Triple-A: A Non-SSD Based Autonomic All-Flash Array for High Performance Storage Systems

Myoungsoo Jung[1], Wonil Choi[1], John Shalf[3], and Mahmut Taylan Kandemir[2]

[1] Department of EE, The University of Texas at Dallas, Computer Architecture and Memory Systems Laboratory
[2] Department of CSE, The Pennsylvania State University, Microsystems Design Laboratory
[3] National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory
{jung, wonil.choi}@utdallas.edu, jshalf@lbl.gov, kandemir@cse.psu.edu

Abstract

Solid State Disk (SSD) arrays are in a position to (as least partially) replace spinning disk arrays in high performance computing (HPC) systems due to their better performance and lower power consumption. However, these emerging SSD arrays are facing enormous challenges, which are not observed in disk-based arrays. Specifically, we observe that the performance of SSD arrays can significantly degrade due to various array-level resource contentions. In addition, their maintenance costs exponentially increase over time, which renders them difficult to deploy widely in HPC systems. To address these challenges, we propose Triple-A, a non-SSD based Autonomic All-Flash Array, which is a self-optimizing, from-scratch NAND flash cluster. Triple-A can detect two different types of resource contentions and autonomically alleviate them by reshaping the physical data-layout on its flash array network. Our experimental evaluation using both real workloads and a micro-benchmark show that Triple-A can offer a 53% higher sustained throughput and a 80% lower I/O latency than non-autonomic SSD arrays.

Categories and Subject Descriptors B.3.1 [*Memory Structures*]: Semiconductor Memories; D.4.2 [*Operating Systems*]: Storage Management; C.4 [*Computer System Organization*]: Performance of Systems

General Terms Solid State Disk; NAND Flash; High Performance Computing; Flash Array Network; Resource Contention; Self Optimizing.

1. Introduction

Over past years, High Performance Computing (HPC) storage systems have increased storage capacity by adding expansion enclosures with low-cost spinning disks. While such a scale-up HPC storage system achieves high aggregate storage processing capacity, its data processing capacity does not increase linearly. In addition, the bandwidth improvements brought by this traditional HPC storage system started to stagnate since a large fraction of spinning disks are underutilized most of the time. For instance, [10] observed that IBM Blue Gene/P system, called Intrepid, achieves only one-third bandwidth of its full capacity in 99% of the time. As the computational power and degree of parallelism of HPC increase, this underutilization becomes a more problematic challenge in meeting HPC users' high bandwidth and low latency demands [10, 30].

To address this problem, several HPC systems employ Solid State Disks (SSDs) [9, 13, 20, 31, 39]. For example, Carver [36], an IBM iDataPlex cluster, of NERSC at Lawrence Berkley National Laboratory employs two high-performance SSDs for each storage server node (known as Lustre Router Node) [7, 33]. These sever nodes are connected to thousands of compute nodes over 4x QDR InfiniBand, and their SSDs are connected to the RAID couplets for the Lustre storage on the back-end as a storage cache. These multiple SSD caches are mainly used for very data-intensive applications such as out-of-core computations [53] that primarily exhibit read-intensive access patterns [9, 27, 31, 53].

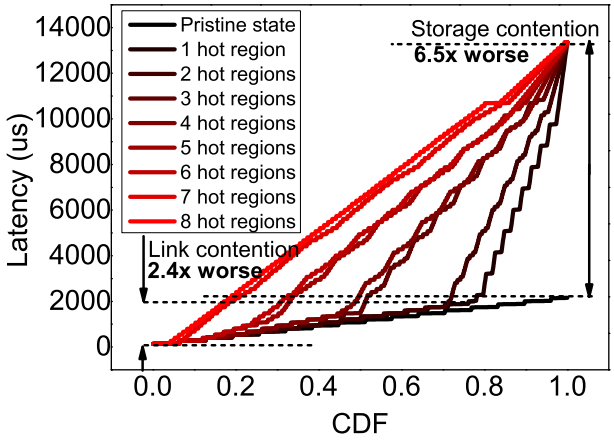


Figure 1: Cumulative distribution function with varying hot regions, which refer to a group of SSDs having at least 10% of the total data in a flash array. Note that, as we increase the number of hot regions (i.e., put more pressure on the flash array), resulting link- and storage-contentions degrade performance by 2.4x and 6.5x, respectively.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541953

In contrast, Argonne National Laboratory and Los Alamos National Laboratory employ multiple SSDs as a burst buffer in I/O nodes at the compute node side [30, 40]. The main purpose of this SSD server is to absorb the heavy write traffic exhibited by check-pointing and some other I/O-intensive HPC applications before storing actual data into back-end file servers and enterprise storage servers over Ethernet and InfiniBand.

One of main challenges behind these HPC SSD use-cases is that they are *not* designed for scalability. Specifically, the SSDs are directly connected to individual HPC compute nodes or server storage nodes and optimized for specific applications. Recently, industry have announced SSD-based, all-flash storage systems, composed of hundreds or thousands enterprise-scale SSDs in the form of one-large-pool storage server. For example, NetApp announced that an all-SSD server for transactional, database-driven applications will be available sometime in 2014 [37]. Pure Storage and EMC proposed a 100% flash storage array composed of multiple SAS-based SSDs [18, 46]. While these SSD arrays offer better performance and scalability as one-large-pool storage systems, *the cost of an SSD is still too high to replace a conventional disk*, and *the maintenance costs of these SSD arrays are unacceptable compared to current main memory technologies* (the costs exponentially increase). In addition, architectural and implementation details of these emerging all-SSD arrays are not known; to our knowledge, there are no publicly available specifications for all-flash arrays.

More importantly, we observe that these SSD arrays face enormous challenges, which are not normally observed in disk-based arrays. Figure 1 plots the cumulative distribution function of an *all-flash* array consisting of 256GB 80 SSDs with varying data localities. One can see from this graph that the performance of the SSD array degrades as the number of hot regions increases – a hot region in this context corresponds to a group of SSDs having at least 10% of the total data in a flash array. There are two main reasons behind this performance degradation: 1) *link contention* and 2) *storage contention*. Even though the target SSD associated with incoming I/O requests may be ready to serve them immediately, if there already exists another SSD delivering data over the I/O bus shared by the target device, the incoming I/O requests have to be stalled until the I/O bus is released. This is referred to as link contention. On the other hand, the incoming I/O requests could also be stalled at a system-level (even though the data path is available), if the target SSD is busy in serving other I/O requests submitted in a previous stage, referred to as storage contention in this paper.

In this paper, we propose *Triple-A*, a *non-SSD based **A**utonomic **A**ll-Flash **A**rray*, which is a self-optimizing, from-scratch NAND flash cluster. Triple-A enables autonomic resource management, which can in turn alleviate both link-contention and storage-contention. Our experimental study shows that the proposed Triple-A offers about 53% and 80% higher sustained bandwidth and lower latency than non-autonomic SSD arrays, respectively. Our main **contributions** in this paper can be summarized as follows:

• *Autonomic link contention management.* Triple-A is able to detect the straggler I/O requests caused by link contention and migrate the corresponding data from the destination flash module(s) of the stragglers to new locations where the local shared bus is almost underutilized. Even though our autonomic link contention management is performed online, the required data migration can be overlapped with the data movement involved in serving the straggler I/O requests.

• *Autonomic storage contention management.* Triple-A autonomically reshapes the physical data-layout for the specific flash module suffering from storage contention, referred to as *Laggard*. In this reshaping, data on different physical blocks are moved from the laggard to adjacent flash modules in the background. As a result, Triple-A can address the storage contention problem and hide the data/storage processing time on reshaping the physical data-layout on its flash array network.

• *A from-scratch, all-flash array architecture.* A unique characteristic of Triple-A is that it employs flash modules rather than SSDs, by taking the bare NAND flash memory out from an SSD box. This *unboxing* the flash modules can deduct the costs of auxiliary internal counterparts from the cost of SSD, which can in turn lead to 50% cost reduction when building and managing hot-swappable and scalable all-flash arrays.

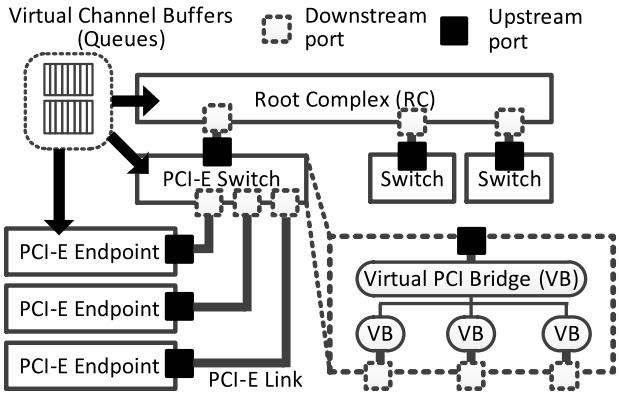


Figure 2: PCI Express topology and corresponding devices.

