Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks

Myoungsoo Jung, Ellis H. Wilson III, Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
{mj, ellis, kandemir}@cse.psu.edu

Abstract

NAND flash storage has proven to be a competitive alternative to traditional disk for its properties of high random-access speeds, low-power and its presumed efficacy for random-reads. Ironically, we demonstrate that when packaged in SSD format, there arise many barriers to reaching full parallelism in reads, resulting in random writes outperforming them. Motivated by this, we propose Physically Addressed Queuing (PAQ), a request scheduler that avoids resource contention resultant from shared SSD resources. PAQ makes the following major contributions: First, it exposes the physical addresses of requests to the scheduler. Second, I/O clumping is utilized to select groups of operations that can be simultaneously executed without major resource conflict. Third, inter-request NAND transaction packing empowers multi-plane-mode operations. We implement PAQ in a cycle-accurate simulator and demonstrate bandwidth and IOPS improvements greater than 62% and latency decreases as much as 41.6% for random reads, without degrading performance of other access types.

1. Introduction

NAND flash-based devices such as Solid State Disks (SSDs) are becoming increasingly popular in a number of markets. Flash has already become the dominant storage technology in mobile devices for its low-power, density, and resilience to shock. Moreover, flash-based SSDs are making significant inroads into consumer computers such as laptops as well as enterprise applications such as Storage-Area-Networks (SANs) and even high performance computing (HPC). Their lack of moving parts – perhaps the biggest problem with magnetic disk – allows them to serve random requests at a far higher rate than disks. However, care needs to be used when writing to these devices, as flash memory cells wear out with overuse. Therefore, write-heavy workloads are not well-suited for these devices. For such reasons, enterprise and HPC areas have strongly considered SSDs for workloads rife with reads, especially for applications that demonstrate mainly random access patterns.

Such use-cases make sense when considering individual flash memory cells, which are biased towards reading, showing typically between ten and forty times better performance for reads than writes. This is due to a significant duration disparity for the three basic operations in flash: read, write, and erase. Perhaps more importantly, this disparity is exacerbated by the requirement that if a block to be written is not already free, an erase must precede the write. Specifically, while reads operate on the tens of microseconds, a write takes hundreds of microseconds, and an erase requires thousands of microseconds. Therefore, if a write occurs to an occupied block, an erase latency plus the write latency is incurred. This problem becomes worse for writing to a subset of a block such that existing data therein will be kept. These situations require all three operations: first perform a read, then an erase, and finally write the block down with a mix of new and old data. For the latter two – erase/write and read/erase/write – the latency of writing to an SSD approaches that of spinning disk.

Due to this vast disparity in latencies, internal research on SSDs (mostly on the flash translation layer such as in [2]) has been concentrated on avoiding the costs of doing such writes. However, since flash cannot yet match the density-to-cost ratio spinning disk excels at, external research considering the *use* of SSDs rather than the *development* of them has focused on their use as a cache to alleviate the I/O-latency gap between memory and disk [20][5][16]. As a cache, researchers capitalize on the presumed efficacy in serving random reads when compared to spinning disk, and often the caching algorithms employed avoid performing large numbers of writes in order to extend the SSDs lifetime and to avoid the high penalties of writes. These two divergent research directions – internal research working to improve writes and external research developing mechanisms to capitalize on improved performance for reads versus writes – have resulted in a poor research landscape.

Ironically, when one examines the *random-read* versus *random-write* performance for SSDs, what is witnessed is often a performance benefit for random writes instead of random reads, due to difficulty experienced in achieving parallelism for reads, but ease in doing so for writes. This particularly strange *reversal of expected performance* tends to be glossed over or completely ignored in many prior stud-

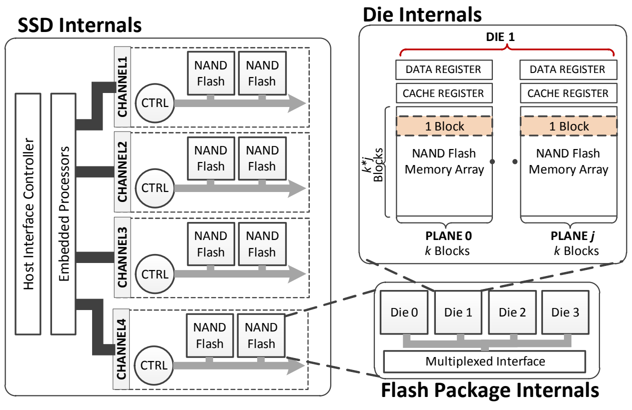


Figure 1. Physical internal architecture of SSD.

ies. However, it is a critical issue that deserves attention especially for enterprise applications seeking to utilize these devices as caches for their efficacy for random-reads.

1.1. Contributions

First, we identify areas where resource contention may exist that cause degraded random-read performance. We then develop and present a new request scheduling algorithm that maximizes performance by avoiding internal resource conflicts. We propose *Physically Addressed Queuing (PAQ)*, a request scheduler that is in part inspired by physical address-based memory scheduling techniques [11, 17, 18, 21, 23] and improves random-read performance by avoiding conflicts for I/O requests. To our knowledge, there is no published work that proposes a scheduler that utilizes physical addresses to optimize performance for SSDs. We summarize our major contributions below:

• *QBM Relocation:* In order to identify and avoid contention for shared SSD resources, we propose to move the queue and buffer management (QBM) functionalities, typically located within the host interface logic, beneath the flash translation layer (FTL). This exposes physical addresses to PAQ, instead of relegating it to work solely with logical block addresses (LBAs).

• *I/O Clumping:* We classify conflicts into groups based on the physical SSD component(s) they share, which provides a framework we use when performing conflict avoidance and improving parallelism within the SSD.

• *I/O Request Rescheduling:* With the ability to identify a request's physical addresses and a framework to "clump" groups of sub-requests together that do not conflict, we present our new queuing algorithm, PAQ, that reschedules requests such that conflicts are avoided.

• *Plane Packing:* The last level of parallelism within the SSD – plane-level parallelism (see Section 2.1) – requires a number of constraints to be satisfied. We show that, given access to the physical addresses of a request's accesses, PAQ can identify more accesses between requests that can benefit from multi-plane mode to improve parallelism.

Using our modified SSD architecture and our proposed PAQ algorithm, *we seek to demonstrate greatly improved read latencies for random accesses.* Our experimental analyses indicate improvements over traditional scheduling in bandwidth, IOPS, and average latency. Specifically, for bandwidth, we see as high as 62.7% and in the average case 32.6% improvements. For IOPS, PAQ demonstrates as high as 62.6% and in the average case 32.7% improvement. And finally, we witness as high as 41.6% and in the average case 25.1% improvement in latency. Further, in all cases tested, PAQ results in performances at least as good as those for traditional scheduling.

2. Background

Flash storage presents the first serious departure from the magnetic storage that were studied for decades. With it come a host of new characteristics and nuances; considerably more than were present in rotational magnetic disk. While many prior studies have already explored the finer details of flash characteristics, it is critical to at least have a basic background on the medium to appreciate the benefits of our proposed Physically Addressed Queuing. To that end, in the following subsections we present a cursory overview of NAND flash architecture and details of one particularly high-potential but underutilized access mode available in SSDs.

