Analysis, Modeling and Design of Flash-based Solid-State Drives

A dissertation presented by

Simona Boboila

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts

December, 2012
NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: Analysis, Modeling and Design of Flash-based Solid-State Drives

AUTHOR: Marcela Simona Boboila

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Date

12/6/2012

Date

12/4/2012

Date

12/4/2012

Date

12/5/2012

Date

GRADUATE SCHOOL APPROVAL:

Date

12/5/2012

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Date

12/6/2012

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Abstract

Flash storage has rapidly grown over the last few years into a redoubtable competitor to disk drives, due to its lower power consumption, lower latencies and lower cost per IOPs. Along with its attractive characteristics, flash also presents challenges, such as limited write endurance and the inability to overwrite a block in place. To address these issues, a software called the flash translation layer (FTL) maps logical blocks to physical locations on flash. Externally, the FTL presents a normal hard disk interface; internally, it implements address translation, garbage collection and wear-leveling algorithms which spread writes uniformly across the device.

In this thesis we provide a detailed investigation of flash device characteristics and internal logic, drawing valuable insight regarding their impact on higher-level properties such as whole-device endurance and throughput efficiency. Based on the derived characteristics, we model device endurance as a function of both the parameters of the chip itself, and the details of the internal algorithms used. We also construct analytic and black-box models to predict performance of solid-state drives under real workloads, and validate them by measurements, both in simulation and on real devices.

In addition, we investigate new design choices, specifically the integration of active computation capability in SSDs. We propose a novel approach, Active Flash, to migrate data analysis in scientific computing to the location of the data, the flash device itself; this can significantly reduce bandwidth, energy and time costs in high-performance computing clusters. We explore energy and performance trade-offs in moving computation from host to storage, demonstrate the ability of embedded controllers to perform data analysis and reduction tasks at acceptable speeds, present a simulation study of scheduling policies, and implement an Active Flash prototype. These results show the viability of the Active Flash model, and its capability to potentially have a transformative impact on scientific data analysis.
Acknowledgements

I would particularly like to thank my advisor, Peter Desnoyers, for his invaluable support, collaboration and advice. Peter has inspired me both as a researcher and an educator, transforming my PhD journey into a great learning and growing experience.

I am also very grateful to Sudharshan Vazhkudai for his ideas and discussions on the active flash work presented here, to Youngjae Kim, whose help and comments have contributed significantly to this work, and to my other collaborators Devesh Tiwari and Xiaosong Ma for their valuable suggestions. Special thanks go to Ravi Sundaram for his mentorship during my first year and for serving on my thesis committee, as well as to my other committee members Gene Cooperman and Ningfang Mi for their helpful comments and suggestions. I’d also like to thank my colleagues from the Solid-State Storage Lab and High-Performance Computing Lab at Northeastern for their friendship and the stimulating research environment they provided during my PhD.

Finally, I’d like to express my gratitude to my family for their love, encouragement and support throughout my education: to my parents Cornelia and Cristea, my grandmother Teodora, my brother Cristian, and my sister-in-law Maria-Vivienne for their caring and always comforting advice; to my grandfather Nicolae for inspiring me to reach far; to my uncle Marian Velcea, whose innovative thinking has been an example to me; to Simi and my cousin Alex for the pleasant get-togethers; and to Vlad, for sharing both the joys and the difficulties during these years.
Contents

Abstract i

Acknowledgements iii

Contents iv

List of Figures vii

List of Tables x

1 Introduction 1
  1.1 Thesis Contributions ............................................. 2
  1.2 Thesis Outline ..................................................... 6

2 Background and Related Work 7
  2.1 Background ......................................................... 7
    2.1.1 General SSD architecture .................................. 7
    2.1.2 Flash Translation Layer (FTL) .............................. 8
  2.2 Related Work .................................................... 12

3 Flash Characteristics, Internal Logic and Endurance 15
  3.1 Reverse Engineering .............................................. 16
List of Figures

2.1 The general architecture of an SSD ................................. 8
3.1 USB Flash drive modified for logic analyzer probing ............. 17
3.2 Generic device page update ........................................... 20
3.3 House device single-page update .................................... 21
3.4 House device multi-page update .................................... 22
3.5 Memorex device page update ....................................... 23
3.6 House device write timing ........................................... 24
3.7 Memorex device garbage collection patterns ....................... 25
3.8 Memorex device static wear-leveling .............................. 26
3.9 Timing patterns indicating static wear-leveling in the newer house-brand flash drive ....................................................... 27
3.10 House device end-of-life signature .................................. 27
3.11 Unscheduled access vs. optimal scheduling for disk and flash ........... 32
4.1 Division of a request into head, middle, and tail fragments .......... 39
4.2 Analytic model for FAST, and validation by simulation ............ 45
4.3 Interpolation-based prediction model for FAST ..................... 47
4.4 Model extension to real workloads ................................... 48
5.9 Computation throughput of the data analysis applications running on the SSD controller, and on the host CPU
5.10 Computation time for 30 GB input data on the controller and on the host CPU
5.11 Slowdown and energy savings
5.12 Limiting factors in Active Flash
5.13 I/O-bound data analysis
5.14 Computation-bound data analysis
5.15 In-storage data analysis during idle times in hybrid schemes
5.16 Data analysis with Garbage Collection management
5.17 Active Flash prototype
5.18 Format of a data analysis request command
5.19 Competing I/O and data analysis tasks. Preemption of data analysis
List of Tables

3.1 Investigated devices .............................................................. 17
3.2 Characteristics of reverse-engineered devices ......................... 18
3.3 Predicted and measured endurance limits .................................. 30
4.1 List of variables: device and workload parameters .................... 39
4.2 SSD parameters in the MSR simulator ..................................... 67
5.1 List of variables: device and data analysis parameters ............... 89
5.2 Measured and derived parameters for speed and power consumption .... 92
5.3 Data analysis applications. Measured timings and data reduction .... 99
5.4 SSD parameters in the MSR simulator ..................................... 104
5.5 Data analysis-related parameters in the SSD simulator ............... 104
5.6 SSD parameters in the OpenSSD platform .................................. 114
5.7 Throughput of data analysis kernels running on the SSD controller within the Active Flash prototype, and on the Host CPU ....................... 120
5.8 Run time splitting between I/O and computation ....................... 121
5.9 Power rate in Active Flash versus Host during idle time, sustained writing to flash, and data analysis ............................................. 122
5.10 Energy consumption in Active Flash versus Host ..................... 122
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Performance prediction method for real workloads.</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>FAST and Pagemap FTLs.</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>BAST FTL.</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>Computing average log utilization.</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Flash storage has grown in the last years, not only as the main technology in embedded media players, photography, portable drives, etc., but also as an important competitor to hard disks in the form of solid state drives (SSDs). NAND flash technology provides persistent storage with remarkable advantages: high density, low power consumption, low latencies.

Flash storage also presents challenges which are difficult to overcome, such as the impossibility of doing in-place updates, and limited write endurance. Flash is electrically addressable, but its capability to store electrical charge, and thus to accurately represent bits, decreases in time, resulting in limited endurance. Also, unlike hard disks, flash can not be written in-place. Each new write has to be redirected to a free page, and the old copy invalidated. An internal firmware called flash translation layer (FTL) running on the flash device addresses these issues by mapping logical blocks to specific physical addresses. The FTL handles out-of-place updates and uniform distribution of writes across the device, in a way that attempts to maximize efficiency and endurance.

Understanding physical and design characteristics of flash devices, and how they ultimately affect real-world performance is a fundamental step in flash storage research, essential for best utilization of flash, and successful integration in new architectures. However, these characteristics are little known, and their implementation is often proprietary.
In addition, flash storage is yet a young field, with tremendous potential for new ideas, optimizations on existing technology and algorithms, and novel design features. One such feature, which we propose and explore in this thesis, is the capability of performing active computations. Due to their high I/O bandwidth, integrated high-speed controllers and low latencies, flash-based solid-state drives are a promising candidate for active computation. Enabling active in-storage computing reduces data movement costs and energy consumption, and frees the host CPU for other tasks. This is highly desirable in various fields, such as high-performance scientific computing (HPC).

