Design Tradeoffs for SSD Performance

Nitin Agrawal[*], Vijayan Prabhakaran, Ted Wobber,
John D. Davis, Mark Manasse, Rina Panigrahy
Microsoft Research, Silicon Valley
[*]*University of Wisconsin-Madison*

Abstract

Solid-state disks (SSDs) have the potential to revolutionize the storage system landscape. However, there is little published work about their internal organization or the design choices that SSD manufacturers face in pursuit of optimal performance. This paper presents a taxonomy of such design choices and analyzes the likely performance of various configurations using a trace-driven simulator and workload traces extracted from real systems. We find that SSD performance and lifetime is highly workload-sensitive, and that complex systems problems that normally appear higher in the storage stack, or even in distributed systems, are relevant to device firmware.

1 Introduction

The advent of the NAND-flash based solid-state storage device (SSD) is certain to represent a sea change in the architecture of computer storage subsystems. These devices are capable of producing not only exceptional bandwidth, but also random I/O performance that is orders of magnitude better than that of rotating disks. Moreover, SSDs offer both a significant savings in power budget and an absence of moving parts, improving system reliability.

Although solid-state disks cost significantly more per unit capacity than their rotating counterparts, there are numerous applications where they can be applied to great benefit. For example, in transaction-processing systems, disk capacity is often wasted in order to improve operation throughput. In such configurations, many small (cost inefficient) rotating disks are deployed to increase I/O parallelism. Large SSDs, suitably optimized for random read and write performance, could effectively replace whole farms of slow, rotating disks. At this writing, small SSDs are starting to appear in laptop computers because of their reduced power-profile and reliability in portable environments. As the cost of flash continues to decline, the potential application space for solid-state disks will certainly continue to grow.

Despite the promise that SSDs hold, there is little in the literature about the architectural tradeoffs inherent in their design. Where such knowledge exists, it typically remains the intellectual property of SSD manufacturers. As a consequence, it is difficult to understand the architecture of a given device, and harder still to interpret its performance characteristics.

In this paper, we lay out a range of design tradeoffs that are relevant to NAND-flash solid-state storage. We then analyze several of these tradeoffs using a trace-based disk simulator that we have customized to characterize different SSD organizations. Since we can only speculate about the detailed internals of existing SSDs, we base our simulator on the specified properties of NAND-flash chips. Our analysis is driven by various traces captured from running systems such as a full-scale TPC-C benchmark, an Exchange server workload, and various standard file system benchmarks.

We find that many of the issues that arise in SSD design appear to mimic problems that have previously appeared higher in the storage stack. In solving these hard problems, there is considerable latitude for design choice. We show that the following systems issues are relevant to SSD performance:

- **Data placement**. Careful placement of data across the chips of an SSD is critical not only to provide load balancing, but to effect wear-leveling.
- **Parallelism**. The bandwidth and operation rate of any given flash chip is not sufficient to achieve optimal performance. Hence, memory components must be coordinated so as to operate in parallel.
- **Write ordering**. The properties of NAND flash present hard problems to the SSD designer. Small, randomly-ordered writes are especially tricky.
- **Workload management**. Performance is highly workload-dependent. For example, design decisions that produce good performance under sequential workloads may not benefit workloads that are not sequential, and vice versa.

As SSDs increase in complexity, existing disk models will become insufficient for predicting performance. In

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

particular, random write performance and disk lifetime will vary significantly due to the locality of disk write operations. We introduce a new model for characterizing this behavior based on cleaning efficiency and suggest a new wear-leveling algorithm for extending SSD lifetime.

The remainder of this paper is organized as follows. In the next section, we provide background on the properties of NAND-flash memory. Section 3 describes the basic functionality that SSD designers must provide and the major challenges in implementing these devices. Section 4 describes our simulation environment and presents an evaluation of the various design choices. Section 5 provides a discussion of SSD wear-leveling and gives preliminary simulator results on this topic. Related work is discussed in Section 6, and Section 7 concludes.