2.1. High-Level Architecture of SSD

A subset of the physical internals relevant to our work are shown in Figure 1, and represent typical hardware used in commercial SSDs [22, 2, 10]. There are four main levels that *increase parallelism* in an SSD:

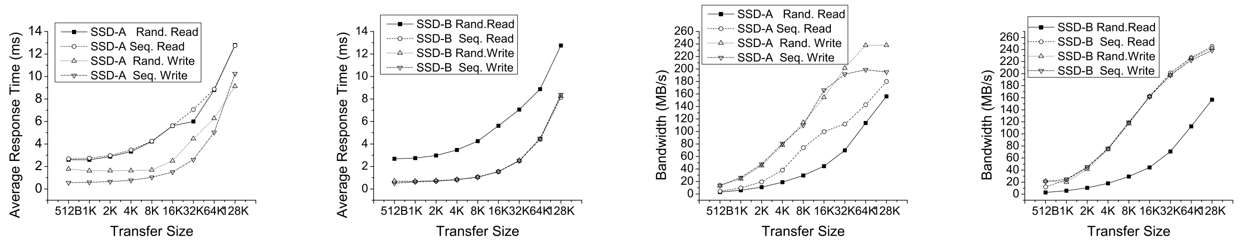
• *Channels.* At the highest level, there exist multiple channels that are operated by embedded processors; each can be operated in a completely independent fashion.

• *Flash Packages.* Each channel is shared by flash packages for transmitting data and operation messages.

• *Dies.* Within each NAND flash package are one or more dies, each sharing one or more buses upon which their communication is interlaced via a chip enable (CE) pin. This leads to a reduction in I/O bus complexity but also adding potential for contention over the CE pin by requests.

• *Planes.* Finally, within each die exist one or more planes, the smallest unit to serve an I/O request in a parallel fashion. Each plane shares a wordline for accessing the flash memory cells, which leads to the important consequence that multiple requests can be served simultaneously in a single wordline access. However, in order to do so, the requests must adhere to the *plane-addressing rule*, a constraint we expand upon in the following section.

A subset of the relevant software layers are shown in Figure 3, and are described below:



(a) Latency (SSD A) (b) Latency (SSD B) (c) Bandwidth (SSD A) (d) Bandwidth (SSD B)

Figure 2. Average response times and bandwidth results as transfer sizes varies under two commercial SSDs.

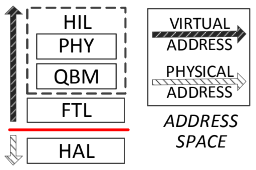


Figure 3. Software stack of an SSD.

• *Host Interface Layer (HIL)*: The HIL is responsible for communication between the host system and underlying layers within the SSD. Specifically, it performs parsing of I/O commands, hand-shaking based on the interface protocols, initiating data transfer, and committing NAND transactions to underlying layers. The raw communication protocol portion of the responsibilities are handled by the Physical layer (PHY), whereas responsibilities related to I/O scheduling and buffering are dealt with by the Queue and Buffer Management layer (QBM).

• *Flash Translation Layer (FTL)*: The FTL is responsible for address translation between the host address space, which contains virtual or logical addresses, and the physical address space, which fully specifies the channel, flash package, and flash die where the data is located.

• *Hardware Abstraction Layer (HAL)*: The HAL is a device driver, which manages physical NAND flash memory. It is charged with committing NAND transactions to the underlying flash memory, and periodically checks each flash package to monitor transaction statuses.

2.2. Multi-Plane Mode Operation

Besides the three basic NAND operations we mention earlier – read, write and erase – there exist a number of advanced modes and operations that seek to improve NAND parallelism but come with constraints that must be adhered to in order to achieve such performance. *Multi-plane mode* , operations serve multiple requests simultaneously by sending them together down the same wordline. This mode has the potential to improve performance n times the number of planes attached to a particular wordline, but comes with the caveat that these requests *must* target the exact same page offset in a block, the exact same die address, and indicate different plane addresses.

3. Random Writes vs. Random Reads

While the objective to simplify SSD design is achieved via resource sharing, what is not readily expressible through the physical diagram is the potential for *conflicts* between requests that end up contending for such a shared resource. While contention is low and parallelism is high for writes due to data layout independence, such parallelism is not as easily achieved for reads. Specifically, in order for a read to occur, there must be some data to be read, and that data must reside in a particular block.This results in a very rigid performance dependence upon data layout and access sequence, leading to queues of requests sprouting up, which strangles parallelism. *In addition, the QBM, in its current position above the FTL, is helpless to do anything about such performance dependence because it solely has knowledge of the virtual address – there is no way for it to intelligently reorder accesses to decrease conflicts.*

To validate our concerns regarding random read performance, we performed tests on two commercial Samsung 470 series SSDs manufactured in 2010, which we hereafter refer to simply as “SSD A” and “SSD B”. In these tests, we used the IOmeter characterization tool [13] to perform sequential writes, sequential reads, random writes and random reads, each for nine separate transfer sizes ranging from 512 bytes to 128 kilobytes for both drives, and allow at most 16 outstanding I/Os to exist at any given time. The amount of data moved by the tool fills about 80% of both SSDs.

Both SSD A and SSD B utilize 32 nm fabrication process NAND flash memory packages, employ a DDR2 flash interface (144Mbs), and employ Samsung’s second generation S3C29MAX01 controller. ARM-based multi-core processors with dual-cache chips manage the SSD internals, and the host interface is connected via Serial ATA Generation 2 (3Gbps). The size of the DRAM cache employed in both devices is 256MB, comprised of two DRAM chips having 667Mbps data rate. The first drive, marked “SSD A,” has a capacity of 64GB and operates using 4 channels and 16 packages. The second drive, marked "SSD B," has a capacity of 256GB and operates using 8 channels and 64 packages. While manufacturers do not publicize exactly how many dies are in each package, single-, dual-, quad-, and octal-die package are all possibilities in production. We suggest that for these devices dual-die package is most likely based on the price point and the date manufactured. The results of our tests are shown in Figure 2. Specifically, Figures 2(a) and 2(b) plot variances in latency as transfer size is increased for SSD A and SSD B respectively, and Figures 2(c) and 2(d) show variance as transfer size increases in terms of overall bandwidth.

In the latency results, for SSD A, random reads and sequential reads perform similarly, but both are still far worse than either types of the writes by at least 25% and as high as 361%. For SSD B, there exists a clear case that while all other operations incur approximately the same latencies, random reads suffer by at least 56% to at most 319% in comparison. The bandwidth results tell a very similar story, but bring to light the increasing gap as transfer size increases. In all cases tested, random writes outperform random reads, in direct contrast to what a majority of the literature on flash memory would lead one to believe. It should be noted that, read operations tested are not affected by any transactions related to garbage collection activities because read and write tests for both latency and throughput are performed in an entirely separate fashion. Also, note that the fraction of DRAM that might be used for buffering I/Os is about 0.48% (SSD A) and 0.12% (SSD B) in our tests. In practice, the fraction of DRAM for buffering I/Os is even smaller because it is concurrently used for maintaining metadata and in-memory data structures of the FTL.

