power consumption by utilizing low-power SSDs instead of huge amounts of memory. Nevertheless, the archetype of separating compute from storage persists – these recent approaches only make this low-latency, high-bandwidth NVM available to the compute nodes via network-attached storage. With rapidly increasing parallelism in NVM solutions such as flash-based SSDs [12, 39, 40], bandwidth of these devices is already beginning to eclipse available point-to-point network bandwidth as shown in Figure 1, and shows great potential to far surpass network bandwidth within the decade. This trend indicates a redefinition of the role of NVM as an acceleration technology for OoC scientific computing is required: *we must begin to envision and find ways to implement NVM as a form of compute-local, large but slow memory, rather than client-remote, small but fast disk.*

It should be noted that there have been a limited number of previous works [25, 28, 29] that also propose utilizing compute-local NVM. However, these works *solely* consider the local NVM as a large and algorithmically-managed caches; they do not propose utilizing it as application-managed memory as we do. This is not merely a superficial difference, as the authors in those works report that these cache solutions may take many hours or even days to "heat up," [17,25] which will nullify any benefits distributed OoC applications and similar scientific applications could reap from them. Specifically, for use of NVM as a general-purpose caching layer to work properly, the fundamental expectation that data is accessed more than once in a constrained window of time must hold true, which is often not the case with many long-running scientific workloads. For instance, some scientific workloads work on huge datasets and never access it twice, whereas others access data multiple times but with such great spans of time between the accesses (i.e., very high *reuse distances*) that the liklihood that it stayed in cache is extremely small. In fact, as the data is cached on the local SSDs, the act of caching and evicting the data itself may very well slow down the execution of OoC-style applications since they are so dependent upon optimal bandwidth.

Therefore, in this work we make the following contributions to remedy these maladies: **First**, we design and present an HPC architecture that co-locates the compute and NVM storage and evaluate such using a real OoC linear algebra workload to prove efficacy of our solution versus the ION-local solutions of the past. Figure 2a demonstrates the basic layout of the former architecture, whereas Figure 2b illustrates changes necessary in the cluster architecture to enact this migration to compute-local NVM. Finding there to be significant improvements for compute-local NVM, but not quite as much as should be theoretically possible, **second**, we demonstrate that even quite modern file systems are not well-tuned for the massively parallel architecture of SSDs, and therefore design and present the novel Unified File System (UFS) to overcome these challenges. **Third**, we explore and expose some of the overheads implicit in many state-of-the-art interfaces and SSD architectures that prevent full utilization of the NVM within. We propose novel architectural solutions to these overheads, and **last** experimentally demonstrate near-optimal performance of the compute-local NVM using a cycle-accurate NVM simulation framework, ultimately achieving a relative improvement of 10.3 times over traditional ION-local NVM solutions.

2. BACKGROUND

Before delving into our proposed solution and implementation, we pause to provide some background. We begin by summarizing the science performed by the OoC application we use for evaluation, and then detail the high performance computing (HPC) architecture such computation is performed upon. Last, we conclude with an explanation of state-of-the-art NVM technologies explored in this work.

2.1 Out-of-Core Scientific Computing

Many HPC applications process very large data sets that cannot fit in main memories of parallel architectures. Optimizing such OoC applications requires reducing I/O latencies and improving I/O bandwidth.

As an illustrative example of OoC computing, our focus in this work is on enabling high-accuracy calculations of nuclear structures via the configuration interaction (CI) method . The CI method utilizes the nuclear many-body Hamiltonian, $\hat{H}$, and due to its sparse nature, a parallel iterative eigensolver is used [14, 15]. However, $\hat{H}$ tends to be massive and further, requires orders of magnitude more

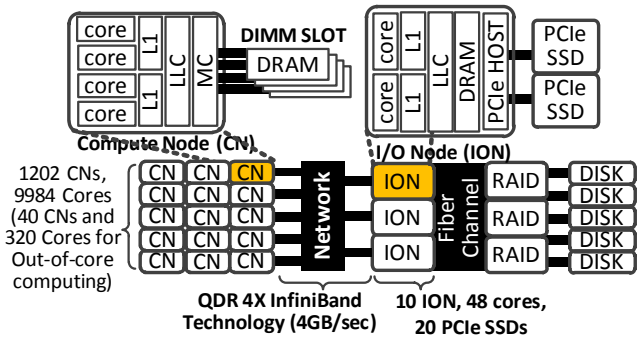


Figure 3: Carver Cluster Architecture at Lawrence Berkeley National Laboratory.

time to compute than a single iteration of the eigensolver. Accordingly, the most pragmatic approach is to preprocess and store $\hat{H}$ in a capacity-rich medium such as traditional magnetic storage.

For computing the eigenpairs, the locally optimal block preconditioned conjugate gradien (LOBPCG) algorithm [42] is used, within which the most time-consuming part is the repeated multiplication of $\hat{H}$ and Ψ (Ψ, for our purposes, need only be known to be a tall, skinny matrix with as many rows as $\hat{H}$ and only about 10-20 columns). As previously alluded to, due to the massive size of the Hamiltonian matrix, traditional solutions to this problem utilize large, distributed main memory to store $\hat{H}$, which are expensive (both in capital cost and power) and place hard limits on the size of $\hat{H}$ that can be stored in-memory. Because of these limitations, some computations are simply not possible even on very large clusters. Even if the required memory is available, this approach can still be very communication-intensive.

Instead of tackling the problem using strictly main memory, recent works [35,36] utilize NVM storage on the IONs of the cluster to store $\hat{H}$ and rely on high performance networks to ferry $\hat{H}_{ij}$ chunks to any compute node desiring them. To effect this in an automated and efficient fashion, [36] presents DOoC, a distributed data storage and scheduler with OoC capabilities. DOoC sits atop DataCutter [20], a middleware that abstracts dataflows via the concept of filters and streams. Filters perform computations on flows of data, which are represented as streams running between producers and consumers. The first of two main components of DOoC is a distributed data storage layer that enables filters to reach data stored on any node in the cluster. This layer supports basic prefetching, automatic memory management, and OoC operations using simplified semantics to improve performance. Specifically, semantics of DOoC are such that large disk-located arrays are immutable once written, removing any need for complicated coherency mechanisms. This nicely complements the read-intensive nature of the aforementioned scientific problem. The second main component of DOoC is a hierarchical data-aware scheduler, which is cognizant of data-dependencies and performs task reordering to maximize parallelism and performance. On the whole, DOoC greatly simplifies the problem of managing many compute and storage nodes efficiently so that the user can instead concentrate on the science being performed.

