Improving NAND Flash Based Disk Caches

Taeho Kgil*   David Roberts   Trevor Mudge
University of Michigan
Advanced Computer Architecture Laboratory
2260 Hayward Street Ann Arbor, MI 48109
E-mail: {tkgil,daverobe,tnm}@eecs.umich.edu

Abstract

Flash is a widely used storage device that provides high density and low power, appealing properties for general purpose computing. Today, its usual application is in portable special purpose devices such as MP3 players. In this paper we examine its use in the server domain—a more general purpose environment. Aggressive process scaling and the use of multi-level cells continues to improve density ahead of Moore's Law predictions, making Flash even more attractive as a general purpose memory solution. Unfortunately, reliability limits the use of Flash. To seriously consider Flash in the server domain, architectural support must exist to address this concern. This paper first shows how Flash can be used in today's server platforms as a disk cache. It then proposes two improvements. The first improves performance and reliability by splitting Flash based disk caches into separate read and write regions. The second improves reliability by employing a programmable Flash memory controller. It can change the error code strength (number of correctable bits) and the number of bits that a memory cell can store (cell density) according to the demands of the application. Our studies show that Flash reduces overall power consumed by the system memory and hard disk drive up to 3 times while maintaining performance. We also show that Flash lifetime can be improved by a factor of 20 when using a programmable Flash memory controller, if some performance degradation (below 5%) is acceptable.

1 Introduction

Today, NAND Flash can be found in handheld devices like cell phones, digital cameras and MP3 players. This has been made possible because of its high density and low power properties. These result from the simple structure of Flash cells and its nonvolatility. Its popularity has meant that it is the focus of aggressive process scaling and innovation.

The rapid rate of improvement in density has suggested other usage models for Flash besides handheld devices, and in recent years, Flash has also emerged as an attractive memory device to integrate into other computing platforms. Flash is used in solid state disks and hybrid disk drives to reduce power consumption and improve performance [1, 3]. Flash is even being considered inside data centers where system memory power and disk drive power are critical concerns. With appropriate support, Flash could dramatically reduce the cost of powering and cooling data centers.

However, Flash manageability and reliability are challenging problems that need to be addressed to fully integrate Flash into data centers. Flash wears out as we increase the number of writes/erases. A combined software/hardware effort is required to mitigate these problems. Reliability is especially important in servers deployed in data centers. This paper proposes a hardware assisted software managed disk cache for NAND Flash. We address the shortcomings due to reliability and manageability. Our contributions are as follows:

1. We extend the power and performance analysis for using Flash based disk caches on server platforms described in [16]. We also show that there is negligible overhead in complexity and performance to using Flash.

2. We show that by splitting Flash based disk caches into read and write regions, overall performance and reliability can be improved.

3. We show that a programmable Flash memory controller can improve Flash cell reliability and extend memory lifetime. The first programmable parameter is error correction code strength. The second is the Flash cell density—changing from multi-level cells to single level cells.

---

*Currently at Intel

IEEE
computer
society

The paper is organized as follows. The next section provides background on Flash and makes the case for using Flash as a disk cache. Section 3 presents the architecture of a Flash based disk cache. Section 4 explains the Flash memory controller architecture that improves reliability. Section 5 describes how our proposed architecture works. Section 6 describes the methodology used to explore the design space and evaluate our proposed architecture. Section 7 presents the results of that exploration. Section 8 presents concluding remarks.

2 Background and the case for Flash based disk caches

2.1 Using NAND Flash to reduce system memory power

Prior work on Flash has been proposed to save power and improve performance in memory systems. It has been proposed as a disk cache [16] and a replacement for disk [26]. Tables 1 and 2 show why Flash is attractive as a component in servers. Table 1 shows the 2007 ITRS roadmap for NAND Flash compared to other types of memory devices. Table 2 shows the density, performance and power consumption of NAND Flash compared to other types of memory and storage devices. It can be seen that the cell size for NAND Flash is much smaller than other memory devices. This is primarily due to the NAND cell structure and the adoption of multi-level cell (MLC) technology. MLC Flash today provides 2 bits per cell, and 4 bit per cell MLC Flash is expected by 2011. Given the trends shown in the ITRS roadmap, it is reasonable to expect NAND Flash to be as much as 8× denser than DRAM by 2015.

Table 2 also shows that NAND Flash consumes much less power than DRAM, but its latency is much greater [25, 18]. On the other hand, compared to a hard disk drive, a NAND Flash has significantly lower access latency and power consumption.

Table 1 shows that Flash is limited in the number of write/erase cycles that a data block can reliably tolerate. There also exists a clear trade-off between Flash density and reliability, since today's SLC Flash tolerates 10 times as many writes as MLC. The limited Flash endurance is due to Flash cell wear out [24, 23, 14].

NAND Flash is organized in units of *pages* and *blocks*. A typical Flash *page* is 2KB in size and a Flash *block* is made up of 64 Flash *pages* (128KB). Random Flash reads and writes are performed on a *page* basis and Flash erasures are performed per *block*. A Flash must perform an erase on a *block* before it can write to a *page* belonging to that *block*. Each additional write must be preceded by an erase. Therefore *out-of-place* writes are commonly used to mitigate wear out. They treat Flash as a log and append new

	2007	2009	2011	2013	2015
NAND Flash-SLC*($\mu m^2/bit$)	0.0130	0.0081	0.0052	0.0031	0.0021
NAND Flash-MLC*($\mu m^2/bit$)	0.0065	0.0041	0.0013	0.0008	0.0005
DRAM Cell density($\mu m^2/bit$)	0.0324	0.0153	0.0096	0.0061	0.0038
Flash write/erase cycles—SLC/MLC[†]	1E+05/ 1E+04	1E+05/ 1E+04	1E+06/ 1E+04	1E+06/ 1E+04	1E+06/ 1E+04
Flash data retention (years)	10-20	10-20	10-20	20	20

[*] SLC - Single level Cell, MLC - Multi Level Cell
[†] write/erase cycles for MLC Flash estimated from prior work[17]

Table 1. ITRS 2007 roadmap for memory technology.