4. Physically Addressed Queuing

In order to improve the poor random-read performance we demonstrated above, we propose a new request scheduling scheme named *Physically Addressed Queueing (PAQ)*. Unlike previous schedulers who do not have access to the physical addresses of requests, with PAQ we propose moving the QBM layer out from the HIL and beneath the FTL to provide such crucial functionality. With exposure of the physical layout to our PAQ scheduler, we are able to positively identify requests that will cause conflicts as they concurrently contend for the same resources. Using these physical addresses, we present a classification system for conflicts, and describe how PAQ can aggregate groups of requests together that do not share conflicts. Such groups of requests without interdependence we term a *clump*[1], and we show that PAQ works to build clumps in a conflict-first bottom-up fashion such that total contention is reduced and SSD performance is improved. Last, we discuss how PAQ can improve multi-plane mode performance immensely given the physical layout of requests, which is an optimization that can orthogonally improve overall performance.

[1]While related, clumping differs from memory request coalescing in that clumping does not combine multiple requests into one, nor does it remove duplicate requests [9]. The main purpose of clumping is to avoid conflicts between requests.

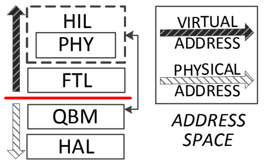


Figure 4. Firmware layers within a SSD.

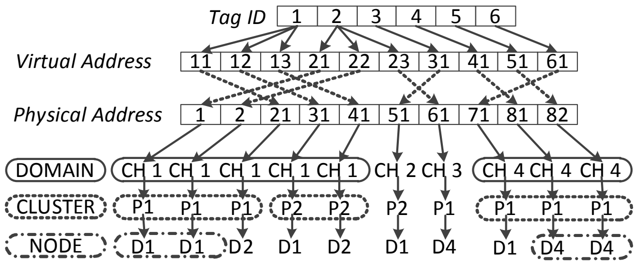


Figure 5. An example that shows PAQ conflict classification. CH, P and D denote channel, package, and die, respectively.

4.1. QBM Migration

Since the QBM layer must be exposed to physical addresses of requests in order to make intelligent decisions that decrease conflict, we propose migrating the QBM out of the HIL and positioning it directly beneath the FTL. This migration is visually depicted in Figure 4, where the horizontal line separates the virtual and physical address spaces.

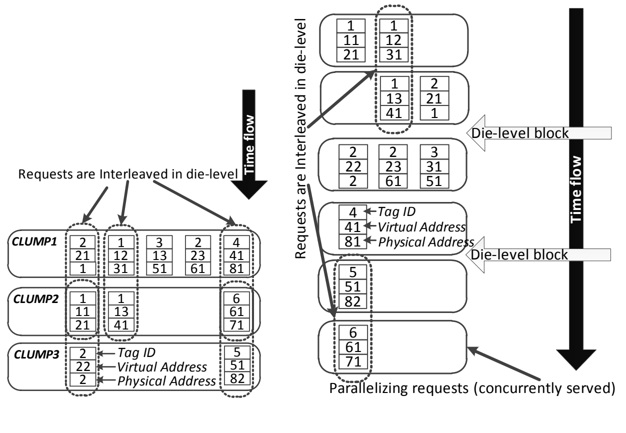
This change is achievable in practice because the PHY and QBM layers operate within distinct protocols. Specifically, in the SATA interface, while the PHY layer resides in the physical, link and transport layers defined in the SATA protocol, the QBM resides in the application layer, allowing for migration of the QBM without necessitating changes to existing manufacturing processes or established protocols.

4.2. Conflict Classification

To build a queuing mechanism that can identify and properly schedule around contention, we first propose the following conflict classification framework:

• *Domain:* a set of requests that require access to the same channel. Concurrently scheduling requests in a domain increases the potential for channel I/O bus contention, but also has the potential to exploit parallelism by interleaving requests across multiple flash packages and dies.

• *Cluster:* a set of requests requiring access to the same flash package. A cluster may contend for the same NAND I/O bus in accessing a package, which incurs domain-level


Requests are Interleaved in die-level
Time flow
CLUMP1
CLUMP2
CLUMP3
Tag ID
Virtual Address
Physical Address
Requests are Interleaved in die-level
Die-level block
Die-level block
Parallelizing requests (concurrently served)

(a) PAQ scheduling (b) VAQ scheduling

Figure 6. An example that compares PAQ scheduling to traditional VAQ scheduling.

conflicts. However, the requests in a cluster enjoy die-level interleaving parallelism if no node-level conflict exists.

• *Node:* a set of requests that require access to the same flash die. A node always incurs resource conflicts, and it also has the potential for a number of resource contentions among the requests in same domain and cluster, including NAND flash register, NAND I/O bus, and channel I/O bus.

The fact that each level incurs conflicts at its level and *all levels above it* is a critical part of conflict classification. To visualize this inclusive hierarchy, Figure 5 provides an example of requests in the command queue, along with the virtual and physical addresses of the transactions within those requests. Beneath the transactions, we show what channels, packages and dies the transactions require access to. We also demarcate potential conflicts using rounded rectangles surrounding all transaction targets that are the same physical component. Domains are indicated using a solid-rounded rectangle, clusters using a dotted-rounded rectangle, and nodes using a dot-dashed-rounded rectangle. The resulting set of clumps PAQ constructs is shown in 6(a).

In the simplest case, for request ID #3, it requires access to LBA 31, physical address 51, channel 2, package 2 and die 1. As it is the only request in the queue seeking to access channel 2 (no node-, cluster- or domain-level conflicts exist), it is free from conflict and may be executed immediately. However, if we look at a more complicated scenario resultant from request ID #2, we see that it is attempting to access physical addresses that exist along two channels, two packages, and two dies. This request causes domain- and cluster-level conflicts with request ID #1, and also results in domain-, cluster- and node-level conflicts with other transactions in the same request. Therefore, even for a command queue with solely a single request, there may exist conflicts between multiple transactions in that request. While conflicts occur for accesses to the same channel, package, or die, it is important to note that the window of conflict is wider for lower levels due to a pipelining of each transaction through the architecture, resulting in a slightly more complex landscape for conflicts.

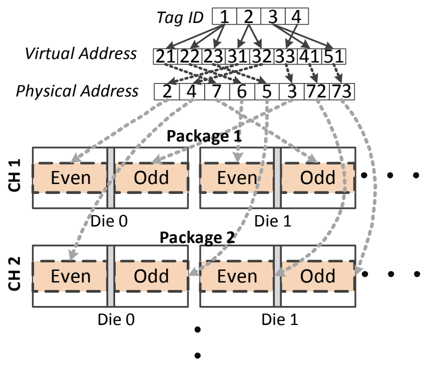


Figure 7. Plane packing in PAQ.