2.2 HPC Architecture

While DOoC works to abstract the system for the physical scientist, in this work it is still critical to understand the fundamental architecture of the HPC computation framework in use. Most HPC architectures, though varied and each burdened with their own nuances and targeting their own applications, tend to have features akin to the Carver Cluster shown in Figure 3 (Carver is an OoC-specialized subcluster of the cluster described in detail in [26]). Carver is housed at the Computational Research Division of Lawrence Berkeley National Laboratory.

Just as Carver does, HPC architectures in general tend to sequester compute from storage, with the compute layer traditionally being numerous nodes with multi-core processors within. Beside these cores, large memories are provisioned to assure applications run on the compute side of the HPC architecture do not run out of memory. In fact, many modern compute nodes are completely diskless, implying that (barring network-attached swap) applications will likely fail or be killed off if they grow larger than available memory. Very high-throughput, low-latency networks such as the QDR 4X InfiniBand network used in Carver connect this compute side of the cluster to the storage side, which is composed of IONs and external storage. These IONs are charged with managing and exposing their storage over the network to the compute nodes. The physical storage either exists within the IONs themselves or more commonly, as is done in Carver, they are housed externally and attached directly to the IONs via Fibre Channel. This enables enhanced manageability of the RAIDed disks and high-availability of the storage in the event of ION or individual disk failure.

Lastly, and critically for our research, as is demonstrated in Carver, there are a subset of the IONs that are equipped with PCIe-attached SSDs. These nodes are dedicated along with a set of the compute nodes for OoC computations exclusively. As the diagram suggests however, the SSDs are *not* local to the compute doing the OoC processing – the QDR 4X InfiniBand plays an integral role in assuring remote accesses to the low-latency, high-bandwidth NVM are completed efficiently. However, as demonstrated in Figure 1, providing this assurance is increasingly difficult as bandwidth available on these networks grows at a stagnant rate compared to emerging NVM.

	SLC	MLC	TLC	PCM
Page Size	2kB	4kB	8kB	64B
Read (us)	25	50	150	0.115-0.135
Write (us)	250	250-2200	440-6000	35
Erase (us)	1500	2500	3000	35

Table 1: Latency comparison to complete various page-size operations for each of the NVM types we consider, based on industry products as specified in [1], [3], [2] and [5] for SLC, MLC, TLC, and PCM, respectively.

2.3 Non-Volatile Memory

In this work, we concentrate on three types of NAND flash to address what is commonly available today, and isolate and study what appears to be a strong candidate for future NVM devices: Phase-Change Memory (PCM) [4,5,9]. Each of these NVM types has varying performance capabilities on read, write, and erase, as documented in Table 1.

NAND Flash Memory. Beginning with NAND flash, there are three common types based on the number of bits that can be stored in a given flash cell: single-level cell (SLC), which stores a single bit, multi-level cell (MLC),

which stores two bits, and triple-level cell (TLC), which stores three bits. The increase in data density MLC enjoys over SLC involves trade-offs: SLC tends to provide much higher durability guarantees to the flash cell and promises lower and more uniform latencies, whereas MLC wears out faster and completes accesses in longer and less uniform times. A corresponding, but further exacerbated, trade-off occurs for the increased density TLC provides compared to MLC. While the physical medium in use among all these NAND-based NVMs is nearly identical, the logic in use reading and writing from this material decides whether one, two, or three bits are stored on the flash cell. All types employ memory cells composed of floating-gate transistors, which have the special property that they must be erased, or discharged, prior to being written again. This characteristic is referred to as *erase-before-write*. Due to limitations in device-level architecture, erases in NAND-flash must occur to an entire block at a time, which typically range between 64kB and 256kB. This can result in an increased wear on specific cells in the flash device, which is dealt with by wear-leveling techniques common in modern devices.

Phase Change Memory. PCM [9], based on the interesting effects of heating and cooling chalcogenide alloy of germanium, antimony and tellurium (GeSbTe), called GST, at different rates, has different performance and durability characteristics than NAND flash. While read operations upon a GST involves only a series of current sensing processes, writes need to perform two different cell-level operations to change the cell to crystalline and amorphous states (i.e., SET and RESET). Therefore, as shown in Table 1, the writing latency in terms of a flash-type's page size is slower than that of SLC or MLC while its read performance drastically out-performs flash. In addition, PCM also becomes worn out with overuse for writing, which in turn makes it well-suited for OoC applications demonstrating read-intensive workloads. It should be noted that, even though PCM offers 10^3 to 10^5 times better endurance than NAND flash, PCM requires wear-leveling at a much lower level, specifically management for each GST, which might result in unreasonable memory consumption on the host to keep counting their erase cycles and mapping information. In practice, industry applies NOR flash memory interface logic to PCM by emulating block-level erase operations and page-based I/O operations. Because of this compatibility with flash, system designers can manage PCM via flash software (e.g., the flash translation layer) and high-level SSD architecture.

Solid State Disks. Last, whether the NVM material in question is flash, PCM, or some other new technology, this will have to be packaged in an SSD type of architecture to be used as standard storage. Specifically, the individual SLC, MLC, TLC, or PCM cells are manufactured into the smallest group, called "dies." Multiple of these dies are then placed into packages, whom are in turn lined up along shared channels. Further, there can even be multiple planes inside of each die in the case of NAND flash. Clearly, there is an extreme level of parallelism to manage in an SSD, and the efficacy of the managing software (either on the SSD or in the host) plays an extremely important role in extracting full performance from the underlying technology. However, if the parallelism is managed well, the extreme disposition for read performance and parallel nature of modern NVM solutions puts them in a perfect position to be leveraged as accelerators for HPC OoC computing.

3. HOLISTIC SYSTEM ANALYSIS

In this section we provide a thorough overview of our approaches to accelerating OoC computing. These approaches are comprised of a three-fold, holistic evaluation of the system for optimizing I/O throughput. First, we take a look at the cluster architecture from a high-level, providing explanation for our proposal of co-located NVM and compute and describing the software infrastructure we utilize to manage the new localized storage. Second, we explore and expose software-based overheads, specifically related to commonly employed file systems that fail to take full advantage of NVM devices, and present our proposed novel, yet simplified, file system approach to maximizing NVM performance. Third and last, we move lower to unearth implicit issues with most modern PCIe-based SSDs, describe how these overheads can be avoided to expose fuller performance, and last explore and demonstrate how future architectural layouts and protocols could truly bring to bear the full potential of NVM.

3.1 Architecture and Software Framework

While previous works [35, 36] have explored the use of NVM in OoC computing, these implementations have only ever considered use of the NVM device alongside traditional magnetic disk on the IONs. This limited scope was likely driven by two main factors: cost and tradition.

First, while the price per gigabyte for SSDs is dropping rapidly as they become commoditized, they are still expensive compared to magnetic disk. Therefore, considering placing them on the compute nodes rather than the fewer numbered IONs may not have been affordable at that time. As SSDs continue to drop in price and skyrocket in bandwidth, our proposition of placing SSDs on the compute nodes will not only become a financially tractable solution, but will be the cost-efficient solution when compared against purchasing high-performance networks to serve SSDs from the IONs.