This thesis presents new performance models of flash storage, with a focus on the following performance factors: endurance, throughput efficiency and active computation capability. We present extended studies of flash device characteristics and internal algorithms, and model their effect on device endurance and throughput efficiency under real workloads. Moreover, we propose an active storage model, and demonstrate its feasibility and performance advantages on realistic HPC applications.

**Thesis Statement**

Flash storage modeling based on performance factors such as endurance, throughput efficiency and active computation capability provides insight into real-world performance and potential design optimizations of flash devices, as demonstrated via implementation and measurements, with real workloads and applications.

1.1 Thesis Contributions

The research contributions of this thesis are three-fold, consisting in modeling and analysis of flash storage endurance, throughput efficiency and active computation capabilities.

Each of these factors contributes to the performance of flash devices. For a better under-
standing of flash performance as a whole, we propose to investigate all these aspects, as they depend on common grounds (i.e. the underlying characteristics of flash), and often interact.

Here are a few examples: (1) data layout on flash affects both endurance and throughput efficiency; (2) active computation and I/O tasks compete for the SSD controller, and require good scheduling strategies to maximize both throughput and active computation performance; (3) throughput efficiency models also solve the converse problem of determining idle times in workloads, during which active computation can be scheduled to run.

We next describe the specific contributions related to each of the studied performance factors:

**Endurance:** We investigate the FTLs (i.e. internal logic implementation) of three low- to high-end USB flash drives, providing detailed parameters of each FTL, such as zone organization, reserved list size, mapping scheme, and wear-leveling methods. The methodology used includes reverse engineering with logic analyzer probing and timing analysis of operation latencies. Based on the derived FTL-related information, we construct a prediction model for overall device endurance as a function of both the parameters of the chip itself, and the details of the internal algorithms used. We further validate this model with actual endurance measurements on USB flash drives.

The results related to this contribution are presented in [BD10].

**Throughput efficiency:** We develop methods to estimate the throughput efficiency of flash devices for real workloads, both as white-box and black-box predictions. The models estimate performance (specifically, throughput efficiency) based on two parameters: sequentiality in workloads, and utilized free space.

We start from the well-behaved case of uniform traffic, with uniformly distributed requests over the address space, and exponentially distributed request lengths. Next, the models are extended to non-uniform workloads, by dividing the workload in small segments and
approximating them to uniform traffic. We propose and analyze methods to define the comprising segments and estimate the equivalent per-segment parameters (i.e. sequentiality and free space).

The white-box predictions approach (i.e. assuming knowledge of device’s internal algorithm), consists of analytic models for two commonly-used flash translation layer algorithms. Uniform writes are modeled probabilistically, by estimating the chance of requests to trigger garbage collection due to eviction from the free list. The accuracy of these models is demonstrated on real workloads incorporating simulation-based approximations.

The black-box predictions approach (i.e. without knowledge of device’s internal algorithm) is represented by an interpolation-based methodology validated on real workloads both in simulation and on real SSDs. First, multiple performance values are measured for different values of free space utilization and sequentiality (i.e. mean request length). Next, the remaining performance values are derived, by interpolating from the measured values. The individual contribution of reads and writes to the overall throughput in real workloads is also evaluated.

In addition, we present a detailed discussion of potential inaccuracy factors of these models. These inaccuracies may emerge from the difficulty to measure or capture in the model information related to: (a) specific device parameters (e.g. page/block size, free space), (b) zone organization, (c) number of channels, allocation policies, striping, (d) flash translation layer algorithm, and (e) specific optimization tweaks in SSDs, buffering policies.

The results related to this contribution are presented in [BD11].

**Active computation capability:** We propose an active storage model on flash, called *Active Flash*, and evaluate its applicability in scientific high performance computing. Typically, scientific HPC applications generate large volumes of data on the compute nodes, store this data on a shared data store (e.g. a centerwide parallel file system such as Lustre [Lus]),
and then perform various analysis tasks on it, such as data reduction, feature extraction, statistical processing and visualization. This leads to many rounds of I/Os between the storage and the host CPU on the node, with severe effects on performance in large scale systems, due to storage bandwidth limitations. Active Flash reduces the data transfer costs by moving the post-processing step closer to where data already resides, from the host CPU onto the storage controller.

We conduct a detailed feasibility study with realistic data analysis applications to demonstrate that the generally slower controller (however increasingly faster with new generation embedded processors and multiple integrated CPUs) can carry out this task in acceptable time. With power consumption in today’s HPC centers already reaching megawatt values, Active Flash addresses this issue by using embedded processors which are significantly more power-efficient compared to host CPUs. The performance-energy trade-offs of the proposed architecture are explored in this thesis. Moreover, we conduct a simulation study of scheduling policies in Active Flash, examining the possibilities of scheduling controller-resident computation along with host I/Os and flash management tasks.

We implemented a proof-of-concept prototype for Active Flash using the Jasmine OpenSSD experimental platform [Jas]. The prototype features a host–controller communication protocol to send and intercept data analysis commands; also, it handles contention of data analysis tasks and I/O tasks on the controller, solved by preemption of data analysis; in addition, it implements four data analysis applications from the field of statistical analysis. The code is co-located with the FTL logic and runs on the SSD controller.

The prototype is described in [TBV+13], and the other results related to this contribution are presented in [BKV+12].
1.2 Thesis Outline

This paper is structured as follows: Section 2 presents background and related work in the field. Section 3 describes our work on investigating flash characteristics and internal logic, along with a model to predict device endurance. Section 4 presents our results in modeling the throughput efficiency of flash devices on real workloads. Section 5 describes a new architectural model, Active Flash, which proposes to run active computations on flash and thus reduce data movement costs. Finally, Section 6 concludes the thesis.
Chapter 2

Background and Related Work

2.1 Background

2.1.1 General SSD architecture

An SSD as shown in Figure 2.1 is a small general-purpose computer, based on one or multiple 32-bit CPUs and typically 64-128 MB of DRAM, or more for high-end PCI-based devices. The CPUs are integrated in the Flash Controller, which handles data transfers and error correction (ECC). In addition an SSD contains Host Interface Logic, implemented over SATA or PCIe, and an array of NAND flash memory chips comprising the storage itself. Other flash devices, e.g. USB drives, are a simplified and less efficient version of an SSD, with the same main components: flash memory, controller, RAM memory, and host interface.

Flash storage is organized in pages, which are read and written as a unit, while pages (typically 4 KB page size) are organized in blocks of 64-256 pages (typically 128) which must be erased as a unit before pages may be re-written; this operation is time-consuming (2 ms or more) but rare in comparison to writes. Host read and write operations consist of a command, a busy period, and a data transfer phase. The busy period (i.e. on-board read/write activity from/to the flash memory chip) typically takes 50 $\mu$s for a page read and 200-300 $\mu$s for a page write. In the case of reads, the busy period is followed by a page
transfer phase from the flash chip to the DRAM, while for writes, the busy period is preceded by the page transfer phase from DRAM to the flash chip; this transfer phase typically takes 40-100 $\mu$s. In order to increase the read and write bandwidth, operations are performed simultaneously to multiple chips over the same bus (multi-way interleaving, in which case the data transfer phase can not occur in parallel for multiple chips, but the write to flash itself can), and across multiple buses (multi-channel interleaving).