4.3. Clump Composition

With the physical addresses available to the QBM layer, and a classification scheme that allows us to identify conflicts and their location, we construct our PAQ scheduling algorithm around the following two intuitions: (i) lower-level conflicts are most costly – PAQ should avoid them if possible, and (ii) if PAQ can schedule a single transaction from an area of conflict while doing other work and only later execute the other contentious operation, it can avoid contention while achieving parallelism. These intuitions lead us to establish the following goals on how PAQ should perform clump composition, the most critical contribution of our work: (Goal #1) Add transactions incurring conflicts in the lowest levels first. (Goal #2) For node- and cluster-level conflicts, never schedule a clump such that either would have greater than one transaction issued concurrently from it. (Goal #3) Continue adding transactions to the clump, prioritizing for low-level conflicts, until no more can be added without breaking Goal #2. Succinctly, *PAQ attempts to build clumps in a bottom-up, conflict-first fashion such that the lowest level with contention does not have conflicting transactions in the clump.*

To properly compare our PAQ clumping strategy, let us observe how requests in Figure 5 would be handled by the *default* (traditional) scheduling scheme, which we name *Virtually Addressed Queuing* (VAQ), when compared with how PAQ handles the exact same situation (depicted in Figure 6(a)). Due to lack of space, we do not go over the example presented in Figure 6 in detail. In short, VAQ handles requests in a FIFO nature because it has no better option without access to the physical addresses, resulting in six separate request sets being issued, of which only two have the opportunity to execute in an interleaved fashion. PAQ, on the other hand, is able to view the physical addresses and therefore optimizes its submissions to solely three separate request sets, all of which can be interleaved.

4.4. Plane Packing

In theory, multi-plane mode operation should allow SSDs to achieve nx speedup (where n is the number of planes in a NAND flash), since n pages can be served simultaneously. However, such a speedup is generally not reached because traditional VAQ is ignorant of physical addresses, which is a prerequisite to intelligently constructing multi-plane-mode operations. Further, since traditionally the underlying FTL is oblivious of the device-level queue and requests therein, it is not possible for the FTL and HIL to collaborate to construct multi-plane mode requests. For example, in Figure 7, even though a VAQ scheduler would reorder commands in an attempt to satisfy the plane-addressing rule, the order of transactions associated with the reordered commands is still not sufficient to build multi-plane-mode operations; without knowledge of the physical addresses, it is purely luck for the FTL to be able to execute multi-plane mode transactions.

4.5. Implementation of PAQ Scheduling

```
head_lpn := tag.lsn % the size of page
tail_lpn := (tag.lsn + tag.length) % the size of page
/* get physical address info and record */
while head_lpn != tail_lpn do
    if tag.req_type = read then
        ppn := ftl.translate_physical_address(head_lpn)
        pair(ch_id, flash_id) := ftl.parse_channel_and_way(ppn)
        pair(die_id, plane_addr) := ftl.parse_die_and_plane(ppn)
        nand_trans := build_trans(ppn)
        add(nand_trans, clump_table[ch_id][flash_id])
        device_queue.push_back(tag)
        send_ack(tag)
    else
        device_queue.push_back(tag)
        send_ack(tag)
        ftl.page_basis_commit(head_lpn)
    head_lpn += 1
```

Algorithm 1: adding_io_request(tag). Note that address translation for read requests occurs before the actual data transfer begins.

Algorithms 1 and 2 describe how our approach is implemented and manages PHY and QBM, respectively, of the HIL. First, for the PHY management, address translation occurs between the time pre-information (called a *tag*, which includes information like logical block address and request size) is received, and the time an acknowledgment is sent (if the request type is read). For requests that are not a read, the PHY sends the acknowledgment first and bypasses the request to the underlying FTL. The reason behind these two different strategies is that the addresses of read are decided at write time, for quick translation. In contrast, in the write case, the decision of the physical target has not yet been determined by the FTL.

Once the PHY translates the physical address for the read request, it adds information along with the physical address into a table, called the *clump table*. It should be noted that

```
foreach ch_id := 0 ... n do
    foreach flash_id := 0 ... m do
        /* build clump */
        prev_tag := check_and_wait(ch_id, flash_id)
        if prev_tag.committed_trans = 0 then
            device_queue.release(prev_tag)
        trans := get_trans_from_node(tag, clump_table[ch_id][flash_id])
        if trans is not assigned then
            trans := get_trans_from_cluster(tag,
            clump_table[ch_id][flash_id])
        if trans is not assigned then
            trans := get_trans_from_domain(tag,
            clump_table[ch_id][flash_id])
        if trans is not assigned then
            pair(tag, trans) :=
            get_another_req(clump_table[ch_id][flash_id])
        /* packing */
        assoc_trans := get_associate_plane_trans(tag,
        clump_table[ch_id][flash_id])
        trans := packing(trans, assoc_trans)
        hal.commit(ch_id, flash_id, trans)
        tag.committed_trans += 1
```

Algorithm 2: read_data_transfer(tag). Serving I/O request and managing QBM.

the latency incurred from the computation overhead in address translation can be hidden by overlapping it with the process of receiving the tag and sending an acknowledgment. When the device-level queue is not empty, from the front of the queue, the QBM commits NAND transactions to the underlying HAL by visiting each entry of the clump table associated with the target tag and attempting to identify and issue requests conflicting in the lowest level possible first. In the case that there are no requests found having node- or cluster- or domain-level conflicts, the QBM will simply issue any transaction headed for the currently selected NAND flash by identifying one in the clump table. Once the QBM commits the transaction, it moves on to perform the same process for the next NAND flash and its associated entries.

5. Experimental Setup

5.1 NAND Flash SSD Simulator

To implement and evaluate PAQ, we required a high-fidelity simulator that was capable of capturing cycle-level interactions between the many components in an SSD. While there exist a few well-known SSD simulators such as Microsoft Research Lab's SSD extension [2] to DiskSim [4] and FlashSim [19], neither of these nor most others deliver the high-fidelity results we require. Consequently, we developed a cycle-accurate NAND flash simulator [2] that provides: (i) Fine-grained NAND command handling, so that conflicts between competing commands are made evident (ii) Advanced command implementation with maintenance

[2]The source code of this simulator [14] can be downloaded from http://www.cse.psu.edu/∼mqj5086/nfs.

	Write I/Os	Read I/Os	Percent random-write	Percent random-read
fin1	4,099,354	1,235,633	97.86	96.98
fin2	653,082	3,046,112	99.2	97.49
web1	212	1,055,236	99.06	93.53
web2	990	4,578,819	100	93.33
web3	1,260	4,260,449	99.84	91.25
usr1	1,333,406	904,483	94.23	92.2
usr2	3,857,714	41,426,266	96.24	88.97
usr3	1,994,612	8,575,434	97.02	82.77
prn1	4,983,406	602,480	76.4	88.6
prn2	2,769,610	8,463,801	97.16	90.5
sql1	1,423,458	606,487	93.5	89.91
sql2	73,833	87,058	16.07	73.66
sql3	38,963	5,136,405	92.95	71.96
sql4	21,330	10,050	46.89	86.95
msnfs1	1,467,625	41,772	87.23	99.79
msnfs2	2,100,032	121,697	66.71	88.8
msnfs3	500	24	0	99
msnfs4	4,014	338	22.52	64.79
msnfs5	3,003,205	9,624,191	97.86	96.98
msnfs6	3,040,098	9,941,612	100	97.51

Table 1. Trace decomposition into the number of writes and reads, and the percentage of random-reads and random-writes issued.

of strong constraints (i.e., to properly evaluate our multi-plane mode optimization) (iii) Awareness of intrinsic latency variations for diverse NAND I/O operations based on the current state of the memory cells. In addition, we built a simulation framework that performs all of the high-level tasks of an SSD, which builds and issues requests to concurrent instances of the NAND flash simulator. This provides us with a cycle-accurate SSD simulator.