	Active Power	Idle Power	Read Latency	Write Latency	Erase Latency
1Gb DDR2 DRAM	878mW	80mW[†]	55ns	55ns	N/A
1Gb NAND-SLC	27mW	6μW	25μs	200μs	1.5ms
4Gb NAND-MLC	N/A	N/A	50μs	680μs	3.3ms
HDD[‡]	13.0W	9.3W	8.5ms	9.5ms	N/A

[†] DRAM Idle power in active mode. Idle power in powerdown mode is 18mW
[‡] Data for 750GB Hard disk drive [5]

Table 2. Performance, power consumption and cost for DRAM, NAND-based SLC/MLC Flash and HDD.

data to the end of the log while old data is invalidated. This is done on a per page basis. Figure 1(a) illustrates the organization of the SLC/MLC dual mode Flash used in this study. A similar design with different page and block sizes was published by Cho et al. [11]. Pages in SLC mode consist of 2048 bytes of data area and 64 bytes of 'spare' data for error correction code (ECC) bits. When in MLC mode, a single SLC page can be split into two 2048 byte MLC pages. Pages are erased together in blocks of 64 SLC pages or 128 MLC pages.

Server applications typically use a large amount of system memory for caching contents in disk. Traditionally part of DRAM is used as a cache to reduce the high access latency to hard disk drives. The disk cache, or page cache, is managed by the OS. Assuming that reliability can be guaranteed through software and hardware techniques, Flash can be used as a way to address the wide latency gap between the hard disk drive and DRAM while saving power. In fact, it has been shown that system memory consumes an appreciable amount of total power—more than 1/4 of the power consumption in a data center platform [20].
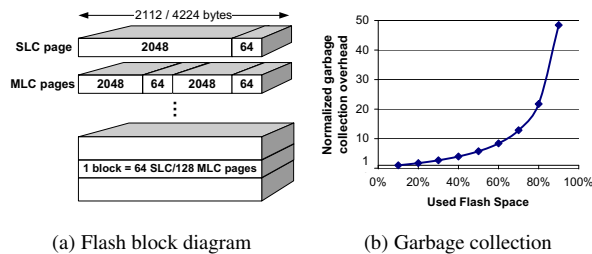
Figure 1. (a) Example dual mode SLC/MLC Flash bank organization (b) Garbage collection (GC) overhead in time versus occupied Flash space in a 2GB Flash.

Figure 2. 1GB DRAM is replaced with a smaller 256MB DRAM and 1GB NAND-based Flash. Additional components are added to control Flash.

2.2 Using NAND Flash based disk caches

In addition to using Flash as a disk cache, there has been a proposal to use it as an auxiliary disk using NOR cells [26]. When that study was performed the now dominant NAND Flash had not been introduced. However, the architecture in [26] could easily be modified to incorporate NAND.

The biggest difficulty with using Flash as a solid-state disk (SSD) in place of a conventional disk is the garbage collection overhead. Flash based file systems require a dedicated software layer that executes file system primitives customized for Flash to emulate FAT32, EXT2 or dedicated Flash file systems like JFFS. These implementations are slow because of the garbage collection that is necessary with the *out-of-place* write policy in Flash.

The overhead in garbage collection increases as less free space is available on Flash. This becomes a significant problem, because garbage collection generates extra writes and erases in Flash, reducing performance and endurance as the occupancy of the Flash increases. The garbage collection overhead is a product of garbage collection frequency and garbage collection latency. Figure 1(b) shows how the time spent garbage collecting increases as more Flash space is used. It is normalized to an overhead of 10%. It can be seen that GC becomes overwhelming well before all of the memory is used. In fact, the study in [26] was only able to use 80% of its storage capacity to suppress garbage collection overhead.

Another difficulty is the memory overhead in maintaining the Flash file system data structures. File systems store file data along with meta-data to manage the file. The size of these structures are appreciable and take up a considerable percentage of memory for small files. Managing the entire file system meta-data in DRAM incurs an appreciable memory overhead. Thus, conventional file systems load only a portion of the file system meta-data in DRAM. This method is similar to demand based paging. Wear out ef-
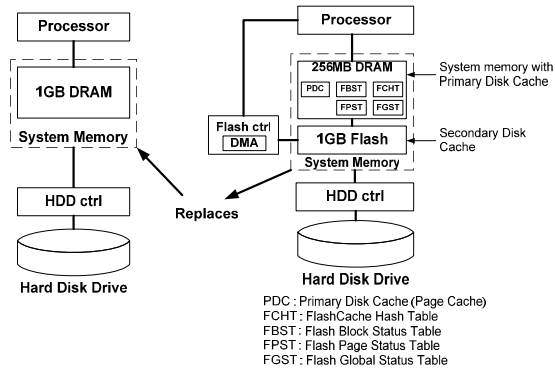
fects and higher access latency mean Flash is less appropriate for storing this meta-data at runtime. File system meta-data structures are frequently updated and result in frequent Flash writes and erases. As a result, the meta-data used in a Flash based file system should be kept in DRAM at runtime. If Flash is used as a full file system rather than as a disk cache, more DRAM is required for meta-data.

A disk cache is much simpler and requires less storage overhead. By managing the contents of a disk at the granularity of pages and using a single item of meta-data to locate each page, the meta-data size is bounded and independent of the size of the underlying file system.

3 Architecture of the Flash based disk cache

The right side of Figure 2 shows the Flash based disk cache architecture. It is an extension of [16]. Compared to a conventional DRAM-only architecture shown on the left side of Figure 2, our proposed architecture uses a two level disk cache, composed of a relatively small DRAM in front of a dense Flash. The much lower access time of DRAM allows it to act as a cache for the Flash without significantly increasing power consumption. A Flash memory controller is also required, for reliability management. We will provide a detailed description of our Flash memory controller in later sections.

Our design uses a NAND Flash that stores 2 bits per cell (MLC) and is capable of switching from MLC to SLC mode using techniques proposed in [11, 19]. Finally, our design uses variable-strength error correction code (ECC) to improve reliability. We will discuss ECC and density control in later sections.

Next, we explain the data structures used in our Flash block and page management. These tables are read from the hard disk drive and stored in DRAM at run-time to reduce access latency and mitigate wear out. The Primary Disk Cache (PDC) also resides in DRAM and caches the most frequently accessed disk pages. Adding Flash allows us to greatly reduce the size of the PDC. The tables are as follows;

- *FlashCache hash table (FCHT)* - stores the mapping between addresses on disk and Flash. The FCHT was first introduced in [16].

- *Flash page status table (FPST)* - stores error correction code (ECC) strength, SLC/MLC mode and a saturating access counter. It also stores a valid bit field.

- *Flash block status table (FBST)* - records the number of erase operations performed on each block as well as the degree of wear out.

- *Flash global status table (FGST)* - tracks average latency and miss rate of the Flash based disk cache.

We augmented the data structures described in [16]. Additional data structures (FPST, FBST, FGST) are required to support our programmable Flash memory controller. The overhead of the four tables described above are less than 2% of the Flash size. The FCHT and FPST are the primary contributors because we need them for each Flash page. For example, the memory overhead for a 32GB Flash is approximately 360MB of DRAM. Our Flash based disk cache is managed in software (OS code) using the tables described above. We found the performance overhead in executing this code to be minimal.