The second factor that likely staved off study of compute-local SSDs in past research is the tradition of keeping storage completely isolated from compute. Administration of coupled compute and storage can be more difficult, but as more and more development is done in areas like the Big Data space to enable that very type of architecture (e.g. Hadoop, Mesos, etc), we believe management tools will evolve and remove much of the burden from the administrator.

Therefore, we present an overview of our modified version of the Carver Cluster architecture in Figure 2b. As can be seen in comparison with the original architecture in Figure 2a, this approach migrates the PCIe SSDs directly to the compute nodes, connecting them via the PCIe host layer. Because this removes the need for frequent data transit over the network besides communication, it enables high-throughput and low-latency storage and enables improved network latencies for computational messages.

All required data should be able to be pre-loaded from network-attached magnetic storage to the compute-local SSDs prior to beginning the computation, moving that I/O out of the critical path and thereby improving compute time. Such data migration can of course be overlapped with previous application execution times to hide the pre-loading duration. Once it is pre-loaded, since most OoC computations are heavily read-intensive and require many iterations to complete, no further data transit to spinning network-attached storage should be required. As alluded to in the background, this data migration is made easier than absolutely manual intervention on the part of the applica-

tion designer by leveraging the distributed out-of-core linear algebra framework (DOoC+LAF) [35] and DataCutter [20]. These frameworks work in a manner that parallels the way in which OpenMP [6] enables efficient parallel processing: By using a set of directives and routines exposed by DOoC+LAF, the OoC application is able to provide the framework enough knowledge about the application's workings to enable DOoC+LAF to transparently handle global and local scheduling of tasks and data migration. In our approach, we extend the functionality of DOoC+LAF in our simulation to enable migration of data between data pools as well as between a monolithic data pool and an individual node's memory.

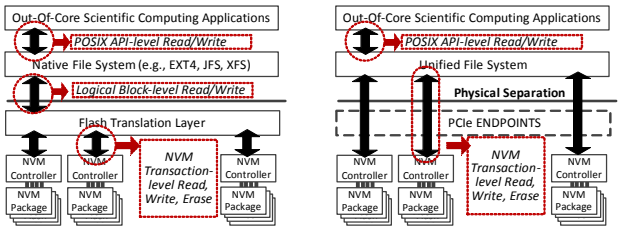
Migration of the NVM device and enabling efficient and easy utilization of such increases the overall I/O bandwidth, but in the process uncovers numerous previously hidden performance issues in software and hardware solutions that exist in-between the application-level and the NVM dies.

3.2 File Systems

Host File Systems. For decades the hard disk has reigned as king of persistent storage in computing, and its properties have correspondingly been optimized for in various layers of the modern Linux kernel. While there is a building movement [11, 19, 46] to improve the existing I/O stack for the increasing variety of storage devices (such as NAND-based SSDs) with different properties than spinning disk, right now the OS optimizations lag well behind the rapid development of new and improved SSD technology. Therefore, in this work, a primary goal in our exploration of the software side of the I/O stack was to determine how well existing file systems perform on state-of-the-art as well as future NVM devices. As we experimentally demonstrate in Section 4, we find that existing file systems are insufficient to fully leverage the capabilities of existing NVM devices, and very poorly suited for the improved speed and parallelism that trends indicate will exist in future devices.

The reasons for these shortcomings vary among file systems, but largely revolve around two main drawbacks: First, all of the examined file systems divide the storage space into small units called blocks, and these blocks tend to be between 512 bytes and 4kB. When a read request is issued from the file system for one of these blocks, sometimes the underlying block device layer will aggregate them into a larger request, but this is not always the case and more importantly, artificial limits are imposed on how large the size of the coalesced request can be. Second, many of the explored file systems perform metadata and/or journalling accesses, some reads and some writes, to the underlying NVM device in the midst of the rest of the data accesses. These random accesses result in increased contention and a corresponding decrease in throughput.

The Unified File System. Acknowledging these issues, and considering our specialized vantage point of advocating for an application-managed storage space, we present our concept of the Unified File System (UFS), as diagrammed in Figure 4b. To best understand how UFS diverges from the norm, let us begin by examining how I/O is handled in the modern Linux I/O stack. As shown in Figure 4a, the OoC application will first perform POSIX-level read and write commands to a specific file as exported by the namespace of the underlying Linux file system. This file system subsequently breaks up that request into logical block-level reads and writes, which is then passed to the Flash Translation Layer (FTL), which further reshapes and reorders the incoming requests. Finally, the requests are handed off to the NVM controllers as NVM transaction-level reads, writes, or erases.



(a) Traditional File Systems. (b) Unified File System.

Figure 4: Diagram of the commonly available bridged-PCIe SSD architecture and a higher-efficiency and throughput native-PCIe SSD architecture.

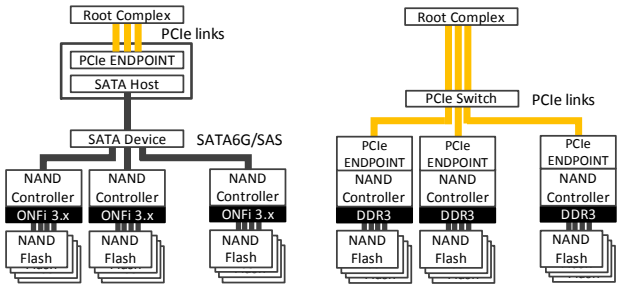
UFS, on the other hand, can be seen to both replace existing file systems but also, and more importantly, the underlying FTL of the SSD. UFS provides *direct, application-managed* access to the NVM media, in terms of raw device addresses rather than human-readable filenames or specialized file-system semantics. Therefore, in many respects, UFS can be seen as less a traditional incarnation of a file system than a new interface providing more direct access to the underlying SSD by taking on many of the roles of the FTL and elevating them to the host-level. This enables a more cohesive and collaborative job of I/O scheduling between the storage medium itself and the host OS. For example, in our use case since UFS will be receiving large read requests directly from our OoC application, it is able to translate and issue those requests directly, which enable the SSD to fully parallelize these larger requests over the many flash channels, packages, and dies. It should be mentioned that this elevation of the FTL to the host OS has already been undertaken and commercialized by Fusion-IO [32].

While this use-case is quite specialized, as we demonstrate in Section 4, it delivers such high performance relative to even tuned traditional file systems, sometimes in excess of 100%, that the extra work is repaid with considerable benefits in execution speed. Moreover, such specialized execution is particularly well suited to large scale OoC applications, whom already perform a high degree of tuning to extract maximum performance out of the extremely expensive cluster they are running on.