5.2. SSD Configuration and Schedulers Tested

In this work, we configure our simulation environment as having 8 channels, 8 flash packages per channel (64 total), dual-die package format (128 total) and a queue size of 32, which is the standard for SATA-based SSDs, with a page-level mapping FTL similar to the one employed in [2, 7]. We believe this represents the typical modern SATA SSD, but we experiment with varying queue and channel counts in the sensitivity section to provide further insight into how our scheme would behave under varying protocols and future SSDs. We evaluate the following queuing strategies:

- **VAQ**: The default queuing scheme.
- **PAQ0**: PAQ, only using plane-packing.
- **PAQ1**: PAQ, only using clumping.
- **PAQ2**: PAQ, using both plane-packing and clumping.

5.3. Traces

We wanted to validate performance across a number of traces from *actual enterprise applications*. To that end, we collected traces of workloads representative of the following enterprise areas (with the corresponding abbreviation we use afterwards in parentheses): online transaction processing (fin), search engines (web), shared home directories (usr), print serving (prn), relational database management systems (sql), and remote file storage servers (msnfs). There exist multiple traces with the same prefix but varying numeric extensions; some of these are traces of different points in the lifetime of the application, and others are from distinct applications that happen to fall into the same category. These traces are available at [3] and [1], and the latter was originally detailed and presented in [24]. In order to give a better insight on the high-level nature of the traces and the overall landscape of our trace selection, we have characterized our traces, as shown in Table 1.

6. Experimental Results

In evaluating PAQ we quantify its impact on overall performance relative to VAQ by measuring total bandwidth, I/Os per second (IOPS), and average latency for all aforementioned traces. To connect those performance improvements to our goal of improving parallelism and utilization, we also measure contention time and idle time.Finally, we demonstrate low-level impacts of PAQ through a graph of individual request latencies and perform a sensitivity analysis along the axes of queue size and channel count.

6.1. Aggregate Performance: Bandwidth, IOPs, and Latency

As can be seen in Figure 8, PAQ improves bandwidth for read-intensive workloads immensely; five of the twenty workloads exceeding a 100MB/s improvement. Furthermore, for all of the web workloads, the improvement achieved is greater than 100MB/s and *such workloads are comprised of greater than 90% of random reads* (see Table 1).

Lastly, PAQ2 never hurts the performance for any workload, regardless of whether it is read- or write-oriented or has mostly random or sequential requests. PAQ0 occasionally hurts performance because, with a default SATA queue size of 32, there is a limited window of requests with which to consider packing. The IOPS measurements shown in Figure 9 tell the other side of our story; those traces that exhibit particularly low bandwidth were generally dominated by much larger numbers of requests whom had smaller sizes than those with high bandwidth (a good example is fin2). Just as with bandwidth, for random-read intensive workloads, PAQ2 does a great job improving performance and never performs worse than VAQ. However, for our worst performing trace, msnfs4, the IOPS are so few that they appear to be missing from the figure. In that case, despite Table 1 describing it as mainly issuing sequential-writes, we find that these sequential writes are intermixed with very small random-read requests.

This write-performance degradation occurs for all schedulers as a result of the great disparity between read and write latency and these random reads undermining the par-

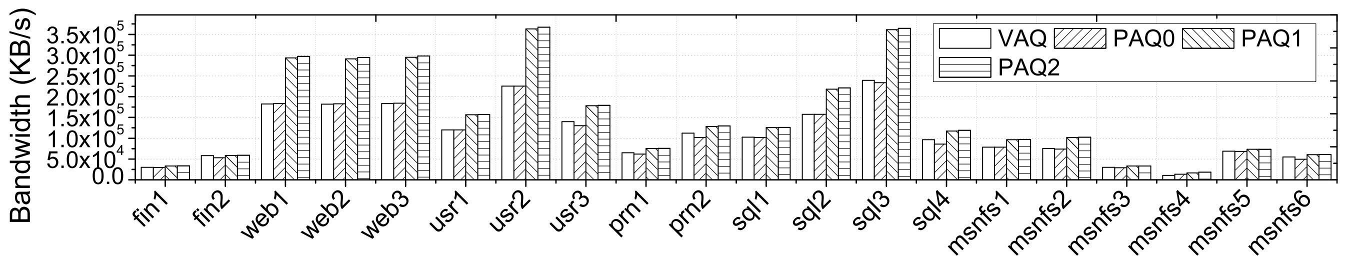


Figure 8. Average bandwidth comparison between VAQ and PAQ.

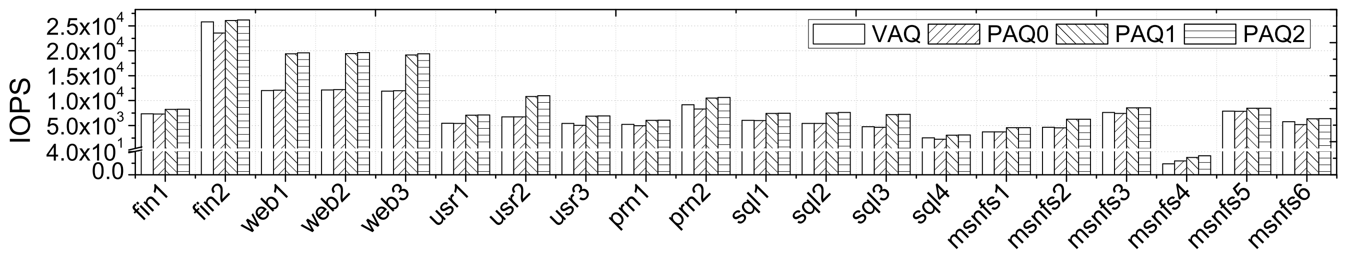


Figure 9. Average I/Os per second (IOPS) comparison between VAQ and PAQ.