3.1 FlashCache hash table (FCHT)

The FCHT is a memory structure that holds tags associated with the Flash pages. A tag is made up of a logical block address (LBA) field and a Flash memory address field. The LBA field points to the location in the hard disk drive and is used to determine whether Flash has cached this data. The corresponding Flash memory address field is used to access Flash.

The FCHT is organized as a fully associative table accessed quickly by performing a hash. Increasing the number of entries that can be indexed in the hash table can increase performance We found that around 100 entries provide close to the maximum system throughput.

3.2 Flash page status table (FPST)

The FPST contains the configuration of each page in Flash. The ECC strength and the SLC/MLC mode fields are used to interface with the Flash memory controller. The

saturating access counter is used to address frequent access behavior. We also keep track of valid page data by setting and resetting the valid bit field (see [15] for details). Its usage is explained in section 5.2.

3.3 Flash block status table (FBST)

The FBST is used to profile the number of erases performed on a particular block. This information is used to perform wear-leveling (section 3.6), which increases the lifetime of the Flash by attempting to erase all blocks uniformly. The FBST maintains the current status of a Flash block. It holds the degree of wear out of a Flash block, and the number of erases performed on this block (see [15] for details). The degree of wear out is a cost function generated from observing the erase count as well as the status of all Flash pages belonging to a Flash block. We defined the degree of wear out for Flash block i as follows. It is a metric used to compare the relative wear out of Flash blocks to make allocation decisions.

$$wear\_out_i = N_{erase,i} + k_1 \times Total_{ECC,i}$$
$$+ k_2 \times Total_{SLC\_MLC,i}$$

where $N_{erase,i}$ is the number of erases to block i, $Total_{ECC,i}$ is the total ECC strength of block i (which is increased as the block wears out). This is the sum of ECC strengths across all pages in a block. The $Total_{SLC\_MLC,i}$ is the total number of pages in block i that are converted to SLC mode due to wear out. The constants k_1, k_2 are positive weight factors. Constant k_2 is larger than k_1 because a mode switch (from MLC to SLC) impacts wear out a lot more than increasing ECC strength. Essentially, $Total_{ECC,i}$ and $Total_{SLC\_MLC,i}$ are the sum of the ECC strength and the SLC/MLC mode fields for all pages in the FPST that belong to block i.

The number of erases is related to the number of Flash based disk cache evictions and writes. Section 3.6 explains the usage of the FBST. The `wear_out` field of the FBST determines whether a block is *old* or *new*.

3.4 Flash global status table (FGST)

The FGST contains summary information regarding how well the entire Flash based disk cache is working. This includes statistics such as miss rate and average access latencies across all Flash blocks. The fields in the FGST are used to address the increase in faults due to wear out (see [15] for details). Its usage is explained in section 5.2.

3.5 Splitting Flash into a read cache and a write cache

Naively managed *out-of-place* writes degrade the performance of a Flash based disk cache. They tend to in-

crease the garbage collection (GC) overhead which in turn increases the number of overall disk cache misses incurring a significant performance overhead. The increase in over-all Flash based disk cache miss rate is due to invalid Flash pages that are generated from *out-of-place* writes.

It is desirable to limit GC and maintain a reasonable access latency to Flash, because the read access latency to a Flash based disk cache is important for overall system performance. To that end we divide the Flash into a read disk cache and a write disk cache. Read caches are less susceptible to *out-of-place* writes, which reduce the read cache capacity and increase the risk of garbage collection. Read critical Flash blocks are located in the read disk cache that may only evict Flash blocks and pages on read misses. The write disk cache captures all writes to the Flash based disk cache and performs *out-of-place* writes. Wear-leveling is applied globally to all regions in the Flash based disk cache.

By separating the Flash based disk cache into a read and write disk cache, we are able to reduce the number of blocks that have to be considered when doing write triggered garbage collection. Figure 3 shows an example that highlights the benefits of splitting the Flash based disk cache into a read and write cache. The left side shows the behavior of a unified Flash based disk cache and the right side shows the behavior of splitting the Flash based disk cache into a read and write cache. Figure 3 assumes we have 5 pages per block and 5 total blocks in a Flash based disk cache. Garbage collection proceeds by reading all valid data from blocks containing invalid pages, erasing those blocks and then sequentially re-writing the valid data. In this example, when the Flash based disk cache is split into a read and write cache, only 2 blocks are candidates for garbage collection. This dramatically reduces Flash reads, writes and erases compared to a unified Flash based disk cache that considers all 5 Flash blocks.

Each read and write cache manages its own set of Flash blocks and its own LRU policy. Section 5.1 provides more detail for how Flash based disk cache hits, misses and writes are handled. Figure 4 shows the miss rate improvement obtained by splitting into a read and write disk cache. Based on the observed write behavior, 90% of Flash is dedicated to the read cache and 10% write cache. The reduced miss rate results in improved performance and shows the benefit of splitting the Flash based disk cache, particularly as disk caches get larger.

3.6 Wear-level aware Flash based disk cache replacement policy

Unlike DRAM, Flash wears out. To mitigate wear out, wear-level management is performed on Flash erases for the Flash based disk cache. Wear-leveling is a commonly used technique in Flash [6, 9]. We adopt a modification of con-
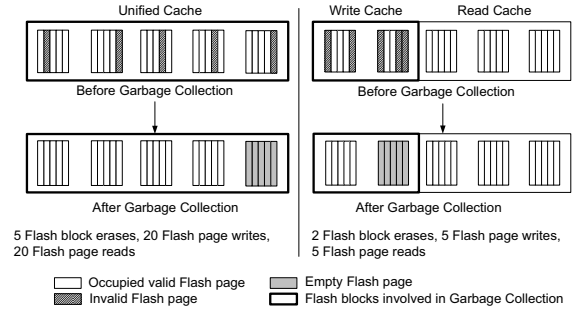

Figure 3. Diagram illustrating the benefits of splitting the Flash based disk cache.
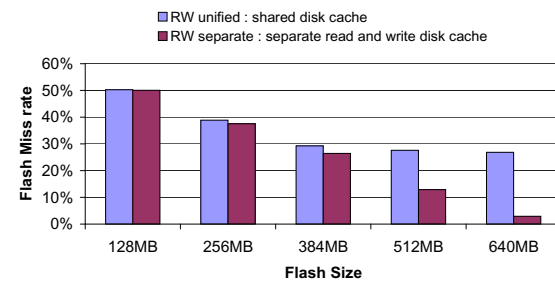

Figure 4. A miss rate comparison executing a dbt2 (OLTP) disk trace for a unified Flash based disk cache and a split read and write Flash based disk cache.

ventional wear-leveling techniques that is customized for a disk cache usage model. For both the read and write cache, we initially select a block to be evicted using an LRU policy on disk cache capacity misses (for the read cache) or capacity writes (*out-of-place* writes that need to first erase blocks, for the write cache). However, if this block wear out exceeds that of the *newest block* by a predetermined threshold, then the block corresponding to the minimum wear out (*newest block*) is evicted to balance the wear-level. *Newest blocks* are chosen from the entire set of Flash blocks. Before we evict the *newest block*, its content is migrated to the *old block*. The degree of wear out of a Flash block is determined by observing the entries in the FBST.