2. Background

Before presenting the details of our proposed system, we discuss background on flash arrays. We first introduce PCI-E devices and their network topology, which is used for interconnecting NAND flash modules in our all-flash array, and then explain the modern NAND flash technology that we leverage in this work. We conclude this section with an explanation of the fundamental functionality of flash software modules and the software stack.

2.1 PCI Express

Unlike a conventional I/O bus that handles data from multiple resources, PCI Express (PCI-E) implements a point-to-point serial connection, which is dual-simplex, high speed and high performance. As shown in Figure 2, this network-like connection makes PCI-E a promising interface to build large-scale high-speed storage mediums. Below, we explain the main PCI-E components employed in building our all-flash array.

Root Complex (RC). PCI-E RC supports one or more PCI ports, each of which is connected to a PCI-E switch device or underlying endpoint devices. PCI-E RC generates I/O transaction requests on behalf of host processor(s) and can route requests from one port to another. In practice, RC implements central computing resources including hot-plug, power management and interrupt controllers. In a flash array

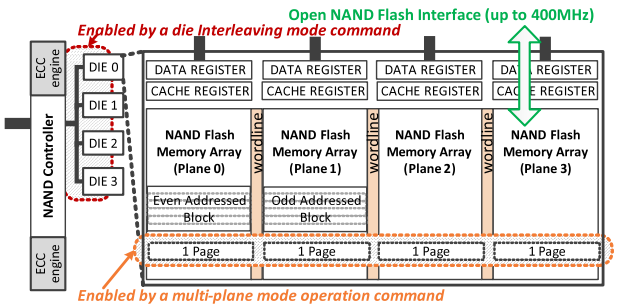


Figure 3: Bare NAND flash memory package.

architecture, root complex can also implement flash software and define the corresponding flash storage stack.

Switch Device. PCI-E switch consists of two or more virtual PCI-E to PCI-E bridges, which can in turn create multiple PCI-E endpoints (peripheral device). Considering Figure 2 as an example, a four-port switch is composed of four virtual bridges internally connected by an undefined bus; the upstream port is the port in the direction of PCI-E RC, and, all other ports are referred to as downstream ports, away from the RC. This switch can forward *packets* encoding the request transactions to the next device or destination based on address routing or device-ID routing through these upstream and downstream ports. It should be noted that this switch device can form a sub-hierarchy underneath RC, which in turn makes PCI-E systems highly scalable.

Flow Control. Each PCI-E device, including RC, switch and endpoints, implements its own *virtual channel buffer*. Therefore, a transmitted PCI-E packet is received into the virtual channel buffer of the target PCI-E device, called receiver. In practice, receivers periodically update transmitters on how much buffer/queue space they have available to serve incoming requests, and the transmitters send new packets only when the receivers have enough queue entries or buffer space to hold them. Clearly, if the destination is unavailable to serve or link associated with the target is occupied by other receivers or transmitters, the packet should be stalled in the virtual channel queue, which normally causes performance degradation.

2.2 Bare NAND Flash Package

Internals. As shown in Figure 3, NAND flash packages usually employ multiple flash dies which can operate in parallel to serve incoming memory requests. Each die is formed by putting multiple planes together over shared wordline(s). Different planes are identified by even or odd block addresses and can also concurrently service multiple requests. In addition, flash packages employ internal cache and data registers to mitigate the latency gap between the I/O interface and the memory array.

Parallelism and Commands. The different internal storage components shown in the figure can be activated in different fashions based on which NAND flash command is received at runtime. For example, the die-interleaving mode flash command can interleave memory requests among multiple dies, whereas the multi-plane mode flash command activates multiple planes in one or more dies in a flash package. Note also that, the cache mode flash command is required to take advantage of the internal cache registers.

Embedded Controller and ECC Engines. The NAND flash controller within flash packages is connected to the NAND internal buses via the ONFi interface. This embedded controller parses the flash commands and handles the protocols associated them. In addition, the controller of a modern flash package typically implements an ECC engine, which implements a 128 bit (or more) error correction capability. This controller decouples reliability management from flash software and simplifies the design of a flash array.

2.3 Flash Software Modules

Hardware Abstract Layer (HAL). To extract the true performance of a bare NAND flash, it is essential to compose flash commands which can take advantage of high degree of internal parallelism. HAL creates flash commands and handles the corresponding NAND flash protocols. Specifically, HAL commits a series of commands to the underlying flash packages, handles data movements, and oversees transaction completions. Thanks to HAL, other flash software modules are free from hardware specific operations and treat multiple flash packages over block device APIs (e.g., they read and write on physical address space that HAL abstracts).

Flash Translation Layer (FTL). FTL is a key component of any flash software stack, which hides the complexities and constraints of the underlying flash medium like the erase-before-write requirement and endurance limits. Generally speaking, FTL translates/remaps addresses between the virtual and physical spaces in order to offer block-level storage interface compatibility with spinning disks. It also performs wear-leveling, garbage collection, and data access parallelization across multiple flash packages.

Flash Software Stack. The flash software modules discussed above are usually implemented within solid state disks (SSDs) in the form of firmware or embedded system software. However, as NAND flash memories are widely deployed across diverse computing domains and become increasingly popular, SSD vendors are increasingly motivated to take out these flash software modules from their devices and place them into the host-side. This software stack reorganization can improve the management of the underlying devices by being aware of the host-level information and by taking advantage of abundant host-level resources (e.g., multicores). In a similar manner, these flash modules can be also implemented in one or more PCI-E devices within a PCI-E network.

3. Non-SSD based All-Flash Array

In this section, we first provide the motivation behind our proposed *non-SSD* flash module approach, and then explain a flash array architecture that can be used to build a highly scalable storage server. Lastly, we describe how such non-SSD flash modules can be organized and discuss how they can be incarnated as a storage cluster beneath the flash array topology.

3.1 Motivation

SSD-based all-flash arrays [18, 37, 46] are expandable by adding more SSD storage shelves, and maintained by replacing the worn-out SSDs with new ones. Clearly, the price of an individual SSD is key to enabling these SSD arrays in HPC or major enterprise-scale systems with high scalability and serviceability. Unfortunately, SSDs are com-

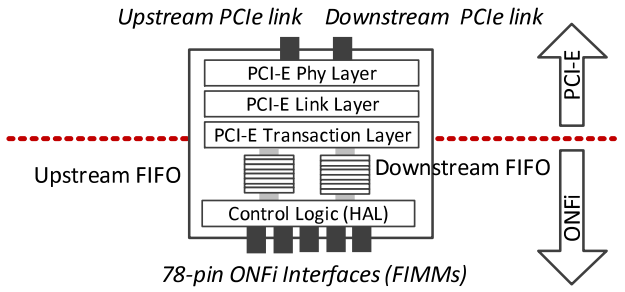


Figure 4: An endpoint device that can be used to build a cluster.

plex systems, which makes them expensive (even though NAND flash costs are expected to continue going down). Specifically, bare NAND flash packages in an SSD only account for 50% $\sim$ 65% of the total SSD cost (in comparison, DRAMs on Dual Inline Memory Module (DIMM) are 98% of the DIMM's cost [1]). This is because SSDs include costly firmware and controllers such as host interface controllers, flash controllers, microprocessors, and internal DRAM buffer modules (256MB $\sim$ 2GB) in addition to the NAND flash storage itself, whereas most of the control logic in DRAM is implemented in outside of DIMM. Consequently, SSD-based all-flash arrays needs to pay more than required for these extra software/hardware modules every time a worn-out SSD need to be replaced, and this cost penalty makes the large-scale deployment of SSD-based arrays difficult in HPC systems.

Motivated by this, we introduce a novel *non-SSD based* all-flash array architecture, which is a highly-scalable storage system built from scratch by *unboxing SSDs and interconnecting them over a PCI-E network*. This from-scratch (i.e., non-SSD based) array architecture can remove about 35% $\sim$ 50% extra hardware/software costs, making it a very promising storage solution for HPC.

