# A Semi-Preemptive Garbage Collector for Solid State Drives

Junghee Lee*, Youngjae Kim†, Galen M. Shipman†, Sarp Oral†, Feiyi Wang†, and Jongman Kim*

*Electrical and Computer Engineering, Georgia Institute of Technology
†National Center for Computational Sciences, Oak Ridge National Laboratory
{jlee36, jkim}@ece.gatech.edu, {kimy1, gshipman, oralhs, fwang2}@ornl.gov

***Abstract*—NAND flash memory is a preferred storage media for various platforms ranging from embedded systems to enterprise-scale systems. Flash devices do not have any mechanical moving parts and provide low-latency access. They also require less power compared to rotating media. Unlike hard disks, flash devices use out-of-update operations and they require a garbage collection (GC) process to reclaim invalid pages to create free blocks. This GC process is a major cause of performance degradation when running concurrently with other I/O operations as internal bandwidth is consumed to reclaim these invalid pages. The invocation of the GC process is generally governed by a low watermark on free blocks and other internal device metrics that different workloads meet at different intervals. This results in I/O performance that is highly dependent on workload characteristics. In this paper, we examine the GC process and propose a semi-preemptive GC scheme that can preempt on-going GC processing and service pending I/O requests in the queue. Moreover, we further enhance flash performance by pipelining internal GC operations and merge them with pending I/O requests whenever possible. Our experimental evaluation of this semi-preemptive GC sheme with realistic workloads demonstrate both improved performance and reduced performance variability. Write-dominant workloads show up to a 66.56% improvement in average response time with a 83.30% reduced variance in response time compared to the non-preemptive GC scheme.**

## I. Introduction

Hard disk drives (HDD) are the primary storage media for large-scale storage systems for a few decades. HDD manufacturers have provided slow but steady improvement in performance while lowering the price (in terms of dollar per GB) with breakthrough technology enhancement such as perpendicular recording [34], [26], [7]. However, HDD technology has some drawbacks, such as the lack of spatial/temporal locality that limit the performance. HDD is a mechanical device where the heads must be moved back and forth across the tracks over the platters for requests with significant randomness. To decrease access times of these random requests rotation speeds have increased but at the cost of higher power consumption, increasing internal air-temperature beyond 100°F [23], [12], [21].

With advancements in the semi-conductor technology, NAND flash memory based solid-state drives (SSD) have become more prevalent in the storage marketplace. Unlike HDDs, SSDs do not have mechanically moving parts. SSDs offer several advantages over HDDs such as lower access latency, higher resilience to external shock and vibration, lower power consumption which results in lower operating temperatures. Other benefits include lighter weight and flexible designs in terms of device packaging. Moreover, recent reductions in cost (in terms of dollar per GB) have accelerated the adoption of SSDs in a wide range of application areas from consumer electronic devices to enterprise-scale storage systems.

One interesting feature of Flash technology is the restriction of write locations. Target address for a write operation should be empty [1], [11]. When the target address is not empty the invalid contents must be erased for the write operation to succeed. Erase operations in NAND flash are nearly an order of magnitude slower than write operations. Therefore, flash-based SSDs use out-of-place writes unlike in-place writes on HDDs. To reclaim stale pages and to create space for writes, SSDs use a Garbage Collection (GC) process. The GC process is a time-consuming task since it copies non-stale pages in blocks into the free storage pool and then erases the blocks that do not store valid data. A block erase operation takes approximately 1-2 milliseconds [1]. Considering that valid pages in the victim blocks (to be erased) need to be copied and then erased, GC overhead can be quite significant.

GC can be executed when there is sufficient idle time (i.e. no incoming I/O requests to SSDs) with no impact to device performance. Unfortunately, prediction of idle times in I/O workloads is challenging and some workloads may not have sufficiently long idle times. In a number of workloads incoming requests may be bursty and an idle time can not be effectively predicted. Under this scenario the queue-waiting time of incoming requests will increase. Server-centric enterprise data center and high-performance computing (HPC) environment workloads often have bursts of requests with low interarrival time [22], [11]. Examples of enterprise workloads that exhibit this behavior include online-transaction processing applications, such as OLTP and OLAP [3], [23]. Furthermore, it has been found that HPC file systems are stressed with write requests of frequent and periodic checkpointing and journaling operations [29]. In our study of HPC I/O workload characterization of the Spider storage system at Oak Ridge National Laboratory, we observed that the bandwidth distributions are heavily long-tailed [22].

In this paper, we propose a semi-preemptive garbage collection scheme (PGC) that enables the SSDs to provide

978-1-61284-368-1/11/$26.00 ©2011 IEEE

sustainable bandwidths in the presence of these heavily bursty and write-dominant workloads. We will show that the PGC can achieve higher bandwidth over the non-preemptive GC scheme by preempting an ongoing GC process and servicing incoming requests. We carry out a detailed and systematic simulated performance study using the Microsoft Research (MSR) SSD simulator [1].

This paper makes the following contributions:

- We empirically observe the performance degradation due to the GC process on commercially-off-the-shelf (COTS) SSDs for bursty write-dominant workloads. Based on our observations, we propose a novel semi-preemptive garbage collection scheme that can easily be implementable within SSDs.
- We identify preemption points that can minimize the preemption overhead. We use a state diagram to define each state and state transitions that result in preemption points. For experimentation we enhance the existing well-known SSD simulator [1] to support our PGC algorithm and show an improvement of up to 66.56% in average response time for overall realistic applications.
- We investigate further I/O optimizations to enhance the performance of SSDs with PGC by *merging incoming I/O requests with internal GC I/O requests* and *pipelining these resulting merged requests*. The idea behind this technique is to merge internal GC I/O operations with I/O operations pending in the queue. The pipelining technique inserts the incoming requests into GC operations to reduce the performance impact of the GC process. Using these techniques we can further improve the performance of SSDs with PGC enabled by up to 13.69% for the Cello benchmark.
- We conduct not only a comprehensive study with synthetic traces by varying I/O patterns (such as request size, inter-arrival times, sequentiality of consecutive requests, read and write ratio, etc.) but also a realistic study with server workloads. Our evaluations with PGC enabled SSD demonstrate up to a 66.56% improvement in average I/O response time and an 83.30% reduction in response time variability.

