# Exploiting Intra-Request Slack to Improve SSD Performance

Nima Elyasi[†] Mohammad Arjomand[†] Anand Sivasubramaniam[†]
Mahmut T. Kandemir[†] Chita R. Das[†] Myoungsoo Jung[‡]

[†]School of Electrical Engineering and Computer Science, Pennsylvania State University, USA
[‡]School of Integrated Technology, Yonsei University, South Korea
{nxe125,mxa51}@psu.edu {anand,kandemir,das}@cse.psu.edu m.jung@yonsei.ac.kr

## Abstract

With Solid State Disks (SSDs) offering high degrees of parallelism, SSD controllers place data and direct requests to exploit the maximum offered hardware parallelism. In the quest to maximize parallelism and utilization, sub-requests of a request that are directed to different flash chips by the scheduler can experience differential wait times since their individual queues are not coordinated and load balanced at all times. Since the macro request is considered complete only when its last sub-request completes, some of its sub-requests that complete earlier have to necessarily wait for this last sub-request. This paper opens the door to a new class of schedulers to leverage such *slack* between sub-requests in order to improve response times. Specifically, the paper presents the design and implementation of a slack-enabled re-ordering scheduler, called *Slacker*, for sub-requests issued to each flash chip. Layered under a modern SSD request scheduler, Slacker estimates the slack of each incoming sub-request to a flash chip and allows them to jump ahead of existing sub-requests with sufficient slack so as to not detrimentally impact their response times. Slacker is simple to implement and imposes only marginal additions to the hardware. Using a spectrum of 21 workloads with diverse read-write characteristics, we show that Slacker provides as much as 19.5%, 13% and 14.5% improvement in response times, with average improvements of 12%, 6.5% and 8.5%, for write-intensive, read-intensive and read-write balanced workloads, respectively.

***CCS Concepts*** • **Hardware** → **External storage**; *Non-volatile memory*

***Keywords*** SSD, Scheduling, Intra-Request Slack



## 1. Introduction

NAND-flash based Solid-State Disks (SSDs) are gaining rapid acceptance as a disk supplement or even a replacement in enterprise applications. They provide substantially lower latency and higher throughput than conventional disks [8], with a continually dropping price per gigabyte to make them increasingly attractive. At the same time, there is a continuing strive to boost SSD performance for the demanding storage needs of the evolving big data applications. Faster SSD hardware incorporating high speed processing logic, low latency storage cells and faster interconnects have leveraged technological advances over the years to provide substantial performance improvements. Another complementary performance-enhancing architectural technique is to incorporate and leverage parallelism in the hardware (multiple flash chips each with multiple dies and planes, multiple channels, etc.) to achieve high degrees of parallelism within and across read/write requests. There has been considerable prior work [13, 15, 18, 19, 25, 31] on scheduling requests in different layers (host software, device and channel levels) to leverage the parallelism offered by such hardware. However, even with these sophisticated schedulers to exploit hardware parallelism, SSD requests can experience considerable inefficiencies. The parallelism could, in fact, accentuate these inefficiencies. One important inefficiency arises from the fact that requests spanning multiple chips (each termed a *sub-request*[1]) necessarily need to wait for the last sub-request to complete, even if one or more sub-requests get serviced early. Such skews between sub-request completion times open the door to a new class of schedulers which can leverage the slack of existing sub-requests, allowing new arrivals to jump ahead without affecting the response times of those already present. This paper presents one such scheduler, *Slacker*, which estimates slack for sub-requests when each request arrives, and leverages this slack to significantly reduce response times.

[1] A (macro) request spans several pages, is translated into several sub-requests, each of which is directed at a flash chip. A Read queue and a Write queue are maintained for each chip to service such sub-requests independently.

Today's SSDs offer multiple layers of hardware parallelism. Within each flash chip, there are multiple dies and planes for parallelism. There are several such chips on each channel that are connected to a flash controller, with several channels themselves that could be independently operated. Schedulers order incoming Read and Write requests to take advantage of such offered parallelism. Numerous such schedulers have been developed over the years, that could be implemented at the host which sends the requests [15, 23, 31], or the device which assigns these requests to different channels [18, 20, 25], and even within the channel where requests are sent to individual chips.

If each Read or Write request spans enough pages to exactly match the offered hardware parallelism, then the scheduler's job is relatively simple since the hardware utilization/ parallelism is automatically maximized regardless of the order in which the requests are serviced. However, two small requests, may not be serviceable together if they intersect on some flash chips. One of them has to wait for its sub-requests on those chips to complete (which can be started only after the other request completes), even if its other sub-requests (to other flash chips) complete earlier (these are referred to as having *slack* in this paper). Further, not all operations take the same latency, especially when comparing writes versus reads where the former can be 10–40 times the latter, exacerbating the problem. As a consequence, we will show that even modern schedulers [25] can result in a highly unbalanced system with chip-level queues exhibiting very high variance, leading to considerable slack between sub-requests of a macro request.

One solution to dealing with the slack problem is to try to mitigate the slack itself. Reducing slack of read requests requires finding time slots where the requisite flash chips are all free, which is akin to gang scheduling the tasks of a parallel program on the processors of a parallel machine [26]. Posing such a restriction on the scheduler can lead to hardware under-utilization due to fragmentation, as is well known in the gang scheduling context [11, 12, 17]. Instead, most current SSD schedulers opportunistically use whatever time slots are available, thereby potentially creating skews/slack between sub-requests that get earlier time slots on some flash chips with the other sub-requests of that macro request getting scheduled later at their respective chips. Writes, unlike reads, offer better slack mitigation opportunities since writes could be opportunistically re-directed to whatever flash chips are free at that instant (since there is no Write-in-Place in flash). Prior research has proposed techniques [9, 14, 27, 29] for such re-direction. Apart from requiring considerable storage overhead for remapping tables, as we will show, these enhancements address only writes, and there is still plenty of slack amongst read requests (which are also usually in the more critical path).

Another solution is to take advantage of whatever slack is present, and re-order incoming (sub)-requests to jump ahead of existing (sub)-requests that have slack. This can give lower response times to incoming requests which can move ahead, without impacting the response times of the ones already in the system. This rationale constitutes the basis of our *Slack-Enabled Reordering (Slacker)* scheduler that is presented in this paper. Though intuitive, there are a number of practical considerations – estimating the slack, figuring out which requests to bypass, developing an elegant implementation that is not very complex (since the high level problem is NP-hard) and avoiding extensive communication across the hardware components – which need thorough investigation. This paper specifically makes the following **contributions**:

- We introduce the notion of slack between sub-requests of macro requests that are directed at different flash chips of an SSD. Across a diverse number of storage workloads, we show that considerable slack exists across the read and write sub-requests that can get as high as several milliseconds.
- The success of a slack-aware scheduler would very much depend on estimating the slack of sub-requests accurately. We present an online methodology for estimating the slack that is based on queue lengths, the type (write/read) of requests, and contention for different hardware resources. We show that our methodology yields fairly accurate slack estimates across these diverse workloads – within 5% for 16 workloads, within 10% for 4 workloads, and exceeds 20% in only 1 workload. Even in the very few cases, where the errors are slightly higher, Slacker still provides response time benefits. We also identify the hardware counters (which are relatively easy to provide) needed for estimating the slack.
- Recognizing the hardness of the problem, we propose a simple heuristic to implement a slack-aware scheduler. Rather than a coordinated global re-arrangement of the distributed queues for each flash chip, which can result in a lot of communication/overheads, our heuristic takes a sub-request at each queue independently and figures out whether to jump ahead of existing ones based on their slack. The resulting complexity is just linear in the number of queued sub-requests for each chip.
- High write latency (relative to reads) accentuate slack since response times of waiting requests can get skewed further based on writes that are ongoing. To mitigate this problem, we adapt Write Pausing [30], to pre-empt ongoing writes if they have sufficient slack to accommodate newer requests.
- We implement Slacker in SSDSim, together with state-of-the-art scheduler (e.g., O3 scheduler [25]) enhancements that take advantage of SSD advanced commands. We show that request response time is improved by 12%, 6.5% and

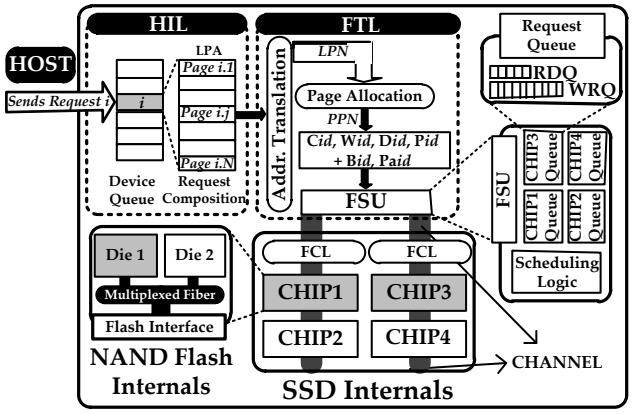



Figure 1: A generic architecture of modern SSDs.