3.2 Flash Array Architecture

Figure 5 illustrates the high-level view of our Triple-A. At a high-level, Triple-A treats each unboxed flash module as a "passive memory device" like DIMMs. It manages the underlying flash modules by employing an *autonomic* flash array management module implemented in external multi-cores and DRAM buffer, which needs no replacement over time. This enables the bare NAND flash packages to form a *Flash Inline Memory Module* (FIMM) with minimum physical-level protocol logic. One or more of these passive memory modules can be connected via a small PCI-E device called endpoint (EP) [8] over ONFI 3.x [38], which is a standard flash interface offering a 400MHz bus clock speed. In this work, the set of FIMMs connected to a PCI-E EP as a hot-swappable device is referred to as *cluster*. While each FIMM within a cluster can expose the true NAND flash performance over ONFI 3.x, the cluster offers an aggregated performance of them to host(s) over the external link of its PCI-E EP – the technical details of the FIMM and cluster are later given in Sections 3.3 and 3.4, respectively. From a system-level viewpoint, a set of clusters help us incarnate a flash array by interconnecting the autonomic flash array management module through multiple PCI-E links and ports on a PCI-E switch [44]. Further, a PCI-E network is formed from multiple switches to root complex's (RC) multi-ports. Finally, multiple compute nodes or hosts can access each FIMM over the RC's multi-ports managed by the autonomic flash array management module.

3.3 Flash Inline Memory Module (FIMM)

Even though unboxing SSDs can help us reduce the costs involved in building and maintaining storage blocks of all-flash arrays, *it is important to design the flash module to be easily connected to one of the conventional SSD interfaces*. Fortunately, ONFi industry workgroup specifies a DDR2-like standard physical connector for NAND flash modules in their 78-pin vertical NV-DDR2 interface design [52]. This NV-DDR2 describes not only the mechanical interface but also the signal groups and pin assignments, which can be used for forming a non-SSD flash module by integrating multiple NAND flash packages into a printed circuit board. Based on this specification, we define a standard NAND flash memory module, referred to as *Flash Inline Memory Module* (FIMM). FIMM minimizes the number of hardware components and simplifies the SSDs' internal complexities. Consequently, a FIMM can be easily replaced (like DIMM) by a new one when it is worn out. The FIMM internal organization is illustrated in Figure 6. Specifically, FIMM has been designed from the ground-up to work with eight NAND flash packages wired with a 16 data-pin channel and connected to the NV-DDR2 interface as depicted in Figure 6a. From a control signal configuration perspective, all the flash packages have separate chip enable control pins so that an external PCI-E device can activate an individual NAND flash package appropriately. However, their ready/busy pins used for checking up the flash chip status are connected using a single control wire as shown in Figure 6b. In this manner, FIMMs can exclude internal resources not related to storage mediums such as microprocessors, control units and extra DRAM buffers, unlike traditional SSDs over the NV-DDR2 interface.

3.4 Endpoint Design for a Cluster

To integrate multiple FIMMs into a cluster as a hot-swappable device of our Triple-A's PCI-E topology, we need to understand two different protocols between the front-end PCI-E and back-end ONFi NV-DDR2 interfaces. To this end, A PCI-E endpoint is implemented at the forefront of the cluster by employing three main components: 1) PCI-E device layers, 2) upstream and downstream buffers, and 3) a control logic including HAL, as shown in Figure 4. While device layers handle the PCI-E interface, HAL in the control logic is responsible for managing the underlying FIMMs over the multiple ONFi 78-pin NV-DDR2 slots. Specifically, PCI-E device layers disassemble a PCI-E packet by striping the header, sequence number and CRC field of each layer (e.g., phy, link, transaction). The disassembled packet is enqueued into the downstream buffer and waits to get serviced if the underlying control logic is busy. HAL dequeues the disassembled packet from the downstream buffer and constructs a NAND flash command based on the core section of the packet. Once the service is performed, a new packet is assembled in a similar manner to what we described above.

4. Autonomic Flash Array Management

In Triple-A, the flash control logic is moved from the inside of conventional SSDs to the autonomic flash array management module that resides in the external multicores (as

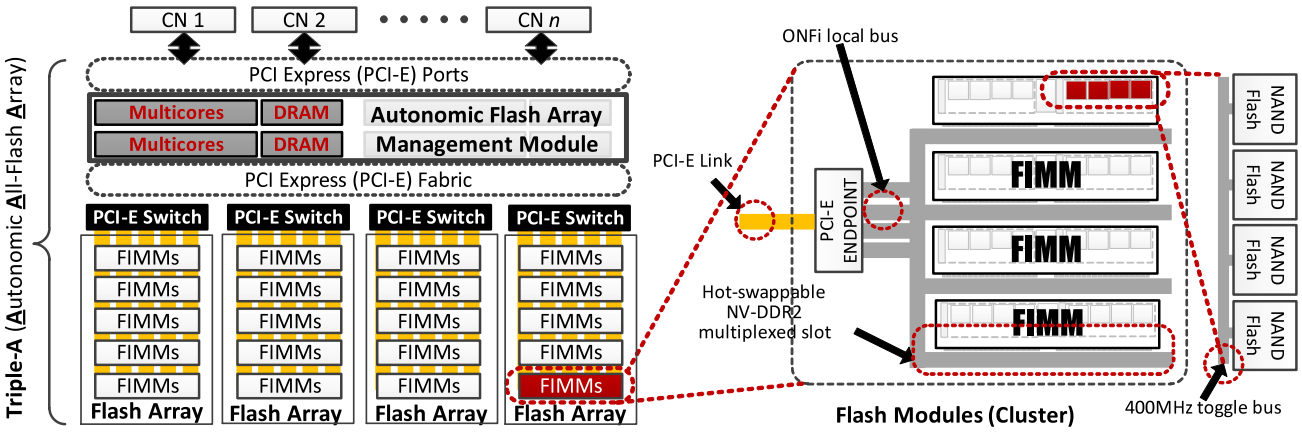
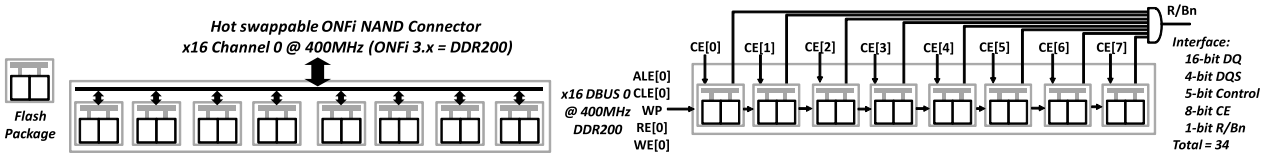


Figure 5: High-level view of our Autonomic All-Flash Array (Triple-A) architecture.



(a) Data Bus Connections

(b) Control Signal Connections

Figure 6: Internal organization of FIMM.

shown on the left part of Figure 5). This flash control logic (e.g., address translation [19, 29, 41], garbage collection [16, 23, 24], IO scheduler [25, 35], and page allocation [21, 22, 42]) can be implemented in many different ways, which we postpone to investigate in a future work. Instead, in this paper, we focus on presenting the details of the autonomic flash array management in the context of addressing the internal resource contention. To address the link contention problem, Triple-A identifies *hot-clusters* whose shared local bus is busy at any given time and excuses them (temporarily) from serving incoming I/O requests. Also, to solve the storage contention problem, Triple-A detects *laggard*, which is a specific FIMM that introduces a high number of stalled I/O requests in a cluster, and redistributes the data originally in the laggard to other FIMMs.

It should be noted that these types of contentions *cannot* simply be addressed by using a static or dynamic network routing methods. This is because, even though a intelligent data path could be selected using a different routing method, the destination would not be changed, and this would still introduce the same link- and storage-contention.

4.1 Link Contention Management

Hot-cluster detection. A cluster is classified as "hot", if the following condition is satisfied when the target FIMM device is available to serve I/O requests:

$$t_{Latnecy} \geq t_{DMA} * (n_{page} + n_{FIMM} - 1) + t_{exe} * n_{page}, \quad (1)$$

where $t_{latency}$ is the device-level latency of an I/O request monitored by the autonomic flash-array module, t_{DMA} is the data movement latency per underlying physical flash page, n_{page} is the number of pages associated with the I/O request, n_{FIMM} is the number of FIMMs in the cluster, and finally t_{exe} is the I/O latency for loading data (or storing data) into a specific location. Note that this expression can capture the cases where 1) the local shared bus is always busy to move data from the underlying FIMMs to the PCI-E EP, and 2) all the FIMMs in the cluster contend to acquire the local shared bus most of the time.

Autonomic data migration. Once the hot-cluster is detected, Triple-A clones the corresponding data from the hot-cluster to a cold-cluster, the local shared bus of which is currently a non-critical resource (as far as serving the I/O requests is concerned), and unlinks its original data location. This process is termed as *autonomic data migration*, and its details are illustrated in Figure 7. The target cold-cluster can be identified with the help of the following expression:

$$u_{Bus} < \frac{n_{size}/t_{DMA}}{n_{Pins} * 1/f_{inf}}, \quad (2)$$

where u_{Bus} is the bus utilization of a cluster, n_{Pins} is the number of I/O pins of a single flash package, n_{size} is the physical NAND flash page size, and f_{inf} is the shared bus clock frequency. This expression implies that u_{Bus} is lower than the case where on average only a single FIMM in the cluster uses the local shared bus.