The rest of this paper is organized as follows. Section II presents a background of SSD technology and the motivation for developing the PGC scheme. Section III provides an overview of the PGC scheme including further optimizations such as live merge and pipelining of GC operations with arriving I/Os. Section IV describes the workloads and metrics used in the evaluation, along with details of simulation configuration, and the results of our study. Section V presents related work. Section VI concludes the paper.

## II. Background and Motivation

### A. Background

*1) Physical characteristics of flash memory:* Unlike rotating media (HDD) and volatile memories (DRAM) which only need read and write operations, flash memory-based storage devices require an erase operation [28]. Erase operations are performed at the granularity of a block which is composed of multiple pages. A page is the granularity at which reads and writes are performed. Each page on flash can be in one of three different states: (i) *valid*, (ii) *invalid* and (iii) *free/erased*. When no data has been written to a page, it is in the erased state. A write can be done only to an erased page, changing its state to valid. Erase operations (on average 1-2 ms) are significantly slower than reads or writes. Therefore, out-of-place writes (as opposed to in-place writes in HDDs) are performed to existing free pages along with marking the page storing the previous version invalid. Additionally, write latency can be higher than the read latency by up to a factor 10.

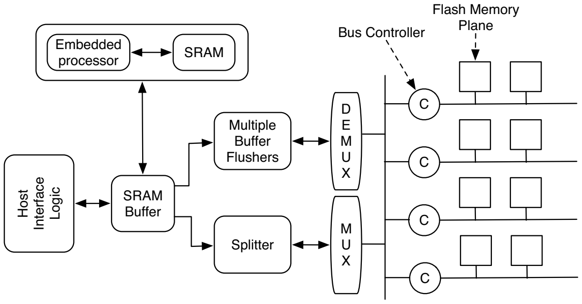


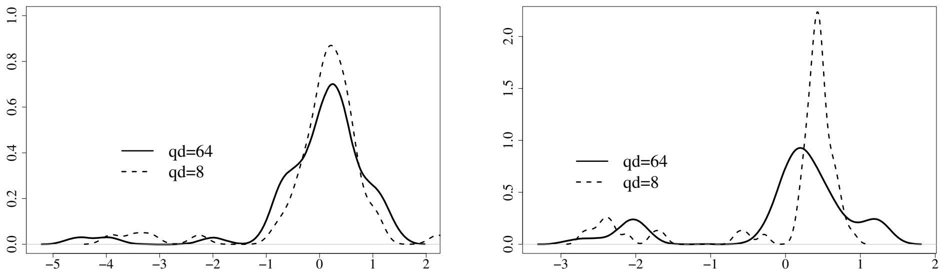
Fig. 1. Flash-based Solid State Disk Drive [27].

The lifetime of flash memory is limited by the number of erase operations on its cells. Each memory cell typically has a lifetime of $10^3$-$10^9$ erase operations [10]. *Wear-leveling* techniques are used to delay the wear-out of the first flash block by spreading erases evenly across the blocks [17], [5].

*2) NAND flash based SSDs:* Figure 1 describes the organization of internal components in a flash-based SSD [27]. It provides a host interface (such as Fiber-Channel, SATA, PATA, and SCSI) to appear as a block I/O device to the host computer. The main controller is composed of two units, the processing unit (such as an ARM7 processor) and fast access memory (such as SRAM). The virtual-to-physical mappings are processed by the processor and the data-structures related to the mapping table are stored in SRAM in the main controller. The software module related to this mapping process is called the Flash Translation Layer (FTL). A part of SRAM can be also used for caching data.

A storage pool in an SSD is composed of multiple flash memory planes. The planes are implemented in multiple dies. For example, the Samsung 4 GB flash memory has two dies. A die is composed of four planes, each of size 512 MB [1]. A plane consists of a set of blocks. The block size can vary (64KB, 128KB, 256KB, etc.) depending on the memory manufacturer. The SSD can be implemented using multiple planes. SSD performance can be enhanced by interleaving requests across the planes, which is achieved by a multiplexer and demultiplexer between SRAM buffer and flash memories [1].

*3) Flash Translation Layer:* The Flash Translation Layer (FTL) is a software layer that translates logical addresses



(a) SSD(A)

(b) SSD(B)

Fig. 2. Throughput variability comparison for SSDs with increasing number of the number of outstanding requests. Y-axis represents normalized frequency. *qd* denotes queue depth. Higher *qd* means requests are more bursty and intense in their arrival rate.

from the file system into physical addresses on a flash device. The FTL helps in emulating flash as a normal block device by performing out-of-place updates thereby hiding the erase operations in flash. Due to out-of-place updates, flash devices must clean stale data for providing free space (similar to log-structured file system [33]). This cleaning process is known as garbage collection (GC). During an ongoing GC process incoming requests are delayed until the completion of the GC if their target is the same flash chip that is busy with GC. Current generation SSDs use a variety of different algorithms and policies for GC that are vendor specific. It has been empirically observed that GC activity is directly correlated with the frequency of write operations, amount of data written, and/or the free space on the SSD [6]. GC process can significantly impede both read and write performance, increasing queueing delay.

The FTL mapping table is stored in a small, fast SRAM. FTLs can be implemented at different granularities in terms of the size of a single entry capturing and address space in the mapping table. Many FTL schemes [8], [24], [19], [25] and their improvement by write-buffering [20] have been studied. A recent page-based FTL scheme called DFTL [11] utilizes temporal locality in workloads to overcome the shortcomings of the regular page-based scheme by storing only a subset of mappings (those likely to be accessed) on the limited SRAM and storing the remainder on the flash device itself. Also, there are several works in progress on the optimization of buffer management in NAND flash based environments [30], [16].

### B. Motivation

*1) Experimental setup:* We use various commercially-off-the-shelf (COTS) SSDs for experiments. Table I shows their detail specifications. We selected the Super Talent 128 GB SSD [38] as a representative of multi-level cell (MLC) SSDs and the Intel 64 GB SSD [15] as a representative of single-level cell (SLC) SSDs. We denote the SuperTalent MLC, and Intel SLC devices as SSD(A), and SSD(B) in the remainder of this study, respectively. All experiments were performed on a single server with 24 GB of RAM and an Intel Xeon Quad Core 2.93GHz CPU [14]. The operating system was Linux with a Lustre-patched 2.6.18-128 kernel. The *noop* I/O scheduler with FIFO queueing was used [32].