3.3 Device Protocols and Interfaces

In this section we uncover and explore solutions to *three major device- and interface-level performance hurdles*. We begin closest to the CPU at the root complex, shown in Figure 5, and move towards the NVM itself.

The first problem we tackle, the overhead for conversion between the SATA protocol and the PCIe protocol, is an artifact of ad-hoc PCIe-based SSD design using bridged flash components that were originally destined for SATA-based SSDs. This artifact is a result of manufacturers of PCIe-based SSD frequently purchasing the internal controllers from specialized controller companies, and since SATA-based SSDs are the prevailing market for NVM storage, the controller companies often only offer a SATA-based controller, shown in Figure 5a. This overhead manifests itself in terms of both protocol re-encoding computation time and protocol



(a) Bridged Architecture. (b) Native Architecture.

Figure 5: Diagram of the commonly available bridged-PCIe SSD architecture and a higher-efficiency and throughput native-PCIe SSD architecture.

re-encoding bandwidth losses.

While the former can be argued to be marginal, and we do not focus on it in this work, in the latter bandwidth losses are unarguably tangible. Specifically, the SATA protocol utilizes an 8/10 bit encoding for the purposes of DC-balance and bounded disparity, meaning that for every 8 bits of data 10 bits are actually transferred, 2 of which are simply wasted in order to assure that there are sufficient state changes to give the clock time to recover. This implies that there is an additional 25% overhead in bandwidth incurred by the internal controllers that is needless; commercially-available PCIe 3.0 protocols only use a 128/130 bit encoding scheme for an overhead of just 1.5%. Moreover, typical modern PCIe-based SSDs utilize PCIe 2.0, which shares the 8/10 bit encoding scheme with SATA and therefore the overheads implicit with that encoding. Therefore, in this work we compare these bridged and PCIe 2.0 protocols against a non-bridged, or native, architecture that allows the PCIe 3.0 protocol non-bridged access to the NAND controller as shown in Figure 5b.

The second problem we examine is that current interface lane-widths also limit the full bandwidth possible. In PCIe 2.0, only 5Gb per lane is available, and therefore, since typical PCIe-based SSDs only provide four PCIe lanes [38], this results in approximately (post-conversion overheads) a 2GBps maximum throughput potential. This ceiling is far lower than the maximum possible throughput of, in aggregate, the internal NVM packages, and therefore we propose and evaluate how expanded-lane architecture that should arrive in the near future will alleviate this bottleneck. We begin by examining devices with 8 lanes, already somewhat beyond typical 4 lane architectures of today, and seeing that even at 8 lanes there still exist major bandwidth bottlenecks, we continue to 16 lane PCIe architectures.

In our third and last problem, NVM interface frequencies limiting bandwidth, we find that even cutting edge protocols such as ONFi major-revision 3 [37] leave bandwidth on the table relative to what the underlying NVM can deliver. Therefore, since ONFi 3 provides a bus interface of 400MHz Single Data Rate (SDR), which is only equal to 200MHz Dual Data Rate (DDR) 2 RAM, we consider future migration to an improved interface similar to DDR3-1600 RAM.

All of these considerations are depicted in the diagrams in Figure 5, and validated in Subsection 4.4.

4. EVALUATION

We begin by discussing our simulation framework, the traces we gathered from our OoC application on various file systems, and the system architectures explored using these traces and our simulator. Finally, we present our results, which elucidate how our approaches improve on existing work and provide a model for future HPC I/O acceleration.

4.1 Experimental Configuration

As careful evaluation of numerous types and configurations of NVM memories at scale is a prohibitively expensive exploration and limits us needlessly to only hardware that is commercially available today, in this work we utilize our simulator to perform our exploration. All of our experiments utilize our NANDFlashSim simulation framework, described in detail in [21], which enables highly accurate timing models for the NVM types we examine. Further, we utilize queuing optimizations within NANDFlashSim as discussed in [22], to refine our findings for future NVM devices. Last, we extended NANDFlashSim to handle TLC and PCM.

The architectures that we simulate using this framework, detailed in Table 2, begin with the approaches taken by previous work at the top where NVM is separated from the compute, and continue down towards increasingly forward-looking architectures. Further, for each of the listed configurations, we examine all four types of NVM discussed: SLC, MLC, TLC and PCM. Each of these NVM types are simulated in equivalent SSD architectures equipped with 8 channels, 64 NVM packages, and a total of 128 NVM dies.

4.2 Tracing Methodology and I/O Patterns

In order to use our simulator, we captured traces from a real-world OoC application [35] that was running on the Carver Cluster previously mentioned. To collect these traces, we executed our OoC application on the Carver cluster, and captured I/O commands at two different levels: First, we collected POSIX-level traces directly under the application but prior to reaching GPFS. This was done on all of the compute-nodes in use for the OoC application on the cluster, however, since it is just a POSIX-level trace, to explore the impact of migration of the NVM devices from IONs to CNs, we have to replay it through a real file system in order to capture the device-level block trace required for input to NANDFlashSim. This extra effort is well worth it, as it enables us to explore how OoC access patterns are mutated by the underlying file system in question, an important consideration for the HPC system designer. Second, we collected device-level block traces completely under GPFS on all of

Location-FileSystem	PCIe Controller	PCIe Interface/ Bus Speed	PCIe Lanes
ION-GPFS	Bridged	2.0/SDR 400MHz	8
CNL-JFS	Bridged	2.0/SDR 400MHz	8
CNL-BTRFS	Bridged	2.0/SDR 400MHz	8
CNL-XFS	Bridged	2.0/SDR 400MHz	8
CNL-ReiserFS	Bridged	2.0/SDR 400MHz	8
CNL-EXT2	Bridged	2.0/SDR 400MHz	8
CNL-EXT3	Bridged	2.0/SDR 400MHz	8
CNL-EXT4	Bridged	2.0/SDR 400MHz	8
CNL-EXT4-L	Bridged	2.0/SDR 400MHz	8
CNL-UFS	Bridged	2.0/SDR 400MHz	8
CNL-UFS	Bridged	2.0/SDR 400MHz	16
CNL-UFS	Native	3.0/DDR 800MHz	8
CNL-UFS	Native	3.0/DDR 800MHz	16

Table 2: List of relevant software and hardware configurations evaluated.