8.5% on the average for write-intensive, read-intensive, and read-write balanced workloads (with improvements up to 19.5%, 13% and 14.5%), respectively.

## 2. Background

Modern SSDs [5, 10, 21] have several NAND flash chips and control units interconnected by I/O channels (Figure 1). Each channel is shared by a set of flash chips and is controlled by an independent controller. Each chip, internally, consists of several dies and a number of planes within each die.

### 2.1 Flow of a Request within the SSD

When the host sends an I/O request, the host interface picks it up and inserts it into a device queue for processing. Since the macro request may span several pages, the host interface parses each request into several page-sized transactions, each with a specific Logical Page Address (LPA). In NAND flash, writes are never done in place. Consequently, mapping tables need to be maintained to keep track of the current location of a page, that is referred to as a Logical Page Number (LPN). LPN is then allocated to flash chips at a specific Physical Page Number (PPN). Translating an LPN to a PPN is accomplished in 2 steps: (i) determining the plane where the block/page resides amongst the numerous choices, and (ii) the eventual physical location of the block/page within that plane. While a single table to accomplish both steps would allow a complete (and possibly dynamic) write-redirection mechanism with full flexibility for placing any page at any location across the flash chips, this takes additional space. Instead, commercial SSDs use a static mapping scheme to determine the chip, die and plane of each LPN, which can be accomplished by simple modulo calculations (instead of maintaining mapping tables). Once the plane is determined by this mechanism, a Flash Translation Layer (FTL) maps the page to a PPN within that plane using a table (a page-level mapping table here again offers maximum flexibility at the overhead of extra space).

Slacker is built on top of a static mapping mechanism for the first step to avoid the additional space overheads. There are several ways of striping LPNs across the channels, chips, dies and planes of the SSD, based on the relative ordering of the dimensions for such striping. Of the different alternatives, the ordering of Channel-first, Chip-second, Die-third and Plane-fourth (CWDP), has been shown to perform the best across a wide range of workloads [28], and is the mechanism for the first-level mapping in Slacker as well. The normal FTL, with a page-level translation table to reach the eventual physical page on the plane, is used for the second step.

After address translation, the FTL Scheduling Unit (FSU) which is part of the FTL firmware, resolves resource contention and schedules requests for maximizing parallelism across the different hardware entities [18, 25]). The O3 scheduler [25], which has been shown to maximize such parallelism, has been used as the baseline in this paper. FSU subsequently uses the well-known *First Read-First Come First Served (FR-FCFS) with conditional queue size* [20] to order the sub-requests at each chip. FR-FCFS is designed to lower the impact of high write latency on reads (write latency is 10–40 times higher than read latency). To do so, the scheduler maintains a separate *Read Queue* (RDQ) and *Write Queue* (WRQ) for each chip, where WRQ is much larger than RDQ. FR-FCFS prioritizes commands in the following order:

1. *Read-first:* Read requests are prioritized over writes unless WRQ is $> \alpha\%$ full, in which case a write is prioritized.
2. *Oldest-first:* Within each queue, FCFS order is preserved.

Slacker builds on top of FR-FCFS to allow some requests to bypass others in the same RDQ/WRQ queues based on slack. Beneath FSU, there is Flash Controller Logic (FCL) which is a mediator between the FTL and the flash chips. FCL serves requests while obeying timing of the flash chips and channels.

## 3. Preamble to Slacker

This section introduces the concept of slack, its origin, and how it can be exploited for performance benefits.

### 3.1 What Is Intra-Request Slack?

An I/O request's size varies from a few bytes to KBs (or MBs). When a request arrives at the SSD, the core breaks it into multiple sub-requests and distributes them over multiple flash chips so that they can get serviced in parallel. Since service of a request is not complete until all its sub-requests are serviced, the sub-request serviced last, called *critical sub-request*, determines the request's eventual response time. For each sub-request, we define *slack time* as the difference between the end time of its service and the end time for the critical sub-request in the same request. Essentially, the slack of

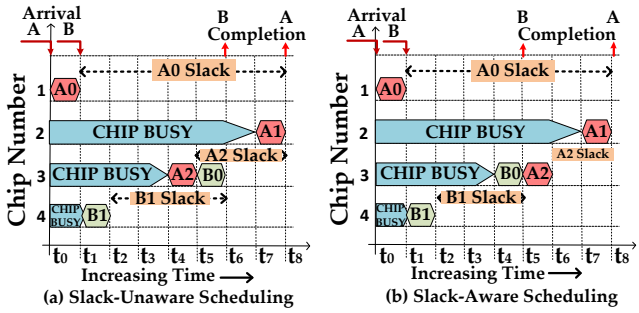



Figure 2: Example of (a) a slack-unaware and (b) a slack-aware scheduler (B completes earlier without impacting A).

a sub-request indicates the latency it could tolerate without increasing the overall latency of the corresponding request. Figure 2.(a) gives an example. The SSD consists of four chips, CHIP1, CHIP2, CHIP3 and CHIP4, where CHIP1 is currently idle while the other three are servicing requests of 7, 4 and 1 sub-requests respectively. Let us assume that servicing a sub-request takes 1 cycle. At time $t_0$, Request $A$ arrives that has sub-requests $A0$, $A1$ and $A2$ each that are in turn mapped to CHIP-1, CHIP-2 and CHIP-3, respectively. With FR-FCFS scheduling, even though $A0$ can get serviced right away, $A1$ and $A2$ have to wait 7 and 4 more cycles, respectively, to get their turns. As a result, $A0$, $A1$ and $A2$ have slacks of 7, 0 and 3 cycles, respectively.

Figure 3.(a) plots the cumulative distribution function (CDF) of slack across sub-requests of several workloads on a 4 channel SSD with 16 flash chips. One can see from these results that, over 50% of the sub-requests have tens of milliseconds of slack. While one could try reducing this slack, this paper examines the more interesting possibility of leveraging this slack for better scheduling.

### 3.2 Why Does Slack Arise?

There are two main causes of slack. First, as can be seen in Figure 3.(b), a majority of the requests are relatively small, with their pages spanning just a few chips leading to lower chip-level parallelism. Large requests, on the other hand, could span all the chips at the same time, leading to higher (flash) chip-level parallelism. Even though several small requests could be serviced at the same time by the higher parallelism offered by the hardware, it is not necessary that these requests be disjoint in the chips that they exercise. This can lead to some sub-requests of a request having to wait their turn for their respective chips while their sibling sub-requests of the same request are being serviced at other chips. The consequent load imbalance across the chips is evident when we observe the average, maximum, and minimum values for the number of sub-requests in request queues of each chip waiting to be serviced in Figure 3.(c). At any instant, the load varies significantly – sometimes the maximum queue length is 72 times larger than the minimum, which in turn can skew the end times of the sub-requests (within a request) directed to different chips. The second reason for the non-uniform end times of sub-requests is the widely differential latency of the basic flash operations: read, write, and erase. Since not all chips are necessarily doing the same operation at the same time, the sub-requests of a request may have different waiting times even if they are all next in line at their respective queues.

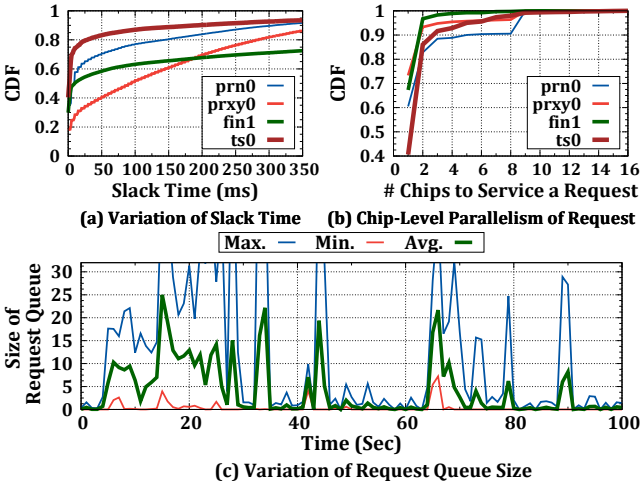



Figure 3: (a) CDF of slack time and (b) CDF of chip-level parallelism of requests for 4 workloads, (c) The maximum, minimum and average request queue sizes over 100sec for prxy0.