4 Architecture of the Flash memory controller

Flash needs architectural support to improve reliability and lifetime when used as a cache. Figure 5 shows a high-level block diagram of a programmable Flash memory con-
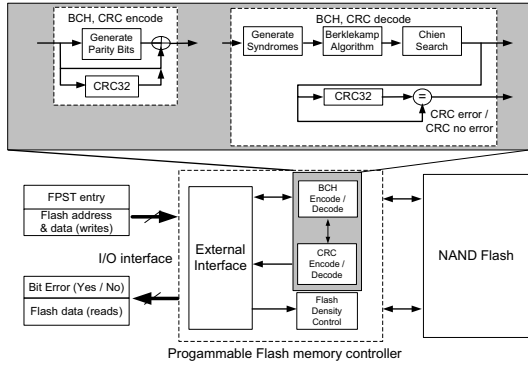
Figure 5. High-level block diagram of a pro-grammable Flash memory controller.

Figure 6. (a) BCH decode latency (b) Maximum tolerable Flash write/erase (W/E) cycles for varying code strength.

troller that addresses this need. It is composed of 2 main components;

- An encoder and decoder for error correction and detection. Unlike conventional Flash logic, we use a variable error correction strength.

- A density controller for SLC/MLC mode selection. This dynamically trades block storage capacity for reliability as the Flash ages, or lower latency for frequently accessed pages.

We use a typical device driver interface to access the Flash memory controller. The device driver reads the error correction code (ECC) strength and SLC/MLC mode fields from the FPST and generates a descriptor (control messages) to access a page in Flash. We describe the details of our Flash memory controller in the following subsections.

4.1 Hardware assisted error correction code support

A common way to recover from errors in Flash is to use an error correction code (ECC). This section describes the error correction and detection scheme in our Flash memory controller. We also show how error correction can extend Flash lifetime, and that the time penalty for the correction process need not impact performance.

ECCs are widely employed in digital storage devices to mitigate the effects of hard (permanent) and soft (transient) errors. Flash typically uses linear block codes like the Bose, Ray-Chaudhuri, Hocquenghem (BCH) code due to its strength and acceptable decode/encode latency. To reduce the false positives that can occur with BCH codes, CRC codes are also employed.

Our architecture shown in Figure 5 uses a BCH encoder and decoder to perform error correction and a 32 bit CRC
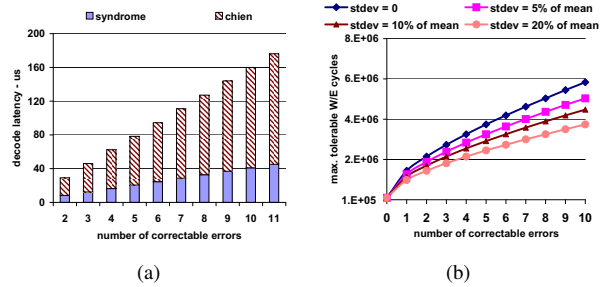
checker to perform error detection. The BCH check bit storage overhead is small considering the high capacity of today's Flash. Devices typically include 64 bytes per page for ECC support bits. The CRC32 code needs 4 bytes, leaving 60 bytes for BCH. Because we limited the number of correctable errors to 12, a maximum of 23 bytes are needed for check bits, per page.

4.1.1 BCH encoder and decoder

Given a message of k bits, we can construct a t-error-correcting BCH code with block length of n bits, such that the number of parity check bits is given by $n - k \geq mt$. Furthermore, the block length n should satisfy $n = 2^m - 1$. We need to append approximately $log(n)$ bits for each error we wish to correct, and the number of parity check bits increase linearly with the number of correctable errors for a fixed code length.

Figure 5 shows an implementation of our BCH encoder and decoder. The Berlekamp and Chien search algorithms in the decoder are widely used due to their simplicity, and they have been proven to be an effective iterative technique in decoding BCH codes [21]. In addition to that, Chien search can be parallelized in a straightforward manner.

We implemented the BCH encoder and decoder in C and measured the amount of time spent encoding and decoding BCH code on a 3.4 GHz Pentium 4 system. Latencies ranging from a tenth of a second to nearly a second, were observed for correcting 2-10 errors. Clearly an accelerator is necessary. In response, we designed one based on ideas in [22]. A Berlekamp acceleration engine and a highly parallelized Chien search engine improve the modular arithmetic and memory alignment found in BCH codes and takes advantage of the parallelism inherent in BCH decoders. The resulting decode latencies are shown in Figure 6(a). These latencies are obtained using a 100 MHz in-order embedded processor with parallelized modular arithmetic

support. Berlekamp algorithm overhead is insignificant and was omitted from the figure.

We developed a design to estimate the cost in area. Our implementation used a 2^{15} entry finite field lookup table as well as 16 finite field adders and multipliers as accelerators to implement the Berlekamp and Chien search algorithm (16 instances of the Chien search engines). BCH codes use finite field operators, which are sufficiently different from standard arithmetic operators to cause a bottleneck in a general purpose CPU without an accelerator. We limit the programmability to a fixed block size (2KB) to avoid memory alignment with different block sizes and limit the maximum number of correctable errors to 12. Our design required about 1 mm^2.

4.1.2 CRC checksum

One of the drawbacks of BCH codes is that they cannot always detect when more errors occur than they have been designed to correct. In some cases the Chien search can find no roots, indicating that more errors occurred. In other cases roots are found creating a false positive. The usual solution to this is to augment them with CRC codes to improve error detection. CRC codes are capable of covering a wide range of error patterns. We used an optimized hardware implementation of a CRC32 functional block. Our design compiler results showed it occupied a negligible amount of die area and added negligible performance overhead (tens of nanoseconds). This agrees with other implementations such as [13].