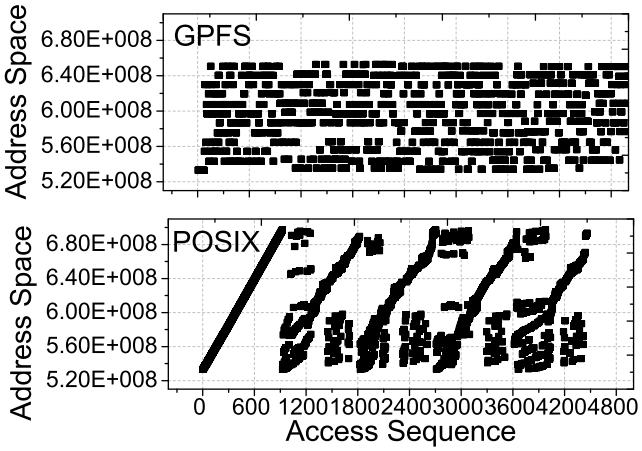


Figure 6: Block access patterns from the beginning of our OoC workload trace from the perspective of the POSIX block access pattern at the compute node (bottom) and the sub-GPFS block access pattern at the IONs (top)

the IONs. Since these traces are at the device-level, they may be directly fed to NANDFlashSim.

These access patterns are compared in Figure 6, and provide an example of how certain workloads (such as OoC applications) can be better served from local storage rather than through a parallel file system such as GPFS, which will break up that sequential stream in order to perform striping. Notably, GPFS divides up what was previously largely sequential in the compute-local trace, which deteriorates performance for NVMs that enjoy best performance when all of the dies are accessed at once (generally only possible with sequential accesses). Larger stripes combat this randomizing trend, but only to limited extents.

4.3 Architecture and File System Results

To begin our exploration of the proposed changes, we first present comparative throughput results for examining how migrating from ION-local to CN-local impacts our OoC application's I/O bandwidth performance. Since the CN-local migration implicitly requires exploration of possible local file systems, we also present results from such in Figure 7a.

In this work, we have explored the following file systems: The General Parallel File System (GPFS), Journaled File System (JFS), B-tree File System (BTRFS), XFS, and the second, third and fourth Extended File Systems (ext2, ext3 and ext4, respectively). [7, 8, 13, 27, 41, 45, 48] These file systems were chosen because they are currently supported, general-purpose file systems available on very modern Linux kernels and have relatively wide user bases. Due to lack of space, we do not delve into each of their specific semantics and the manner in which they change access patterns when passing data from the incoming application to the outgoing block device layer. The important take-away is the degree to which they individually impact throughput for the underlying NVM device, and in the aggregate that file systems need to be at least tuned and, ultimately, rewritten, to perform optimally with NVM devices.

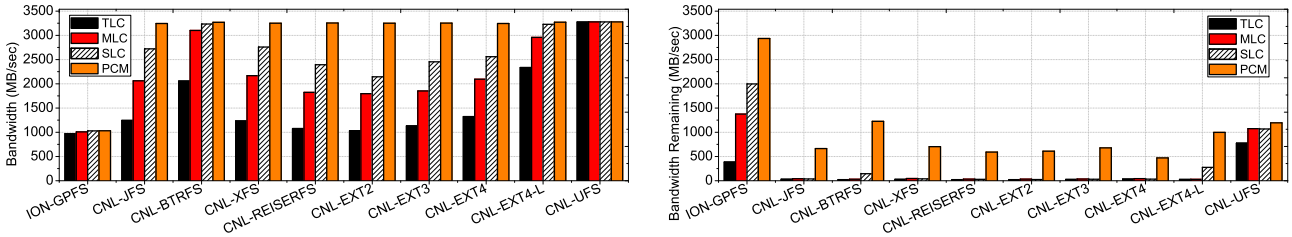
Beginning our discussion with the ION-local case, we see that it runs up against the throughput limit for QDR Infiniband. Even in the worst performing file systems for the CN-local approaches, improvements over the ION-GPFS setup are 7%, 78%, and 108% for TLC, MLC, and SLC, respectively. As network throughput expands as 40 Gigabit Ethernet and similarly high-bandwidth networking technologies drops in price, one could argue that they will help to overcome this bottleneck. However, the chart in Figure 1 suggests the rate at which the network catches up lags well behind the rate at which NVM media is expanding in bandwidth. Therefore, between these poor results for ION-local NVM and the lack of confidence that such a bottleneck will diminish in the near future, we believe these results make a cogent case for compute-local NVM.

Moving to consideration of the very interesting dynamics between file systems as shown in Figure 7a, we conclude that first, bandwidth varies widely for NAND flash media between file systems and second, that simple tuning of the file system and block device layer can go a long way towards improving performance. In the former conclusion, the difference between the lowest performing file system ext2, to the highest performing, non-tuned file system BTRFS, we can see an increase in bandwidth by a factor of 2 when considering TLC. Further, in the case of ext4-L, which stands for ext4 with "large request sizes," we show that by simply turning a few kernel knobs (in this case knobs related to the number of file system requests that can be coalesced together at the block device layer in the kernel), we demonstrate an improvement of about 1GB/s. These differences, and improvements in the latter case, are all strongly related with the size of the requests that actually reach the underlying SSD. As a direct result of this, by preserving the size and sequentiality of the requests from the OoC application, UFS is able to reach the maximal throughput available under PCIe 2.0 with eight lanes. A last observation we make is that, due to the much higher read speeds of PCM, it is able to obscure the differences between file systems as the PCIe 2.0 8x bottleneck becomes the only limit of interest.

To understand where to go from these bandwidth results, in Figure 7b we measure and document just how much bandwidth performance is left over in each architecture and for each NVM type. By looking at the remaining bandwidth for the NVM media, we are able to assess just where the bottlenecks might be. For example, all of the media in the ION-GPFS case is leaving a lot of performance untouched because of the major bottleneck being the network; the media spends a lot of time simply idling waiting for the data to be sent or received across the Infiniband. Somewhat less obvious, it is interesting to see that while most of the CNL file systems leave little behind for the NAND flash memories, UFS does. This gives a better example of how this graph should be interpreted – by using the underlying NVM media more efficiently, and reaching a bandwidth bottleneck in the PCIe interface, UFS-controlled media completes its requests faster and therefore ends up idling, which is why the bandwidth left-over is much higher than in any other file system. This remaining bandwidth leads us to examine how to overcome the bottlenecks in the hardware to extract that remaining performance, as we do in the next subsection.

4.4 Results of Device Improvement

As discussed in Section 3.3, we take a three-part approach to exploring how to reduce overheads and eliminate bottlenecks we discovered and discussed earlier. First, we examine how removing the needless SATAe bridging in common PCIe devices and migrating to a much lower encoding cost interface such as PCIe 3.0 will help reduce bandwidth losses. Sec-



(a) Bandwidth Achieved. (b) Bandwidth Remaining.