### 3.3 How to Get the Most out of this Slack?

In the presence of slack for sub-requests within a request, the request service time is determined by the completion of the critical sub-request. One can tackle slack in two ways:

- **Reduce slack:** Reducing slack, in general, requires finding time slots where all requisite chips for a request are free, akin to gang scheduling. Waiting for such time slots can lead to significant under-utilization as has been well studied [11]. SSDs, however, offer some unique opportunities for reducing slack within write requests owing to their "no-write-in-place" property, i.e., if the chip is busy, the corresponding write sub-request could be directed to some other chip that is opportunistically free. Techniques such as Dynamic Write Mapping [14, 27, 29] and Write Order Base Mapping [9] could be employed to perform such re-direction, which could lower slack within write requests. A significant drawback with such write re-direction is that additional mapping tables need to be maintained to re-direct a subsequent request to the appropriate channel, chip, die and plane where it has been re-mapped, i.e., a static strategy such as CWDP which does not require such a table can no longer be employed. Consider for example a page-level mapping for an SSD with configuration in Table 1 used in this paper which has 1TB capacity, 16 flash chips, 4 dies per flash chip and 2 planes per die. Dynamic

re-direction requires maintaining 7 bits per page (4 bits for chip number, 2 bit for die and 1 bit for plane). With 8KB sized pages, we need $\frac{1TB}{8KB} = 2^{27}$ entries, each of 7 bits, putting the additional storage requirement at 117MB which is 23% of the 512MB RAM capacity on state-of-the-art SSDs. Already, the on-board memory storage is very precious, and needs to be carefully rationed amongst the different needs – caching data pages, caching FTL translations, temporary storage for garbage collection, etc. Sacrificing a fourth of this capacity (or adding the required capacity) can be a significant overhead. Additionally, re-direction is not an option for reads. Prior work [14, 29] has shown that write re-direction can actually hurt reads which are usually in the critical path. Our experimental results will also concur with this observation.

- **Exploit Slack:** In contrast, in this paper, we explore how to leverage any existing slack between sub-requests to improve response time. The basic idea is to identify sub-requests in the request queue with high slack and de-prioritize them to benefit some others. Such re-ordering is done without impacting the waiting times of those being de-prioritized by leveraging knowledge of their slack. For instance, consider Figure 2.(a) where the baseline (slack-unaware FR-FCFS) scheduler prioritizes $A2$, because of its earlier arrival, thereby delaying $B0$. In contrast, if the scheduler is aware of the sub-request slacks, it would prioritize $B0$ over $A2$ as the latter can be delayed without affecting the response time of $A$. Figure 2.(b) shows this. Doing so reduces the response time of Request $B$ *without* increasing the response time of Request $A$ (its critical sub-request, $A1$, remains unchanged), thereby improving the overall SSD response time.

# 4. Slacker Mechanisms

We propose *Slacker*, that makes slack-aware scheduling decisions at FSU to improve response time. The main idea of Slacker is to identify how much the service for the sub-requests waiting in the queue can be delayed (based on their slack) to accelerate the service of newer sub-requests upon their arrival.

## 4.1 Reordering

Slacker works on a generic SSD architecture illustrated in Figure 1 that uses FR-FCFS algorithm at FSU. Reordering sub-requests waiting in the queue, using their slack time, is performed by de-prioritizing sub-requests already ordered by FR-FCFS. As FR-FCFS is composed of two prioritization rules, *read-first* and *oldest-first*, de-prioritization intuitively has two dimensions: relaxing the *read-first* order and relaxing the *oldest-first* order.

**Relaxing *read-first* order.** If the slack of a read sub-request in RDQ is larger than write latency, this slack can be used by a write sub-request (in WRQ) to be serviced ahead of that read sub-request. However, we do not expect significant improvement in response time of such writes due to two reasons: (1) by analyzing a wide range of workloads, we observed that the average slack seen by a read sub-request is typically much lower than the flash write latency (slack of microseconds compared to write latency of hundreds of microseconds to milliseconds, see Table 2 in Section 6), and (2) this approach can at best reduce the response time of a write sub-request by a time equal to that of a read latency, which is much smaller. Hence, this relaxation is not likely to provide meaningful benefits.

**Relaxing *oldest-first* order.** Oldest-first policy in FR-FCFS performs FIFO scheduling in RDQ for read sub-requests and in WRQ for write sub-requests. With this relaxation, we propose reordering requests in each of the queues independently. If a read sub-request has slack, we use this slack for allowing other reads in RDQ to bypass this one. We call this scheme, *Read Bypassing* (**Rbyp**) which can improve read response time. Similarly, slack of a write sub-request is only used to accelerate service of other write sub-requests in WRQ, and is referred to as *Write Bypassing* (**Wbyp**).

### 4.1.1 Reordering Algorithm

At a high level, the scheduling of sub-requests across the RDQs and WRQs of flash chips can be posed as a two-dimensional constrained bin-packing problem – filling in the time slots with sub-requests directed at each flash chip, so that average response times of requests can be minimized, as shown in the matrix in Figure 4. While it may be advantageous to co-schedule sub-requests of a macro request in the same time slot (same row in Figure 4) to avoid slack and having to wait for the last sub-request to complete (similar to Gang scheduling of parallel processors [26]), such restrictions may fragment rows of this matrix, leading to under-utilized slots. Opportunistically using such slots, without being restricted to co-scheduling, would improve the utilization but can result in the slack problem that has been discussed until now. Regardless, this two-dimensional constrained bin-packing problem is NP-hard [22]. A brute-force evaluation of all permutations at each request arrival in the flash HIL/FSU would be highly inefficient (the matrix contains several dozen rows and columns). Further, since each FSU maintains its own queues, coordinatedly permuting the entries across all these distributed queues can incur tremendous overheads. Instead, Slacker employs a simple heuristic that can be implemented within each FSU to order its request queues independent of the queues of the other FSUs. Also, each FSU takes only $O(N)$ time for inserting a sub-request in its queue (i.e., the column in the matrix of Figure 4, where $N$ is the current queue length).

Upon a request arrival, each FSU has little information on the queues of other FSUs to try and co-schedule its assigned sub-request with its sibling sub-requests at other FSUs in the same time slot. Having discussed earlier, queue lengths can vary widely across the chips at any instant and hence, simply adding the sub-request at the tail of each FSU queue

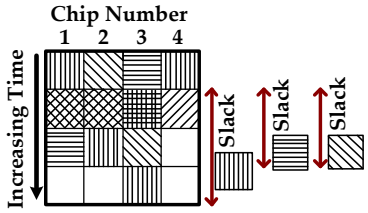



Figure 4: 2-D bin packing of sub-requests.

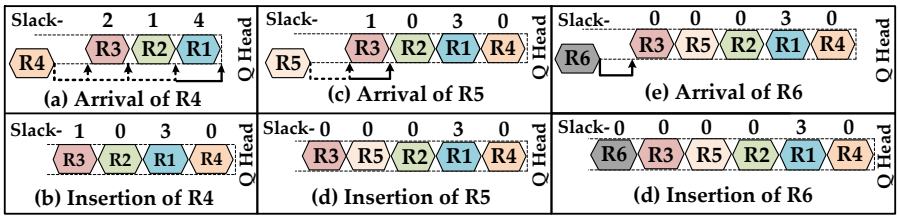



Figure 5: Reordering incoming sub-requests at FSU.

as in FR-FCFS can lead to wide slacks without availing the flexibility that such slacks allow in scheduling. Instead, in Slacker, each FSU individually examines whether each incoming sub-request can jump ahead of the sub-requests already waiting in its queue, starting from the tail. It can jump ahead as long as the slack of the waiting sub-request is higher than the estimated service time of this new sub-request (i.e., delaying the waiting sub-request by servicing the incoming one will not delay the overall response time of the request it belongs to). As it jumps ahead of each sub-request, their slack is accordingly reduced. The new sub-request is inserted in the queue position, where it cannot jump ahead any more. After the sub-requests of the new macro request are inserted in the respective FSU queues, their slacks can be estimated/computed (slack estimation is explained later). This mechanism is illustrated in Figure 5 for an existing queue, with an incoming stream of 3 new sub-requests. Since the incoming sub-request would at best jump over $N$ waiting sub-requests in the queue, the work at each FSU is $O(N)$.

#### 4.1.2 Examples of Rbyp and Wbyp

Figure 6.(a) shows how Rbyp improves performance with an example. It depicts the RDQ in baseline (①) and a system with Rbyp (②). In this example, read request *RA* has two sub-requests (*RA1* and *RA2* with slack values of 2 and 0), read request *RB* has two sub-requests (*RB1* and *RB2* with slack values of 1 and 0), and read request *RC* has two sub-requests (*RC1* and *RC2* with slack values of 1 and 0). At time $t_3$, when *RB* and *RC* arrive, Rbyp pushes *RB2* and *RC2* forward (both had 0-cycle slack before reordering) and pushes *RA1* back (previously had 2-cycle slack). As a result, the response times of requests *RB* and *RC* improve while the response time of *RA* remains unchanged (since the response time of its laggard sub-request did not change). Hence, Rbyp can improve the response time of a read request if there exists enough slack in other requests.

Figure 6.(b) shows the possibility of performance improvements with Wbyp using an example. The status of WRQ in baseline and a system with Wbyp are shown in ③ and ④, respectively. In this scenario, we have three write requests. *WA* is a single-page request that has no slack. *WB*, on the other hand, has three sub-requests: *WB1*, *WB2* and *WB3* with response times of 9, 5 and 4 cycles, respectively and slacks of 0, 4 and 5. *WC* is also a single-page write request with response time of 8 and slack of 0. *WC1* can be serviced earlier in Wbyp than the baseline by prioritizing it over *WB2*, resulting in 4 cycles faster response time for *WC*. Note that, the response time of request *WB* does not get worse as the service of *WB2* is delayed at most by its slack.