Consequently, the request that causes a cluster to be hot will be served by a different location from the next I/O access; this in turn shortens the device-level latency and improves throughput by removing the stalled I/O requests heading to the hot cluster in the switch-level queue. Further, the stalled I/O requests targeting the hot-cluster can also deduct the cycles imposed by $t_{DMA} * n_{page}$ from their $t_{Latency}$, which can eventually turn a hot-cluster into a cold one.

Shadow cloning. Triple-A needs to check $t_{Latency}$ in order to detect a hot-cluster. In other words, the corresponding data is already available in the PCI-E EP of the hot-cluster when it is detected (no need to additionally access the underlying

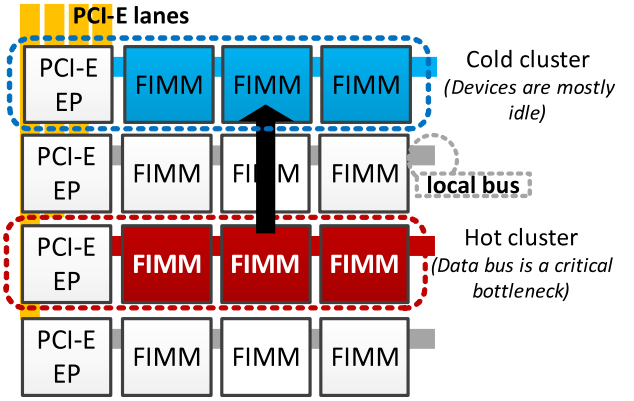


Figure 7: Autonomic hot cluster management. Once the hot-cluster is detected, Triple-A clones the corresponding data from the hot-cluster to a cold-cluster, the local shared bus of which is currently a non-critical resource.

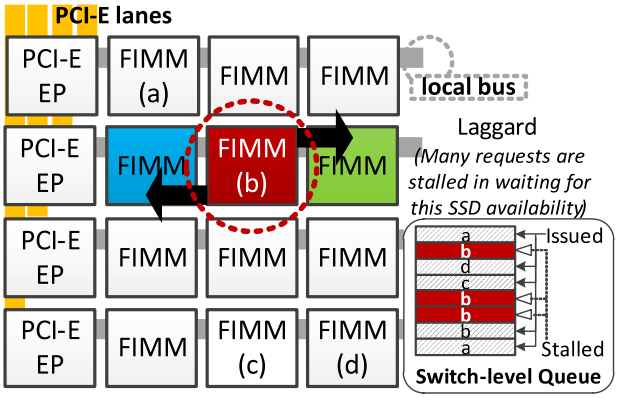


Figure 8: Laggard handling. FIMM marked using 'b' introduces most of the stalled I/O requests, which in turn makes it a laggard.

FIMMs). Since the I/O service routine employed by the flash array is moving the data from PCI-E EP to a host over the internal PCI-E fabric (to serve the current I/O request), there is a room to *overlap* this data movement with the autonomic data migration. Thus, Triple-A can migrate data from the hot-cluster to the cold-cluster, while the data is on its way to the host over the network. This *shadow cloning* can reduce the potential overheads brought by the autonomic data migration.

4.2 Storage Contention Management

Reads can be stalled at a switch-level queue while the target FIMM (i.e., laggard) is busy in serving the previous I/O requests. This can lead to higher latencies on the cluster associated with the target device. In contrast, writes might be buffered and return immediately even though a laggard exists, which allows the corresponding cluster to be temporarily available. However, during an I/O congestion period, the writes could also be stalled if the buffer needs to evict an I/O request to the underlying FIMM(s) in order to secure available room. Consequently, for both reads and writes, the longer latency imposed by storage-contention can potentially violate quality-of-service (QoS) requirements or service-level-agreements (SLAs). From a flash array perspective, if the queue is fully-occupied by a number of stalled I/O requests targeting the laggard, this can degrade the performance until all the stalled requests are flushed. Our autonomic storage contention management addresses these challenges by reshaping the physical data-layout.

Laggard detection. Triple-A can detect laggards based on two different strategies: 1) *latency-monitoring* and 2) *queue-examination*. While the latency-monitoring strategy can detect laggard(s) based on a QoS/SLA violation, the queue-examination strategy can detect them based on the flash array level serviceability. Let t_{SLA} be the time target aimed by SLA (or QoS), which Triple-A has to meet. *Stalled* I/O requests correspond to the requests at a switch-level queue that are not issued to an underlying FIMM yet. In contrast, issued I/O requests correspond to the requests that the target FIMM is currently processing, but not completed yet. Thus, a laggard can be detected by monitoring latency through the following expression:

$$t_{SLA} < \sum_{i=0}^{n_{stalled}} (t_{DMA} + t_{exe}) * n^i_{page}, \qquad (3)$$

where $n_{stalled}$ is the number of stalled I/O requests targeting the same FIMM and n^i_{page} is the number of pages associated with the i^{th} stalled request. On the other hand, with the queue-examination strategy, Triple-A counts the number of stalled requests for each FIMM when the queue has no more room for accepting incoming I/O requests. It thus identifies the FIMM that has the most stalled I/O requests as the laggard. Considering the queue shown on the right side of Figure 8 as an example, FIMM marked using 'b' introduces most of the stalled I/O requests (occupies 37% of the total queue entries), which in turn makes it a laggard. Note that in cases where all the target FIMMs have similar number of stalled I/O requests, Triple-A classifies all of them as laggards; normally this should not happen very frequently, but if it happens, the cluster will be managed by the autonomic data-layout reshaping scheme described next.

Autonomic data-layout reshaping. Unlike data migration on link contention, Triple-A reshapes the physical data-layout within the cluster by moving the data associated with all the stalled I/O requests from the laggard to other FIMMs, as shown in Figure 8. This is because the link is not the performance bottleneck in such cases and the contention solely stems from the actual data location. As mentioned earlier, in cases where all the FIMMs are laggards, the data will be migrated to another cluster across their PCI-E EPs, similar to what the autonomic data migration would do in the hot-cluster management. The data movement to reshape the physical data-layout can be performed by leveraging the shadow copying method if the laggard detection is invoked on reads. On the other hand, for writes, since the data is partially in its PCI-E EP or will arrive at the PCI-E EP from a host, Triple-A can redirect the stalled I/O requests to adjacent FIMMs *within the same cluster*. It then updates the address mapping information managed by the flash control logic, which is an essential functionality of any NAND flash memory storage system, for all incoming I/O requests.

It should be noted that if the bus-utilization keeps increasing after reshaping the physical data-layout (which may in turn introduce link contention), it will be handled by the autonomic link contention module again. Consequently, the loads between link contention and storage contention would be naturally balanced.

5. Experimental Setup

To evaluate our proposed autonomic flash array management strategies, we need a large flash memory array server, which can offer a high-fidelity evaluation environment and generate reproducible results. To the best of our knowledge, currently, there is no publicly-available all-flash array system. Therefore, in this work, we perform a simulation-based study with real workloads collected at thousand nodes on an IBM iDataPlex machine [36, 51] and published by SNIA [2] and UMASS [6]. In this section, we discuss the details of our simulation framework, the configurations we tested, and our workloads.

5.1 Flash Array

Flash Array Network Simulation Model. We modeled a PCI-E based mesh network for the all-flash array I/O simulation. Our simulation framework[1] can capture all the PCI-E specific characteristics on an all-flash array, including PCI-E data movement delay, switching and routing latencies, and I/O request contention cycles. In addition, it employs a reconfigurable network simulation based on numerous architecture/system parameters such as network size, link width, and routing strategy.

All-Flash Array Configuration. To collect our results, we configured 16TB *non-autonomic* and *autonomic* all-flash arrays. In our default flash-array configuration, the cluster is composed of four 64GB FIMMs connected to a single PCI-E EP. Our *baseline* network consists of four PLX technology PCI-E switches, each composed of sixteen EP clusters (e.g., 4x16 configuration). In this configuration, a single RC (root complex) has four ports, each connected to an individual switch over the PCI-E fabric. We used 3.3 μsec as our SLA value, and employed 650 $\sim$ 1000 queue entries for RC. For our sensitivity experiments presented later, we also evaluated diverse network sizes with varying number of clusters ranging from 4x8 (32 clusters, 8TB) to 4x20 (80 clusters, 20TB).