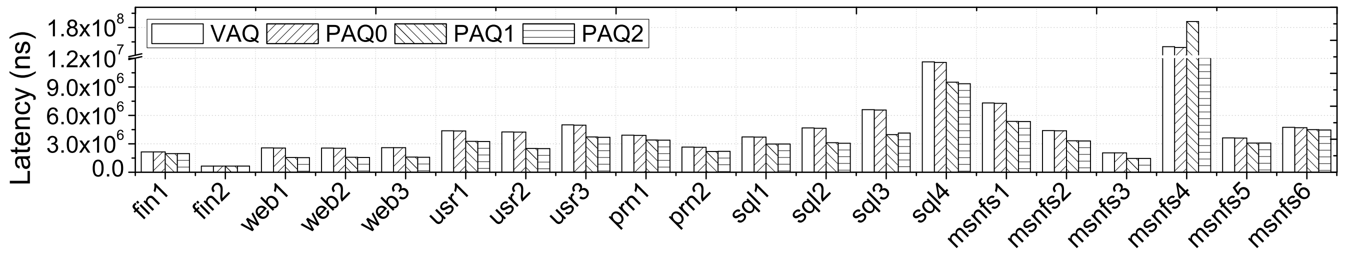


Figure 10. Average latency comparison between VAQ and PAQ.

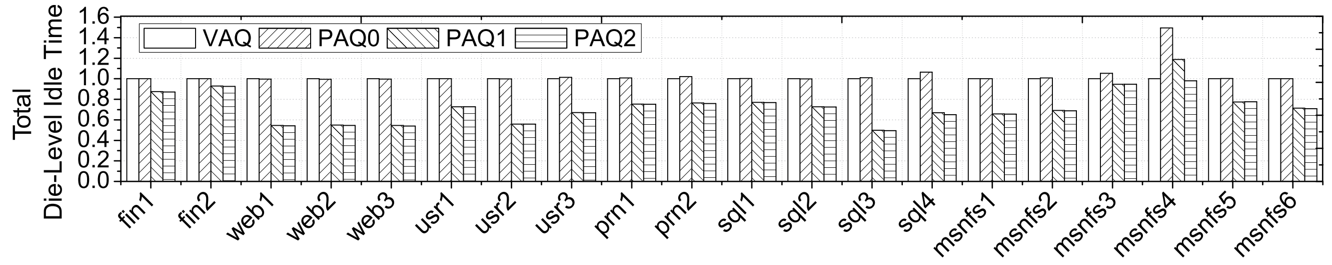


Figure 11. Idle times for VAQ and PAQ.

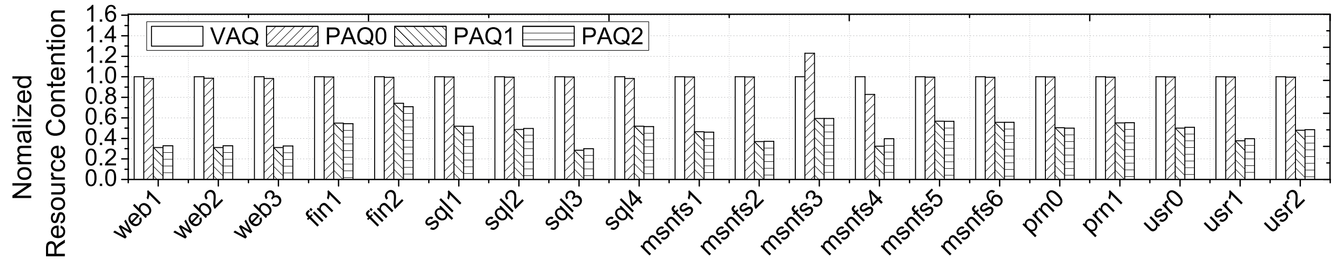
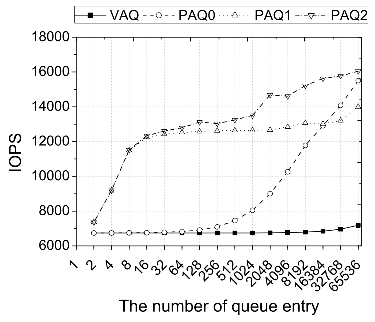
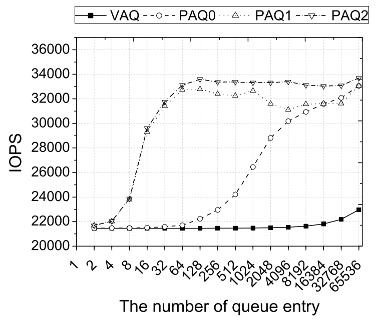


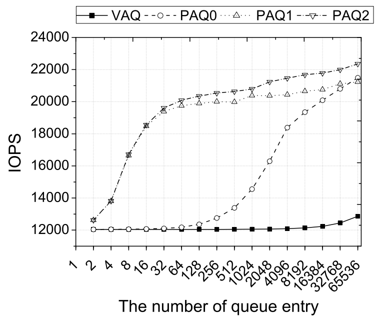
Figure 12. Normalized total contention time comparison between VAQ and PAQ.



(a) 4 Channels

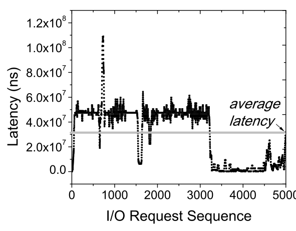


(b) 8 Channels

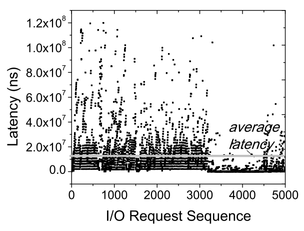


(c) 16 Channels

Figure 14. IOPs sensitivity to varying queue and channel sizes.



(a) Latency (VAQ)



(b) Latency (PAQ)

Figure 13. All latencies incurred for requests from trace sql3 for VAQ and PAQ. Note that average latencies are shown using a horizontal line in each case.

allelism of the sequential-write requests. Even in the face of such random-read interference, PAQ delivers 1.41 times the IOPS and bandwidth of VAQ.

6.2. Quantifying Parallelism: Idle Time and Contention

While raw throughput and total I/Os completed per second are important metrics, it is still critical that individual transactions from operations are not delayed so long that average latency increases a great deal. Interestingly, what we find as shown in Figure 10 is that, *in the average case*, PAQ2 actually decreases latency fairly significantly. Moreover, PAQ2 improves performance in a similar proportion *across nearly all workloads, regardless of their access type breakdown*. Last, we see that while in bandwidth and IOPS PAQ1 and PAQ2 performed very similarly, in the case of msnfs4, using just plane-packing or clumping alone did not improve performance as much as together, giving credence to our belief that both are needed for best performance.

We originally conjectured that the poor performance we observed in real SSDs in Section 3 for random reads was a direct result of difficulty achieving parallelism in the device. While we have demonstrated aggregate performance improvements using PAQ2, we further seek to directly demonstrate that these improvements were correlated to increasing utilization of the individual dies (thereby reducing idle time) and reducing total contention time. Idle time is measured as the total time each die spent without serving any transaction, and is shown again for each trace in Figure 11. It has been normalized to 1 so that all improvements can be clearly seen. Decreases in total idle time for PAQ2 around 20% are witnessed on average, with a few read-oriented traces reaching improvements as high as approximately 60% and a few write-oriented traces experiencing almost no improvements. Contention time is somewhat harder to measure than idle time, but we formalize it as *the amount of time a NAND transaction spends waiting on the I/O bus within a flash package to get to a specific die.* As the results in Figure 12 reflect, contention time does not directly reflect idle time. While PAQ2 does not always result in a great reduction of idle time, it does result in a very significant reduction in contention time across *all* traces.