### 2.1.2 Flash Translation Layer (FTL)

Unlike HDD drives supporting re-writable sectors, and thus performing in-place updates, SDDs provide a fundamentally different mechanism based on out-of-place updates. Write to flash can only occur on a clean page; however, cleaning is an expensive block-level operation. Performing a block erasure for every page update would be undesirable. To address this issue, a flash management firmware called the Flash Translation Layer (FTL) implements an out-of-place updates mechanism: the updates are temporarily stored in separate log blocks which are page-mapped, and later merged to keep only the valid writes (most recent copy to a logical page) and erase the stale data. Thus the role of a flash translation layer is to
provide a re-writable disk-like block abstraction on top of flash devices. The FTL runs on the internal controller, which ranges from e.g. 80 MHz ARM CPU (Indilinx Barefoot) in low-end consumer devices, to higher processor speeds to reduce operation latencies in high-end SSDs, e.g. four 780 MHz Tensilica cores in the OCZ RevoDrive X2.

**Merging and garbage collection:** Whenever units smaller than an erase block are mapped, there can be *stale* data: data which has been replaced by writes to the same logical address (and stored in a different physical location) but which has not yet been erased. Recovering these pages consists of merging log blocks with blocks containing stale data, and programming the result into one or more free blocks. These operations are of the following types: switch merges, partial merges, and full merge [GKU09a].

a) **Switch merge:** occurs during sequential writing; the log block contains a sequence of pages exactly replacing an existing data block, and may replace it without any further operation; the old block may then be erased.

b) **Partial merge:** copies valid pages from a data block to the log block, after which the two may be switched.

c) **Full merge:** is needed when data in the log block is out of order; valid pages from the log block and the associated data block are copied together into a new free block, after which the old data block and log block are both erased.

**Mapping schemes:** Common FTL mapping schemes include Page-mapped, BAST (Block Associative Sector Translation) and FAST (Fully Associative Sector Translation):

a) **Page-mapped FTL:** no specific ordering is preserved for logical-to-physical page mappings, thus requiring large mapping tables with an entry for each page: for example, a 64 GB device requires 32 to 64 MB of RAM (even more for older small-page devices).
b) Block Associative Sector Translation, or BAST FTL [KKN+02b], (also called Hybrid Log Block): the logical space is organized in fully ordered block-mapped data blocks. The data blocks are written consecutively from beginning to end, such that the pages within the physical block are in logical order. All updates to a particular data block are first redirected to the same log block, which is later merged with its corresponding data block to reclaim space and restore the logical page order. In the best case the log block is written consecutively from beginning to end, and can be switched into place with no overhead; for non-sequential or partial writes it is necessary to merge valid pages from the log and data blocks into a third block, possibly incurring significant overhead. In particular, if a log block is merged after $n$ pages have been written, this will result in a write amplification factor, or ratio of internal physical writes to external logical writes, of $B/n$ for block size $B$. For randomly distributed small writes, virtually every write will evict a log block containing a single page, for a write amplification factor of $B$ (where $B$ can be as high as 256).

c) Fully Associative Sector Translation, or FAST FTL [LPC+07] attempts to maximize the utilization of log blocks by allowing pages from multiple data blocks to share a log block. The log blocks are arranged in a queue; at the tail, any pages not invalidated while in the queue are merged with their corresponding data blocks. This performs well for workloads with spatial locality, as pages are invalidated while on the log block queue; in effect the queue functions as a fully associative cache in front of the data blocks (vs. BAST, where associativity is only within single-block sets). For random traffic, however, very few pages will be invalidated in the queue, resulting in a write amplification factor of nearly $B$. Thus, while FAST addresses BAST’s issue of potentially inefficient log list utilization, it introduces another downside: since logs contain pages from multiple data blocks, multiple merges will be needed to reclaim space, unless most pages are invalidated while being in the log list (as in the case of workloads
with high locality).

Other FTLs proposed to date include Park’s N/K adaptive algorithm [PCK+08], which combines elements of BAST and FAST, Superblock [KJKL06], which uses page mapping within groups of data blocks, and FASTer [LLM10], which enhances FAST with a hot-cold page separation policy.

**Wear-leveling:** In addition to address mapping and garbage collection, another important function of the FTL is wear-leveling. Unlike disks, flash memory presents limited write endurance, meaning that bits in a flash chip will fail after a limited number of writes, typically quoted at $10^4$ to $10^5$. To avoid over-use of a small number of blocks (e.g. may occur for applications with high write locality) and early failure, flash devices implement wear-leveling algorithms which vary the logical-to-physical address mappings to ensure that writes are spread uniformly across the entire device. Wear-leveling algorithms [GT05] are classified as either dynamic or static.

a) Dynamic wear-leveling operates only on over-written blocks, rotating writes between blocks on a free list; thus if there are $m$ blocks on the free list, repeated writes to the same logical address will cause $m + 1$ physical blocks to be repeatedly programmed and erased.

b) Static wear-leveling spreads the wear over both static and dynamic memory regions, by periodically swapping active blocks from the free list with static randomly-chosen blocks. This movement incurs additional overhead, but increases overall endurance by spreading wear over the entire device.
2.2 Related Work

Our flash investigation work combines both physical-level methods (i.e. the reverse engineering of read/write/erase operations) and system-level methods (i.e. timing analysis, whole-device measurements, performance modeling) to provide an extensive study of flash characteristics, endurance and performance. Previous physical-level studies include measurements of program/erase cycling and their effect on electrical characteristics [LCPK03, PAC\textsuperscript{+}09, YKP\textsuperscript{+}06], as well as empirical characterization of basic operations performance, power consumption, and reliability [Des09, GCC\textsuperscript{+}09].

System-level studies have instead examined characteristics of entire flash-based storage systems, by running multiple benchmarks to understand flash device performance under various IO patterns [AMMT08, HCK\textsuperscript{+}08, OSSP08]. A recent example is uFLIP [BJB09], a benchmark used to measure a wide range of devices and quantify the degraded performance observed for random writes in many such devices. While benchmarking studies use synthetic traces with predefined patterns, we extend our performance modeling to real traces covering workloads from different activity domains.

Analytic performance models for disk drives and disk arrays have been studied for many years [LK93, RW94, UAM, VMX\textsuperscript{+}03, VMXQ04]. Shiver et al. [SMW98] develop analytic models of disk drives, where performance prediction for a storage device is a function of both the storage device itself and the workload. Their workload modeling, which attempts to capture temporal and spatial locality in request streams, is analogous to our work; however, the models are designed for disks, with specific mechanically-determined characteristics which are far different from those of flash storage systems.

Analytic FTL performance models have been studied by [Rob96, BT06, HEH\textsuperscript{+}09, BI10, Des12, XK12]. All these studies address single-page uniformly distributed writes for page-mapped FTL with LRU cleaning and Greeding cleaning (i.e. the least recently written block,
or the block with most invalid pages, respectively, is selected for cleaning). LRU cleaning performance is modeled by Robinson in the context of log-structured file systems [Rob96], however he does not provide a closed-form solution; two more recent studies of Desnoyers [Des12] and Xiang [XK12] provide an analytic expression for LRU cleaning. Greedy cleaning performance is studied by Bux using Markov models [BI10], but no analytic expression is given, while Desnoyers derives an analytical model which also provides a closed-form expression [Des12].

This previous body of work describes analytic models for page-mapped FTLs with single-page uniformly distributed writes. Instead, in this thesis we are modeling non-ideal FTLs (BAST and FAST), for data with spatial locality, as is typical of real workloads (i.e. write bursts rather than single-page writes) and presenting a methodology for applying these models to real workloads (i.e. non-uniform traffic).

Our methodology to use interpolated measurement-based performance functions is related to black-box performance models, such as those proposed for disk drives using statistical machine learning techniques [WAA+04, YUK06]. Huang and Li [HLST11, LH10] study black-box performance models for SSDs, which utilize statistical machine learning algorithms to capture correlations between workload characteristics and observed performance values. However, their experiments are mainly based on synthetic traces, with a predefined set of access patterns, while we provide a more extensive study of the performance models on real traces, which are significantly harder to predict, and also analyze potential inaccuracy factors on real devices.