TABLE I
CHARACTERISTICS OF SSDS USED IN OUR EXPERIMENTS.

| Label | SSD(A) | SSD(B) |
|---|---|---|
| Company | Super-Talent | Intel |
| Model | FTM28GX25H | SSDSA2SH064G101 |
| Type | MLC | SLC |
| Interface | SATA-II | SATA-II |
| Capacity (GB) | 120 | 64 |
| Erase (#) | 10-100K | 100K-1M |
| Power (W) | 1-2 | 1-2 |

We examine the I/O bandwidth of individual COTS SSDs for write-dominant workloads. To measure the I/O performance we use a benchmark that exploits the *libaio* asynchronous I/O library on Linux. Libaio provides an interface that can submit one or more I/O requests in one system call *iosubmit()* without waiting for I/O completion. It also can perform reads and writes on raw block devices. We used the direct I/O interface to bypass the operating system I/O buffer cache by setting the *O-DIRECT* and *O-SYNC* flags in the file *open()* call.

*2) Performance degradation of SSDs by GCs:* Figure 2 illustrates the impact of GC on I/O bandwidth. In order to compare the bandwidth variability of individual SSDs for different arrival rates of requests, we measured I/O bandwidth for 512KB write requests by varying I/O queue depth (QD). We normalize the I/O bandwidth of each configuration with a Z-transform [18] and then curve-fitted and plotted their density functions. We observe that the performance variability increases with respect to the arrival rate of requests. The SSD is not able to guarantee bandwidth in the face of these workloads that are characterized by bursty arrival of I/O requests. This performance variability is attributable to the GC process. While the inter-arrival time would allow for some garbage collection during I/O idle time, the GC process is unable to take advantage of this. This insight led to our design and development of a preemptive garbage collector.

The basic idea of the proposed technique is to service an incoming request even while GC is running. However, allow-

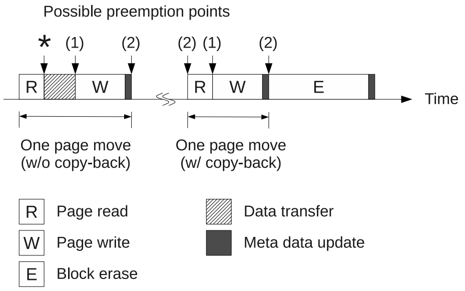


Fig. 3. Description of operation sequence during garbage collection.

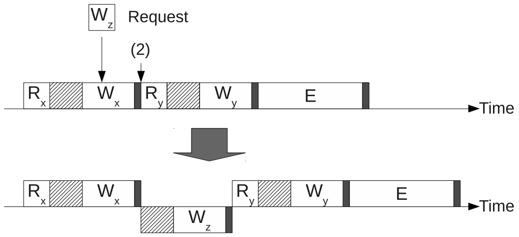


Fig. 4. A semi-preemption. R, W, and E denote read, write, and erase operations, respectively. The subscripts indicate the page number accessed.

ing preemption of GC at any time may incur an extra context switching overhead. Thus, we only allow semi-preemption at certain points.

## III. Preemptive Garbage Collection

### A. Semi-Preemptive GC

Figure 3 shows a typical garbage collection process when a page-based logical-to-physical page mapping is used. Although we explain our proposed technique based on the page-based mapping scheme, it can be easily applied to block-based or hybrid techniques because GC preemption is orthogonal to address mapping schemes.

Once a victim block is selected during GC, all the valid pages in that block are moved into an empty block and the victim block is erased. A moving operation of a valid page can be broken down to *page read*, *data transfer*, *page write*, and *meta data update* operations. If both the victim and the empty block are in the same plane, the data transfer operation can be omitted by copy-back operation [1] if the flash device support the operation.

We identify two possible preemption points in the GC sequence marked as (1) and (2) in Figure 3. Preemption point (1) is within a page movement and (2) is in-between page movement. Preemption point (1) is just before a page is written and (2) is just before a new page movement begins. We may also allow preemption at the point marked with a (*), but the resulting operations are the same as those of (1) as long as the preemption during data transfer stage is not allowed. Although preemption point (2) can service any kind of incoming request, (1) cannot because the registers are already occupied by the previous read page operation. At preemption point (1) an incoming request of reading a page cannot be serviced.

Figure 4 illustrates the semi-preemption scheme. The subscripts of *R* and *W* indicate the page number accessed. Suppose that a write request on page $z$ arrives while writing page *x* during GC. With a conventional non-preemptive GC, the request should be serviced after GC is finished. If GC is fully preemptive, the incoming request may be serviced immediately. To do so, the on-going writing process on *x* should be canceled first. This will incur an additional write operation on *x* after servicing the incoming request on page $z$. In PGC, the preemption occurs only at preemption points. As shown in the bottom of Figure 4, the incoming request on page $z$ is inserted at preemption point (2).

If more preemption points were allowed, the response time of incoming requests would be shortened further but may incur excessive overhead. Page read, write, and erase operations marked as R, W, and E, respectively, are not ”preemptive-friendly” as preemption of these types of operations are not supported by the flash device. To preempt them, they would be canceled first and then re-executed again after the preemption. Moreover, preempting GC at any time requires an interrupt upon receipt of the incoming request. Each such interrupt incurs context switching overhead.

Our proposed semi-preemption does not require an interrupt. Due to the small number of preemption points it can be implemented by a polling mechanism. At every preemption point, the GC process looks up the request queue. This may involve a function call, a small number of memory accesses to look up the queue, and a small number of conditional branches. Assuming 20 instructions and 5 memory access per looking up, $10ns$ per instruction $(100MHz)$, $80ns$ per memory access, it takes $600ns$. One page move involves at least one page read which takes $25\mu s$ and one page write which takes $200\mu s$ [1]. Since there are two preemption points per one page move, the overhead of looking up the queue per one page move can be estimated as $1.2\mu s/225\mu s = 0.53\%$.

To resume GC after servicing the incoming request, the context of GC needs to be stored. The context to be stored at the preemption points (1) and (2) is very small. In case of (1), the victim block and page information needs to be stored in the registers. In case of (2), only the victim block needs to be stored. Because the meta data is already updated, the incoming request can be serviced based on the mapping information. Thus, the memory overhead for preempting GC is very small and negligible.