### 4.2 Slack-aware Write Pausing

Until now, similar to prior scheduling proposals, our optimization have only looked at requests in the queues without pre-empting sub-requests already being serviced. Going a step further, it is possible that even sub-requests that have started being serviced could have slacks and thus become a candidate for reordering. Utilizing this slack, the controller may decide to cancel service of the currently being processed sub-request to favor an incoming request. However, a simple cancellation midway through service would throw away a lot of the accomplished work. Instead, we look into options for pre-emption of the request being currently serviced in favor of another sub-request in the queue and then restart the canceled sub-request in later cycles. In effect, we are simply advancing the re-ordering algorithm described earlier by one more step - to include the sub-request currently being serviced.

Pausing a read sub-request, in favor of another read or a write sub-request, is not expected to be beneficial as both read latency and slack of a read sub-request are small. With higher write latency, there may be merit to pre-empt ongoing writes in favor of other reads. Pre-empting a write to service another write is not possible in current hardware[2], and is also not expected to provide significant benefits either since both operations take equally long time. Instead, we only consider pre-empting an ongoing write to service incoming read sub-requests if the former has sufficient slack. We adapt a previously proposed write pausing (WP) technique [30] for our slack-aware reordering. In [30], the authors have shown that the write implementation in modern flash technologies has features that can be leveraged for pausing. The first feature is that read operation does not modify write point [7]. So after reading a page, the write point still refers to the place of the paused write, and we can thus serve the read in the middle

[2] Writes in flash are sequential, and the write point [7] gets lost.

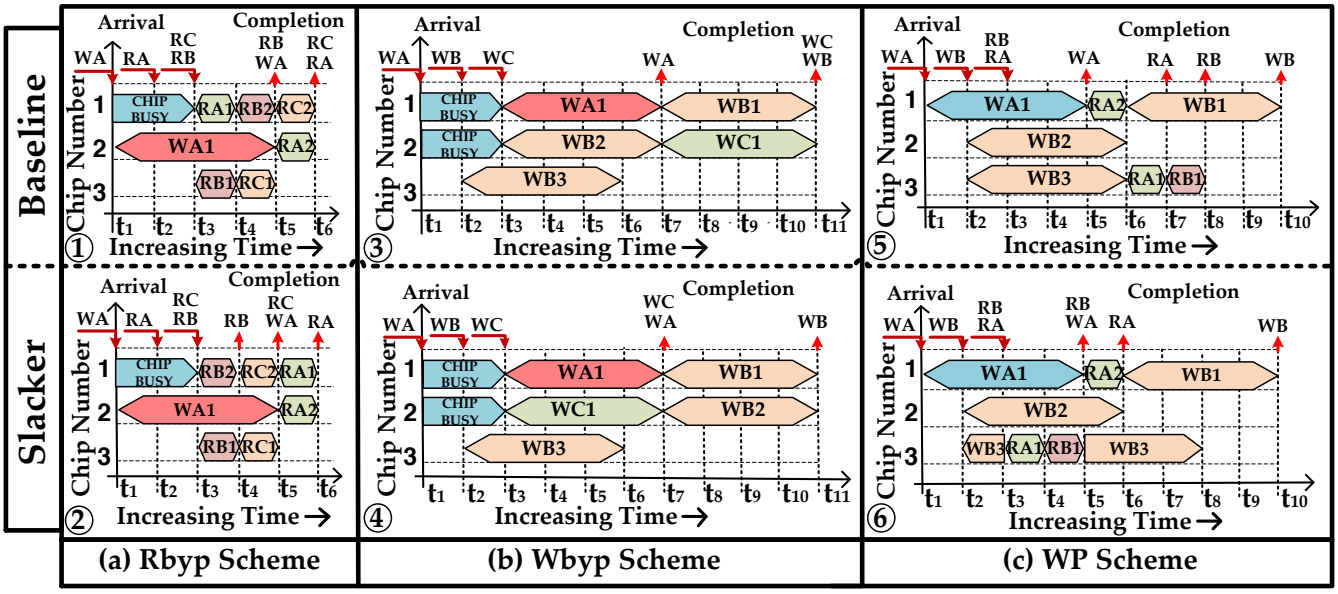



Figure 6: Examples for (a) Rbyp (RB and RC finished 1 cycle earlier without impacting RA and WA), (b) Wbyp (WC finished 4 cycles earlier without impacting WA and WB), and (c) WP (RA and RB finished 1 and 3 cycles earlier without impacting WA and WB). RA, RB, and RC are read requests, WA, WB, and WC are write requests. RAi is the $i^{th}$ sub-request in RA.

and resume the write later. The second feature comes from the programming algorithm, *write-and-verify* technique [6]. As the name suggests, a write-and-verify programming technique consists of applying a programming pulse followed by a read stage which is used to decide whether to stop or to compute the new programming signal to be sent. It is possible to pre-empt an on-going write at the end of each such iteration for servicing a pending read sub-request. Therefore, at each potential pause point, the scheduler checks if there is a pending read request, service it, and resume the write from the point it was left based on the slack of the ongoing write.

Figure 6.(c) shows the possibility of improving read response time via write pausing using an example. In the baseline (⑤), there are two writes: one (*WA*) with no slack and one (*WB*) with 4-cycles of slack for two of its sub-requests. With WP (⑥), the controller cannot pause *WA1* for sub-request *RA2*, but *WB3* is paused for two cycles to enable the service of *RA1* and *RB1*, thereby reducing the read latency of *RA* and *RB*, without affecting the response time of *WB*.

### 4.3 Key Points to Note

Here are some salient points to keep in mind about Slacker:

- Each queue is individually re-ordered once slacks are computed, with no global coordination.
- When the sub-requests of a request are added to the respective chip queues, their individual slacks are computed (as described in the next section). They will not change subsequently, unless some other requests jump ahead of them in the same queue or they are paused while being serviced.
- Computing the slack of a sub-request needs initial information about its sibling sub-request response times (calculated later) when they are inserted in the respective queues. There is no subsequent global information exchange.
- No slack is ever allowed to become negative, thereby not delaying any request beyond its original scheduled completion time. The only reason this constraint would be violated is when slacks are mis-estimated. As we will show next, our estimates are fairly accurate.

## 5. On-line Estimation of Slack

To accomplish reordering and pausing, it is of utmost importance to accurately quantify the slack of each sub-request. If the slack is not accurate, the proposed enhancements could delay some sub-requests by more than their actual slack, increasing their response times. Despite knowing the queue positions of all sub-requests and reasonable estimates of response times of sub-requests before them in an isolated setting, precise estimation is difficult due to non-deterministic and unpredictable behavior of requests contending for shared resources (channels and chips), utilizing advanced commands [4], performing GC, and arrival of future read requests that can get prioritized over writes. To overcome this inaccuracy, we propose a *stochastic model* working in an online fashion to approximate response time of each sub-request of a request. Having estimated response times of each sub-request of a request, it is straightforward to calculate its slack time by subtracting its response time from the critical sub-request response time. This is calculated as soon as all the sub-requests of an incoming request are in-

serted into their respective queues availing of information about sub-requests that are ahead in the queues.

### 5.1 Estimating Slack

**Calculating response time.** The response time of a read/write sub-request, $T_{RD/WR}^{Response}$, is composed of the wait time seen by the sub-request in a RDQ/WRQ, $T_{RD/WR}^{Wait}$, and its own service time, $Latency_{RD/WR}$. Therefore, we have:

$$T_{RD/WR}^{Response} = T_{RD/WR}^{Wait} + Latency_{RD/WR} \quad (1)$$

Obtaining accurate estimates for $T_{RD/WR}^{Wait}$ is challenging because (1) FCFS is violated as the new incoming read sub-requests can bypass existing write requests, (2) performing advanced commands affects our estimated time by servicing multiple requests at the same time, (3) performing GC, and (4) there is potential interference between command and data messages sent to different flash chips sharing a channel. The wait time for a sub-request consists of four parts: the first is the delay due to actual queuing latency, $T^{Queue}$; the second is the average stall-time of the chip due to GC, $T^{GC}$; the third is the blocking time over the channel, $T^{Interference}$; and the fourth is the time remaining to complete the ongoing operation on the chip, $T^{Residual}$. Therefore we can write:

$$T_{RD/WR}^{Wait} = T_{RD/WR}^{Queue} + T^{GC} + T_{RD/WR}^{Intereference} + T^{Residual} \quad (2)$$

Below, we calculate each of these components.