6.3. Overheads of PAQ

As PAQ involves advanced scheduling to improve performance, it is worthwhile to consider overheads scheduling may cause. The process of checking the clump table described in Section 4.5 is theoretically bounded by O($n * m$) for each read transfer, where n is channel count (state-of-the-art is 4$\sim$16) and m is flash package count (state-of-the-art is 4$\sim$16). This results in a search space of approximately a few thousand choices, which is inexpensive to iterate through. Our estimations show this overhead to be approximately 1% (on an SSD with 72MHz microprocessor and 64 dual-die flash packages), which does not affect our conclusions. Specifically, a read operation takes 180 μs (including NAND flash I/O bus activities), and we estimate iterating through the search space to take 1.77 μs. Furthermore, modern SSDs are using multi-core processors, which are often underutilized and could be employed for queue management with even less impact on performance.

6.4. Time Series: In-Depth Analysis of a Database Trace

Next, we examine the differences in latencies between VAQ and PAQ2 in greater detail since that was the area of most uncertainty. For this, we focus on the sql3 trace, and present latencies incurred for its first 5000 I/O requests (about the first 10% of its execution) in Figure 13 for both VAQ and PAQ. It can be observed that, while VAQ demonstrates very consistent but moderately high latencies, PAQ is able to decrease latencies a great deal for a majority of the requests. However, there do exist some requests PAQ has delayed due to their conflicting nature; in a small subset of cases their response times exceed twice the latency of VAQ. We believe this characteristic is a natural side-effect of PAQ's performance-enhancing optimizations, but suggest that a balance can be struck and a bound on such spikes set by giving priority to those requests approaching a defined Quality-of-Service (QoS) threshold. Such examination is deferred for future work.

6.5. Sensitivity Analysis

We wanted to expose how PAQ's performance might be impacted by varying current protocols and flash devices such as flash connected via not only our examined SATA protocol, but also SAS, PCI-Express and other emerging connecting protocols. Therefore, we performed an IOPS and waiting time sensitivity analysis for VAQ and our PAQ varieties on varying channel counts and queue sizes, two parameters we believe will fluctuate the most in upcoming devices and between different protocols respectively.

While the IOPS sensitivity analyses shown in Figure 14 demonstrates a predictable increase in IOPS as the numbers of components increases and as queue size increases, there are two less obvious take-aways: First, VAQ shows almost no improvements in performance as queue size increases, whereas all versions of PAQ demonstrate significant gains. Since VAQ does not utilize the queue for anything but FIFO storage, whereas PAQ utilizes the entire queue in performing its optimizations, this observation is expected. Second, while we saw little evidence that plane-packing by itself in the earlier results gave benefits, as we increase queue size in this analysis we see rapidly increasing performance for greater queue sizes. Such improvements are a result of an increased space for the optimization to search through and select packable blocks from.

Figure 15 illustrates an *Average Waiting Time* (AWT) sensitivity test. AWT is the average time a request waits for a spot in the device-level queue, when that queue is full. Interestingly, in contrast to the IOPS sensitivity test, as we increase the number of queue entries, AWT of PAQ0 slightly increases, worse than VAQ at some points. This phenomenon occurs as a result of a delay experienced prior to PAQ0 taking advantage of multi-plane-mode operations. Even though multi-plane-mode operations can boost system throughput 1.53 times when compared to VAQ, it requires data movement between the HIL and the target flash, which preempts channel I/O bus while transferring data and therefore imposes a delay. PAQ2 also employs the plane packing scheme, but, AWT of PAQ2 is better than VAQ's. This is because the benefits of I/O clumping covers relatively long channel I/O bus time of plane packing. As a result, PAQ2 improves both throughput and AWT.

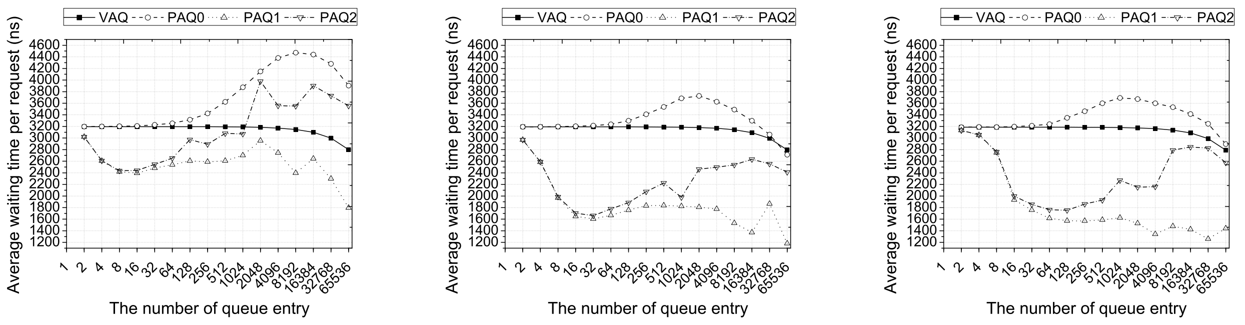
7. Related Work

In [28], the authors attempt to uncover a specific resource contention that occurs in SSDs, the areas where parallelism is far below optimal, and present a dynamic request rescheduling scheme to improve performance. However, in their work they do not take into account the fact that the addresses which correspond to the requests they "reschedule" are virtual. That is, while they claim that their scheme increases parallelism by avoiding conflict and capitalizing on multi-plane mode in flash dies, they actually have no knowledge of where those addresses point to since physical addresses are not available except beneath the FTL.

[28] actually is one paper in a larger body of works that have attempted to improve the read performance of SSDs without grappling with the key issue we point out in our work: the disparity between virtual and physical addresses. Another work suffering from a similar oversimplification is [7], where the researchers observe suboptimal access trends in their traces and claim improvements are possible via the coalescing of read requests. The problem here is again that these traces are of LBAs, not of the physical addresses, and therefore coalescing may or may not end up improving performance internally within the SSD. Any improvements observed are likely the result of a clean-room testing environment leading to artificial alignment of LBAs and physical addresses; in a real installation performance improvements may or may not result from such coalescing.

It is also critical to emphasize major differences between flash-based SSDs and other flash mediums. As we discussed, SSDs are subjected to the very distinct nuances of operating through traditional storage interfaces such as SATA and SAS. Because of this, proposals aimed at optimizing flash-based storage connected via PCI buses or otherwise operating in a byte-addressable manner will likely not work when such optimizations are applied to flash-based SSDs. Therefore, while works such as [6] may appear to tackle a similar problem to what we target, assumptions they make, such as "the memory technology has performance... to that of DRAM and that it presents a DRAM-like interface," pour a foundation that is entirely distinct from the base of assumptions we work from to improve flash-based SSDs.