### B. Merging Incoming Requests into GC

While servicing incoming requests during GC, we can optimize the performance even further. If the incoming request happens to access the same page in which the GC process is attending, it can be merged. Figure 5 illustrates a situation where the incoming request of read or write on page *x* arrives while page *x* is being read by the read stage of GC. The read request can be directly serviced from the registers and the write request can be merged by updating data in the registers.

In case of copy-back operations, the data transfer is omitted, but to exploit merging, it cannot be omitted. As for the read request, data in the register should be transferred to service the

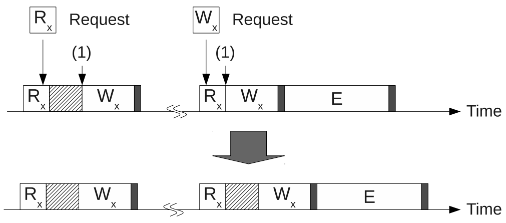


Fig. 5. Merging an incoming request to GC.

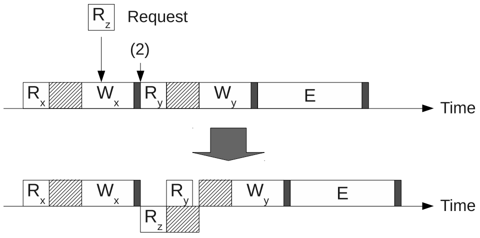


Fig. 6. Pipelining an incoming request with GC.

read request. For the write request, the requested data should be written to the register.

We can increase the chance of merging by re-ordering the sequence of pages to be moved from the victim block. For example, pages *x*, *y*, and *z* are supposed to be moved in that order. For GC, the order of pages to be moved does not matter. Thus, when a request on page *z* arrives, it can be reordered as *z*, *x*, and *y*.

### C. Pipelining Incoming Requests with GC

The response time can be further reduced even if the incoming request is on a different page. To achieve this we take advantage of the internal parallelism of the flash device. Depending on the type of the flash device, internal parallelism and its associated operations can be different. In this paper, we consider pipelining [31] as an example. If two consecutive requests are of the same type, i.e. read after read, or write after write, these two requests can be pipelined.

Figure 6 illustrates a case where an incoming request is pipelined with GC. As an example, lets assume that there is a pending read operation on page *z* at the preemption point (2) where a page read on page *y* is about to begin. Since both operations are read, they can be pipelined. However, if the incoming request is write, they can not be pipelined at preemption point (2) as two operations need to be issued at (2) and they are not of the same type. In this case, the incoming request should be inserted serially as shown in Figure 4.

It should be noted that pipelining is only an example of exploiting the parallelism of an SSD. An SSD has multiple packages, where each package has multiple dies, and each die has multiple planes. Thus, there are various opportunities to insert an incoming requests into GC as means of exploiting parallelism at different levels. We may interleave servicing requests and moving pages of GC in multiple packages or issue a multi-plane command on multiple planes [31]. According to the GC scheme and the type of operations the flash device supports, there are many instances of exploiting parallelism.

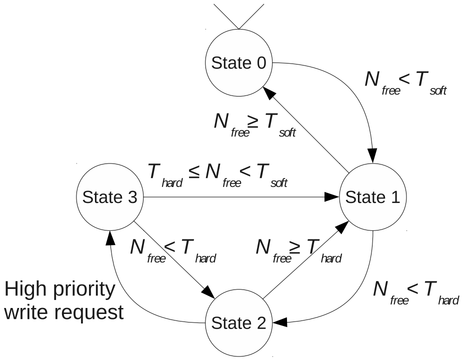


Fig. 7. State diagram of semi-preemptive GC.

### D. Level of Allowed Preemption

The drawback of preempting GC is that the completion time can be delayed which may incur a lack of free blocks. If the incoming request does not consume free blocks, it can be serviced without depleting the free block pool. However, there may be a case where the incoming request is write whose priority is high but there are not enough free blocks. In such a case, GC should be finished as soon as possible.

Based on these observations, we identify four states of GC:

- **State 0** GC execution is not allowed.
- **State 1** GC can be executed but all incoming requests are allowed.
- **State 2** GC can be executed but all free block consuming incoming requests are prohibited.
- **State 3** GC can be executed but all incoming requests are prohibited.

Conventional non-preemptive GC has only two states: 0 and 3. Generally, switching from State 0 to State 3 is triggered by threshold or idle time detection. Once the number of free blocks falls below a pre-defined threshold or an idle time is detected, GC is triggered. We use this threshold for switching from State 0 to 1 and from State 1 to 2. We call the conventional non-preemptive threshold as *soft* but in our proposed design the system allows for the number of free blocks to fall below the soft threshold. We define a new threshold called *hard* which prevents a system crash by running out of free blocks. Switching from State 2 to 3 is triggered by the type of incoming requests. If the incoming request is write whose priority is high, it switches to State 3. How high the priority should be depends on requirements of the system.

Figure 7 illustrates the state diagram. If the number of free blocks ($N_{free}$) becomes less than the soft threshold ($T_{soft}$), the state is changed from 0 to 1. If $N_{free}$ recovers to be larger than $T_{soft}$, then the system switches back to state 0. If $N_{free}$ becomes even less than the hard threshold ($T_{hard}$), the system switches to State 2 or remains in State 1 otherwise. In state 2, the system will move to State 1 if $N_{free}$ becomes larger than $T_{hard}$. If there is an incoming request whose priority is high, the system switches to State 3. While in State 3, after completing current GC and servicing the high priority request, the system will switch to State 1 or 2 according to $N_{free}$.

## IV. Experimental Results

### A. Experimental Setup

We evaluate the performance of the PGC scheme using Microsoft Research's SSD simulator [1]. MSR SSD simulator is event-driven and based on the Disksim 4.0 [2] simulator. MSR SSD simulator has been used in several SSD related researches [31], [36]. In this paper, we simulated a NAND flash based SSD. SSD specific parameter values used in the simulator are given in Table II.