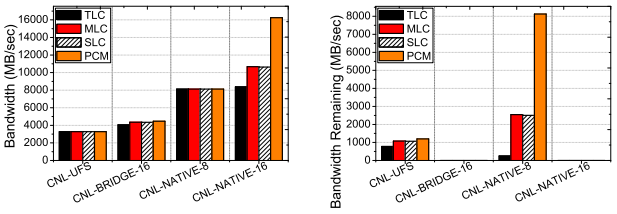
Figure 7: Performance achieved and left-over comparison between traditional ION-local architecture on GPFS and CNL architecture using various file systems and four different NVM types.

ond, we examine how improved lane widths, such as moving to use of all of the lanes available in PCIe (i.e., 16), will alleviate the performance overheads we observe. Third and last, we examine how improved NVM bus frequencies beyond the state-of-the-art ONFi 3 standard might finally expose full performance of the NVM media.

Results of this exploration and experimentation are shown in Figure 8a. First, we show the UFS results from the previous bandwidth graph as a comparison point; again, those results are limited by the bandwidth of PCIe 2.0 8x. Second, expanding the lanes from 8 to 16 in CLN-BRIDGE-16, but staying with PCIe 2.0, we see that bandwidth only increases marginally, even though the bandwidth is theoretically twice the base case. We determine this to be a result of the new limit becoming the overheads of the 8/10b encoding utilized by PCIe 2.0, indicated by the BRIDGE keyword in the name, and the slow speeds of the NVM bus.

Third, moving beyond PCIe 2.0 to a non-bridged, lower encoding overhead interface of PCIe 3.0 that uses a 128/130b encoding, and a faster, DDR bus interface that operates at 800MHz, we can see that CNL-NATIVE-8 outperforms CNL-BRIDGE-16 by a factor of 2, *despite having only half as many PCIe lanes available.* This provides credence to the idea that, even in a smaller focus such as within the SSD itself, a holistic approach to improving the hardware must be taken. Improving the lane-widths did very little compared to the broader approach of increasing the NVM bus speeds and migrating to a lower overhead interface.

Last, as we observed bandwidth being left over even with this vastly improved architecture, evinced in Figure 8b, we realized the full 16 lanes of PCIe 3.0 could be utilized to finally reach full performance of the NVM media within. Therefore, the last set of results in Figure 8a show just how fast this NVM media can perform if the bottlenecks and overheads between the OoC application and the individual



(a) Bandwidth Achieved. (b) Bandwidth Remaining.

Figure 8: Performance achieved and left-over beginning with the basic UFS architecture and extending through increased PCIe lanes and improved NVM bus frequency architectures.

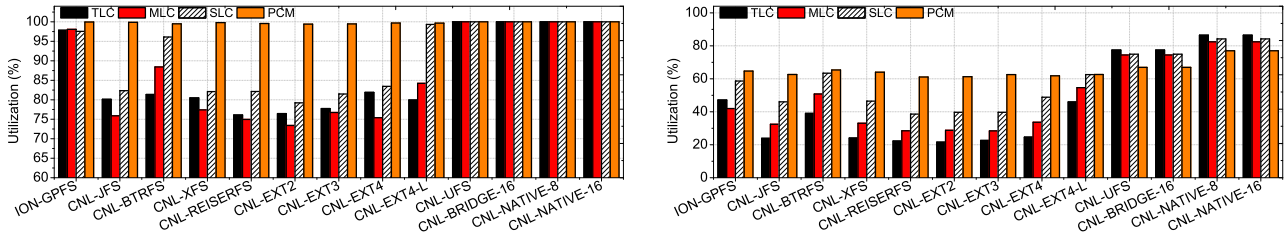
NVM packages are alleviated. In the case of PCM, an incredible factor of 16 improvement is observed between the initial ION-GPFS results and the CNL-NATIVE-16, *despite the underlying NVM media not changing whatsoever.* This highlights just how seriously bottlenecked current NVM device solutions are today. Even when considering the relatively slow media of TLC, we observe an increase of 8 times between those two architectures. Moreover, if considering compute-local against compute-local to perhaps be a bit more fair, we can see that the improvements are still many times the performance of an untuned, traditional file system running on limited hardware compared to our custom UFS on expanded hardware architecture.

4.5 Digging Deeper

While the high-level findings in terms of overall bandwidth and bandwidth foregone for each architecture examined were helpful in our exploration of the configuration and architecture space, these only scratch the surface of what is really going on within the SSD and fails to tell the entire story. Therefore, in this subsection we dig deeper into these findings by first analyzing utilization at a lower level, in terms of channel utilization, and then even lower at the per-package level. Further, we examine the two particularly interesting NVM types, TLC and PCM, and measure how much time is spent by each device on specific operations and what level of parallelism is extracted for each architecture.

We define "channel-level utilization" as the percent of total channels kept busy throughout the execution of the OoC application, whereas "package-level utilization" is the percent of active NVM packages kept busy serving requests throughout the execution of the OoC application. Beginning with average utilization for each type of NVM and every architecture explored, an interesting and altogether unexpected result shows itself in Figure 9a for the ION-GPFS architecture: while the low performance exhibited by this architecture and the bottlenecking on the network would lead one to believe the channel utilization should be low, quite the opposite occurs, coming in second across all architectures except for results involving UFS. As it turns out, the reason behind this is the striping in GPFS, which results in more randomized accesses and more channels being utilized simultaneously. However, it is important to note that high utilization does not necessarily mean the channels are being used *efficiently.* As the deeper package-level utilization chart shows in Figure 9b, while the ION-GPFS architecture utilized its channels well, the utilization of the underlying packages is quite low compared to UFS.

A second observation is that, with the exception of GPFS, whom stripes quite unlike all of the local file systems, higher



(a) Channel-Level Utilization.

(b) Package-Level Utilization.

Figure 9: Average channel and package utilizations across all considered architectures and file systems.

utilizations seem to correlate with a larger fraction of the total performance being extracted, and, as can be seen, the UFS-based architectures manage to achieve near full utilization of the channels and reach greater than 80% of the average package bandwidth. Last, we observe that even small increases in efficient utilization of the underlying NVM media can result in multiple gigabytes per second of bandwidth in real increases, giving credence to the importance of tuning even the finest details of the SSD to assure that the NVM media is fully leveraged.

As a last way to dig into the true reasons behind our findings, we have inserted probes into our simulator to try and quantify the time each architecture spends performing particular operations in the I/O process and additionally, at what level of parallelism the architecture is operating at on average. Results from such are shown in Figure 10.

For our operation quantification, we decompose all time spent in the device into six major categories:

- *Non-Overlapped DMA:* Time spent in data movement between SSD and the host, which includes thin interface (SAS), PCIe bus, and network.
- *Flash-Bus Activation:* Time spent in data movement between registers (or SRAM) in NVM packages and the main channel.
- *Channel-Bus Activation:* Time spent in data movement on the data bus shared by NVM packages.
- *Cell Contention:* Time spent waiting on an NVM package already busy serving another request.
- *Channel Contention:* Time spent waiting on a channel already busy serving another request.
- *Cell Activation:* Time spent actually performing a read, write, or erase operation on an NVM cell. This includes time spent moving the data between NVM internal registers (or SRAM) and the cell array.