Active storage techniques that move computation closer to data originated with the Gamma database machines [DH81] in the ’80s which used special-purpose hardware to speed up scans and other simple database operations. However, it was a decade later when the shift to 32-bit controllers for disk drives inspired Active Disks, the first proposals to move computation on the storage controller itself [KPH98, RGF98]. Simple database tasks, filtering,
image processing, etc. were demonstrated in this environment, but they had little success due to multiple limitations: low and hardly increasing computation power of disk-resident controllers, low I/O bandwidth and latencies of disks, difficulty in programming necessary interfaces.

More recently, active storage concepts have also been pursued in the context of parallel file systems [PNF07, SLC+10] such as the Lustre parallel file system, harnessing the computing power of the storage nodes to reduce the data transfer between the storage and compute nodes. In addition, cluster architectures have been explored, consisting of low-power CPUs coupled to small local flash storage [gor09, AFK+09] to achieve fast, parallel access to data.

Recent work by Kim et al. [KOP+11] has studied the feasibility of SSD-resident processing on flash in the context of database scan operations with promising results, also proposing a dedicated hardware logic to speed up scans. Our research extends this work with general models for performance–energy and computation–I/O tradeoffs, experimental verification of these models, and a proof-of-concept prototype implementing a host-controller communication mechanism and computation tasks co-located with the FTL logic running on the SSD controller. In addition, simulation of the SSD and I/O traffic is used to explore potential scheduling policies in detail, examining contention between I/O and computation and its effect on the application data generation rate that can be handled by the controller-resident data analysis.
Unlike hard disks, flash devices present limited write endurance, meaning that bits can only be stored reliably on flash for a limited number of times, usually quoted at $10^4$ to $10^5$ writes depending on the specific device. After that, flash cells are not able to store electrical charge reliably, and writes fail. Internal algorithms – called wear-leveling algorithms – implemented in the Flash Translation Layer are designed to spread writes uniformly across the device, attempting to maximize the life of the device.

The endurance of a flash-based storage system such as a USB drive or SSD is a function of both the parameters of the chip itself, and the details of the wear-leveling algorithm (or Flash Translation Layer, FTL) used. To date there is little insight into the endurance characteristics of flash devices, as measured endurance data is closely guarded by semiconductor manufacturers, and FTL details are typically proprietary and hidden within the storage device.

Designing systems that use flash devices most efficiently requires knowledge of flash internal characteristics and logic, and their impact on device-level properties such as endurance and throughput efficiency. This work examines design characteristics, internal logic and endurance of flash devices, using various techniques: reverse engineering, timing analysis, and whole device testing. We concentrate here on relatively small mass-market USB drives due to
the difficulties inherent in reverse-engineering and destructive testing of more sophisticated devices.

The contributions of this work include:

- Two methodologies for investigating flash device characteristics: a low-level method using logic analyzer probing, and a high-level method consisting in timing analysis of operation latencies,

- Characterization of three low- to high-end USB flash drives. Detailed parameters of each FTL are provided, such as zone organization, reserved list size, mapping scheme, and wear-leveling methods,

- Prediction model for flash devices endurance, based on the derived characteristics,

- Validation of the model by actual endurance measurements on USB flash drives,

- Discussed implications of this work for storage systems, including space management (scheduling of I/O requests), and applicability of our investigation methods to a larger class of flash devices.

3.1 Reverse Engineering

**Methodology:** We reverse-engineered FTL operation in three different USB drives, as listed in Table 3.1: *Generic*, an unbranded device based on the Hynix HY27US08121A 512Mbit chip, *House*, a MicroCenter branded 2GB device based on the Intel 29F16G08CANC1, and *Memorex*, a 512MB Memorex “Mini TravelDrive” based on an unidentified part.

In Figure 3.1 we see one of the devices with probe wires attached to the I/O bus on the flash chip itself. Reverse-engineering was performed by issuing specific logical operations from a Linux USB host (by issuing direct I/O reads or writes to the corresponding block.
Table 3.1: Investigated devices.

<table>
<thead>
<tr>
<th>Device</th>
<th>Size</th>
<th>Chip Signature</th>
<th>USB ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>512Mbit</td>
<td>HY27US08121A</td>
<td>1976:6025</td>
</tr>
<tr>
<td>House</td>
<td>16Gbit</td>
<td>29F16G08CANC1</td>
<td>125F:0000</td>
</tr>
<tr>
<td>Memorex</td>
<td>4Gbit</td>
<td>MF12G2BABA</td>
<td>12F7:1A23</td>
</tr>
</tbody>
</table>

device) and using an IO-3200 logic analyzer to capture resulting transactions over the flash device bus. From this captured data we were then able to decode the flash-level operations (read, write, erase, copy) and physical addresses corresponding to a particular logical read or write.

![USB Flash drive modified for logic analyzer probing.](image)

**Figure 3.1:** USB Flash drive modified for logic analyzer probing.

We characterize the flash devices based on the following parameters: *zone organization* (number of zones, zone size, number of free blocks), *mapping schemes*, *merge operations*, *garbage collection frequency*, and *wear-leveling algorithms*. Investigation of these specific attributes is motivated by their importance; they are fundamental in the design of any FTL [Ban95, Ban99, Int98, KJKL06, KL02, KKN+02a, LSKK08], determining space requirements, i.e. the size of the mapping tables to keep in RAM (zone organization, mapping schemes), overhead/performance (merge operations, garbage collection frequency), device
endurance (wear-leveling algorithms). The results are summarized in Table 3.2, and discussed in the next sections.

**Table 3.2:** Characteristics of reverse-engineered devices.

<table>
<thead>
<tr>
<th></th>
<th>Generic</th>
<th>House</th>
<th>Memorex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>16 zones</td>
<td>4 zones</td>
<td>4 zones</td>
</tr>
<tr>
<td><strong>Zone size</strong></td>
<td>256 physical blocks</td>
<td>2048 physical blocks</td>
<td>1024 physical blocks</td>
</tr>
<tr>
<td><strong>Free blocks list size</strong></td>
<td>6 phys. blocks/zone</td>
<td>30-40 phys. blocks/zone</td>
<td>4 phys. blocks/zone</td>
</tr>
<tr>
<td><strong>Mapping scheme</strong></td>
<td>Block-level</td>
<td>Block-level / Hybrid</td>
<td>Hybrid</td>
</tr>
<tr>
<td><strong>Merge operations</strong></td>
<td>Partial merge</td>
<td>Partial merge / Full merge</td>
<td>Full merge</td>
</tr>
<tr>
<td><strong>Garbage collection frequency</strong></td>
<td>At every data update</td>
<td>At every data update</td>
<td>Variable</td>
</tr>
<tr>
<td><strong>Wear-leveling algorithm</strong></td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
</tbody>
</table>

**Zone organization:** The flash devices are divided in zones, which represent contiguous regions of flash memory, with disjoint logical-to-physical mappings: a logical block pertaining to a zone can be mapped only in a physical block from the same zone. Since the zones function independently from each other, when one of the zones becomes unusable, other zones on the same device can still be accessed. Each zone has its own free list. We determine the size of the free list for each zone by repeatedly writing to a particular page in that zone, and counting the number of distinct log blocks cycled through in a round-robin fashion, to store the write. We detect zone boundaries by writing to different pages across the device, and observing when a different free list is used. We report actual values of zone sizes and free list sizes for the investigated devices in Table 3.2.

**Mapping schemes:** In a page-level FTL mapping scheme, the logical page requested at upper level can be mapped to any physical page on the flash. On the other hand, in a block-level FTL mapping scheme the logical page has an invariant offset within a block: the logical page number is always equal to the physical page number. For this reason, block-mapped FTLs require smaller mapping tables to be stored in RAM, compared to
page-mapped FTLs. Therefore, the block-level mapping scheme is more practical and was identified in both Generic and multi-page updates of House flash drives. For single-page updates, House uses the simplified hybrid mapping scheme (which we will describe next), similar to Ban’s NFTL [Ban99]. The Memorex flash drive uses hybrid mapping: the data blocks are block-mapped and the log blocks are page-mapped.