4.1.3 Impact of BCH code strength on Flash lifetime

Flash cell lifetime displays an exponential relationship with oxide thickness [24]. In this exponential model Flash cell lifetime W can be defined as:

$$W = 10^{C_1 \cdot t_{ox}}$$

where C_1 is a constant. Most Flash specifications refer to the probability of a cell failing after 100,000 write/erase (W/E) cycles. This probability is usually of the order of 10^{-4}, and allows us to calculate the constant in the cell lifetime formula above. We further assume, in common with other studies, that oxide thickness is normally distributed with three standard deviations equal to 15% of the mean oxide thickness. Combining these facts with number of cells in a page and the code strength (the number of errors that can be corrected) allows us to derive a distribution for the lifetime in W/E cycles. An exponential analytical model is employed. We assume Flash page size to be 2KB and first point of failure to occur at 100,000 W/E cycles. See [15] for details of the derivation.

Using the *exponential model*, we plotted the Flash W/E cycles versus ECC code strength in Figure 6(b). As can
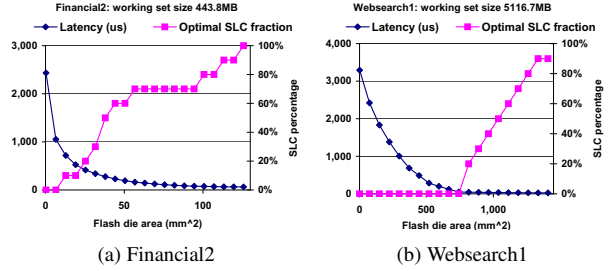

Figure 7. Optimal access latency and SLC/MLC partition for various multimode MLC Flash sizes.

be seen, ECC code strength extends lifetime. We also note that we see diminishing return from increasing ECC code strength for both models. Spatial variation negatively impacts code strength because our assumptions in plotting Figure 6(b) assumed all Flash pages had to be recoverable for a certain ECC code. As bad Flash cells display a higher spatial correlation, bad cells cluster in groups on a page resulting in an increasing number of pages that cannot recover using a particular ECC.

4.2 Hardware assisted density control

MLC Flash cells take longer to read and write. Multiple levels imply a narrower gap between different logical values and this is the main cause of reduced endurance. To reduce this drawback, there has been work on enabling MLC to operate in SLC mode to improve Flash endurance and latency [11, 19]. Samsung has recently also announced One-FlexNAND that dynamically controls the SLC to MLC multimode operation. Our programmable Flash memory controller assumes that one can dynamically control the density of a Flash at the page level by slightly modifying the sense amplifier circuitry found in a MLC [11, 19]. Therefore, the primary role of a density controller is to indicate the mode of the requested page. The density configuration of the requested page can be found in the density descriptor generated from the SLC/MLC mode field in the FPST. Density control benefits Flash performance and endurance, because we are able to reduce access latency for frequently accessed pages and possibly improve endurance for aging Flash pages by changing MLC pages into SLC pages as needed.

To show the potential improvement of Flash performance by controlling density, we present a study using real disk traces. Using disk activity traces from [8] for financial and web search applications, we analyzed the average access latency for different SLC/MLC partitions, for several Flash sizes.

We show that a hybrid allocation of SLC and MLC provides minimum access latency. Figure 7 shows the average access latencies (left y-axis) achieved for an optimal partition (right y-axis) between SLC and MLC. The x-axis varies the Flash size (die area) based on a recent MLC Flash implementation [12]. Although the total chip capacity in [12] is fixed at 1GB, we assumed their implementation could scale and further assumed that the control circuitry area scales linearly with the number of Flash cells. The x-axis extends to the entire working set size, given in Figure 7 (see graph title). As expected, when the size of the cache approaches the working set size, latency reaches a minimum using only SLC. The optimal SLC/MLC partition is dependent on the nature of the workload and Flash size, thus programmability is important. In the two benchmarks of Figure 7, it can be seen that in (a) 70% can be SLC while in (b) almost all the cells are MLC for a Flash size that is approximately half the working set size.

5 Putting it all together

5.1 Accessing the Flash based disk cache

In this section we discuss how hits, misses and writes are handled in a Flash based disk cache with respect to the entire system. When a file read is performed at the application level, the OS searches for the file in the primary disk cache located in DRAM. On a primary disk cache hit in DRAM, the file content is accessed directly from the primary disk cache (no access to Flash related data structures). On a primary disk cache miss in DRAM, the OS searches the FCHT to determine whether the requested file currently exists in the secondary disk cache. Because it is a file read, the OS searches the FCHT. If the requested file is found, then a Flash read is performed and a DMA transaction is scheduled to transfer Flash content to DRAM. The requested address to Flash is obtained from the FCHT.

If a read miss occurs in the FCHT search process, we first look for an empty Flash page in the read cache. If there is no empty Flash page available, we first select a block for eviction to generate empty Flash pages. We enforce our wear-level aware replacement policy to select the block to evict. Wear-level aware replacement is performed based on section 3.6. Concurrently, a hard disk drive access is scheduled using the device driver interface. The hard disk drive content is copied to the primary disk cache in DRAM and the read cache in Flash. The corresponding tag in the FCHT belonging to the read disk cache is also updated.

File writes are more complicated. If we write to a file, we typically update/access the page in the primary disk cache and this page is periodically scheduled to be written back to the secondary disk cache and later periodically written back to the disk drive. When writing back to Flash, we first de-
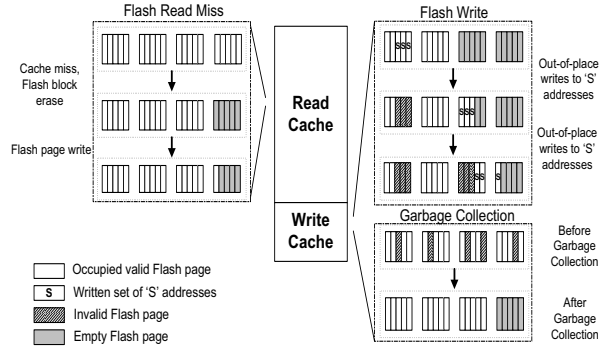

Figure 8. Flash based disk cache handling read cache misses, writes and garbage collection.

termine whether it exists on Flash by searching the FCHT. If it is found in the write cache, we update the page by doing an *out-of-place* write to the write cache. If it is found in the read cache, then we invalidate this page (reset the valid bit field in the FPST) and allocate a page in the write cache. Garbage collection (GC) may also be triggered in the read cache when the read cache capacity goes below 90%. If it is not found in the Flash, we allocate a page in the write cache. When we have no empty pages in the write cache, we either trigger GC (common case) or we select a block for eviction to generate empty Flash pages. We only evict blocks when the total number of invalid pages is less than the total number of pages in a Flash block. The disk is eventually updated by flushing the write disk cache. All GCs are performed in the background. When erasing a block during GC, wear-level management is performed.