TABLE II
Parameters of SSD model.

| Parameter | Value |
|---|---|
| Total capacity | 32 GB |
| Reserved free blocks | 15 % |
| Minimum free blocks | 5 % |
| Cleaning policy | Greedy |
| Flash chip elements | 8 |
| Planes per element | 8 |
| Blocks per plane | 2048 |
| Pages per block | 64 |
| Page size | 4 KB |
| Page read latency | 0.025 ms |
| Page write latency | 0.200 ms |
| Block erase latency | 1.5 ms |

To conduct a fair performance evaluation of our proposed PGC algorithm we fill the entire SSD with valid data prior to collecting performance information. Filling the entire SSD ensures that GC is triggered as new write requests arrive during our experiments. Specifically, for GC, we use a greedy algorithm that is designed to minimize the overhead of GC. The greedy algorithm selects a victim block to be erased whose number of valid pages is minimal. The more valid pages there are in the victim block, the longer it takes for GC to complete as the GC process needs to move more pages.

Our preemptive GC algorithm can be applied to any existing GC schemes, such as idle-time or reactive. In the idle-time GC scheme the GC process is triggered when there are no new incoming requests and all queued requests are already serviced. In the reactive scheme GC is invoked based on the number of available free blocks without regard the incoming request status. If the number of available free blocks is less than the set threshold then the GC process is triggered, otherwise, it continues servicing request. The reactive GC scheme is the default in the MSR SSD simulator and we use it as our baseline (non PGC) GC scheme. The lower bound of the threshold in our simulations is set as the 5% of available free blocks. On-going GC is never preempted in the baseline GC scheme in our simulations.

*1) Workloads:* We use a mixture of real-world and synthetic traces to study the efficiency of our semi-preemptive garbage collection scheme. We use synthetic workloads with varying parameters such as request size, inter-arrival time of requests, read access probability, and sequentiality probability in access.[1] The default values of the parameters that we use in our experiments are shown in Table III. We use an exponential distribution for varying request sizes and a Poisson distribution for varying inter-arrival time of requests. We vary one parameter while other parameters are fixed.

TABLE III
Default parameters of synthetic workloads.

| Parameter | Value |
|---|---|
| Request size | 32 KB |
| Inter-arrival time | 3 ms |
| Probability of sequential access | 0.4 |
| Probability of read access | 0.4 |

TABLE IV
Characteristics of realistic workloads.

| Workload | Average Request Size (KB) | Read (%) | Arrival Rate (IOP/s) |
|---|---|---|---|
| Financial [37] | 7.09 | 18.92 | 47.19 |
| Cello [35] | 7.06 | 19.63 | 74.24 |
| TPC-H [39] | 31.62 | 91.80 | 172.73 |
| OpenMail [13] | 9.49 | 63.30 | 846.62 |

We use four commercial I/O traces, whose characteristics are given in Table IV. We use write dominant I/O traces from an OLTP application running at a financial institution made available by the Storage Performance Council (SPC), referred to as the Financial trace, and from Cello99, which is a disk access trace collected from a time-sharing server exhibiting significant writes which was running the HP-UX operating system at Hewlett-Packard Laboratories. We also examine two read-dominant workloads. Of these two, TPC-H is a disk I/O trace collected from an OLAP application examining large volumes of data to execute complex database queries. Finally, a mail server I/O trace referred as OpenMail is evaluated.

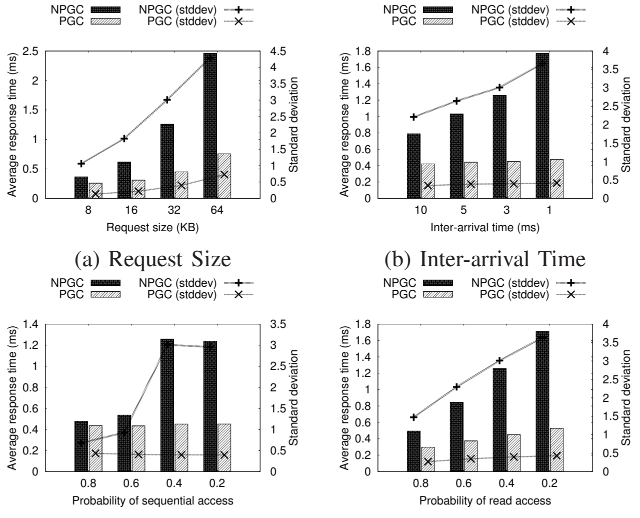
*2) Metrics:* While the device service time captures the overhead of GC, it does not include queueing delays for pending requests. Additionally, using an average service time does not capture response time variances. In this study we utilize (i) the system service response time measured at the block device queue and (ii) the variance in response times. Our measurement captures the sum of the device service time and the additional time spent waiting for the device (queueing delay) to begin to service the request.

*3) GC schemes:* The following garbage collection schemes are evaluated:

- **NPGC** A non-preemptive garbage collection scheme.
- **PGC** A semi-preemptive garbage collection scheme.
- **PGC+Pipeline** A pipelining technique enabled PGC.

Although we implemented the I/O merging technique on PGC and evaluated it against PGC and PGC+Pipeline we found the improvement to be minimal therefore we do not include the I/O merging technique results in the paper.

[1] If a request starts at the logical address immediately following the last address accessed by the previously generated request, we consider it a sequential request; Otherwise, we classify it as a random request.



(a) Request Size

(b) Inter-arrival Time

(c) Sequentiality

(d) Read Ratio

Fig. 8. Performance improvements of preemptive GC for synthetic workloads. Average response times and standard deviations are shown with different parameters of synthetic workloads.

### B. Results

*1) Performance analysis for synthetic workloads:* GC may have to be performed while requests are arriving. Recall that GC is not preemptable in the baseline GC scheme and incoming requests during GC are delayed until the on-going GC process is complete. Figure 8 shows the performance improvements when enabling GC preemption.

*a) Request size:* Figure 8(a) shows the improvements of performance and variance by PGC for different request sizes In this experiment, we vary the request size as 8, 16, 32, and 64 KB. These values are chosen because the average request size of realistic workloads is between 7 and 31 KB, as given in Table IV. For a small request size (8 KB) we see the improvement in response time by 29.44%. Furthermore, the variance of average response times decreases by 87.31%. As the request size increases, we see further improvements. For a large request (64 KB), the response time decreases by up to 69.21% while its variance decreases by 83.03%.