5.2 Workloads

Important characteristics of the real workloads we tested are given in Table 1. This table gives, for each workload, the read/write ratio, the percentage of randomness for both reads and writes, the number of hot-clusters, and the ratio between I/O requests heading to the hot-clusters and the total number of I/O requests. It should be pointed out that the most real workloads have 20% $\sim$ 60% of their I/O requests destined for the hot-clusters in our all-flash array, except for *cfs* and *web* workloads. For presentation purposes, we categorize these workloads into three different groups as explained below.

Enterprise Workload. We evaluate our all-flash array using a set of enterprise-scale workloads. Read and write ratios vary based on the application characteristics. For example, *proj* and *usr* exhibit write-intensive patterns (write ratio is around 70%) whereas *prn* and *web* are read-intensive workloads (read ratio is almost 100%). In these workloads, we allocated all-flash array as one large pool storage server that offers a single global address space to multiple worker threads.

Workload	Read ratio (%)	Read randomness (%)	Write randomness (%)	# of hot clusters	I/O ratio on hot clusters (%)
cfs	76.5	94.1	73.8	0	0
fin	50.2	90.4	99.1	5	55.7
hm	55.1	93.3	99.2	5	43.7
mds	25.9	80.2	94.8	4	54.1
msnfs	52.8	90.9	84.9	4	28.8
prn	97.1	94.8	46.6	2	50.9
proj	29.1	80.7	8.5	6	61.3
prxy	61.1	97.3	59.4	3	39.3
usr	28.9	90.3	96.9	5	40.1
web	100	95	0	0	0
websql	54.3	73.9	67.6	4	50.6
g-eigen	100	17.1	0	6	70.6
l-eigen	100	17.1	0	11	48.1

Table 1: Important characteristics of our workloads.

HPC Workload. We used an HPC workload generated by Eigensolver application [53] that simulates many-fermion dynamics in nuclear physics on one thousand IBM iDataPlex compute nodes [51]. Specifically, the workload was collected under Global Parallel File System (GPFS) [47], which is installed in ten Luster Router Nodes of Carver cluster at National Energy Research Scientific Computing Center [36]. Based on our observations, a majority of the I/O requests in this application (named *g-eigen*) are read, and many of them tend to head to a specific set of clusters (hot) in our all-flash array. We also evaluated this Eigensolver application in a different configuration, referred to as *l-eigen*. This version allocates a separate PCI-E switch to each router node, which can in turn expose multiple local address spaces of flash array.

Micro-benchmark. We also used a micro-benchmark for our specific evaluations, including sensitivity analysis and execution breakdown analysis. In our micro-benchmark, a workload composed of only random read I/O requests is called as *read*, whereas *write* indicates a workload that includes only random writes. Most I/O request sizes are 4KB, which is the maximum payload size in PCI-E 3.0 [43].

6. Experimental Results

We first evaluate our real workloads with both non-autonomic all-flash array and Triple-A in order to show the performance impact brought by our proposal. We then dig deeper into detailed analyses using our micro-benchmark, which include execution breakdown study and sensitivity tests with varying number of hot clusters and network sizes.

[1] This simulation framework will be available in the public domain for free download.

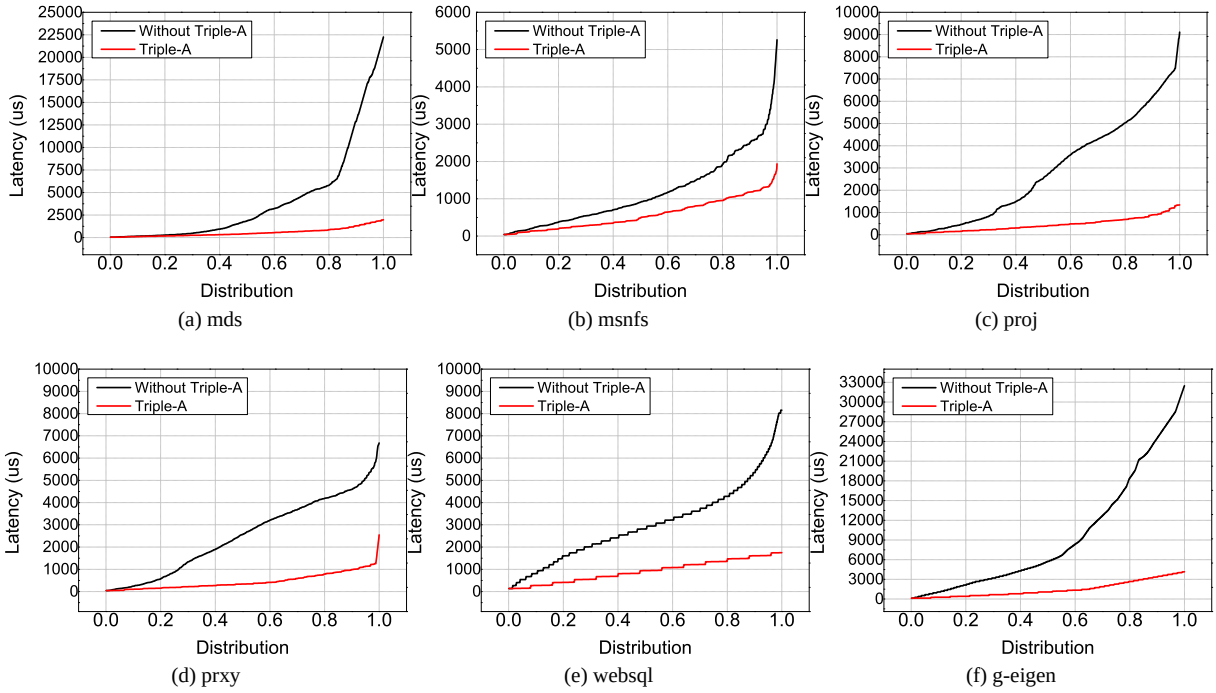


Figure 11: Cumulative distribution function of the real workload latency on a non-autonomic all-flash array and Triple-A. Triple-A shortens a large portion of latencies, particularly, the long tail is significantly improved.

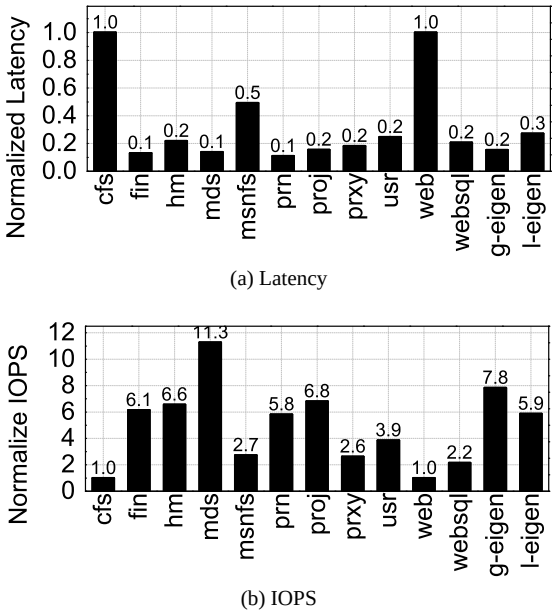


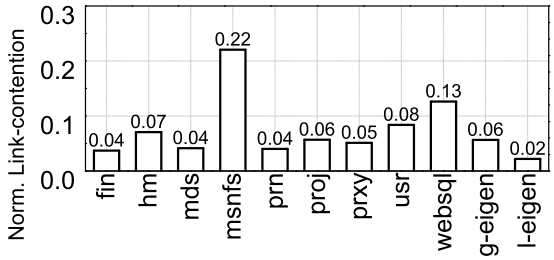
Figure 9: Latency and IOPS values normalized with respect to a non-autonomic all-flash array.

Workload	Avg. La-tency (us)	IOPS	Avg. link-cont. time (us)	Avg. storage-cont. time (us)	Avg. queue stall time (us)
cfs	940	229K	173	637	811
fin	5,457	39K	4,814	550	5,365
hm	1,806	112K	1,270	461	1,731
mds	4,064	35K	3,573	419	3,992
msnfs	1,172	152K	742	341	1,084
prn	6,733	42K	6,115	532	6,648
proj	2,860	87K	2,384	411	2,796
prxy	2,521	119K	2,021	422	2,444
usr	1,772	134K	1,364	338	1,702
web	1,148	181K	239	768	1,008
websql	4,737	57K	4,242	431	4,674

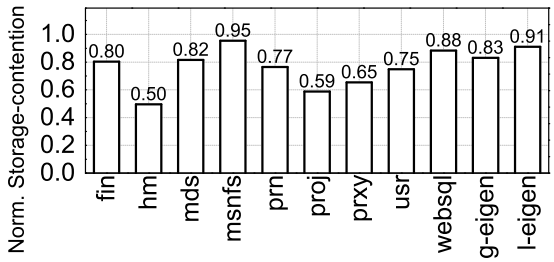
Table 2: Absolute values for our major performance metrics on a 4x16 non-autonomic all-flash array.