Figures 10a and 10c are extremely useful dissections of the execution time for each architecture and clearly describe just what operations represent the biggest bottleneck to improving performance. Specifically, we can see that in the ION-local cases, a significantly larger proportion of time is spent in non-overlapped DMA than any other case, a direct result of the network bottleneck. Similarly, while internal bus activities, demarcated by flash bus activation and channel activation, dominate the proportion of time spent on operations in traditional file systems due to their errant division of requests, we witness how UFS truly leverages the underlying NVM by drastically reducing the time spent on those operations. Last, towards the right side of both of the PCM and TLC figures, we can see that the time spent actually performing the read, write, or erase on the underlying NVM cells grows significantly, becoming the dominant operation in the case of TLC. This is a nearly ideal case – the closer one can get to waiting solely on the NVM itself, the better.

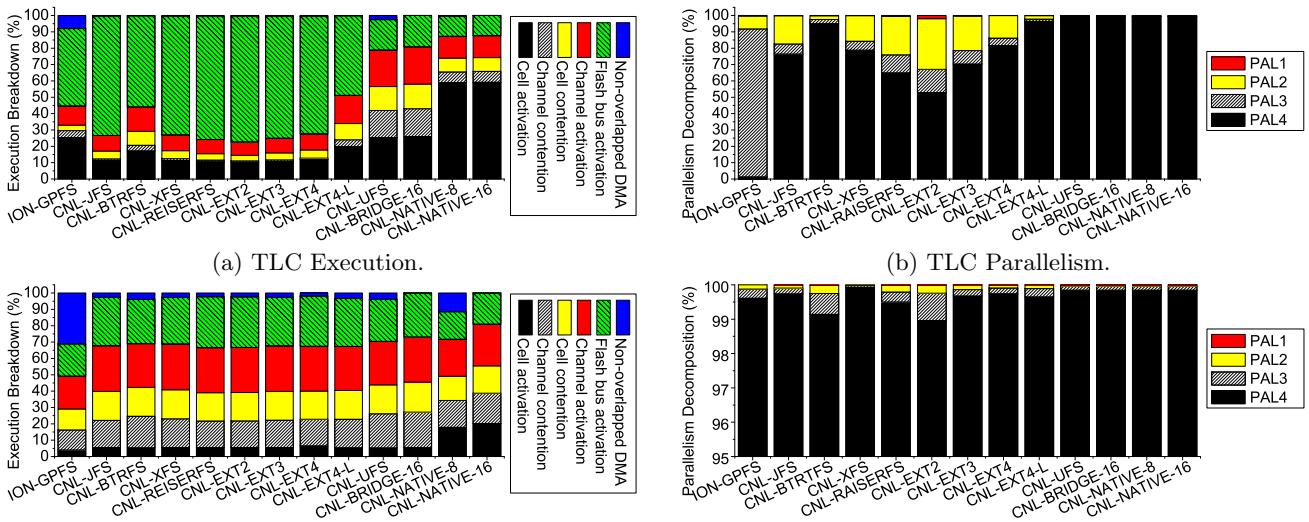
Then, we classify parallelism into four types:

- *PAL1:* System-level parallelism via solely channel striping and channel pipelining.
- *PAL2:* Die (Bank) interleaving on top of PAL1.
- *PAL3:* Multi-plane mode operation on top of PAL1.
- *PAL4:* All previous levels above.

Parallelism is an important and distinct metric to examine apart from utilization or operation time breakdown as reported in the last sets of graphs – it provides an idea of the effectiveness of an architecture *from the perspective of the request, rather than from the perspective of all of the hardware.* However, this simultaneously means that a small request may be issued that blocks a subsequent large request; both of these will achieve full parallelism when they are run, but they conflict on the targeted NVM and therefore utilization may be low. Similarly, time spent in contentious states in the execution breakdown figures may be high, so ultimately all graphs must be concurrently examined for a complete understanding of the situation. The important take-aways from the parallelism graphs plotted in Figure 10 are first that ION-local NVM storage runs into issues attempting to parallelize requests; because the parallel file system above it decomposes sequential accesses into stripes this can easily lead to needlessly small and unparallelizable accesses. As the TLC graph in Figure 10b shows, ION-local PCIe stays almost completely parallelism type PAL3, and almost never makes it to the full parallelism of PAL4. Second, UFS-based architectures are able to almost entirely reach parallelism state PAL4, largely a result of the increased sequentiality exposed by UFS compared with traditional file systems. The PCM-based graph is almost entirely in state PAL4, a direct result of the much smaller page sizes than in the NAND-based NVM types. In other words, incoming I/O requests can be easily be spread across all dies in the SSDs, which improves the degree of parallelism. Last, due to the fast access speeds of PCM, such SSDs are able to fully utilize their internal channels and reach near-to-theoretical maximums in performance, irrespective of the file system employed.

5. RELATED WORK

There is a rapidly expanding body of work on accelerating scientific applications, improving the state-of-the-art for solid-state storage, and exploring how SSDs can be used to improve scientific workloads. However, in that collection there are few works that parallel our approach to utilize compute-local NVM *storage* (not server or client caches).



(a) TLC Execution.

(b) TLC Parallelism.

(c) PCM Execution.

(d) PCM Parallelism.

Figure 10: Breakdowns in terms of time spent in particular states in the SSD and at what level of parallelism was extracted during the execution, for both TLC and PCM NVM types.

Beginning from the scientific vantage point, there have been quite a few investigations to determine if NVM, particularly flash-based SSDs, are effective alternatives or supplements to traditional HPC storage for scientific workloads. As a sampling of such research, Cohen et al. [16] comparatively evaluated an SSD array against local disk and a file server using three supercomputing benchmarks. Further, considering a wider range of SSD types and parallel I/O traces, [30,40] provided a broader understanding of SSD I/O performance for various scientific data-intensive workloads. Moving to the field, several research laboratories are making efforts to improve the performance of their supercomputers with the SSD-based storage. For instance, San Diego Supercomputing Center deployed SSDs in their supercomputer, *Gordon* [18, 24, 31], to reduce the latency gap between the memory and disks, as has the Tokyo Institute of Technology for their supercomputer, *TSUBAME* [43].