*b) I/O arrival rate:* Similar to the improvement with respect to varying request sizes, we also see an improvement with respect to varying the arrival rate of I/O requests. Typical response time of a request on a page is less than 1 ms without GC while it can be as high as 3-4ms when the page request is queued up due to GC. Based on this observation, we vary the inter-arrival time between 1 and 10 ms in our experiments. In Figure 8(b), it can be seen that PGC is minimally impacted by intense arrival rate. In contrast, the system response times and their variances for the baseline (NPGC) increase with respect to the request arrival rate.

*c) Sequential access:* Random workloads (where consecutive requests are not next to each other in terms of their access address) are known to be likely to increase the fragmentation of SSD, causing a GC overhead increase [20], [11]. We experiment with PGC and NPGC by varying the sequentiality of requests. Figure 8(c) illustrates the results. As can be seen, NPGC exhibits a substantial increase in system response time and its variance for a 60% sequential workload while PGC performance levels remain constant for all levels of sequentiality.

*d) Write percentage:* Writes are slower than reads in SSDs because flash page writes are slower than reads (recall unit access latency for reads and writes, 25us and 200us, respectively) and GC can incur further delays. In Figure 8(d), we see the improvement of PGC as the percentage of writes within the workload increases. Overall, we observe that PGC exhibits a marginal increase in response time and variance compared to the NPGC scheme. For example, PGC performance slows down by only 1.77 times for an increase of writes in workloads (from 80% to 20% of reads) while NPGC slows down by 3.46 times.

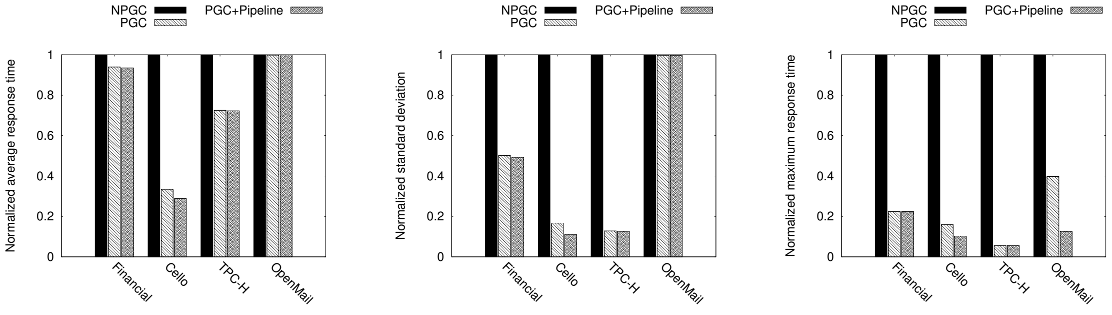
*2) Performance analysis for realistic server workloads:* Figure 9 presents the improvement of system response time and variance over time for realistic workloads. For write-dominant workloads, we see an improvement in average response time by 6.05% and 66.56% for Financial and Cello, respectively (refer to Figure 9(a)). Figure 9(b) shows a substantial improvement in the variance of response times. PGC reduces the performance variability by 49.82% and 83.30% for each of the workloads. In addition to the improvement in performance variance, we observe that PGC can further reduce the maximum response time of NPGC by 77.59% and 84.09% for Financial and Cello traces as illustrated in Figure 9(c).

For the OpenMail trace PGC does not show a significant improvement for performance and variance, as we expected for read-dominant traces. However, PGC reduces the maximum response time by 60.26%. Interestingly for TPC-H, although it is a read dominant trace, we observe a substantial improvement for performance and variance. TPC-H is a database application. The disk trace includes a phase of application run that inserts tables into a database, which is shown as a series of large write requests (around 128 KB) for database insert operations.

Moreover, we observe further improvement by the pipelining technique on PGC in the Figure 9. For Cello, an improvement is observed in the average response time of PGC by 13.69% and its performance variance by 33.53%. For OpenMail, our proposed algorithm can obtain a significant reduction of maximum response time, as much as 68.30%.

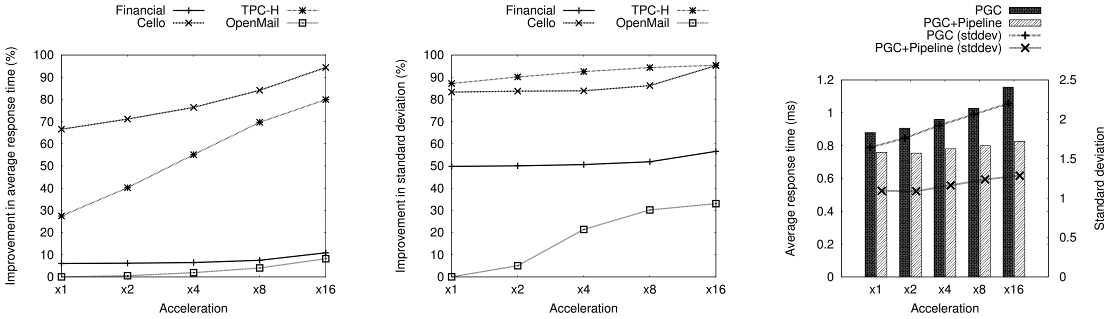
*3) Exploring a wider range of workload characteristics:* We have seen the improvement of performance and variance from our experiments with realistic workloads. Next we determine if PGC is robust enough to sustain periods of increased I/O intensity. For this experiment, we increased the I/O arrival rates of the original traces. Without modifying sizes and offset addresses of requests, we shortened the inter-arrival times of subsequent requests by factors of 2, 4, 8, and 16.

As shown in figures 10(a) and (b), with respect to increasing arrival rate, average response time and variance also improve. In particular, improvements in response times can be seen for write-dominant workloads (Financial and Cello) compared



(a) Average Response Time (b) Variance of Response Times (c) Maximum Response Time

Fig. 9. Performance improvements of PGC and PGC+Pipelining for realistic server workloads.



(a) Average Response Time (b) Variance of Response Times (c) GC pipelining for Cello

Fig. 10. Scalability tests by increasing the arrival rate of I/O requests.

to read-dominant workloads in Figure 10(a). For TPC-H, we see a gradual improvement for the performance variability. Overall, we observe that PGC can increase the performance and improve the variance up to 90% for a 16 times more bursty workload (i.e. the I/O arrival rate is increased by 16 times)

Figure 10(c) shows further improvements of the GC pipelining technique. In this figure, improvements in average response time and its variance for Cello can be clearly observed. In particular, we observe that the gaps of performance and variance are widened as the arrival rate of I/O requests increases. In other words, the GC pipelining technique makes PGC enabled SSDs robust enough to provide a sustained level of performance.