6.1 Results with Real Workloads

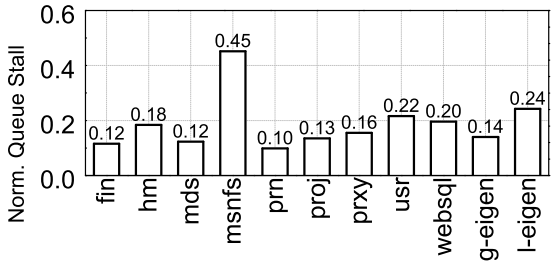
We use IOPS and average I/O latency as our main metrics. In the view of the overall performance of an all-flash array, IOPS is an appropriate system-level metric. However, since our all-flash array is essentially a networked system where a request moves around, the latency of each I/O request is also a big concern, irrespective of the entire system performance.



(a) Link-contention



(b) Storage-contention



(c) Queue stall time

Figure 10: Contention and queue stall time comparison between a non-autonomic all-flash array and our Triple-A.

To better understand the performance differences between non-autonomic and autonomic all-flash arrays, we *normalized* both IOPS and latency value observed with Triple-A to the corresponding values of the non-autonomic all-flash array. The absolute values of our performance metrics under the non-autonomic all-flash array are given in Table 2.

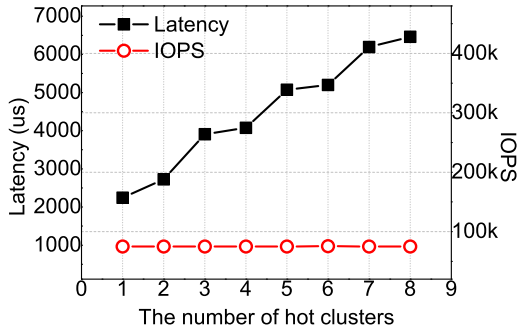
Figures 9a and 9b plot the normalized latency and IOPS, respectively, under the enterprise and HPC workloads. One can see from these figures that Triple-A achieves significant performance improvements, thanks to its autonomic data migration and data-layout reshaping. Specifically, Triple-A offers 5 times shorter latency, on average, compared to the non-autonomic flash array, and improves IOPS by about 2 times. The performance gains achieved by Triple-A are more pronounced when there is a small number of hot-clusters with high I/O rates. For example, Triple-A cuts the average latency by 98% and improves throughput by 7.8 times under *g-eigen* workload. This is because the performance degradations in a flash array network are mainly caused by resource contentions, not SSDs themselves. In contrast, there is no performance gain in the case where the all-flash array has no significant resource contention induced by the hot-clusters (e.g., *cfs* and *web* workloads). Interestingly, even though *websql* results in four hot-clusters with a high I/O request rate (see Table 1), the IOPS improvement brought by Triple-A is limited to around 2x. One of the reasons why the performance improvements under *websql* are not as high as other applications is that all the hot clusters (four) caused by *websql* belong to the same PCI-E switch. As a result, the storage space that Triple-A targets to move (for data migration) is limited. Recall that Triple-A does not migrate data across clusters located in different PCI-E switches in an attempt to avoid extra contentions. However, we want to emphasize that the latency (and also throughput) values even in this worst-case are still much better than the non-autonomic flash array (see Section 6.3).

6.2 Quantifying Contention Handling

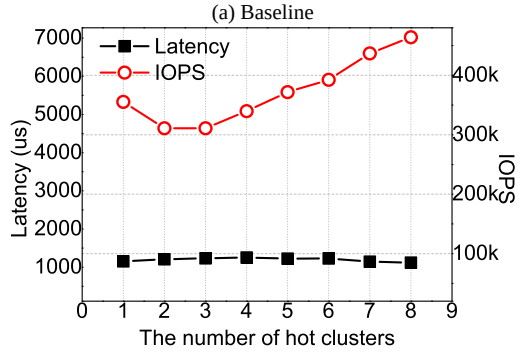
Figures 10a, 10b, and 10c present the normalized latency related to link contention, storage contention, and queue stall, respectively. The queue stall time indicates the amount of time an I/O request in a queue should wait until the I/O request submitted in the previous step is completed. These latencies are measured by the same evaluation scenario used in the previous section. Note that the average contention values and the average queue stall times resulting from the non-autonomic all-flash array are given in Table 2. It can be observed from Figures 10a and 10b that most of the overheads caused by the link contention can be dramatically reduced by Triple-A. In comparison, Triple-A reduces storage contention by only 15%. This is because Triple-A tries to reshape data-layout within a cluster first, and moves them to other clusters *only if* all the FIMM in clusters are classified as laggards. As one can see from Figure 10c, Triple-A can shorten the queue stall time by about 85% by using these minimum data movement strategy – we further analyze the stall times in our execution time breakdown study in Section 6.4.

6.3 The Details of Latency Improvements

The cumulative distribution functions for six workloads on the non-autonomic all-flash array and Triple-A given in Figure 11 indicate Triple-A's great capability to improve the I/O request latency originating from the resource contentions. The long-tailed distribution of these real workload latencies on the non-autonomic all-flash array illustrates the seriousness of the resource contention that takes place in all-flash arrays. The latency gap between the non-autonomic all-flash array and Triple-A indicates that our proposed approach copes well with the contentions and brings significant performance benefits. In addition to slight variations in the overall performance improvement, there are some interesting observations to remark. Compared to other workloads, the degree of improvement in *msnfs* is relatively low even though four hot clusters are caused by its access patterns. We see from Table 1 that the ratio of I/O requests heading to the hot clusters in *msnfs* is not very high. This means the clusters are detected as hot, but less hot than the (hot) clusters found in other workloads. In *msnfs*, we note that Triple-A's data migration and physical data-layout reshaping are less effective in the workloads with these less-hot clusters. In contrast, *proj* shows great performance improvement based on a number of relatively hotter clusters. In addition to the degree of hotness, the distribution of hot clusters is a crucial element in determining Triple-A's effectiveness on a given workload. The four hot clusters of *websql* are hot enough to shorten the latency more, while there is little improvement in IOPS. Since all the hot clusters are located in the same



(a) Baseline



(b) Triple-A

Figure 12: Hot cluster sensitivity: IOPS and latency of *read* micro-benchmark with varying number of hot clusters on a non-autonomic all-flash array (a) and Triple-A (b). Note that, as the number of hot clusters increases, Triple-A tends to show stable latency and improved IOPS.

switch, the imbalance of the number of I/O requests among PCI-E switches dictates the total execution time, resulting in low IOPS. Therefore, both the hotness and the distribution of hot clusters characterized by each workload affect Triple-A's performance.

Emerging HPC environments are likely to employ all-flash arrays as their main storage system. In this context, we pay attention to Triple-A's effectiveness on *g-eigen*, which is a version of the Eigensolver scientific application widely-used in HPC. This read-intensive workload results in many hot clusters, which in turn creates opportunities for Triple-A. As shown in Figure 11f, the magnitude of latency improvement brought by our approach is significant in this workload.

6.4 Sensitivity Analysis

Hot-cluster Size. Figure 12 plots the aggregate performance variation of the non-autonomic all-flash array and Triple-A under varying number of hot clusters. One can see from Figure 12a that increasing the number of hot clusters in the non-autonomic all-flash array worsens the average latency of the workload. Since I/O requests heading to the hot clusters compete with one another for the shared resources, more hot clusters we have, higher the ratio of I/O requests that suffer from the resource contention. In other words, the number of I/O requests benefiting from Triple-A increases as the number of hot clusters increases. As can be seen in Figure 12b, Triple-A achieves a stable latency with better IOPS under the increasing number of hot clusters. Therefore, we can expect that our technique would work well and improve

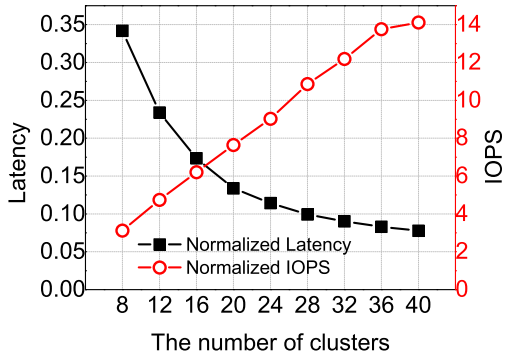


Figure 13: Network size sensitivity: IOPS and latency of *read* micro-benchmark normalized to a non-autonomic all-flash array under varing network sizes. As the network size increases, Triple-A shows better performance.