Figure 8 shows several examples of how our Flash based disk cache handles misses, writes and garbage collection. The left side of the diagram shows how Flash is accessed when a read miss occurs. Because there are no empty Flash pages available, a block is evicted and erased, making room for the new page of data. The write region is illustrated in the right side. Three existing pages labeled 'S' have their values updated. The new data is placed in three empty pages and the old pages are invalidated ready for garbage collection (*out-of-place* write). Writing those three pages for a second time causes three new pages to be written and the three previous pages are invalidated.

5.2 Reconfiguring the Flash memory controller

This section describes how the Flash descriptors (control messages) are generated and how we update the Flash data structures (FPST) used to generate these descriptors. Before

we schedule a Flash access to a certain page, descriptors for that page are generated and sent to our programmable Flash memory controller. These page descriptors are generated by looking up the FPST. Based on the observed behavior of a Flash page, the FPST is updated accordingly to maximize the fields found in the FGST (overall performance). The updated page settings are applied on the next erase and write access. In addition, whenever we access a valid Flash page, we update the field that keeps track of the number of accesses (saturating counter) in the FPST, and all the fields in FGST accordingly.

There are two main triggers for an error correction code (ECC) or density mode change. These are 1) an increase in the number of faulty bits, 2) a change in access (read) frequency. Each trigger is explained below. If a Flash page reaches the ECC code strength limit and density limit (SLC mode), the block is removed permanently and never considered when looking for pages to allocate in a disk cache.

5.2.1 Response to increase in faults

When new bit errors are observed and fail consistently due to wear out, we reconfigure the page. This is achieved by enforcing a stronger ECC or reducing cell density from MLC to SLC mode. The change in overall latency caused by each of these options are denoted by Δt_{cs} (stronger ECC) and Δt_d (reducing cell density) respectively. We choose the option with the minimum latency increase by comparing heuristic approximations. The heuristics are derived as follows;

$$\Delta t_{cs} = freq_i \times \Delta code\_delay$$
$$t_1 = t_{miss} \times miss\_rate + t_{hit} \times (1 - miss\_rate)$$
$$t_2 = t_{miss} \times (miss\_rate + \Delta miss) + (t_{hit} - \Delta t_{hit}) \times (1 - (miss\_rate + \Delta miss))$$
$$\Delta t_d = t_1 - t_2$$
$$\approx \Delta miss \times (t_{miss} + t_{hit}) + freq_i \times \Delta SLC$$

The parameter $freq_i$ is the relative access frequency of the current page i and t_{miss} is the average miss penalty for accessing disk. t_{hit} is the average hit latency for our Flash based disk cache. The latency increase for enforcing a stronger ECC is $\Delta code\_delay$, the reduction in average hit latency is Δt_{hit}, the increase in miss rate is $\Delta miss$ and the reduction in Flash page read latency due to switching from MLC to SLC is ΔSLC. t_1 and t_2 represent the overall average latency before and after a density mode change. $\Delta code\_delay$ and ΔSLC are constants. $freq_i$, t_{hit}, t_{miss} and $\Delta miss$ are measured during run-time and derived from the FPST and FGST. The decision to select Δt_{cs} versus Δt_d is influenced by workload behavior and Flash age.

	Configuration parameters
Processor type	8 cores, each core single issue in-order
Clock frequency	1GHz
Cache size	L1: 4 way 16KB each, L2: 8 way 2MB
DRAM	$128 \sim 512$MB ($1 \sim 4$ DIMMs), $t_{RC} = 50$ns
NAND Flash	256MB $\sim$ 2GB random read latency: $25\mu s$(SLC), $50\mu s$(MLC) write latency: $200\mu s$(SLC), $680\mu s$(MLC) erase latency: 1.5ms(SLC), 3.3ms(MLC)
BCH code latency	$58\mu s \sim 400\mu s$
IDE disk	average access latency: 4.2ms[2]

Table 3. Configuration Parameters

Name	Type	Description
uniform	micro	uniform distribution of size 512MB
alpha1, alpha2, alpha3	micro	zipf distribution of size 512MB α=0.8, 1.2, 1.6, $x^{-0.8}$, $x^{-1.2}$, $x^{-1.6}$
exp1, exp2	micro	exponential distribution of size 512MB with $\lambda = 0.01, 0.1$, $e^{-0.01x}$, $e^{-0.1x}$
dbt2	macro	OLTP 2GB database
SPECWeb99	macro	1.8GB SPECWeb99 disk image
WebSearch1, WebSearch2	macro	Search Engine disk access pattern 1 and 2 from [8]
Financial1, Financial2	macro	Financial application access pattern 1 and 2 from [8]

Table 4. Benchmark descriptions

5.2.2 Response to increase in access frequency

If a page is in MLC mode and the entry in the FPST field that keeps track of the number of read accesses to a page saturates, we migrate that Flash page to a new empty page in SLC mode. If there is no empty page available, a Flash block is evicted and erased using our wear-level aware replacement policy. The FPST field that keeps track of the number of accesses for the new Flash page in SLC mode is set to a saturated value. One cannot reduce density at all if a page is already in SLC mode. The SLC/MLC mode field in the FPST is updated to reflect this change.

Reassigning a frequently accessed page from MLC mode to SLC mode improves performance by improving hit latency. Because many accesses to files in a server platform are spatially and temporally a tailed distribution (Zipf), improving the hit latency to frequently accessed Flash pages improves overall performance despite the minor reduction in Flash capacity. We note this analysis was presented in section 4.2.

6 Methodology

6.1 Modeling the Flash and the Flash memory controller

Our proof of concept Flash memory controller and Flash device are implemented on top of a full system simulator called M5 [10]. The M5 simulation infrastructure is used to generate access profiles for estimating system memory and
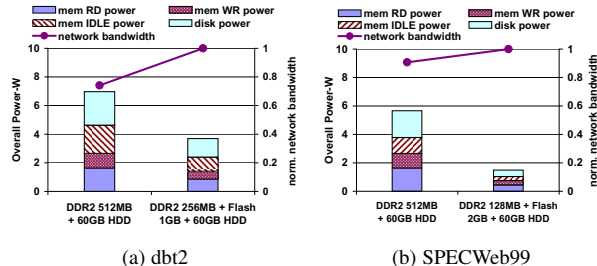
(a) dbt2   (b) SPECWeb99

Figure 9. Breakdown in system memory and disk power and network bandwidth for architecture with/without a Flash based disk cache.