**Garbage collection:** For the Generic drive, garbage collection is handled immediately after each write, eliminating the overhead of managing stale data. For House and Memorex, the hybrid mapping allows for several sequential updates to be placed in the same log block. Depending on specific writing patterns, garbage collection can have a variable frequency. The number of sequential updates that can be placed in a 64-page log block (before a new free log block is allocated to hold updated pages of the same logical block) ranges from 1 to 55 for Memorex and 1 to 63 for House.

We illustrate how garbage collection works after being triggered by a page update operation.

*The Generic flash drive* implements a simple page update mechanism (Figure 3.2). When a page is overwritten, a block is selected from the free block list, and the data to be written is merged with the original data block and written to this new block in a partial merge, resulting in the erasure of the original data block.

*The House drive* allows multiple updates to occur before garbage collection, using an approach illustrated in Figure 3.3. Flash is divided into two planes, even and odd (blocks B-even and B-odd in the figure); one log block can represent updates to a single block in the data area. When a single page is written, meta-data is written to the first page in the log block and the new data is written to the second page; a total of 63 pages may be written to the same block before the log must be merged. If a page is written to another block in the plane, however, the log must be merged immediately (via a full merge) and a new log
Figure 3.2: Generic device page update. Using block-level mapping and a partial merge operation during garbage collection. LPN = Logical Page Number. New data is merged with block A and an entire new block (B) is written.

We observe that the House flash drive implements an optimized mechanism for multi-page updates (Figure 3.4), requiring 2 erasures rather than 4. This is done by eliminating the intermediary storage step in log blocks B-even and B-odd, and writing the updated pages directly to blocks C-even and C-odd.

The Memorex flash drive employs a complex garbage collection mechanism, which is illustrated in Figure 3.5. When one or more pages are updated in a block (B), a merge is triggered if there is no active log block for block B or the active log block is full, with the following operations being performed:

- The new data pages together with some settings information are written in a free log block (Log\textsubscript{B}).

- A full merge operation occurs, between two blocks (data block A and log block Log\textsubscript{A}) that were accessed 4 steps back. The result is written in a free block (Merged\textsubscript{A}). Note that the merge operation may be deferred until the log block is full.
Figure 3.3: House device single-page update. Using hybrid mapping and a full merge operation during garbage collection. LPN = Logical Page Number. LPN 4 is written to block B, “shadowing” the old value in block A. On garbage collection, LPN 4 from block B is merged with LPNs 0 and 2 from block A and written to a new block.

- After merging, the two blocks (A and Log A) are erased and added to the list of free blocks.

In addition to the page of settings that is written to a log block, the Memorex flash drive also dedicates one full block to storing settings. At every write, one page is written to the settings block and is read back before the next update of the same data. The settings block is written sequentially. When it gets filled, a new settings block is allocated from the list of free blocks, and the old block is erased.

Reverse engineering also brought out some aspects concerning space utilization in log blocks for the Memorex flash drive. First, log blocks are not filled with data pages to their full capacity: up to 55 sequential updates can be placed in a 64-pages log block before a new log block is allocated. Second, small size logical pages (e.g. 512 bytes) are always mapped
at the beginning of the 2K physical page in the log block, the remaining portion of the 2K physical page remaining unused.

**Wear-leveling aspects:** From the reverse-engineered devices, static wear-leveling was detected only in the case of the Memorex flash drive, while both Generic and House devices use dynamic wear-leveling. As observed during the experiments, the Memorex flash drive is periodically (after every 138th garbage collection operation) moving data from one physical block containing rarely updated data, into a physical block from the list of free blocks. The block into which the static data has been moved is taken out of the free list and replaced by the rarely used block.
Figure 3.5: Memorex device page update. Using hybrid mapping and a full merge operation during garbage collection. LPN = Logical Page Number. LPN 2 is written to the log block of block B and the original LPN 2 marked invalid. If this requires a new log block, an old log block (Log_A) must be freed by doing a merge with its corresponding data block.

Conclusions: The three devices examined were found to have flash translation layers ranging from simple (Generic) to somewhat complex (Memorex). Our investigation provided detailed parameters of each FTL, including zone organization, free list size, mapping scheme, and static vs. dynamic wear-leveling methods. We will later demonstrate the use of these parameters to predict overall device endurance.

3.2 Timing Analysis

Additional information on the internal operation of a flash drive may be obtained by timing analysis—measuring the latency of each of a series of requests and detecting patterns in the results. This is possible because of the disparity in flash operation times, typically 20µs,
200-300μs, and 2-4ms for read, write and erase respectively [Des09]. Selected patterns of writes can trigger differing sequences of flash operations, incurring different delays observable as changes in write latency. These changes offer clues which can help infer the following characteristics: (a) wear-leveling mechanism (static or dynamic) and parameters, (b) garbage collection mechanism, and (c) device end-of-life status. Thus, by analyzing write latency for varying patterns of operations we are able to determine properties of the underlying flash translation algorithm, some of which were also be verified by reverse engineering.

**Methodology:** Timing analysis uses sequences of writes to logical addresses \(\{A_1, A_2, \ldots, A_n\}\) which are repeated to provoke periodic behavior on the part of the device. The most straightforward sequence is to repeatedly write to the same address (e.g. write a specific page repeatedly); these writes completed in constant time for the Generic device, while results for the House device are seen in Figure 3.6. These results correspond to the FTL algorithms observed in Section 3.1 above; the Generic device performs a full block copy and erase for every page write, while the House device is able to write to Block B (see Figure 3.3) 63 times before performing a merge operation and corresponding erase.

![Figure 3.6: House device write timing. Write address is constant; peaks every 63 operations correspond to the merge operation (including erasure) described in Section 3.1.](image-url)

More complex flash translation layers require more complex sequences to characterize them. The hybrid FTL used by the Memorex device maintains 4 log blocks, and thus pauses
infrequently with a sequence rotating between 4 different blocks; however, it slows down for every write when the input stream rotates between addresses in 5 distinct blocks. In Figure 3.7 we see two patterns: a garbage collection after 55 writes to the same block, and then another after switching to a new block.

![Figure 3.7: Memorex device garbage collection patterns. Access pattern used is \( \{A_1 \times n, A_2 \times n, \ldots \} \) for \( n = 55, 60, 64 \) writes/block.](image)

**Organization:** In theory it should be possible to determine the zones on a device, as well as the size of the free list in each zone, via timing analysis. Observing zones should be straightforward, although it has not yet been implemented; since each zone operates independently, a series of writes to addresses in two zones should behave like repeated writes to the same address. Determining the size of the free list, \( m \), may be more difficult; variations in erase time between blocks may produce patterns which repeat with a period of \( m \), but these variations may be too small for reliable measurement.

**Wear-leveling mechanism:** Static wear-leveling is indicated by combined occurrence of two types of peaks: smaller, periodic peaks of regular write/erase operations, and higher, periodic, but less frequent peaks that suggest additional internal management operations. In
particular, the high peaks are likely to represent moving static data into highly used physical blocks in order to uniformly distribute the wear. The correlation between the high peaks and static wear-leveling was confirmed via logic analyzer, as discussed in Section 3.1 and supported by extremely high values of measured device-level endurance, as reported next.

For the Memorex flash drive, Figure 3.8 shows latency for a series of sequential write operations in the case where garbage collection is triggered at every write. The majority of writes take approximately 45 ms, but high peaks of 70 ms also appear every 138th write/erase operation, indicating that other internal management operations are executed in addition to merging, data write and garbage collection. The occurrence of high peaks suggests that the device employs static wear-leveling by copying static data into frequently used physical blocks.