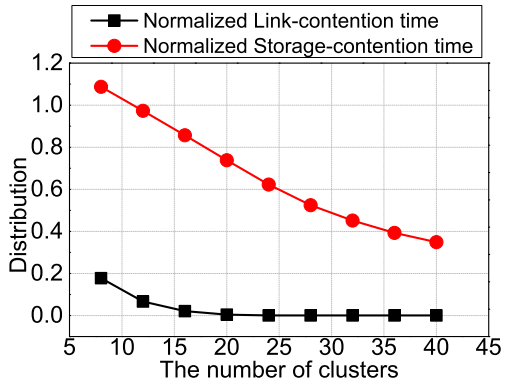
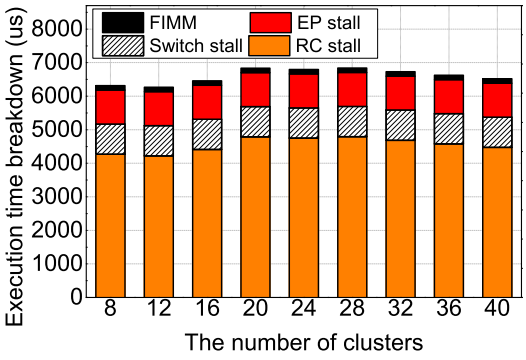


Figure 14: Two types of contention time of *read* micro-benchmark normalized to a non-autonomic all-flash array under varying network sizes. The big decrease in both contention times proves the effectiveness of Triple-A.

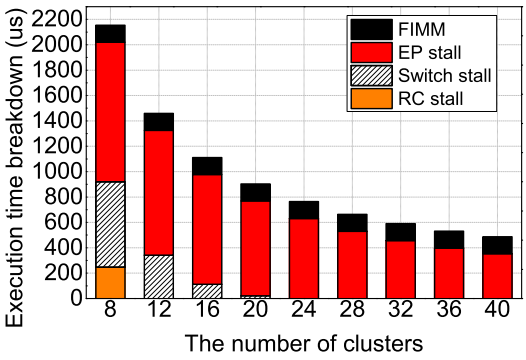
performance more in cases where the target application has an access pattern that results in a high number of hot clusters.

Network Size. Figure 13 shows that the performance (both IOPS and latency) of Triple-A gets better as the size of the all-flash network increases (the x-axis indicates the number of clusters under each PCI-E switch). Because Triple-A tries to redistribute the excessive I/O requests on the hot clusters to other (cold) clusters under the same switch, increasing the network size means employing more neighboring clusters in sharing the heavy load on the hot clusters. Theoretically speaking, the degree of the latency improvement would saturate when the cumulative I/O load is evenly distributed across a large number of clusters under the same switch.

To analyze the behavior of Triple-A better, we quantified the link contention time and storage contention time in the non-autonomic all-flash array. As shown in Figure 14, Triple-A successfully reduces both kinds of contentions by distributing the extra load of the hot clusters through data migration and physical data-layout mapping, as the number of clusters increases. In particular, the link contention time is almost completely eliminated while the storage contention time is steadily reduced as the number of clusters increases. This is because the storage contention time is bounded by



(a) Without Triple-A



(b) With Triple-A

Figure 15: Breakdown of the average execution time of *read* micro-benchmark on a nonautonomic all-flash array (a) and Triple-A (b) under varying network sizes. Various queue stall times representing the resource contentions are largely decreased by Triple-A.

the number of I/O requests to the target clusters, but the link contention time is not.

Execution Time Breakdown. The average execution time breakdown can show us how Triple-A actually optimizes I/O latency. Figures 15a and 15b plot the breakdown of the execution time on the non-autonomic all-flash array and Triple-A. respectively. We see that the significant reductions in both the link and storage contention times are the main contributors for improved latency. Excluding the FIMM throughput, which is not Triple-A's concern, stall times at the three different queues are gradually reduced as the network gets bigger. For instance, when using Triple-A, switch stalls and RC stalls are completely eliminated beyond a cluster size of 20.

6.5 Data Migration Overheads

Latency. Figure 16a gives the series of latency of *read* traces on the non-autonomic all-flash array while Figure 16d gives the same with Triple-A. As can be observed from these plots, Triple-A improves the performance by employing autonomic data migration, which migrates data from hot-clusters to cold-clusters. However, since the data migration activity shares the resources of the all-flash array with the normal I/O requests, it has an impact on overall performance. Therefore, it is worthwhile to consider the overheads our data migration strategy may bring.

Note that a single data migration involves reading the data from the hot-cluster, transmitting the data towards the switch, transmitting the data towards the target EP, and writing the data into the cluster. This overhead, particularly the data read from hot-clusters, can seriously hurt Triple-A's performance. Figure 16b shows the increased latency of I/O requests suffering from the interferece by data migrations. In order to reduce this overhead, shadow cloning described in Section 4.1 overlaps the original read I/O request and the data read as a part of data migration. The minimized latency overhead by shadow cloning is shown in Figure 16c.

Wear-out. Data migraion for high-performance flash arrays incurs extra writes, which reduces the lifetime of flashes by accelerating wear-out speed. According to our quantitative approximation, in the worst case, the data migration generates 34% additional writes, which leads to 23% decrease in lifetime and, in turn, makes the flash replacement earlier. However, the 50% cost reduction of our all-flash array architecture can offset the increased flash replacement frequency caused by the data migration. It should be noted that our Triple-A's architecture focuses on improving the system performance by sacrificing the flash lifetime. This a worthwhile trade-off since all-flash array significantly cut the cost by unboxing flash modules.

6.6 Effectiveness of DRAM Relocation

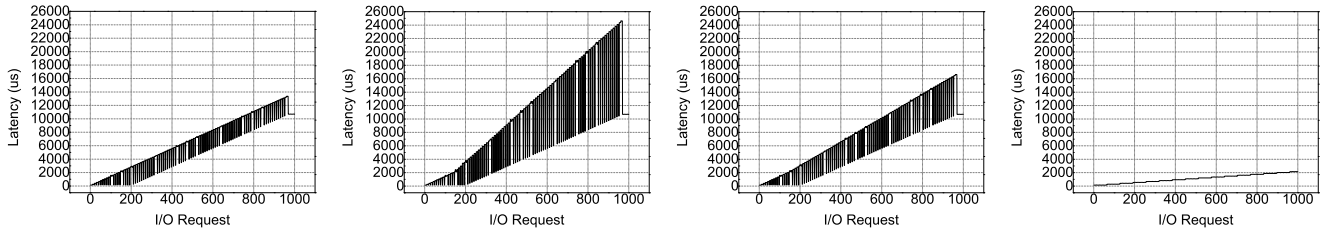
In Triple-A's architecture, all on-board DRAMs are removed; instead, large DRAMs are placed with the management module on top of all-flash arrays as described in Figure 5. The purpose of DRAM relocation is the cost reduction of flash arrays. Note that DRAM needs no replacement over time, whereas flash cells have limited lifetime due to the wear-out characteristic. We expect that this DRAM relocation does not degrade the system performance due to two reasons. Firstly, on-board DRAMs cannot resolve the link and storage contentions. Regardless of on-board DRAM's data caching, multiple successive I/O requests destined to the same SSD should share I/O bus and system-level data path, which is the main cause for contentions. (Instead, flash read/write latency can be eliminated when the requests make DRAM hit). In addition, the relocated and aggregated large DRAMs in Triple-A has the potential of reducing the number of I/O requests which can cause two types of contentions before they travel over the flash array network. We want to emphasize that the basic functionality of DRAMs like caching and alignment and its effects can be preserved since they are only relocated.

6.7 Wear Leveling

As shown in Figure 5, FTL functions including wear leveling are moved from individual SSDs to the central management module on top of the flash array network. Compared to conventional SSDs where FTL functions target only inside flashes, FTL functions of Triple-A can manage the flash arrays in a global fashion. Since the management module is able to centrally know both hotness information and erase counts of each cluster, there is certainly a chance to overlap garbage collection and data migration by considering wear leveling, which is left as a future work.

7. Related Work

Recently, all-flash arrays have been an active area of development in industry. Even though shipping all-flash arrays is



(a) Latency of non-autonomic all-flash array.

(b) Effectiveness of naive data migration.

(c) Effectiveness of data migration using shadow cloning.

(d) Improved latency of Triple-A.