(a) SPECWeb99   (b) dbt2

Figure 10. Average throughput as a function of ECC strength. The system used 256MB of DRAM and 1GB of Flash.

disk drive power consumption. We used the Micron DRAM power calculator [7], NAND Flash datasheet [4] and Hitachi disk specification guide [2] to estimate power. Our die area estimates are derived from the 2007 ITRS roadmap and recently published DRAM and Flash implementations [12]. M5 is also used to generate some of the disk access traces used in this study. Because full system simulators are slow, for reliability and disk cache miss rate experiments where very long traces are necessary, we developed a light weight trace based Flash disk cache simulator. We note that our performance evaluations still use full-system simulation. Given the limitations in our simulation infrastructure, a server workload that uses a large working set of 100's~1000's of gigabytes is not supportable. We scaled our benchmarks, system memory size, Flash size and disk drive size accordingly to run on our simulation infrastructure. Because our disk drive size is small we used power numbers for a typical laptop disk drive [2]. Our configuration is shown in Table 3.

To model wear out behavior we used our exponential model described in section 4.1.3, which models the variation of Flash lifetime. This method is sufficient to understand the impact of a programmable Flash memory controller. Latency values for BCH and CRC codes were taken from the data we presented in section 4.1.

6.2 Benchmarks

As noted previously, we use two different simulators in our experiments. Our disk traces were extracted from micro-benchmarks and macro-benchmarks. These were fed into the Flash disk cache simulator. The binaries were executed on M5. Table 4 summarizes the benchmarks.

We generated micro-benchmark disk traces to model synthetic disk access behavior. It is well known that disk access behavior is often found to follow a power law. Thus we generated disk traces for a Zipf distribution with
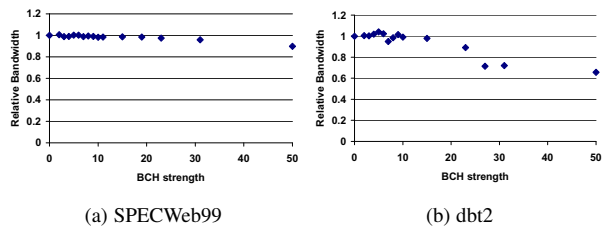
varying alpha values. Additionally for comparison reasons, disk access behaviors that modeled an exponential distribution with a variable exponential coefficient and a uniform distribution were generated. The intent of these micro-benchmarks was to show that many of the macro-benchmarks displayed similar behavior.

For macro-benchmark disk traces, we used disk traces from [8]. These disk traces were intended to model the disk behavior on enterprise level applications like web servers, database servers and web search. To measure performance and power, we used dbt2 (OLTP) and SPECWeb99 which generated representative disk/disk cache traffic.

7 Results

7.1 Overall Flash energy efficiency

Figure 9 shows a breakdown of power consumption in the system memory and disk drive (left y-axis). Figure 9 also shows the measured network bandwidth (right y-axis). We calculated power for a DRAM-only system memory and a heterogenous (DRAM+Flash) system memory that uses a Flash as a secondary disk cache with hard disk drive support. We assume equal die area for a DRAM-only system memory and a DRAM+Flash system memory. Figure 9 shows the reduction in disk drive power and system memory power that results from adopting Flash. Our primary power savings for system memory come from using Flash instead of DRAM for a large amount of the disk cache. The power savings for disk come from reducing the accesses to disk due to a bigger overall disk cache made possible by adopting a Flash. We also see improved throughput with Flash because it displays lower access latency than disk.

7.2 Impact of BCH code strength on system performance

We have shown in Figure 6 that BCH latency incurs an additional delay over the initial access latency. We simu-
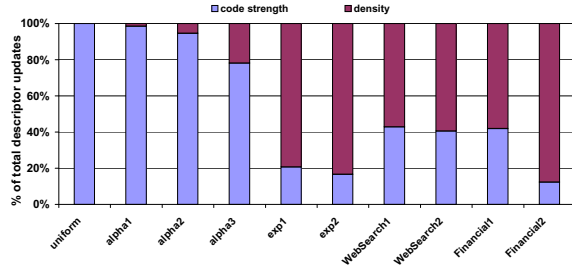
336

Figure 11. Breakdown of page reconfiguration events.

Figure 12. Normalized expected lifetime for a given access rate and the point of total Flash failure.

lated the performance of the SPECWeb99 and dbt2 benchmarks to observe the effect of increasing code strength that would occur as Flash wears out. It is assumed that all Flash blocks have the same error correction code (ECC) strength applied. We also measured performance for code strengths (more than 12 bits per page) that are beyond our Flash memory controller's capabilities to fully capture the performance trends.

From Figure 10 we can see that throughput degrades slowly with ECC strength. dbt2 suffers a greater performance loss than SPECWeb99 after 15 bits per page. The disk bound property of dbt2 makes it more sensitive to ECC strength.

7.3 Flash memory controller sensitivity analysis

Figure 11 shows the breakdown of page reconfiguration events. This can either be a decision to increase ECC strength or switch the block from MLC to SLC mode. The objective is to minimize the latency cost function explained in section 5. The size of Flash was set to half the working set size of the application. These simulations were measured near the point where the Flash cells start to fail due to programs and erases. The results confirm the benefits of a programmable Flash memory controller, because the response to each benchmark is significantly different. The figure also suggests that as the tail length of a workload increases, we see fewer transitions from MLC to SLC, because Flash based disk cache capacity is more important for long tailed distributions. In fact, for a uniform distribution which is an extreme case of a long tailed distribution ($\alpha = 0$), we found almost all descriptor updates are changes in ECC strength and not transitions from MLC to SLC. For exponential distributions, which are an extreme case of short tailed distributions, we see that density (MLC to SLC) changes dominate, because the increased miss rate due to a reduction in density is small. For the macro-benchmarks,
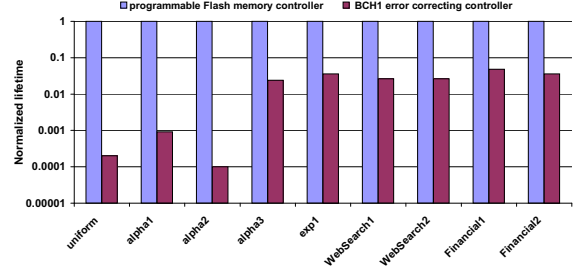
we see a behavior that is fairly high variance, like the micro-benchmarks.

7.4 Improved Flash lifetime with reliability support in Flash memory controller

Figure 12 shows a comparison of the normalized number of accesses required to reach the point of total Flash failure where none of the Flash pages can be recovered. We compare our programmable Flash memory controller with a BCH 1 error correcting controller. Our studies show that for typical workloads, our programmable Flash memory controller extends lifetime by a factor of 20 on average. For a workload that would previously limit Flash lifetime to 6 months, we show it can now operate for more than 10 years using our programmable Flash memory controller. This was accompanied by a graceful increase in overall access latency as Flash wore out.