2 Background

Our discussion of flash memory is based on the latest product specifications for Samsung's K9XXG08UXM series NAND-flash part [29]. Other vendors such as Micron and Hynix offer products with similar features. For the remainder of this paper, we treat the 4GB Samsung part as a canonical exemplar, although the specifics of other vendors' parts will differ in some respects. We present the specifications for single-level cell (SLC) flash. Multi-level cell (MLC) flash is cheaper than SLC, but has inferior performance and lifetime.

Figure 1 shows a schematic for a flash package. A flash package is composed from one or more dies (also called chips). We describe a 4GB flash-package consisting of two 2GB dies, sharing an 8-bit serial I/O bus and a number of common control signals. The two dies have separate chip enable and ready/busy signals. Thus, one of the dies can accept commands and data while the other is carrying out another operation. The package also supports interleaved operations between the two dies.

Each die within a package contains 8192 *blocks*, organized among 4 *planes* of 2048 blocks. The dies can operate independently, each performing operations involving one or two planes. Two-plane commands can be executed on either plane-pairs 0 & 1 or 2 & 3, but not across other combinations. Each block in turn consists of 64 4KB *pages*. In addition to data, each page includes a 128 byte region to store metadata (identification and error-detection information). Table 1 presents the operational attributes of the Samsung 4GB flash memory.

2.1 Properties of Flash Memory

Data reads are at the granularity of flash pages, and a typical read operation takes 25μs to read a page from the media into a 4KB data register, and then subsequently shift it out over the data bus. The serial line transfers data at 25ns per byte, or roughly 100μs per page. Flash media blocks must be *erased* before they can be reused for new data. An erase operation takes 1.5ms, which is considerably more expensive than a read or write operation. In addition, each block can be erased only a finite number of times before becoming unusable. This limit, 100K erase cycles for current generation flash, places a premium on careful block reuse.

Page Read to Register	25μs
Page Program (Write) from Register	200μs
Block Erase	1.5ms
Serial Access to Register (Data bus)	100μs
Die Size	2 GB
Block Size	256 KB
Page Size	4 KB
Data Register	4 KB
Planes per die	4
Dies per package (2GB/4GB/8GB)	1,2 or 4
Program/Erase Cycles	100 K

Table 1: Operational flash parameters

Writing (or programming) is also done at page granularity by shifting data into the data register (100μs) and then writing it out to the flash cell (200μs). Pages must be written out sequentially within a block, from low to high addresses. The part also provides a specialized copy-back program operation from one page to another, improving performance by avoiding the need to transport data through the serial line to an external buffer.

In this paper, we discuss a 2 x 2GB flash package, but extensions to larger dies and/or packages with more dies are straightforward.

2.2 Bandwidth and Interleaving

The serial interface over which flash packages receive commands and transmit data is a primary bottleneck for SSD performance. The Samsung part takes roughly 100μs to transfer a 4KB page from the on-chip register to an off-chip controller. This dwarfs the 25μs required to move data into the register from the NAND cells. When these two operations are taken in series, a flash package can only produce 8000 page reads per second (32 MB/sec). If interleaving is employed within a die, the maximum read bandwidth from a single part improves to 10000 reads per second (40 MB/sec). Writes, on the other hand, require the same 100μs serial transfer time per page as reads, but 200μs programming time. Without interleaving, this gives a maximum, single-part write rate of 3330 pages per second (13 MB/sec). Interleaving the serial transfer time and the program operation doubles the overall bandwidth. In theory, because there are two independent dies on the packages we are considering, we can interleave three operations on the two dies put together. This would allow both writes and reads to progress at the speed of the serial interconnect.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