Figure 16: Latency series of I/O requests under the non-autonomic array (a), the data migrations (b) and (c), and Triple-A (d). The difference between (b) and (c) is that latter employs shadow cloning to minimize migration overheads.

in an early-stage, some vendors (e.g., Pure Storage) already have their one-large pool storage system composed by 100% NAND flash based SSDs. One of the challenges that render the all-flash array difficult to widely deploy is the high price per GB of individual SSDs. To make the flash array more competitive compared to conventional disk arrays, many inline deduction techniques are being applied, which include deduplication [17, 54], compression [45, 45], and thin provisioning [32, 48], among others. These inline deduction techniques are expected to reduce the amount of data by 5x to 10x, which can in turn drive $/GB down.

In the meantime, academia also begins to explore the ways in which efficient SSD arrays can be built. [11] proposed QuickSAN, which is an SSD-based storage area network. In this approach, SSDs can communicate with one another to serve I/O requests as quickly as possible, which can in turn reduce system intervention and array-level software costs. Since QuickSAN integrates a network adopter into SSDs, it will increase the costs of individual storage components in an all-flash array approach. [4, 15] proposed a high performance SSD architecture built from emerging non-volatile memories. [12, 14] studied the benefits brought by NVM on high performance I/O intensive computing. In comparison, [5] proposed an SSD-optimized RAID system, which offers better reliability for individual SSDs than RAID-4 and RAID-5 by creating age disparities within arrays. [34] investigated the effectiveness of SSD-based RAID and discussed the potential benefits and drawbacks in terms of reliability. We believe that these SSD-RAID techniques (designed for an SSD appliance composed by small number of SSDs) could be extended to all-flash arrays as well.

Even though all these prior studies and industry efforts greatly contribute to bring all-flash array to the real world, there exists plenty of room for improvement. The inline deduction schemes are general techniques and can be applied to any storage devices (not just for all-flash array). Therefore, it is desirable to develop an approach that fundamentally reduces $/GB of all-flash arrays, which can employ all the inline modules as well. The architectural concept behind Triple-A can significantly reduce the maintenance costs in terms of hardware and software and drive down the prices of individual SSDs by building flash modules from scratch.

Very recently, a few vendors have already started to announce commercial products employing flash array similar to Triple-A's architecture. Texas Memory Systems's [49] is 2D flash-RAID with high availability, which is suitable for database, HPC, and cloud infrastructures. Violin Memory [50] also has a flash memory array architecture on the market. The cluster of 1000s of flash cells with multiple controllers provides high throughput, which is suitable for transactional business applications and big data analytics environments. Compared to these systems, however, our Triple-A brings further improvements on the performance of this architecture by dealing with I/O congestion problems caused by resource sharing in all-flash arrays.

8. Conclusions and Future Work

The primary goal of this paper was to address the link and storage contentions in emerging all-flash arrays. Our results showed that Triple-A can offer 53% higher sustained throughput and 80% lower latency with 50% less cost than non-autonomic SSD arrays. Our ongoing efforts include 1) developing different large-scale flash modules such as array-level garbage collection schedulers, wear-levelers, queuing mechanisms and flash translation algorithms, 2) expanding Triple-A to a reconfigurable network-based all-flash array, and 3) developing different dynamic I/O routing mechanisms for all-flash arrays. We also plan integrate our PCI-E I/O simulation framework into other flash simulation models such as DiskSim+SSD extension [3], NANDFlashSim [26] and FlashSim [28].

Acknowledgments

This research is supported in part by University of Texas at Dallas Start-up Grants, NSF grants 1302557, 1017882, 0937949, and 0833126 as well as DOE grant DE-SC0002156, and DESC0002156. It also used resources of the National Energy Research Scientific Computing Center supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] http://www.dramexchange.com/.

[2] SNIA IOTTA repository. URL `http://iotta.snia.org/tracetypes/3`.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[4] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: a protoype phase change memory storage array. *HotStorage*, 2011.

[5] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *Trans. Storage*, 2010.

[6] K. Bates and B. McNutt. Umass Trace Repository. URL `traces.cs.umass.edu/index.php/Main/Traces`.

[7] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading hpc i/o performance using a scientific-application derived benchmark. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[8] R. Budruk, D. Anderson, and E. Solari. *PCI Express System Architecture*. Pearson Education, 2003. ISBN 0321156307.

[9] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 2010.

[10] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 2011.

[11] A. M. Caulfield and S. Swanson. Annual international symposium on computer architecture (isca). In *QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks*, 2013.

[12] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[13] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[14] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proceedings of SC*, 2010.

[15] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO*, 2010.

[16] S. Choudhuri and T. Givargis. Deterministic service guarantees for NAND flash using partial block cleaning. In *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis*, 2008.

[17] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *USENIX ATC*, 2010.

[18] EMC. Vnx5500-f unified storage flash array.

[19] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[20] J. He, J. Bennett, and A. Snavely. DASH-IO: An empirical study of flash-based IO for HPC. *TeraGrid' 10*, August 2010.

[21] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *ISC*, 2011.

[22] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed SSDs. In *USENIX HotStorage*, 2012.

[23] M. Jung and J. Yoo. Scheduling garbage collection opportunistically to reduce worst-case I/O performance in solid state disks. In *Proceedings of the International Workshop on Software Support for Portable Storage*, 2009.

[24] M. Jung, R. Prabhakar, and M. Kandemir. Taking garbage collection overheads off the critical path in ssds. In *Middleware*, 2012.

[25] M. Jung, E. Willson, and M. Kandemir. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *ISCA*, 2012.

[26] M. Jung, E. H. Wilson, D. Donofrio, J. Shalf, and M. Kandemir. NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *IEEE Conference on Massive Data Storage*, 2012.

[27] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *International Symposium on Computer Architecture*, 2008.

[28] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. Flashsim: A simulator for NAND flash-based solid-state drives. In *SIMUL*, 2009.

[29] S. W. Lee, W. K. Choi, and D. J. Park. FAST: An efficient flash translation layer for flash memory. In *EUC Workshops Lecture Notes in Computer Science*, 2006.

[30] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *IEEE Conference on Massive Data Storage*, 2012.

[31] Y. Liu, J. Huang, C. Xie, and Q. Cao. Raf: A random access first cache management to improve SSD-based disk cache. *NAS*, 2010.

[32] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. scc: cluster storage provisioning informed by application characteristics and slas. *FAST'12*, 2012.

[33] N. Master, M. Andrews, J. Hick, S. Canon, and N. Wright. Performance analysis of commodity and enterprise class flash devices. In *PDSW*, 2010.

[34] S. Moon and A. L. N. Reddy. Hotstorage). In *Dont Let RAID Raid the Lifetime of Your SSD Array*, 2013.

[35] E. H. Nam, B. Kim, H. Eom, and S.-L. Min. Ozone (O3): An out-of-order flash memory controller architecture. *Computers, IEEE Transactions on*, 2011.

[36] NERSC. In *http://www.nersc.gov/users/computational-systems/carver/*.

[37] NetApp. Netapp EF540 flash array.

[38] ONFI Working Group. Open nand flash interface 3.0. In *http://onfi.org/*, 2012.

[39] Y. Ou, T. Härder, and P. Jin. Cfdc: a flash-aware replacement policy for database buffer management. In *the Fifth International Workshop on Data Management on New Hardware*, 2009.

[40] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing checkpoint performance with staging I/O and SSD. In *International Workshop on Storage Network Architecture and Parallel I/O*, 2010.

[41] C. Park, W. Cheon, Y. Lee, W. C. Myoung-Soo Jung, and H. Yoon. A re-configurable FTL architecture for NAND flash based applications. In *IEEE/IFIP International Workshop on Rapid Prototyping*, 2007.

[42] C. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee. Exploiting internal parallelism of flash-based ssds. In *Computer Architecture Letters*, page 9, January 2010.

[43] PCI-SIG. PCI express base 3.0 specification. 2012.

[44] PLX Technology. PLX 8796 PLX expresslane gen 3 PCI Express compliant switches.

[45] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference*, 2004.

[46] Pure Storage. Fa-300 series technical specifications.

[47] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[48] P. Shivam, A. Demberel, P. Gunda, D. Irwin, L. Grit, A. Yumerefendi, S. Babu, and J. Chase. Automated and on-demand provisioning of virtual machines for database applications. In *ACM SIGMOD international conference on Management of data*, 2007.

[49] Texas Memory Systems. Texas memory systems RamSan-820.

[50] Violin Memory. Violin memory 6000 Series Flash Memory Arrays.

[51] D. Watts and M. Bachmaier. *Implementing an IBM System x iDataPlex Solution*. Redbooks, 2012.

[52] O. C. Workgroup". Open nand flash interface specification: NAND connector.

[53] Z. Zhou, E. Saule, H. Aktulga, C. Yang, E. Ng, P. Maris, J. Vary, and U. Catalyurek. An out-of-core dataflow middleware to reduce the cost of large scale iterative solvers. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, 2012.

[54] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.