![Figure 3.8: Memorex device static wear-leveling. Lower values represent normal writes and erasures, while peaks include time to swap a static block with one from the free list. Peaks have a regular frequency of one at every 138 write/erasure.](image)

Additional tests were performed with a fourth device, House-2, branded the same as the House device but in fact a substantially newer design. Timing patterns for repeated access indicate the use of static wear-leveling, unlike the original House device. We observed peaks of 15 ms representing write operations with garbage collection, and higher regular peaks of 20 ms appearing at approximately every 8,000 writes (Figure 3.9). The 5 ms time difference from common writes to the highest peaks is likely due to data copy operations implementing
static wear-leveling.

![Graph](image)

**Figure 3.9:** Timing patterns indicating static wear-leveling in the newer house-brand flash drive.

**End-of-life signature:** Write latency was measured during endurance tests, and a distinctive signature was seen in the operations leading up to device failure. This may be seen in Figure 3.10, showing latency of the final $5 \times 10^4$ operations before failure of the House device. First the 80ms peaks stop, possibly indicating the end of some garbage collection operations due to a lack of free pages. At 25000 operations before the end, all operations slow to 40ms, possibly indicating an erasure for every write operation; finally the device fails and returns an error.

![Graph](image)

**Figure 3.10:** House device end-of-life signature. Latency of the final $5 \times 10^4$ writes before failure.
Conclusions: By analyzing write latency for varying patterns of operations we have been able to determine properties of the underlying flash translation algorithm, which have been verified by reverse engineering. Those properties include wear-leveling mechanism and frequency, as well as number and organization of log blocks. Additional details which should be possible to observe via this mechanism include zone boundaries and possibly free list size.

3.3 Device-level Endurance

By device-level endurance we denote the number of successful writes at logical level before a write failure occurs. Endurance was tested by repeated writes to a constant address (and to 5 constant addresses in the case of Memorex) until failure was observed. Testing was performed on Linux 2.6.x using direct (unbuffered) writes to the block devices. Several failure behaviors were observed:

- **silent**: The write operation succeeds, but read verifies that data was not written. A read operation after device failure may return either garbage data, or the data written before failure.

- **unknown error**: On multiple occasions, the test application exited without any indication of error. In many cases, further writes were possible. Thus, the behavior becomes unpredictable, the device sustaining multiple writing cycles after the first failure or exiting. We recorded an order of $10^6$ more writes for Generic, and $10^4$ for House before the next failure.

- **error**: An I/O error is returned by the OS. This was observed for the House flash drive; further write operations to any page in a zone that had been worn out failed, returning error.
• **blocking:** The write operation hangs indefinitely. This was encountered for both Generic and House flash drives, especially when testing was resumed after failure.

**Endurance limits with dynamic wear-leveling:** We measured an endurance of approximately $106 \times 10^6$ writes for House; in two different experiments, Generic sustained up to $103 \times 10^6$ writes and $77 \times 10^6$ writes, respectively. As discussed in Section 3.1, the House flash drive performs 4 block erasures for 1-page updates, while the Generic flash drive performs only one block erasure. However, the list of free blocks is about 5 times larger for House (see Table 3.1), which may explain the higher device-level endurance of the House flash drive.

**Endurance limits with static wear-leveling:** Wearing out a device that employs static wear-leveling (e.g. the Memorex and House-2 flash drives) takes considerably longer time than wearing out one that employs dynamic wear-leveling (e.g. the Generic and House flash drives). In the experiments conducted, the Memorex and House-2 flash drives had not worn out during the test run, reaching more than $3.7 \times 10^7$ writes and $2.6 \times 10^9$ writes, respectively.

**Endurance predictions and measurements:** The primary insight from these measurements is that wear-leveling techniques lead to a significant increase in the endurance of the whole device, compared to the endurance of the memory chip itself, with static wear-leveling providing much higher endurance than dynamic wear-leveling.

Table 3.3 presents a synthesis of predicted and measured endurance limits for the devices studied. We use the following notation:
\( N \) = total number of erase blocks,
\( k \) = total number of pages in the erase block,
\( h \) = maximum number of program/erase cycles of a block
(i.e. the chip-level endurance),
\( z \) = number of erase blocks in a zone, and
\( m \) = number of free blocks in a zone.

Ideally, the device-level endurance is \( Nkh \). In practice, based on the FTL implementation details presented in Section 3.1 we expect device-level endurance limits of \( mh \) for Generic, between \( mh \) and \( mkh \) for House, and \( zkh \) for Memorex. In the following computations, we use the program/erase endurance values, i.e. \( h \), from [Des09], and \( m \) and \( z \) values reported in Table 3.2. For Generic, \( mh = 6 \times 10^7 \), which approaches the actual measured values of \( 7.7 \times 10^7 \) and \( 10.3 \times 10^7 \). For House, the lower and higher limits are \( mh = 3 \times 10^7 \) and \( mkh = 30 \times 64 \times 10^6 = 1.9 \times 10^9 \), with the measured device-level endurance of \( 1.06 \times 10^8 \) falling between these two limits. For Memorex, we do not have chip-level endurance measurements, but we will use \( h = 10^6 \) in our computations, since it is the predominant value for the tested devices. We estimate the best-case limit of device-level endurance for Memorex to be \( zkh = 1024 \times 64 \times 10^6 \approx 6 \times 10^{10} \), which is about three orders of magnitude higher than that of Generic and House devices, demonstrating the major impact of static wear-leveling.

**Table 3.3:** Predicted and measured endurance limits.

<table>
<thead>
<tr>
<th>Device</th>
<th>Parameters</th>
<th>Predicted endurance ((E_P)) value</th>
<th>Measured endurance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic</strong></td>
<td>( m = 6, h = 10^7 )</td>
<td>( mh )</td>
<td>( 6 \times 10^7 )</td>
</tr>
<tr>
<td><strong>House</strong></td>
<td>( m = 30, k = 64, h = 10^6 )</td>
<td>( mh \leq E_P \leq mkh )</td>
<td>( 3 \times 10^7 \leq E_P \leq 1.9 \times 10^9 )</td>
</tr>
<tr>
<td><strong>Memorex</strong></td>
<td>( z = 1024, k = 64, h = 10^6 ) (est.)</td>
<td>( zkh )</td>
<td>( 6 \times 10^{10} )</td>
</tr>
</tbody>
</table>
3.4 Implications for Storage Systems

**Space management:** Space management policies for flash devices are substantially different from those used for disks, mainly due to the following reasons. Compared to electromechanical devices, solid-state electronic devices have no moving parts, and thus no mechanical delays. With no seek latency, they feature fast random access times and no read overhead. However, they exhibit asymmetric write vs. read performance. Write operations are much slower than reads, since flash memory blocks need to be erased before they can be rewritten. Write latency depends on the availability (or lack thereof) of free, programmable blocks. Garbage collection is carried out to reclaim previously written blocks which are no longer in use.

Disks address the seek overhead problem with scheduling algorithms. One well-known method is the elevator algorithm (also called SCAN), in which requests are sorted by track number and serviced only in the current direction of the disk arm. When the arm reaches the edge of the disk, its direction reverses and the remaining requests are serviced in the opposite order.

Since the latency of flash vs. disks has entirely different causes, flash devices require a different method than disks to address the latency problem. Request scheduling algorithms for flash have not yet been implemented in practice, leaving space for much improvement in this area. Scheduling algorithms for flash need to minimize garbage collection, and thus their design must be dependent upon FTL implementation. FTLs are built to take advantage of temporal locality; thus a significant performance increase can be obtained by reordering data streams to maximize this advantage. FTLs map successive updates to pages from the same data block together in the same log block. When writes to the same block are issued far apart from each other in time, however, new log blocks must be allocated. Therefore, most benefit is gained with a scheduling policy in which the same data blocks are accessed
successively. In addition, unlike for disks, for flash devices there is no reason to reschedule reads.