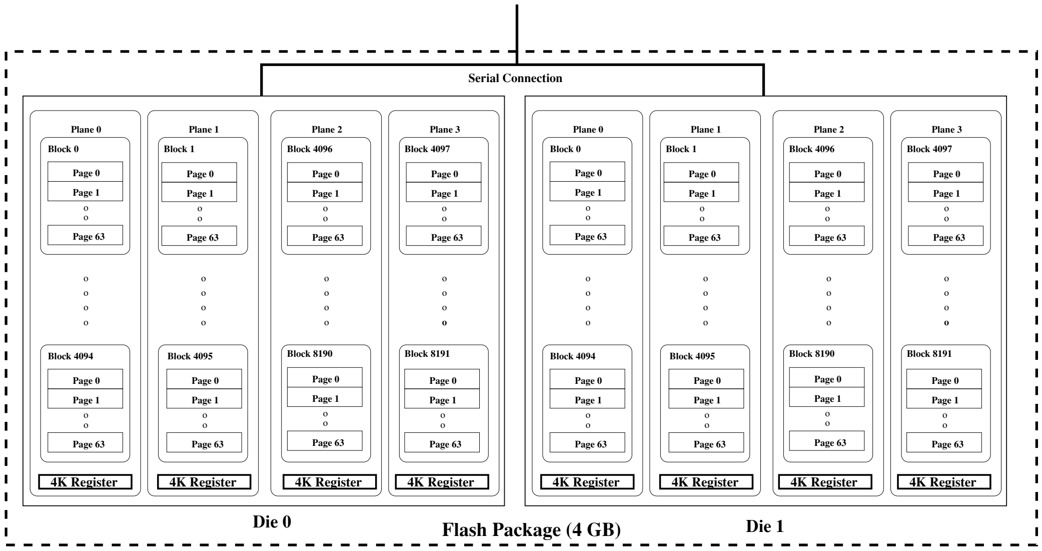

Figure 1: Samsung 4GB flash internals

Interleaving can provide considerable speedups when the operation latency is greater than the serial access latency. For example, a costly erase command can in some cases proceed in parallel with other commands. As another example, fully interleaved page copying between two packages can proceed at close to 100μs per page as depicted in Figure 2 in spite of the 200μs cost of a single write operation. Here, 4 source planes and 4 destination planes copy pages at speed without performing simultaneous operations on the same plane-pair and while optimally making use of the serial bus pins connected to both flash dies. Once the pipe is loaded, a write completes every interval (100μs).

Even when flash architectures support interleaving, they do so with serious constraints. So, for example, operations on the same flash plane cannot be interleaved. This suggests that same-package interleaving is best employed for a choreographed set of related operations, such as a multi-page read or write as depicted in Figure 2. The Samsung parts we examined support a fast internal copy-back operation that allows data to be copied to another block on-chip without crossing the serial pins. This optimization comes at a cost: the data can only be copied within the same flash plane (of 2048 blocks). Two such copies may themselves be interleaved on different planes, and the result yields similar performance to the fully-interleaved inter-package copying depicted in Figure 2, but without monopolizing the serial pins.

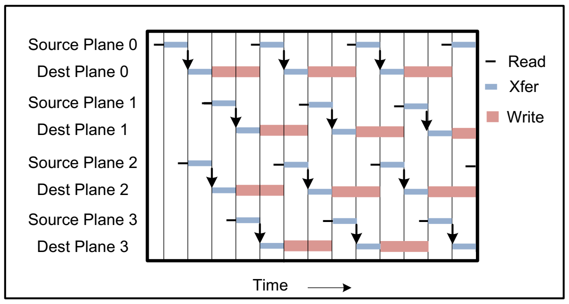

Figure 2: Interleaved page copying

3 SSD Basics

In this section we outline some of the basic issues that arise when constructing a solid-state disk from NAND-flash components. Although we introduce a number of dimensions in which designs can differ, we leave the evaluation of specific choices until Section 4.

All NAND-based SSDs are constructed from an array of flash packages similar to those described in the previous section. Figure 3 depicts a generalized block diagram for an SSD. Each SSD must contain host interface logic to support some form of physical host interface connection (USB, FiberChannel, PCI Express, SATA) and logical disk emulation, like a *flash translation layer* mechanism to enable the SSD to mimic a hard disk drive. The bandwidth of the host interconnect is often a critical constraint on the performance of the device as a whole, and it must be matched to the performance available to and from the flash array. An internal buffer manager holds pending and satisfied requests along the primary data path. A multiplexer (Flash Demux/Mux) emits commands and handles transport of data along the serial connections to the flash packages. The multiplexer can include additional logic, for example, to buffer commands and data. A processing engine is also required to manage the request flow and mappings from disk logical block