**Estimating queuing latency.** The scheduler always prioritizes read requests over writes.Then, the queuing latency for a read sub-request, $T_{RD}^{Queue}$, is different from that for a write sub-request, $T_{WR}^{Queue}$: $T_{RD}^{Queue}$ is the sum of the service times of all preceding read entries in RDQ, and $T_{WR}^{Queue}$ is the sum of the service times of all read sub-requests in RDQ and the preceding write sub-requests in WRQ. So:

$$\begin{aligned} T_{RD}^{Queue} &= Num_{RDQ} \times Latency_{RD} \\ T_{WR}^{Queue} &= T_{RD}^{Queue} + Num_{WRQ} \times Latency_{WR} \end{aligned} \quad (3)$$

where $Num_{RDQ}$ and $Num_{WRQ}$ refer to the number of read and write entries in the request queues, respectively. $Num_{RDQ}$ and $Num_{WRQ}$ are obtained after the sub-requests have been inserted in the appropriate queues by the scheduler. On servicing a request, modern controllers check the possibility of employing multi-plane or multi-die operations. This affects the queuing latency of a sub-request by reducing the service times of the requests scheduled earlier. To measure the effect of advanced operations, the controller keeps four counters per-chip: $Cntr_{RD}$, $Cntr_{WR}$, $Cntr_{ARD}$, and $Cntr_{AWR}$. The controller increases $Cntr_{RD}$ and $Cntr_{WR}$ when it services a read sub-request or a write sub-request, respectively, and increments $Cntr_{ARD}$ and $Cntr_{AWR}$ when it commits $N$ read or write requests, respectively, by an advanced command. Assuming for now that the controller knows the values of these counters for each chip, it calculates the probability of executing advanced commands in RDQ or WRQ as $Pr_{RD\ or\ WR}^{AdC} = \frac{Cntr_{ARD\ or\ AWR}}{Cntr_{RD\ or\ WR}}$. So we update Eq. 3 as:

$$\begin{aligned} T_{RD}^{Queue}(New) &= T_{RD}^{Queue}(Old) \\ &\quad \times ((1 - Pr_{RD}^{AdC}) + Pr_{RD}^{AdC}/N) \\ T_{WR}^{Queue}(New) &= T_{RD}^{Queue}(New) + T_{WR}^{Queue}(Old) \\ &\quad \times ((1 - Pr_{WR}^{AdC}) + Pr_{WR}^{AdC}/N) \end{aligned} \quad (4)$$

**Estimating chip stall-time due to GC.** When a chip is busy with GC, it cannot service any other request. GC latency has two parts: (1) the latency of moving valid pages, $T^{Move}$; and (2) the erase latency, $Latency_{Erase}$. The former changes over time and the controller keeps two counters per chip, $Cntr_{Move}$ and $Cntr_{ER}$, to compute it as $T^{Move} = (Cntr_{Move}/Cntr_{ER}) \times (Latency_{RD} + Latency_{WR})$. $Cntr_{Move}$ and $Cntr_{ER}$ count the total number of page movements during GC and the number of erases, respectively. Thus, $T^{GC}$ can be estimated as:

$$T^{GC} = Pr^{GC} \times (T^{Move} + Latency_{Erase}) \quad (5)$$

where $Pr^{GC}$ is the probability of executing GC, and is computed as $Pr^{GC} = \frac{Cntr_{ER}}{Cntr_{RD} + Cntr_{WR}}$.

**Estimating interference latency.** When a read or write sub-request is sent over the channel to a chip, it keeps the channel busy for $T^{Xfer}$ cycles (for a page size of $P$ bytes and a channel width of $W$ bytes, $T^{Xfer} = \frac{P}{W \times 333MT/s}$ for ONFi 3.1). During this time, the FSU is not able to schedule a command to any other chips on that channel, even though several commands might be ready to be scheduled. Hence, $T^{Interference}$ for each sub-request can be estimated as:

$$T^{Interference} = E[\#ReadyRequests] \times T^{Xfer} \quad (6)$$

where $E[\#ReadyRequests]$ is the average number of sub-requests contending for the same shared channel at the same time. The controller maintains two counters per-channel, $Cntr_{Ready}$ and $Cntr_{Total}$, and computes $E[\#ReadyRequests] = Cntr_{Ready}/Cntr_{Total}$. $Cntr_{Ready}$ is incremented when a command is ready to be issued but stalled due to a busy channel. $Cntr_{Total}$ counts the total number of sub-requests mapped to the flash chips connected to the channel.

**Calculating residual time.** Upon arrival of a new flash request, the residual time of the current operation to be completed in the target chip is calculated as:

$$\begin{aligned} T^{Residual} &= \overline{Flag_{RD/WR}} \times Latency_{RD} \\ &\quad + Flag_{RD/WR} \times Latency_{WR} \\ &\quad - (T^{Now} - T^{Start}) \end{aligned} \quad (7)$$

$Flag_{RD/WR}$ and $T^{Start}$ are attributes of the current operation on the chip: $Flag_{RD/WR}$ is per-chip flag determining

type of the operation ("0": read and "1": write); $T^{Start}$ is a per-chip register holding the start time of the operation.

**Calculating slack time.** After estimating the response times of the $N$ sub-requests of a macro request, the slack for the $i^{th}$ sub-request, $T_i^{Slack}$, can be calculated as

$$T_i^{Slack} = Max(T_1^{Response},\ T_2^{Response},\ \ldots,\ T_N^{Response}) - T_i^{Response} \quad (8)$$

where $T_i^{Response}$ is response time of $i^{th}$ sub-request from Eq. 1.

### 5.2 Accuracy of Estimation

Slack estimation primarily depends on $T^{Response}$ of sub-requests. To verify the accuracy of our $T^{Response}$ estimation described above, we compare the estimates with actual response times of requests in a number of workloads. As can be seen in Figure 7, our estimates are quite accurate, with errors less than 1% for a number of workloads. Even in the few cases where the errors exceed 5%, we will show that our solution is still able to improve response times[3].

### 5.3 Hardware Support and Latency Overhead

Slack estimation described above, requires additional counters, registers and flags. For each chip, eight up/down counters ($Num_{RDQ}$, $Num_{WRQ}$, $Cntr_{RD}$, $Cntr_{WR}$, $Cntr_{ER}$, $Cntr_{ARD}$ and $Cntr_{AWR}$, $Cntr_{Move}$), a flag ($Flag_{RD/WR}$), and two registers for $T^{Start}$ and $T^{Now}$ are needed.In addition, for each channel, two counters ($Cntr_{Ready}$ and $Cntr_{Total}$) are needed. This information should be maintained at the controller (at FSU) and the imposed overhead is relatively small, taking only a few additional bytes to provide the requisite information.

In our Slacker implementation, the latency overheads of slack estimation and re-ordering is around 100 cycles that becomes less than 1us with SSD configuration in the next section; i.e., negligible compared to the read and write latencies.

## 6. Experimental Setting

**Evaluation platform.** We model a state-of-the-art SSD using SSDSim [14], a discrete-event trace-driven SSD simulator. SSDSim has detailed implementations of page allocation strategies and page mapping algorithms and captures inter-flash and intra-flash parallelism. It also allows studying different SSD configurations with multiple channels, chips, dies, and planes. The accuracy of SSDSim has been validated via hardware prototyping [14].

[3] The estimation of wait time in Eq. 2 may still have inaccuracies.Using an error term, the wait time of a sub-request can be updated as: $T^{Wait}_{RD/WR}(New) = T^{Wait}_{RD/WR}(Old) + T(Err)$, where $T(Err)$ is average time difference between the actual wait time and the estimated wait time for all the requests serviced in the last second (i.e., a moving average of the estimation error). With this error term, we experienced insignificant reduction in estimation error (less than 1%) compared to Figure 7. Thus, we did not consider it in our model.

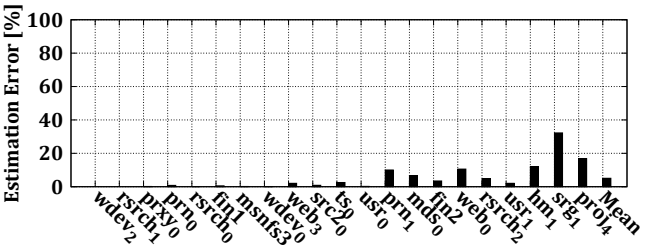



Figure 7: Estimation error.