Agnostic of the application, there exist a number of works that consider flash-based SSDs as caches. The works are too numerous to identify all of them in this space, but notable works that are actually designed for use in a distributed setup include [25,28,29]. All of these differ from ours in that they use the flash-based SSD strictly as an extension of the traditional DRAM buffer cache. While the effect of a cache might appear on the surface to be equivalent, or even easier to effect, than allowing manual management of the NVM as we do in this work, the major issue is the extremely long time to properly heat up the cache. Because the working set size of OoC computing is so large, and data is not as frequently doubled-back on as in other arenas, we actively chose against implementing our approaches as caches because it would have such low hit rates. In a similar bin as these caching works, the authors of [33] utilize flash-based SSDs as write-back caches for checkpointing. Again, in this solution the application is again oblivious of the rather different medium it is sending checkpoints to and therefore, cannot leverage it towards improved steady-state performance; this will only improve checkpoint performance. We are not aware of any works where the flash is kept as a compute-local, application- or framework-managed pre-load space, which is critical for the efficacy of our approach in OoC computing.

Last, in [10] the authors design an interface for non-volatile memory that could be seen as similar to our UFS design. However, this similarity is merely superficial – not only does their proposal break our prerequisite that NVM should be colocated with every compute node in use, but more importantly they insert yet another interposing layer to allow the NVM to be accessed in byte-addressable fashion, directly anathema to our goal of reducing interposing layers and improving performance.

6. ACKNOWLEDGEMENTS

This research is supported in part by NSF grants 1017882, 0937949, 0833126, OCI- 0904809 and DOE grant DE-SC0002156. This work was also supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231. and partially supported by Scientific Discovery through Advanced Computing program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Nuclear Physics.

7. CONCLUSION AND FUTURE WORK

As NVM bandwidth begins to exceed point-to-point network capacity, we need to rethink existing approaches to utilizing NVM storage in HPC architectures. In this paper, we present an alternative role for NVM as a form of compute-local, large but slower memory, which we show is capable of removing current performance bottlenecks without major modifications to existing HPC architectures. Further, we proposed novel software and hardware optimizations that improve the compute-local acceleration and work to more directly expose the NVM for OoC applications. Our comprehensive evaluation, in which we consider a diverse set of potential NVM storage mediums, device and storage interface configurations, revealed that our compute-local SSD approach alone offers on average 108% performance enhancement compared to conventional client-remote SSD approaches, and our software- and hardware-optimized SSDs improve an additional 52% and 250%, respectively, on the base-line compute-local SSD approaches.

8. REFERENCES

[1] NAND flash memory MT29F32G08ABAAA, MT29F64G08AFAAA SLC datasheet. Technical report, Micron Technology, Inc.

[2] NAND flash memory MT29F64G08EBAA TLC datasheet. Technical report, Micron Technology, Inc.

[3] NAND flash memory MT29F8G08MAAWC, MT29F16G08QASWC MLC datasheet. Technical report, Micron Technology, Inc.

[4] P5q serial phase change memory (PCM) datasheet. Technical report, Micron Technology, Inc.

[5] P8p parallel phase change memory (PCM) datasheet. Technical report, Micron Technology, Inc.

[6] OpenMP Architecture Review Board. Openmp specification c/c++ version 2.0. `http://www.openmp.org`.

[7] A. Mathur et. al. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, 2007.

[8] A. Sweeney et. al. Scalability in the xfs file system. In *ATEC '96*.

[9] B. Lee et. al. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09*.

[10] Chao Wang et. al. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. IPDPS '12, 2012.

[11] E. Kim et. al. Frash: hierarchical file system for fram and flash. ICCSA'07.

[12] F. Chen et. al. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA' 11*.

[13] F. Schmuck et. al. GPFS: a shared-disk file system for large computing clusters. In *FAST'02*.

[14] Hasan M Aktulga et. al. Improving the scalability of a symmetric iterative eigensolver for multi-core platforms. *Concurrency and Computation: Practice and Experience*.

[15] Hasan M Aktulga et. al. Topology-aware mappings for large-scale eigenvalue problems. *Euro-Par Parallel Processing*, 2012.

[16] J. Cohen et. al. Storage-intensive supercomputing benchmark study. Technical report, Lawrence Livermore National Laboratory.

[17] J. Do et. al. Turbocharging dbms buffer pool using ssds. SIGMOD '11.

[18] J. He et. al. Dash: a recipe for a flash-based data intensive supercomputer. SC '10.

[19] J. Kim et. al. Flashlight: A lightweight flash file system for embedded systems. *ACM Trans. Embed. Comput. Syst.*

[20] M. Beynon et. al. Distributed processing of very large datasets with datacutter. *Parallel Comput.*

[21] M. Jung et. al. NANDFlashSim: Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*.

[22] M. Jung et. al. Physically addressed queueing (PAQ): improving parallelism in solid state disks. In *ISCA'12*.

[23] M. Marques et. al. Using graphics processors to accelerate the solution of out-of-core linear systems. In *ISPDC' 09*.

[24] M. Norman et. al. Accelerating data-intensive science with gordon and dash. TeraGrid '10.

[25] M. Saxena et. al. Flashtier: a lightweight, consistent and durable storage cache. In *EuroSys '12*.

[26] R. Canon et. al. GPFS on a cray XT.

[27] R. Card et. al. Design and implementation of the second extended filesystem. `http://web.mit.edu/tytso/www/linux/ext2intro.html`.

[28] R. Koller et. al. Write policies for host-side flash caches. FAST '13.

[29] S. Byan et. al. Mercury: Host-side flash caching for the data center. MSST '12.

[30] S. Park et. al. A performance evaluation of scientific i/o workloads on flash-based ssds. In *CLUSTER '09*.

[31] S. Strande et. al. Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer. In *XSEDE '12*.

[32] W. Josephson et. al. Dfs: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 2010.

[33] Xiangyu Dong et. al. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 2011.

[34] Y.-Yu Chen et. al. Local methods for estimating pagerank values. In *CIKM' 04*.

[35] Z. Zhou et. al. An out-of-core eigensolver on ssd-equipped clusters. In *CLUSTER '12*.

[36] Zheng Zhou et. al. An out-of-core dataflow middleware to reduce the cost of large scale iterative solvers. In *ICPPW '12*.

[37] Open NAND Flash Interface Working Group. Open NAND flash interface specification revision 3.1. `http://www.onfi.org/`.

[38] M. Jung and M. Kandemir. Challenges in getting flash drives closer to CPU. In *HotStorage '13*.

[39] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *HotStorage '12*.

[40] M. Jung and M. Kandemir. Revisiting widely-held expectations of ssd and rethinking implications for systems. In *SIGMETRICS '13*.

[41] D. Kleikamp. Journaled file system. `http://jfs.sourceforge.net/`.

[42] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 2001.

[43] S. Matsuoka. Making tsubame2.0, the world's greenest production supercomputer, even greener: challenges to the architects. In *ISLPED '11*.

[44] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *ESA' 02*.

[45] N/A. Btrfs. `http://btrfs.wiki.kernel.org`.

[46] J. No. Nand flash memory-based hybrid file system for high i/o performance. *J. Parallel Distrib. Comput.*

[47] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External memory algorithms*, 1999.

[48] S. Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.