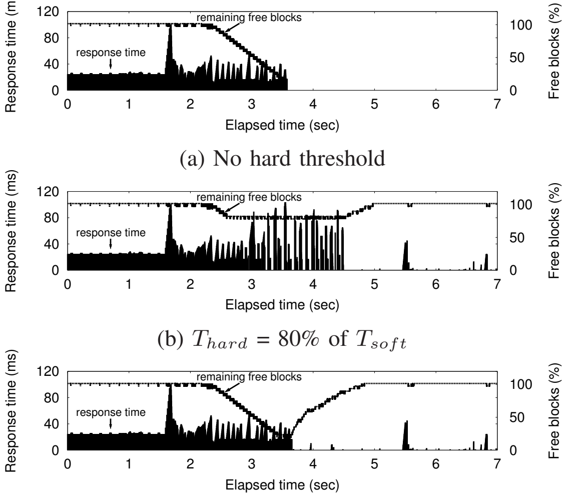
*4) Handling starvation of free blocks:* Continuous GC preemption can cause starvation of free blocks. Thus, we develop a mechanism that can avoid a situation where an entire system becomes completely unserviceable because no free blocks are available. For this, we implement our PGC algorithm with a hard limit of available free blocks. Our algorithm now has two thresholds, one is for triggering the GC process and the other is for stopping preemption. Once the number of free blocks reaches $T_{hard}$, SSD stops GC preemption. A hard limit ($T_{hard}$) is set for a lower bound of the number of free blocks available in SSD.

To evaluate the effect of our extra threshold, we use an amplified Cello trace where the arrival rate of I/O requests are 16 times higher and the average request size is set to be 300 KB on average. In Figure 11(a), we see the situation where there are no free blocks left due to continuous GC preemption and the SSD is not available to service the I/O requests. It captures a zoomed-in region for 7 seconds of entire simulation run. The remaining free blocks indicate the ratio of the number of available free blocks over the minimum number of free blocks. The minimum number of free blocks corresponds to the soft threshold ($T_{soft}$) which is 5% of the total number of blocks as shown in Table II. On the contrary, in Figure 11(b)(c), we see that the SSD handles the starvation of free blocks in the SSD by adjusting $T_{hard}$. We see that the lower $T_{hard}$ shows better response time while it exhausts more free blocks.

Since there exists a trade-off between the number of free blocks and response times, we evaluate the impact of performance in terms of response time according to $T_{hard}$. Figure 12 shows the cumulative distribution function of response time for different $T_{hard}$. As we lower $T_{hard}$, we see overall response time improve. For example, we observe 18% improvement when we lower $T_{hard}$ from 80% to 20% of $T_{soft}$.

## V. Related Work

Preemptive GC is discussed in [4] as a possible method to meet the constraints of a real-time system equipped with NAND flash. They proposed creation of a GC task for each real-time task so that the corresponding GC task can prepare enough free blocks in advance. In a real-time environment both GC tasks and real-time tasks need to be preemptive. However, since NAND flash operations can not be interrupted, these are

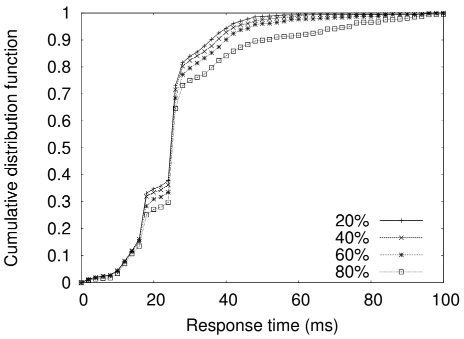


(a) No hard threshold

(b) $T_{hard}$ = 80% of $T_{soft}$

(c) $T_{hard}$ = 20% of $T_{soft}$

Fig. 11. Impact of hard threshold.



Avg. Resp. Times = {23.8, 24.4, 25.7, 29.1}

Fig. 12. A trade-off between response time and hard limit. The average response times (in ms) are shown below each graph in the order of increasing the percentage of hard limit ($T_{hard}$).

defined as atomic operations. In contrast, our work provides a comprehensive study on the impact of the preemptive GC in an SSD environment (compared to real-time environment) and we emphasize on optimizing performance by exploiting the internal parallelism of the NAND flash device (e.g. the multi-plane command and pipelining [31]).

Multiple planes on the same die can execute the same type of command concurrently, which is called a multi-plane command. Pipelining can be exploited if consecutive commands are of the same type. While transferring data from/to registers, a NAND flash operation can be executed. Exploiting the internal parallelism of the NAND flash gives us the ability to further optimize the performance of PGC. For a non-PGC SSD, pipelining implies reordering the sequence for consecutive incoming requests so that there are as many consecutive requests of the same type as possible. In contrast, we pipeline an incoming request with an operation of GC. If pipelining is not possible, we insert the request into GC in a serial manner. We reorder neither the sequence of incoming requests nor the sequence of GC operations.

In the HDD domain, semi-preemptive I/O has been evaluated [9] and its extension to RAID arrays also has been studied [9] by allowing preemption of on-going I/O operations to service a higher-priority request. To enable preemption, each HDD access operation (seek, rotation, and data transfer) is split into distinct operations. In-between these operations, a higher-priority I/O operation can be inserted. In the case of PGC, we allow preemption of GC to service any incoming request. We split GC operations into distinct operations and insert incoming requests in between them. In addition, we provide further optimization techniques while inserting requests.

Similar to GC scheme in flash based SSDs, Java GC reclaims obsolete memory space to produce available free memory space; it splits the heap regions of memory space into two memory regions (young and old generations) and develops the GC algorithms that can efficiently use the memory space and effectively perform GC. Our preemptive GC scheme is different from Java GC schemes in that our basic idea is to split incoming I/O requests in a series of page-level operations and insert operations appropriately in-between page-level operations of GC. Our preemption idea not only can work with any GC algorithm, but also requires only minimal modification to these general GC algorithms.