8 Conclusions

This paper presents an architecture that integrates Flash into a server platform. Flash is an attractive candidate for integration because it reduces power consumption in system memories and disk drives. This in turn can reduce the operating cost of a server platform. By carefully managing the Flash and using it as a secondary disk cache and by also splitting the disk cache into a separate read cache and write cache, we observed a dramatic improvement in power consumption and performance. We also showed that a Flash memory controller with reliability support greatly improves Flash lifetime. We found that the best configuration of a Flash memory controller is largely dependent upon the access patterns resulting from the application. For example, we found that the typical workload with Zipf access behavior was best served by a Flash configured such that the

heavily accessed contents would be located in regions composed of reliable low latency single level cells. In general, we found that variable error correction code strength (ECC) gracefully extended Flash lifetime, and that the overhead of ECC is minimized with configurable density.

Acknowledgments

References

[1] Flash Solid State Drive. http://www.samsung.com/Products/Semiconductor/FlashSSD/index.htm.

[2] Hard Disk Drive Specification Hitachi Travelstar 7K60 2.5 inch ATA/IDE Hard Disk Drive Model: HTS726060M9AT00. http://www.hitachigst.com/tech/techlib.nsf/techdocs/53989D390D44D88F86256D1F0058368D/$file/T7K60_sp2.0.pdf.

[3] Hybrid Hard Drives with Non-Volatile Flash and Longhorn. http://www.samsung.com/Products/HardDiskDrive/news/HardDiskDrive_20050425_0000117556.htm.

[4] Samsung NAND Flash Memory Datasheet. http://www.samsung.com/products/semiconductor/NANDFlash/SLC_LargeBlock/8Gbit/K9K8G08U0A/K9K8G08U0A.htm.

[5] Seagate Barracuda. http://www.seagate.com/products/personal/index.html.

[6] Technical Note: TrueFFS Wear-Leveling Mechanism(TN-DOC-017). http://www.embeddedfreebsd.org/Documents/TrueFFS_Wear_Leveling_Mechanism.pdf.

[7] The Micron System-Power Calculator. http://www.micron.com/products/dram/syscalc.html.

[8] University of Massachusetts Trace Repository. http://traces.cs.umass.edu/index.php/Storage/Storage.

[9] Wear Leveling in Single Level Cell NAND Flash Memories. http://www.st.com/stonline/products/literature/an/10122.pdf.

[10] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, Jul/Aug 2006.

[11] T. Cho, Y. Lee, E. Kim, J. Lee, S. Choi, S. Lee, D. Kim, W. Han, Y. Lim, J. Lee, J. Choi, and K. Suh. A dual-mode NAND flash memory: 1-Gb multilevel and high-performance 512-mb single-level modes. *IEEE Journal of Solid State Circuits*, 36(11), Nov 2001.

[12] T. Hara, K. Fukuda, K. Kanazawa, N. Shibata, K. Hosono, H. Maejima, M. Nakagawa, T. Abe, M. Kojima, M. Fujiu, Y. Takeuchi, K. Amemiya, M. Morooka, T. Kamei, H. Nasu, C. Wang, K. Sakurai, N. Tokiwa, H. Waki, T. Maruyama,

S. Yoshikawa, M. Higashitani, T. D. Pham, Y. Fong, and T. Watanabe. A 146mm$_2$ 8Gb Multi-Level NAND Flash Memory With 70-nm CMOS Technology. *IEEE Journal of Solid State Circuits*, 41(1), Jan 2006.

[13] R. Hobson and K. Cheung. A High-Performance CMOS 32-Bit Parallel CRC Engine. *IEEE Journal of Solid State Circuits*, 34(2), Feb 1999.

[14] D. Ielmini, A. S. Spinelli, A. L. Lacaita, and M. J. van Duuren. A Comparative Study of Characterization Techniques for Oxide Reliability in Flash Memories. *IEEE trans. on device and materials and reliability*, 4(3), Sep 2004.

[15] T. Kgil. *Architecting Energy Efficient Servers*. PhD thesis, University of Michigan, 2007.

[16] T. Kgil and T. Mudge. FlashCache: a NAND flash memory file cache for low power web servers. In *Proc. Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[17] K. Kim and J. Choi. Future Outlook of NAND Flash Technology for 40nm Node and Beyond. In *Workshop on Non-Volatile Semiconductor Memory*, pages 9–11, Feb 2006.

[18] J. Lee, S. Lee, O. Kwon, K. Lee, D. Byeon, I. Kim, K. Lee, Y. Lim, B. Choi, J. Lee, W. Shin, J. Choi, and K. Suh. A 90-nm CMOS 1.8-V 2-Gb NAND Flash Memory for Mass Storage Applications. *IEEE Journal of solid-state circuits*, 38(11), Nov 2003.

[19] S. Lee, Y. Lee, W. Han, D. Kim, M. Kim, S. Moon, H. Cho, J. Lee, D. Byeon, Y. Lim, H. Kim, S. Hur, and K. Suh. A 3.3V 4Gb Four-Level NAND Flash Memory with 90nm CMOS Technology. In *Proc. Int'l Solid-State Circuits Conference*, pages 52–53, 2004.

[20] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.

[21] S. Lin and D. Costello. *Error Control Coding, Second Edition*. 2004.

[22] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. D. Martino, L. D. Onofrio, A. Gambardella, E. Grillea, G. Guerra, D. Kim, C. Missiroli, I. Motta, A. Prisco, G. Ragone, M. Romano, M. Sangalli, P. Sauro, M. Scotti, and S. Won. A 4Gb 2b/cell NAND Flash Memory with Embedded 5b BCH ECC for 36MB/s System Read Throughput. In *Proc. Int'l Solid-State Circuits Conference*, pages 497–506, Feb 2006.

[23] N. Mielke, H. Belgal, I. Kalastirsky, P. Kalavade, A. Kurtz, Q. Meng, N. Righos, and J. Wu. Flash EEPROM Threshold Instabilities due to Charge Trapping During Program/Erase Cycling. *IEEE trans. on device and materials and reliability*, 4(3), Sep 2004.

[24] A. Modelli, A. Visconti, and R. Bez. Advanced Flash Memory Reliability. In *Proc. of Int'l Conf. on Integrated Circuit Design and Technology*, pages 211–218, 2004.

[25] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim. A Low-cost Memory Architecture With NAND XIP for Mobile Embedded Systems. In *Proc. Int'l Conf. on HW-SW Codesign and System Synthesis(CODES+ISSS)*, Oct 2003.

[26] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Oper. Sys.*, Oct. 1994.