To illustrate the importance of scheduling for performance as well as the conceptually different aspects of disk vs. flash scheduling, we look at the following simple example (Figure 3.11).

<table>
<thead>
<tr>
<th>Disk - unscheduled</th>
<th>Address rounded to:</th>
<th>X = seek (disk), or garbage collection (flash)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start track</td>
<td>- track number (disk)</td>
<td>R = read, W = write</td>
</tr>
<tr>
<td></td>
<td>- erase block number (flash)</td>
<td></td>
</tr>
<tr>
<td>Disk - optimal scheduling</td>
<td>R 70 X R 10 X R 50 X W 70 X W 10 X W 50 X R 70 X R 10 X R 50 X W 70 X W 10 X W 50</td>
<td>R = seek (disk), or garbage collection (flash)</td>
</tr>
<tr>
<td>Flash - unscheduled</td>
<td>R 70 R 10 R 50 W 70 R 10 W 70 R 50 W 70 R 10 W 70 R 50 W 70 R 10 W 70 R 50 W 70 R 10 W 70</td>
<td>X = seek (disk), or garbage collection (flash)</td>
</tr>
<tr>
<td>Flash - optimal scheduling</td>
<td>R 70 R 10 R 50 W 70 R 70 W 70 R 10 W 10 W 50 X R 70 W 70 R 10 W 10 W 50 X R 70 W 70 R 10 W 10 W 50</td>
<td>R = seek (disk), or garbage collection (flash)</td>
</tr>
</tbody>
</table>

**Figure 3.11:** Unscheduled access vs. optimal scheduling for disk and flash. The requested access sequence contains both reads (R) and writes (W). Addresses are rounded to track numbers (disk), or erase block numbers (flash), and “X” denotes either a seek operation to change tracks (disk), or garbage collection to erase blocks (flash). We ignore the rotational delay of disks (caused by searching for a specific sector of a track), which may produce additional overhead. Initial head position (disk) = track 35.

*Disk scheduling.* Let us assume that the following requests arrive: R 70, R 10, R 50, W 70, W 10, W 50, R 70, R 10, R 50, W 70, W 10, W 50, where R = read, W = write, and the numbers represent tracks. Initially, the head is positioned on track 35. We ignore the rotational delay of searching for a sector on a track. Without scheduling, the overhead (seek time) is 495. If the elevator algorithm is used, the requests are processed in the direction of the arm movement, which results in the following ordering: R 50, W 50, R 50, W 50, R 70, W 70, R 70, W 70, (arm movement changes direction), R 10, W 10, R 10, W 10. Also, the requests to the same track are grouped together, to minimize seek time; however, data integrity has to be preserved (reads/writes to the same disk track must be processed in the
requested order, since they might access the same address). This gives an overhead of 95, which is 5x smaller with scheduling vs. no scheduling.

*Flash scheduling.* Let us assume that the same sequence of requests arrives: R 70, R 10, R 50, W 70, W 10, W 50, R 70, R 10, R 50, W 70, W 10, W 50, where R = read, W = write, and the numbers represent erase blocks. Also assume that blocks are of size 3 pages, and there are 3 free blocks, with one block empty at all times. Without scheduling, 4 erasures are needed to accommodate the last 4 writes. An optimal scheduling gives the following ordering of the requests: R 70, R 10, R 50, W 70, R 70, W 70, W 10, R 10, W 10, W 50, R 50, W 50. We observe that there is no need to reschedule reads; however, data integrity has to be preserved (reads/writes to the same block must be processed in the requested order, since they might access the same address). After scheduling, the first two writes are mapped together to the same free block, next two are also mapped together, and so on. A single block erasure is necessary to free one block and accommodate the last two writes. The garbage collection overhead is 4x smaller with scheduling vs. no scheduling.

**Applicability:** Although we have explored only a few devices, some of the methods presented here (e.g. timing analysis) can be used to characterize other flash devices as well. FTLs range in complexity across devices; however, at low-end there are many similarities. Our results are likely to apply to a large class of devices that use flash translation layers, including most removable devices (SD, CompactFlash, etc.), and low-end SSDs. For high-end devices, such as enterprise (e.g. the Intel X25-E [Int09b] or BiTMICRO Altima [BiT09] series) or high-end consumer (e.g. Intel X25-M [Int09a]), we may expect to find more complex algorithms operating with more free space and buffering.

As an example, JMicron’s JMF602 flash controller [JMi08] has been used for many low-end SSDs with 8-16 flash chips; it contains 16K of onboard RAM, and uses flash configurations with about 7% free space. Having little free space or RAM for mapping tables, its flash
translation layer is expected to be similar in design and performance to the hybrid FTL that we investigated above.

At present, several flash devices including low-end SSDs have a built-in controller that performs wear-leveling and error correction. A disk file system in conjunction with a FTL that emulates a block device is preferred for compatibility, and also because current flash file systems still have implementation drawbacks (e.g. JFFS2 has large memory consumption and implements only write-through caching instead of write-back) [Mem09].

Flash file systems could become more prevalent as the capacity of flash memories increases. Operating directly over raw flash chips, flash file systems present some advantages. They deal with long erase times in the background, while the device is idle, and use file pointers (which are remapped when updated data is allocated to a free block), thus eliminating the second level of indirection needed by FTLs to maintain the mappings. They also have to manage only one free space pool instead of two, as required by FTL with disk file systems. In addition, unlike conventional file systems, flash file systems do not need to handle seek latencies and file fragmentation; rather, a new and more suited scheduling algorithm as described before can be implemented to increase performance.

3.5 Concluding Remarks

We have investigated write endurance on three USB flash drives, due to their accessibility; however, our results and methods are expected to be applicable across flash devices of all sizes that implement an FTL logic. The three devices examined were found to have flash translation layers ranging from simple (Generic) to somewhat complex (Memorex). We used both reverse-engineering of actual devices, as well as non-intrusive timing-based methods, to investigate detailed parameters of each FTL, including zone organization, free list size, mapping scheme, and wear-leveling algorithms. In addition, we developed a prediction model
that estimates whole-device endurance as a function of the FTL parameters and chip-level measurements. The model was shown to give close predictions to measured endurance results on real devices.
There is a wide gap between the potential performance of NAND flash-based solid state drives (SSDs) and their performance in many real-world applications; understanding this gap requires knowledge of their internal algorithms and behavior for various workloads.

In this chapter, we study performance models of SSDs for real workloads. This task is complex, first, because performance is workload-dependent (and real workloads are not well-behaved), and, second, because it highly depends on the Flash Translation Layer (FTL) implementation (often little known). We approach this problem from the inside out, starting from well-known FTL algorithms, deriving analytic expressions for their behavior under stationary workloads, and validating these models via simulation. Building up from this, we extend the model to real, non-ideal workloads. Finally, we present and validate a method to extrapolate from calibrated measurements and thereby predict performance for arbitrary workloads, giving results for both simulation and real devices.

Previous performance studies of SSDs used benchmarking (e.g. uFLIP [BJB09]), looking at specific I/O patterns from synthetic workloads, and trace-driven simulations (e.g. [KKN+02b, LPC+07, KJKL06]), by running each trace entirely to get exact performance values per trace. Thus while previous benchmarking work usually uses synthetic traces limited to specific I/O patterns, we investigate real workloads with real I/O patterns. Moreover, our
solutions circumvent the need to run the entire trace on the SSD, as required in trace-driven simulations, giving performance estimations instead of exact performance values.

Throughput efficiency is often referred to in the literature as simply device performance [HLST11, BI10, Des12], although conceptually performance is a more general term encompassing many aspects. In this section, we use the two terms interchangeably, noting that here device performance refers to throughput efficiency specifically.