## VI. Conclusions

In this paper, we argue that existing SSD technology has a performance variability problem due to GC overheads and the problem can become severe for bursty write-dominant workloads. As a solution to this problem we have developed a preemptive garbage collector (PGC) that allows on-going GC preemption to prioritize serving incoming requests over regular GC activity. Our experimental evaluations demonstrate the efficiency of PGC as it offers (i) improved performance, (ii) reduced performance variability, and (iii) increased system robustness. Moreover, we further improved the PGC scheme by merging or pipelining incoming requests with internal I/O operations for GC. We have shown that the average response time and the variability of response times of realistic workloads can be reduced by up to 66.56% and 83.30%, respectively.

As future work, we plan to evaluate PGC with idle-time based GC schemes. Also we will analyze the performance of PGC for RAID arrays of SSDs. We expect that minimizing GC overheads in individual SSDs will not only provide improved aggregate RAID level bandwidth but also decrease the variability of observed aggregate throughput of the RAID array.

## Acknowledgment

We would like to thank the anonymous reviewers for their detailed comments which helped us improve the quality of this paper. This work was started during Junghee Lee's summer internship supported by the 2010 HERE (Higher Education Research Experiences) program at Oak Ridge National Laboratory. The GaTech authors were funded in part by KORUS

Tech Program grant, KORUSTECH(KT)-2008-DC-AP-FS0-0003. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

## References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Technical Conference*, June 2008.

[2] John S. Buch, Jiri Schindler, Steven W. Schlosser, Gregory R. Ganger, and et al. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. http://www.pdl.cmu.edu/DiskSim/, 2008.

[3] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *IASDS '09: Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[4] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, November 2004.

[5] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the 44th Annual Conference on Design Automation*, pages 212–217, New York, NY, USA, 2007. ACM.

[6] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[7] Jinghuan Chen and Jaekyun Moon. Detection signal-to-noise ratio versus bit cell aspect ratio at high areal densities. *IEEE Transactions on Magnetics*, 37(3):1157–1167, May 2001.

[8] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. System software for flash memory: A survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 394–404, August 2006.

[9] Zoran Dimitrijevi, Raju Rangaswami, and Edward Chang. Design and implementation of semi-preemptible IO. In *USENIX File and Storage Technologies*, March 2003.

[10] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Survey*, 37(2):138–163, 2005.

[11] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240, New York, NY, USA, 2009. ACM.

[12] Sudhanva Gurumurthi, Anand Sivasubramaniam, and Vivek K. Natarajan. Disk drive roadmap from the thermal perspective: A case for dynamic thermal management. In *ISCA'05: Proceedings of the International Symposium on Computer Architecture*, 2005.

[13] HP-Labs. *The OpenMail trace*. http://tesla.hpl.hp.com/opensource/openmail/.

[14] Intel. Intel Xeon Processor X5570 8M Cache, 2.93 GHz, 6.40 GT/s Intel QPI. http://ark.intel.com/Product.aspx?id=37111.

[15] Intel. Intel X25-E Extreme 64GB SATA Solid-State Drive SLC. http://www.intel.com/design/flash/nand/extreme/index.htm.

[16] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[17] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164, New York, NY, USA, 2007. ACM.

[18] Eliahu Ibrahim Jury. *Theory and Application of the Z-Transform Method*. Wiley-Interscience, 1964.

[19] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, New York, NY, USA, 2006. ACM.

[20] Hyojun Kim and Seongjun Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, February 2008.

[21] Youngjae Kim, Jeonghwan Choi, Sudhanva Gurumurthi, and Anand Sivasubramaniam. Managing thermal emergencies in disk-based storage systems. *Journal of Electronic Packaging, Transactions of the ASME*, 130(4), 2008.

[22] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, David A. Dillow, Zhe Zhang, and Bradley W. Settlemyer. Workload characterization of a leadership class storage. In *5th Petascale Data Storage Workshop Supercomputing '10 (PDSW'10)*, November 2011.

[23] Youngjae Kim, Sudhanva Gurumurthi, and Anand Sivasubramaniam. Understanding the performance-temperature interactions in disk i/o of server workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 179–189, Febuary 2006.

[24] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[25] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, 2008.

[26] M. Mallary, A. Torabi, and M. Benakli. One terabit per square inch perpendicular recording conceptual design. *IEEE Transactions on Magnetics*, 38(4):1719–1724, July 2002.

[27] Mtron. 2.5-Inch Mtron SATA Solid State Drive - MSP 7000, 2008. http://www.mtron.net/English/Product/ec_msp7000.asp.

[28] H. Niijima. Design of a solid-state file using flash EEPROM. *IBM Journal of Research and Developement*, 39(5):531–545, 1995.

[29] Sarp Oral, Feiyi Wang, David A. Dillow, Galen M. Shipman, and Ross Miller. Efficient object storage journaling in a distributed parallel file system. In *FAST '10: Proceedings of the Annual Conference on File and Storage Technology*, February 2010.

[30] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, New York, NY, USA, 2006. ACM.

[31] Seon-Yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based SSDs. *Computer Architecture Letters*, 9(1):9–12, January-June 2010.

[32] Steven L. Pratt and Dominique A. Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Linux Symposium*, July 2004.

[33] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[34] N. Schirle and D. F. Lieu. History and trends in the development of motorized spindles for hard disk drives. *IEEE Transactions on Magnetics*, 32(3):1703–1708, May 1996.

[35] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real-Time System*, 22(1/2):9–48, 2002.

[36] Ji-Yong Shin and et al. FTL design exploration in reconfigurable high-performance SSD for server applications. In *ICS'09: Proceedings of the 23rd International Conference on Supercomputing*, pages 338–349, 2009.

[37] Storage-Performance-Council. *OLTP trace from umass trace repository*. http://traces.cs.umass.edu/index.php/Storage/Storage.

[38] Super Talent. Super Talent 128GB UltraDrive ME SATA-II 25 MLC. http://www.supertalent.com/products/ssd_detail.php?type=UltraDrive%20ME.

[39] Jianyong Zhang, Anand Sivasubramaniam, Hubertus Franke, Natarajan Gautan, Yanyong Zhang, and Shailabh Nagar. Synthesizing representative I/O workloads for TPC-H. In *HPCA'04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.