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

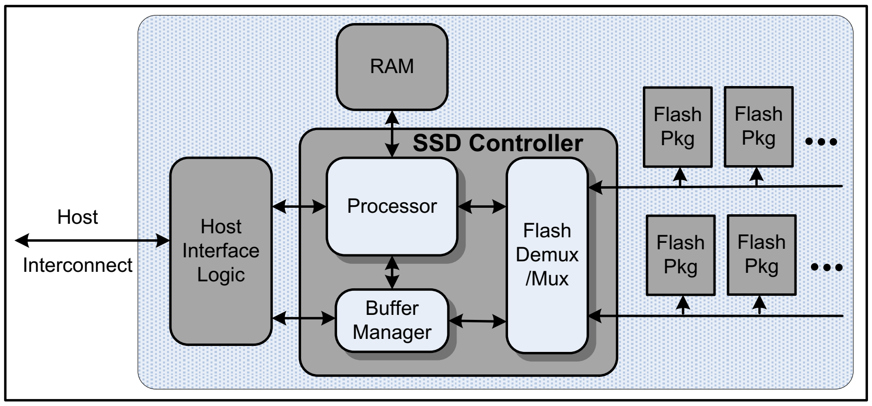

Figure 3: SSD Logic Components

address to physical flash location. The processor, buffer-manager, and multiplexer are typically implemented in a discrete component such as an ASIC or FPGA, and data flow between these logic elements is very fast. The processor, and its associated RAM, may be integrated, as is the case for simple USB flash-stick devices, or standalone as for designs with more substantial processing and memory requirements.

As described in Section 2, flash packages export an 8-bit wide serial data interface with a similar number of control pins. A 32GB SSD with 8 of the Samsung parts would require 136 pins at the flash controller(s) just for the flash components. With such a device, it might be possible to achieve full interconnection between the flash controller(s) and flash packages, but for larger configurations this is not likely to remain feasible. For the moment, we assume full interconnection between data path, control logic, and flash. We return to the issue of interconnect density in Section 3.3.

This paper is primarily concerned with the organization of the flash array and the algorithms needed to manage mappings between logical disk and physical flash addresses. It is beyond the scope of this paper to tackle the many important issues surrounding the design and layout of SSD logic components.

3.1 Logical Block Map

As pointed out by Birrell et al. [2], the nature of NAND flash dictates that writes cannot be performed in place as on a rotating disk. Moreover, to achieve acceptable performance, writes must be performed sequentially whenever possible, as in a log. Since each write of a single logical-disk block address (LBA) corresponds to a write of a different flash page, even the simplest SSD must maintain some form of mapping between logical block address and physical flash location. We assume that the logical block map is held in volatile memory and reconstructed from stable storage at startup time.

We frame the discussion of logical block maps using the abstraction of an *allocation pool* to think about how an SSD allocates flash blocks to service write requests. When handling a write request, each target logical page (4KB) is allocated from a pre-determined pool of flash memory. The scope of an allocation pool might be as small as a flash plane or as large as multiple flash packages. When considering the properties of allocation pools, the following variables come to mind.

- **Static map**. A portion of each LBA constitutes a fixed mapping to a specific allocation pool.
- **Dynamic map**. The non-static portion of a LBA is the lookup key for a mapping within a pool.
- **Logical page size**. The size for the referent of a mapping entry might be as large as a flash block (256KB), or as small as a quarter-page (1KB) .
- **Page span**. A logical page might span related pages on different flash packages thus creating the potential for accessing sections of the page in parallel.

These variables are then bound by three constraints:

- **Load balancing.** Optimally, I/O operations should be evenly balanced between allocation pools.
- **Parallel access.** The assignment of LBAs to physical addresses should interfere as little as possible with the ability to access those LBAs in parallel. So, for example if LBA0..LBAn are always accessed at the same time, they should not be stored on a component that requires each to be accessed in series.
- **Block erasure.** Flash pages cannot be re-written without first being erased. Only fixed-size blocks of contiguous pages can be erased.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

The variables that define allocation pools trade off against these constraints. For example, if a large portion of the LBA space is statically mapped, then there is little scope for load-balancing. If a contiguous range of LBAs is mapped to the same physical die, performance for sequential access in large chunks will suffer. With a small logical page size, more work will be required to eliminate valid pages from erasure candidates. If the logical page size (with unit span) is equal to the block size, then erasure is simplified because the write unit and erase unit are the same, however all writes smaller than the logical page size result in a read-modify-write operation involving the portions of the logical page not being modified.