Table 1: Main characteristics of simulated SSD.

| **Evaluated SSD Configuration** |
|---|
| 4×4 Dimension (4 Channels and 4 Flash Chips per Channel), Channel Width = 8 bits, NAND Interface Speed = 333 MT/s (ONFI 3.1), Page Allocation Scheme = Channel-Way-Die-Plane (CWDP) |
| **NAND flash (Micron [24])** |
| Page Size=8KB, Metadata Size=448B, Block Size=256 pages, Planes per Die=2, Dies per Chip=4, Flash Chip Capacity=64GB, Read Latency=75$\mu$s, Typical Program Latency=1300$\mu$s, Worst Program Latency=5000$\mu$s, Erase Latency=3.8ms |

**Configurations studied** The baseline configuration consists of four channels, each of which is connected to four NAND flash chips. Each channel works on ONFi 3.1 [4]. Table 1 provides specifications of the modeled SSD (which is very similar to [3]) along with parameters of the baseline configuration. The evaluated SSD systems use modern protocols and schedulers at different levels: NVM-e [16] at HIL and an out-of-order scheduler [25] for parallelism-aware scheduling at FSU. In our experimental analysis, we evaluate six systems:

- ***Baseline*** uses FR-FCFS for micro scheduling at FSU.
- ***Wbyp*** uses our write bypassing scheme on top of FR-FCFS.
- ***Rbyp*** uses our read bypassing scheme on top of FR-FCFS.
- ***WP*** is a system with only write pausing on top of FR-FCFS.
- ***Rbyp+WP*** applies read bypassing and write pausing.
- ***Slacker*** is a system with both read and write bypassing, as well as write pausing.

We report the amount of reduction in request response time (read and write) as the performance metric. The response time is calculated as the time difference between the arrival of the request at the host interface and the completion of its service.

Table 2: Characteristics of the evaluated I/O traces.

| Trace | WR-RD Ratio | RD Req. Size (KB) | | WR Req. Size (KB) | | RD Slack (mSec) | | WR Slack (mSec) | |
|---|---|---|---|---|---|---|---|---|---|
| | | Mean | SD | Mean | SD | Mean | SD | Mean | SD |
| **Write Intensive Disk Traces** | | | | | | | | | |
| wdev2 | 0.99 | 13.4 | 8.6 | 16.2 | 11.7 | 0.0 | 0.0 | 144.8 | 205.4 |
| rsrch1 | 0.99 | 25.6 | 16.4 | 18.7 | 15.7 | 0.1 | 0.1 | 155.9 | 183.2 |
| prxy0 | 0.95 | 18.3 | 14.9 | 38.6 | 30.6 | 1.5 | 1.7 | 577.9 | 725.1 |
| prn0 | 0.94 | 22.3 | 18.7 | 16.3 | 19.5 | 2.5 | 6.7 | 130.0 | 164.9 |
| rsrch0-p | 0.9 | 18.9 | 46.7 | 16.5 | 12.2 | 6.8 | 10.2 | 43.9 | 49.5 |
| fin1 | 0.77 | 11.3 | 4.6 | 12.6 | 10.4 | 0.9 | 1.4 | 41.1 | 22.3 |
| msnfs3 | 0.76 | 23.6 | 23.1 | 23.2 | 25.4 | 2.1 | 1.7 | 26.9 | 65.8 |
| **Balanced Read-Write Disk Traces** | | | | | | | | | |
| wdev0 | 0.7 | 16.8 | 14.5 | 17.1 | 14.4 | 2.6 | 2.2 | 63.7 | 109.8 |
| web3 | 0.68 | 82.9 | 241.5 | 28.9 | 14.6 | 2.0 | 4.5 | 6.2 | 10.4 |
| src2-0-p | 0.64 | 18.2 | 11.8 | 22.7 | 20.3 | 5.1 | 14.3 | 193.9 | 160.7 |
| ts0-p | 0.56 | 19.6 | 14.7 | 19.6 | 17.7 | 1.4 | 1.5 | 52.8 | 61.3 |
| usr0-p | 0.43 | 66.9 | 16.5 | 17.7 | 13.3 | 46.7 | 48.2 | 37.1 | 11.2 |
| prn1-p | 0.42 | 16.3 | 14.2 | 17.4 | 13.3 | 0.5 | 1.1 | 72.2 | 178.6 |
| **Read Intensive Disk Traces** | | | | | | | | | |
| mds0-p | 0.21 | 43.5 | 26.3 | 18.4 | 15.6 | 2.8 | 5.2 | 14.1 | 74.1 |
| fin2 | 0.18 | 10.3 | 5.1 | 11.0 | 12.3 | 1.6 | 1.6 | 74.4 | 134.8 |
| web0-p | 0.18 | 46.9 | 26.3 | 16.6 | 14.1 | 1.3 | 1.7 | 15.7 | 43.2 |
| rsrch2-p | 0.08 | 12.0 | 4.0 | 12.2 | 4.0 | 1.3 | 1.4 | 67.9 | 161.2 |
| usr1-p | 0.06 | 56.3 | 26.3 | 14.5 | 7.8 | 30.9 | 29.4 | 36.1 | 93.9 |
| hm1 | 0.05 | 22.9 | 19.2 | 27.8 | 32.5 | 2.2 | 4.9 | 35.1 | 17.9 |
| stg1-p | 0.02 | 68.5 | 13.6 | 15.7 | 11.9 | 0.4 | 0.9 | 4.5 | 3.2 |
| proj4 | 0.005 | 32.8 | 28.5 | 18.4 | 17.7 | 0.3 | 0.4 | 6.4 | 2.7 |

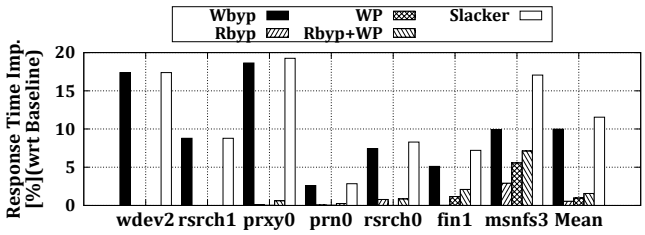
**I/O workload characteristics** We use a diverse set of real disk traces: Online Transaction Processing (OLTP) applications [2] and traces released by Microsoft Research Cambridge [1]. In total, we study 21 disk traces to ensure that we cover a diverse set of workloads. Table 2 summarizes characteristics of our disk traces in terms of write-to-read-ratio (*WR-RD*), average/standard deviation of request sizes, and average/standard deviation of slack across the sub-requests within a request for the baseline system. To better understand the benefits of our slack-aware scheduler, we categorize our disk traces into three groups based on their *write intensity* as it directly contributes to the efficiency of each proposed scheme. In our categorization, a disk trace is (a) *Write Intensive* if its write-to-read ratio is greater than 0.70; (b) it is *Balanced Read-Write* if its write-to-read ratio is between 0.30–0.70; otherwise, (c) it is *Read Intensive*.

# 7. Experimental Evaluation

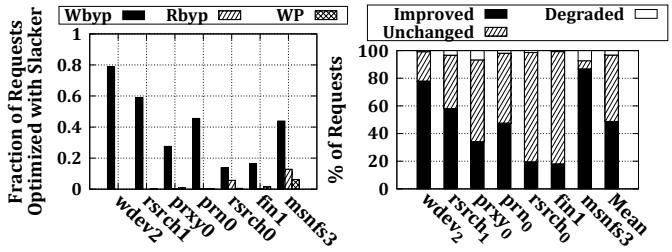
In the next 3 subsections, we analyze the performance results for the three workload categories separately. For each, we present performance results (Figures 8, 9 and 10) in terms of **(a)** the percentage improvement in response times of all requests over the baseline system; **(b)** the fraction of requests that benefit from each of Wbyp, Rbyp and WP. This gives scope of requests that could benefit from these enhancements. **(c)** the percentage of requests which had response times lowered, unchanged and increased with respect to the baseline. Note that even though our proposals intend to not impact requests that are bypassed due to their slack, mis-estimation of slack can sometimes inadvertently impact them and hence it is important to quantify this effect.

## 7.1 Results for Write-Intensive Traces

**Impact of Wbyp**. Wbyp is targeted to improve write request response times. We can see the consequent effect that yields between 3% to 19% improvement across all requests for these write intensive workloads. Workloads such as `prxy0`, `wdev2`, `msnfs3`, and `rsrch1` provide high opportunities for leveraging Wbyp, allowing more incoming write sub-requests to jump ahead of ones with higher slack that are already in WRQ. These are also the workloads with higher overall response time improvements. At the same time, it should be noted that simply having higher opportunities for Wbyp does not automatically translate to better performance. For instance, `prn0` and `fin1` give only 5% performance improvement, even though 50% and 20% of requests benefit from Wbyp in these respective workloads. Despite opportunities for reordering, the gains for each such




(a) Response Time Improvement

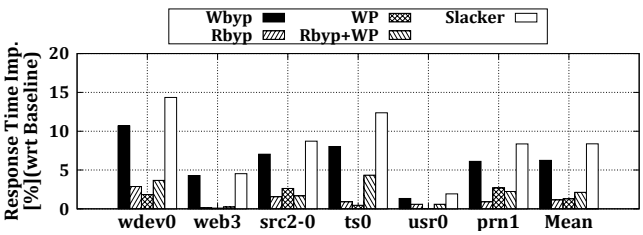



(b) Fraction of Requests Optimized

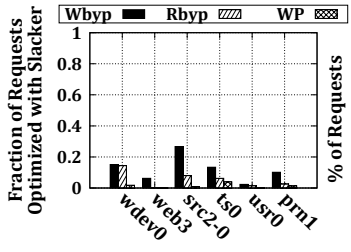



(c) Percentage of Requests Benefit/Unchanged/Hurt

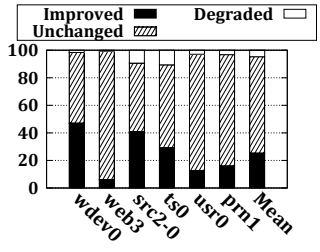
Figure 8: Results for write-intensive workloads.