In [8], the authors demonstrate that modern SSD performance is less dependent upon access patterns than interferences between and within accesses and how the data is physically laid out on the dies. They also demonstrate that



(a) 4 Channels (b) 8 Channels (c) 16 Channels

Figure 15. Waiting time sensitivity to varying queue and channel sizes.

write accesses are faster than for reads in some cases and are far less dependent on access pattern and data layout than reads are. These realizations serve as an important foundation for our work. Using our PAQ scheme, we are able to decrease the impacts of the two major performance limiting factors they identify: interference and data layout.

In [15] and [10], the authors recognize the need to exploit parallelism in flash-based SSDs and propose three techniques for doing so over multiple independent channels: striping, interleaving and pipelining. Other, similar strategies have been presented to increase the parallelism of accesses over multiple NAND-flash packages, such as ganging [2], superpaging [10], and multi-plane mode. While these schemes are capable of achieving significant performance improvements for higher parallelism than more serial alternatives, it is important to recognize that all such strategies only achieve parallelism for certain types accesses. As such, while they work well for writes, since these may go anywhere and therefore achieving parallelism is relatively easy, these schemes often fail to improve the performance for reads since many reads do not perform accesses to strictly sequential physical addresses. We present and quantify the shortcomings of these advanced operations in our VAQ results, which utilizes 8 channels (superpage-based striping), 8 flash packages-per-channel (ganging), and 2 dies-per-package utilizing interleaved-die and multi-plane-mode operations, and yet fails to achieve peak performance. Therefore, the novelty of our work is not akin to the previously discussed strategies, which provide complex mechanisms to improve performance *only for certain access patterns*. PAQ's novelty lies mainly in reordering and reorganizing the accesses to enable mechanisms such as striping and ganging to perform their best.

While scheduling using physical addresses is novel in the context of SSDs, DRAM controllers already employ physical address based scheduling. Static as well as hardware-assisted dynamic memory access reordering strategies [11, 21], and various other DRAM scheduling algorithms [12, 27] have been investigated in order to maximize DRAM bandwidth. Zuravleff and Robinson [29] also proposed a DRAM controller that improves data throughput in DRAM by reordering requests coming to a memory controller without changing the actual outcomes. Examples of QoS-aware controllers are fair-queuing memory systems [25], stall-time fair queuing [23], and start-time fair queuing [26]. These controllers were designed to be fair to applications that share a limited memory bandwidth and provide QoS guarantees if needed. With the increase in on-chip memory controllers, the idea of coordinated control of memory channels emerged. In [17], a method that achieves fairness by keeping track of attained service information for applications running simultaneously is proposed. By prioritizing threads with the least attained service, a fair memory scheduling scheme is obtained. In [18], fairness and throughput are considered simultaneously. The proposed thread cluster memory scheduling scheme isolates latency sensitive and bandwidth sensitive applications.

These DRAM scheduling works are motivated by parallel memory architecture, which has similarities to modern SSDs. However, there are three important differences: First, unlike DRAM, SSD schedulers must adhere to idiosyncrasies of NAND flash such as erase-before-write, endurance, asymmetric I/O speeds, and diverse NAND flash command protocols. Second, SSDs are connected through thin interfaces, which introduce more limitations in I/O scheduling than DRAM controllers, including different levels of queue management, I/O handshaking, host-device data movement for various I/O lengths, and I/O completion protocol. Lastly, an SSD scheduler should recognize underlying flash firmware features such as page- or block-level address remapping and garbage collection. These *differences in characteristics, interface nuances, and the responsibilities schedulers carry out* drive research on scheduling mechanisms in each domain in separate directions.

8. Conclusion

As NAND flash-based storage devices, such as SSDs, become increasingly considered as caching mediums for their ability to serve random-reads at a high rate, extracting full performance for such workloads is only possible if barriers to parallelism are overcome. Our presented scheme to improve random-read performance, PAQ, demonstrates the need for the queuing scheduler to have access to physical addresses of requests, and we discuss how moving the QBM beneath the FTL would achieve such. Further, we present a conflict classification methodology, I/O clumping, which PAQ utilizes to effect efficient and low-contention I/O request scheduling. Lastly, PAQ uses knowledge of the physical addresses of I/O requests to better enable multi-plane mode commands via our *plane-packing* optimization. In this work, we implemented PAQ in a cycle-accurate SSD simulator and evaluated its performance on a diverse set of traces taken from actual enterprise workloads. Our extensive experiments demonstrate bandwidth and IOPS improvements exceeding 62% and decreases in latency as far as 41.6% on random reads when compared to the traditional queue scheduler, without slowing writes or sequential accesses.

9. Acknowledgements

We thank our shepherd, Dr. Kai Li, for his help and careful revisions in improving our paper. We also thank anonymous reviewers for their constructive feedback. This work is supported in part by NSF grants 1017882, 0937949, and 0833126 and DOE grant DE-SC0002156.

References

[1] SNIA IOTTA repository.

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

[3] Bates, K., and McNutt, B. Umass Trace Repository.

[4] Bucy, J. S., Schindler, J., Schlosser, S. W., and Ganger, G. R. The disksim simulation environment version 4.0 reference manual.

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

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

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

[8] Chen, F., Lee, R., and Zhang, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA* (2011).

[9] Davidson, J. W., and Jinturkar, S. Memory access coalescing: a technique for eliminating redundant memory accesses. *SIGPLAN Not.* (1994).

[10] Dirik, C., and Jacob, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *ISCA* (2009).

[11] Hong, S. I., McKee, S. A., Salinas, M. H., Klenke, R. H., Aylor, J. H., and Wulf, W. A. Access order and effective bandwidth for streams on a direct rambus memory. *HPCA* (1999).

[12] Hur, I., and Lin, C. Adaptive history-based memory schedulers for modern processors.

[13] Intel. http://www.iometer.org/.

[14] Jung, M., Wilson, E. H., Donofrio, D., Shalf, J., and Kandemir, M. NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *MSST* (2012).

[15] Kang, J.-U., Kim, J.-S., Park, C., Park, H., and Lee, J. A multi-channel architecture for high-performance NAND flash-based storage system.

[16] Kgil, T., Roberts, D., and Mudge, T. Improving NAND flash based disk caches. In *ISCA* (2008).

[17] Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA* (2010).

[18] Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO* (2010).

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

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

[21] Mckee, S., and Wulf, W. Access ordering and memory-conscious cache utilization. *HPCA* (1995).

[22] Micron, Inc. MT29F8G08MAAWC.

[23] Mutlu, O., and Moscibroda, T. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO* (2007).

[24] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys* (2009).

[25] Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. Fair queuing memory systems. In *MICRO* (2006).

[26] Rafique, N., Lim, W.-T., and Thottethodi, M. Effective management of DRAM bandwidth in multicore processors. In *PACT* (2007).

[27] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. Memory access scheduling. In *ISCA* (2000).

[28] yeong Park, S., Seo, E., Shin, J.-Y., Maeng, S., and Lee, J. Exploiting internal parallelism of flash-based ssds. *IEEE CAL.* (2010).

[29] Zuravleff, W. K., and Robinson, T. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. *U.S. Patent No: 5,630,096* (1997).