RAID systems [26] often stripe logically contiguous chunks of data (e.g. 64KB or larger) across multiple physical disks. Here, we use striping at fine granularity to distribute logical pages (4K) across multiple flash dies or packages. Doing so serves both to distribute load and to arrange that consecutive pages will be placed on different packages that can be accessed in parallel.

3.2 Cleaning

Fleshing out the design sketched by Birrell et al. [2], we use flash blocks as the natural allocation unit within an allocation pool. At any given time, a pool can have one or more *active blocks* available to hold incoming writes. To support the continued allocation of fresh active blocks, we need a garbage collector to enumerate previously-used blocks that must be erased and recycled. If the logical page granularity is smaller than the flash block size, then flash blocks must be cleaned prior to erasure. Cleaning can be summarized as follows. When a page write is complete, the previously mapped page location is *superseded* since its contents are now out-of-date. When recycling a candidate block, all non-superseded pages in the candidate must be written elsewhere prior to erasure.

In the worst case, where superseded pages are distributed evenly across all blocks, $N - 1$ cleaning writes must be issued for every new data write (where there are N pages per block). Of course, most workloads produce clusters of write activity, which in turn lead to multiple superseded pages per block when the data is overwritten. We introduce the term *cleaning efficiency* to quantify the ratio of superseded pages to total pages during block cleaning. Although there are many possible algorithms for choosing candidate blocks for recycling, it is always desirable to optimize cleaning efficiency. It's worth noting that the use of striping to enhance parallel access for sequential addresses works against the clustering of superseded pages.

For each allocation pool we maintain a free block list that we populate with recycled blocks. In this section and the next, we assume a purely greedy approach that calls for choosing blocks to recycle based on potential cleaning efficiency. As described in Section 2, NAND flash sustains only a limited number of erasures per block. Therefore, it is desirable to choose candidates for recycling such that all blocks age evenly. This property is enforced through the process known as wear-leveling. In Section 5, we discuss how the choice of cleaning candidates interacts directly with wear-leveling, and suggest a modified greedy algorithm.

In an SSD that emulates a traditional disk interface, there is no abstraction of a free disk sector. Hence, the SSD is always full with respect to its advertised capacity. In order for cleaning to work, there must be enough spare blocks (not counted in the overall capacity) to allow writes and cleaning to proceed, and to allow for block replacement if a block fails. An SSD can be substantially *overprovisioned* with such spare capacity in order to reduce the demand for cleaning blocks in foreground. Delayed block cleaning might also produce better clustering of superseded pages in non-random workloads.

In the previous subsection, we stipulated that a given LBA is statically mapped to a specific allocation pool. Cleaning can, however, operate at a finer granularity. One reason for doing so is to exploit low-level efficiency in the flash architecture such as the internal copy-back operation described in Section 2.2, which only applies when pages are moved within the same plane. Since a single flash plane of 2048 blocks represents a very small allocation pool for the purposes of load distribution, we would like to allocate from a larger pool. However, if an active block and cleaning state per plane is maintained, then cleaning operations within the same plane can be arranged with high probability.

It might be tempting to view block cleaning as similar to log-cleaning in a Log-Structured File System [28] and indeed there are similarities. However, apart from the obvious difference that we model a block store as opposed to a file system, a log-structured store that writes and cleans in strict disk-order cannot choose candidate blocks so as to yield higher cleaning efficiency. And, as with LFS-like file systems, it's altogether too easy to combine workloads that would cause all recoverable space to be situated far from the log's cleaning pointer. For example, writing the same sets of blocks over and over would require a full cycle over the disk content in order for the cleaning pointer to reach the free space near the end of the log. And, unlike a log-structured file system, the disk here is always "full", corresponding to maximal cleaning pressure all the time.

3.3 Parallelism and Interconnect Density

If an SSD is going to achieve bandwidths or I/O rates greater than the single-chip maxima described in Sec-

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.