(a) Response Time Improvement




(b) Fraction of Requests Optimized




(c) Percentage of Requests Benefit/Unchanged/Hurt

Figure 9: Results for read-write balanced workloads.

reordering is relatively small in these workloads (see Section 7.4 for more details).

**Impact of Rbyp, WP and Rbyp+WP.** Both Rbyp and WP enhancements mainly target reads. As a result, in these write-intensive workloads, these enhancements do not provide significant improvements. Of these workloads, `msnfs3` has the highest read intensity, and is the only case where any reasonable fraction of reads benefit from these enhancements. This results in a 3%, 6% and 7% (16% collectively compared to 10% with Wbyp) reduction in response times given by Rbyp, WP and Rbyp+WP, respectively, for `msnfs3`.

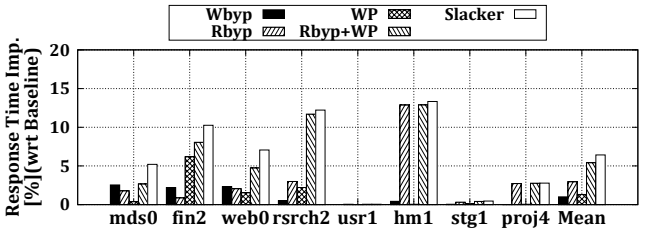
**Slacker.** The results for Slacker, which incorporates all 3 enhancements (Wbyp, Rbyp and WP) gives an interesting insight - one enhancement does not counter-act another, thereby all three can be collectively used to reap additive benefits. In the first 5 workloads, read enhancements (Rbyp and WP) are not providing any benefits, and Slacker defaults to Wbyp which provides good improvements. In the last 2 (can be visibly seen for `msnfs3`), Slacker's benefits are additive over each of the individual improvements. Overall, Slacker gives between 3% to 19.5% (12% on average) reduction in total request response time compared to the baseline. From Figure 8.(c), we can see that Slacker can improve 50% of all requests on the averages. Even though there is a danger of some requests getting slowed down due to mis-estimation, the results confirm this fraction is negligible (less than 3%).

### 7.2 Results for Read-Write Balanced Traces

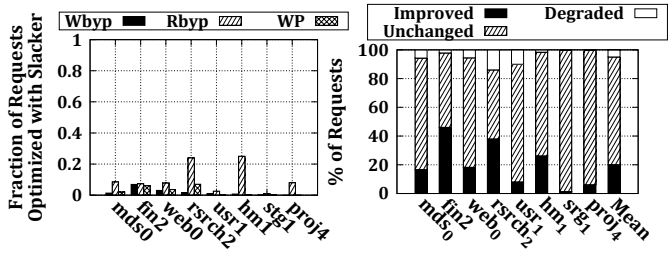
**Impact of Wbyp.** With a lower write intensity in these workloads, the improvements with Wbyp are a little smaller than in the previous case. We see that opportunities for applying Wbyp have gone below 20% (Figure 9.(b)), compared to values reaching 60–80% in write intensive workloads. Still, Wbyp is able to reduce response time of all requests by 2% to 11% (6% on the average). Among the workloads, those with medium-sized requests such as `wdev0`, `src2-0`, `ts0`, and `prn1`, exhibit higher improvements (up to 11%). Note that with larger requests, there is higher slack amongst the sub-requests, that span more chips, to benefit further from Wbyp. In workloads such as `web3` and `usr0`, write slack is much smaller giving less than 5% improvements with Wbyp.

**Impact of Rbyp, WP and Rbyp+WP.** With a higher fraction of reads, Rbyp and WP provide higher response time improvements in these workloads compared to the earlier set. Amongst these, `wdev0` and `ts0` have a larger fraction of requests availing of Rbyp and WP (in Figure 9.(b)), which translates to a 4% overall response time reduction in Rbyp+WP system. While increasing request sizes lead to larger number of sub-requests spanning more chips (to create higher slack), there is a point beyond which the slack times can drop. As all requests become large enough to span all the chips, that is already sufficient parallelism and slack-aware reordering is not likely to provide additional benefits. For instance, in `web3` (with mean read request size of 83KB) and `usr0` (with mean read request size of 66KB), the performance gains with Rbyp and WP are much smaller, with their read request sizes that are significantly higher than the rest (see Table 2).

**Slacker.** Wbyp is still giving the highest rewards of the three optimizations in this set of workloads (since write latency are much higher than reads), though the others do contribute to reasonable improvements. Slacker still does enjoy the additive benefits of the three to some extent, with overall response time improvements that range between 2%–14.5% (with an average benefit of 8.5%). As in the earlier set, there




(a) Response Time Improvement




(b) Fraction of Requests Optimized (c) Percentage of Requests Benefit/Unchanged/Hurt

Figure 10: Results for read-intensive workloads.

is a negligible fraction of requests (less than 5%) that suffer any response time degradation with slacker (Figure 9.(c)). At the same time, the average number of requests that have benefited from Slacker has gone down to 25% in this set, compared to an average of 50% in the previous set.

### 7.3 Results for Read-Intensive Traces

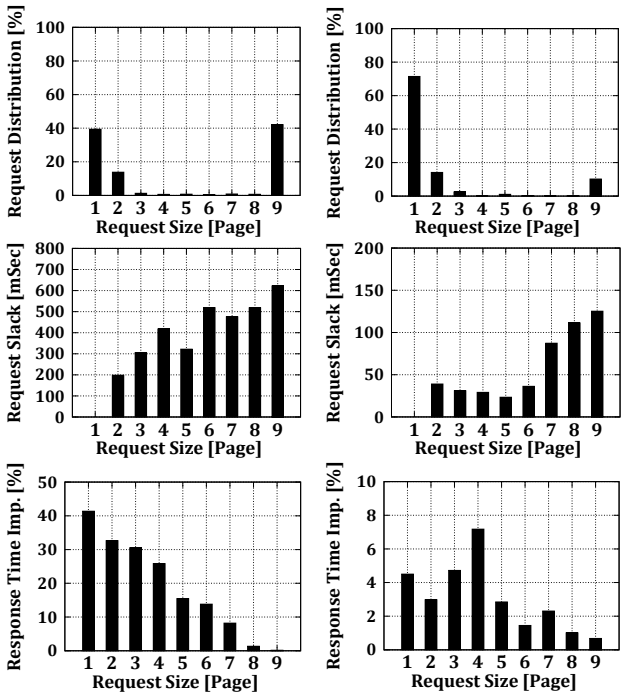
**Impact of Wbyp**. As can be expected, with the low fraction of write requests in these workloads, there is little opportunity to benefit from Wbyp.

**Impact of Rbyp, WP and Rbyp+WP**. The fraction of requests that benefit from these read optimization is higher than those in the previous set. The fraction of requests benefiting from these optimization is over 20% in `rsrch2` and `hm1`. These are also the workloads which reap the highest improvements (of 12% and 13% for Rbyp+WP). Between Rbyp and WP, since the write requests are less prevalent, the latter enhancement does not have much scope in this set of workloads. So the read enhancements are mainly contributed to by Rbyp.

**Slacker**. Based on the above observations, Slacker's benefits in this workload set is mainly driven by Rbyp. Overall, response time gains are up to 13% (average of 6.5%). As in the previous two workload sets, very few requests (less than 5%) suffer any performance degradation with Slacker (Figure 10.(c)), while 20% of all requests benefit on the average.

### 7.4 Slack Distribution and Slack Exploitation

We need to address two important questions related to the benefits of Slacker. First, *how much slack remains for a request after it is scheduled by Slacker?* Second, *how much slack is utilized by requests of different types and sizes?* We




(a) **prxy0** (b) **prn0**

Figure 11: Detailed analysis of prxy0 and prn0.

answer these questions by studying the slack distribution and slack exploitation for different request sizes in two workloads: one that significantly benefits from Slacker (`prxy0`) and one that slightly benefits from Slacker (`prn0`). For each workload, Figure 11 reports three statistics: *request size distribution*, *the average amount of slack that requests of different sizes accrue*, and *the average response time improvement associating with requests of different sizes.* In Section 7.1, we showed that `prxy0` achieves around 19% response time improvement by Slacker, while the fraction of requests optimized is around 30%. Figure 11.(a) reveals the reason: in `prxy0`, request sizes of 1-page, 2-pages and 9-pages are dominant while the first two request sizes have the maximum performance improvements (41% and 32%, respectively). This large amount of performance improvement directly relates to the amount of slack remains after reordering by Slacker. The case is opposite for `prn0` – the fraction of requests optimized by Slacker is 0.45, the performance improvement is 3%. Figure 11.(b) reveals the reason. From these results, we should note two points. First, large requests usually have large intra-request slack, because their sub-requests are stripping over more chips and their service time skews in time with high probability. Second, small requests get higher benefits than large ones.

### 7.5 Tail Latency Analysis

To have a very accurate picture of the Slacker's benefits, Figure 12 shows the 95th percentile of response time im-

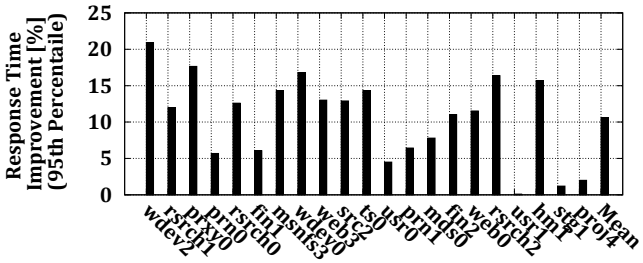



Figure 12: Tail latency analysis.

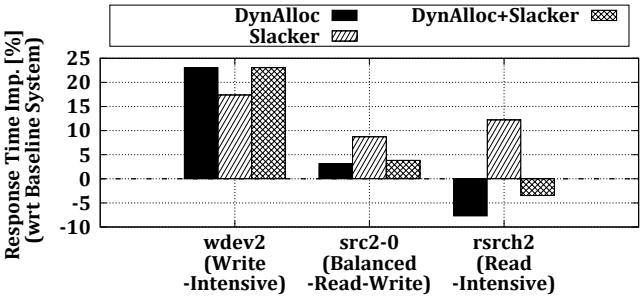



Figure 13: Comparison of dynamic redirection and Slacker.

provement[4]. From this figure, one can find that, for many evaluated workloads, the 95th percentile performance gain (in terms of total response time reduction) by Slacker is higher than mean value reported in preliminary sections. This specifically happens for workloads such as `wdev2`, `rsrch1`, and `prn0`, with 5%, 5% and 3%, respectively, higher improvement in 95th percentile analysis compared to the mean analysis results in Sections 7.1–7.3.

### 7.6 Comparing with Dynamic Redirection

In Figure 13, we compare results of dynamic write redirection (**DynAlloc**) with those of Slacker for 3 representative workloads, one chosen from each of the 3 workload categories identified earlier. In addition to comparisons with Slacker, we also show response times of a scheme, **DynAlloc+Slacker**, that combines write redirection with Slacker that can exploit any remaining slack. As can be expected, dynamic write redirection works very well for the write intensive workload. In this case, the slack reduction achieved by dynamic write redirection results in 23% reduction in response times, compared to Slacker which exploits the slack to give 17.4% savings. Note that even in this case, Slacker does not incur additional storage overheads to maintain redirection tables. Combining the two schemes is not providing any more scope than what is achieved with just write redirection. As the write intensity reduces in the more balanced workload, `src2-0`, the improvements with write redirection are considerably reduced, giving only 3% reduction. Exploiting the remaining slack after such redirection does not have much scope either. On the other hand, exploiting the original slack directly gives much more scope, giving two times the improvement as write redirection. Moving to the read intensive workload, we get re-confirmation of earlier observations [29] that such write optimization can actually hurt reads. We get 7.69% performance degradation, compared to Slacker which gives 12.2% improvement.

### 7.7 Sensitivity Analysis

We have also conducted extensive analysis of the sensitivity of Slacker benefits to different hardware (pages sizes, chips per channel and other parallelism parameters, read and write latency, etc.) and workload (request sizes, inter-request times, read to write ratios, etc.) parameters. In the interest of space, rather than presenting detailed results, we briefly summarize the overall observations from those experiments. Growing chip densities and higher MLC levels can worsen read/write latency, accentuating the slack. Even if technology improvements drive down latency of these operations, workload intensities would also increase in the future, continuing to stress the importance of slack exploitation techniques. Pages sizes do not have as much impact on slack exploitation for the range of realistic page sizes studied. When the hardware offered parallelism within a channel increases substantially for a given load, the request queue lengths at each chip drops, thus reducing slack. However, as the load also commensurately increases, slack exploitation continues to remain important.

[4] The 95th percentile is the value below which 95% of the response time is being found.

## 8. Concluding Remarks

We presented Slacker, a mechanism for estimating and exploiting the slack present in any sub-request while it is waiting in the queue of a flash chip. We have shown that Slacker provides fairly accurate slack time estimates with low error percentages. Slacker incorporates a simple heuristic that avoids coordinated shuffling of multiple queues, allowing incoming sub-requests at each queue to independently move ahead of existing ones with sufficient slack. This can benefit incoming requests without impacting the completion times of existing ones. The results show that Slacker gives average response time improvements of 12%, 6.5% and 8.5% for write-intensive, read-intensive and read-write balanced workloads respectively.

## Acknowledgments

We thank the reviewers for their valuable suggestions. This work is supported in part by NSF grants 1302557, 1213052, 1439021, 1302225, 1629129, 1526750, and 1629915 and a grant from Intel. Myoungsoo Jung also acknowledges grants NRF 2016R1C1B2015312/2015M3C4A7065645 and MSIP IITP-2015-R0346-15-1008.

## References


[1] Microsoft research cambridge traces. http://iotta.snia.org/traces/list/BlockIO.

[2] UMass trace repository. http://traces.cs.umass.edu.

[3] Crucial bx100 ssd. http://www.crucial.com/usa/en/storage-ssd-bx100.

[4] Open NAND flash interface specification 3.1. http://www.onfi.org/specifications.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[6] K. Arase. Semiconductor nand type flash memory with incremental step pulse programming, 1998. URL `http://www.google.com/patents/US5812457`. US Patent 5,812,457.

[7] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 217–228, Mar 2009.

[8] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[9] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *International Symposium on High-Performance Computer Architecture*, pages 266–277, 2011.

[10] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *International Symposium on Computer Architecture*, pages 279–289, 2009.

[11] D. Feitelson and L. Rudolph. Wasted resources in gang scheduling. In *5th Jerusalem Conference on Information Technology*, pages 127–136, Oct 1990.

[12] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[13] C. Gao, L. Shi, M. Zhao, C. Xue, K. Wu, and E.-M. Sha. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. In *International Conference on Mass Storage Systems and Technologies*, pages 1–11, 2014.

[14] 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 *International Conference on Supercomputing*, pages 96–107, 2011.

[15] M. Huang, Y. Wang, Z. Liu, L. Qiao, and Z. Shao. A garbage collection aware stripping method for solid-state drives. In *Asia and South Pacific Design Automation Conference*, pages 334–339, 2015.

[16] A. Huffman. NVM express 1.1a specifications. http://www.nvmexpress.org, Sep 2013.

[17] M. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Supercomputing Conference*, pages 54–54, 1997.

[18] M. Jung, E. H. Wilson, III, and M. Kandemir. Physically addressed queueing (PAQ): improving parallelism in solid state disks. In *International Symposium on Computer Architecture*, pages 404–415, Jun 2012.

[19] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir. HIOS: a host interface IO scheduler for solid state disks. In *International Symposium on Computer Architecuture*, pages 289–300, 2014.

[20] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *International Conference on Embedded Software*, pages 295–304, 2009.

[21] S. Kung. Naive PCI SSD controllers. http://www.marvell.com/storage/system-solutions/native-pcie-ssd-controller/assets/Marvell-Native-PCIe-SSD-Controllers-WP.pdf, Jan 2012.

[22] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1-3):379–396, 2002.

[23] R. Love. Kernel korner – I/O schedulers. *Linux Journal*, 2004 (118):10–, 2004.

[24] Micron Technology, Inc. NAND flash memory MLC data sheet, MT29E512G08CMCCBH7-6 NAND flash memory. http://www.micron.com/.

[25] E. H. Nam, B. Kim, H. Eom, and S. L. Min. Ozone (O3): an out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, 60(5):653–666, 2011.

[26] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[27] C. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee. Exploiting internal parallelism of flash-based SSDs. *IEEE Computer Architecture Letters*, 9(1):9–12, Jan 2010.

[28] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. Ftl design exploration in reconfigurable high-performance ssd for server applications. In *23rd International Conference on Supercomputing*, pages 338–349, 2009.

[29] A. Tavakkol, M. Arjomand, and H. Sarbazi-Azad. Unleashing the potentials of dynamism for page allocation strategies in SSDs. In *ACM International Conference on Measurement and Modeling of Computer Systems*, pages 551–552, 2014.

[30] G. Wu and X. He. Reducing SSD read latency via nand flash program and erase suspension. In *10th USENIX Conference on File and Storage Technologies*, Feb 2012.

[31] Q. Zhang, D. Feng, F. Wang, and Y. Xie. An efficient, QoS-aware I/O scheduler for solid state drive. In *International Conference on High Performance Computing and Communications*, pages 1408–1415, 2013.