While the models described here are presented relative to the throughput performance metric, other metrics can be easily derived. One such metric is the idle time in workloads. In Chapter 5 we propose new ways to efficiently utilize this idle time by performing active computations on the controller. Idle time prediction follows directly from our performance models, as the difference between the total trace time $T$ and the effective run time $T_{\text{eff}}$: $T_{\text{idle}} = T - T_{\text{eff}}$, where $T_{\text{eff}}$ is a function of the data volume in the workload $W$, and throughput $H$: $T_{\text{eff}} = W/H$.

The contributions of this work include:

- Analytic models for two non-ideal FTLs (BAST and FAST),
- Black-box performance models that do not require knowledge of the underlying FTL,
- A methodology for applying these models to real workloads (i.e. non-uniform traffic),
- Validation of these models on real workloads, in simulation and on real SSDs,
- A study of the potential inaccuracy factors in performance prediction.

### 4.1 Analytic Models for Uniform Traffic

We examine the case of a write-only workload, aligned in starting address and length to the page size. We assume writes are uniformly distributed across the logical address space, with
lengths exponentially distributed with mean length $w$. This is different from the uniformly distributed single-page write model used in [Rob96, BT06, HEH+09, BI10, Des12, XK12], and better models the spatial locality found in realistic workloads.

We then divide each request into head, middle, and tail fragments as shown in Figure 4.1, where each fragment is the portion of the write falling within a single erase block.

![Division of a request into head, middle, and tail fragments.](image)

**Figure 4.1:** Division of a request into head, middle, and tail fragments.

<table>
<thead>
<tr>
<th>Device parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_P$</td>
<td>number of physical blocks</td>
</tr>
<tr>
<td>$V_L$</td>
<td>number of logical blocks</td>
</tr>
<tr>
<td>$f$</td>
<td>free space ratio $(\frac{V_P-V_L}{V_L})$</td>
</tr>
<tr>
<td>$B$</td>
<td>erase block size in pages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload parameters:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$</td>
<td>mean write length, in pages</td>
</tr>
<tr>
<td>$T$</td>
<td>total number of writes in pages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other variables:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>number of erasures</td>
</tr>
<tr>
<td>$M_s, M_p, M_f$</td>
<td>number of switch, partial, full merges</td>
</tr>
<tr>
<td>$F_h, F_m, F_t$</td>
<td>number of head, middle, tail fragments</td>
</tr>
<tr>
<td>$H_t$</td>
<td>mean head length in pages</td>
</tr>
<tr>
<td>$L_f$</td>
<td>total number of fragments per log (FAST)</td>
</tr>
<tr>
<td>$L_v$</td>
<td>number of valid fragments per log (FAST)</td>
</tr>
</tbody>
</table>

More formally, each request may be represented as a sequence of consecutive pages $(p_0, p_0 + 1, \ldots, p_j, \ldots)$, and fragments are the largest sub-sequences $(p_i, \ldots, p_j)$ such that
\[ \lfloor p_i/B \rfloor = \lfloor p_j/B \rfloor \] where \( B \) is the block size in pages. The conditions for head, middle, and tail fragments are:

- **Middle**: \( |(p_i, \ldots, p_j)| = B \) (a full block); otherwise:
- **Tail**: \( p_i \equiv 0 \mod B \) (starts on block boundary); otherwise:
- **Head**.

See Table 4.1 for a full list of variables referenced. First we approximate fragment distributions:

**Heads**: Although heads which begin on block boundaries are classified as middles or tails, this number is small for \( B \gg 1 \); so the number of heads is the number of sequences:

\[
F_h \approx \frac{T}{w} \tag{4.1}
\]

where \( T \) is total page writes and \( w \) is the mean write length.

**Middles**: Given a random variable \( X \), modeled by an exponential distribution of rate \( \lambda \) (and mean \( 1/\lambda \)), the probability density function is defined as \( P(X = x) = \lambda e^{-\lambda x} \), and the cumulative distribution as \( P(X \leq x) = 1 - e^{-\lambda x} \) [Ros09]. In this paper, the write length is modeled as an exponentially distributed random variable \( W \) of mean \( w \). Since the approximate number of blocks in a \( T \)-page write is \( T/B \), the total number of middles is thus:

\[
F_m = \frac{T}{B} \cdot Pr(W \geq B) = \frac{T}{B} \cdot (1 + \frac{1}{w})e^{-\frac{B}{w}} \tag{4.2}
\]

**Tails**: Similarly, tails are given by:

\[
F_t = \frac{T}{B} \cdot Pr(W < B) = \frac{T}{B} \cdot (1 - (1 + \frac{1}{w})e^{-\frac{B}{w}}) \tag{4.3}
\]

For hybrid block-mapped FTLs, the class of a fragment is the primary determinant of its performance, which may be measured in several equivalent ways, e.g. write amplification, number of writes to flash, or number of erasures; we model erasures, but other metrics may be
similarly derived. We present results below for the number of erasures for a given workload in both BAST and FAST.

**BAST:** As described in Section 2.1.2, with this scheme each logical block is stored as a fully ordered (i.e. logical page numbers and physical page numbers match) *data block*, and may in addition be associated with a *log block* for unordered update pages to that block. We approximate switch, partial and full merges from survival probabilities of head, middle, and tail fragments in the log list. Since the free space fraction $f$ is comprised of all the log blocks, for $f \leq 1$ it is also the probability of uniform writes to hit a logical block that already has a log block allocated. Switch merges result from middles arriving to data blocks with no allocated log block:

\[
M_s = F_m (1 - f) = \frac{T}{B} \cdot (1 + \frac{1}{w}) e^{-\frac{B}{w}} (1 - f)
\]

(4.4)

Partial merges result from tails arriving to data blocks with no allocated log block in the case where no additional arrivals to this block occur before it is evicted:

\[
M_p = F_t (1 - f)^2 = \frac{T}{B} \cdot (1 - (1 + \frac{1}{w}) e^{-\frac{B}{w}}) (1 - f)^2
\]

(4.5)

Full merges result from middles and tails arriving to data blocks with allocated log blocks (causing fragmentation), and heads that do not have an allocated log block (these are counted once when the log is allocated, since several heads or tails can be mapped to a log, but lead to a single full merge).

\[
M_f = (F_m + F_t) f + F_h (1 - f) = \frac{T}{B} f + \frac{T}{w} (1 - f)
\]

(4.6)

The total number of erasures for the BAST scheme is:

\[
E = M_s + M_p + 2M_f
\]

(4.7)

**FAST:** FAST uses a dedicated log called the sequential block for writes that start on a block
boundary. Each middle thus translates to a switch merge, and each tail to a partial merge:

\[ M_s = F_m = \frac{T}{B} \cdot (1 + \frac{1}{w})e^{-\frac{w}{B}} \]  
\[ M_p = F_t = \frac{T}{B} \cdot (1 - (1 + \frac{1}{w})e^{-\frac{w}{B}}) \]  

(4.8)

(4.9)

Full merges result from head fragments, and the number of erasures is determined by the number of valid fragments in a log block. Let \( L_f \) be the number of fragments per log (both valid and invalid):

\[ L_f = \frac{B}{H_l} \]  

(4.10)

where \( H_l \) is the expected head length. For uniform traffic, there is a \( 1/B \) chance of the head to start in any page of the block. Each possible head length (from 1 to \( B - 1 \)) is achieved with some probability depending on the write length and the start page. We approximate the summation with the integral:

\[ H_l = \frac{1}{B} \int_{h=1}^{B-1} (hPr(W > h) + h(B - h)Pr(W = h))dh \]  

(4.11)

where, as before, the random variable \( W \) models the write length. Solving Equation 4.11 gives \( H_l \):

\[ H_l = \frac{1}{B}(e^{-\frac{1}{B}}(wB - w^2 + B - w - 1) + e^{\frac{1}{B}}(w^2 - w - B + 1)) \]  

(4.12)

We estimate the probability of fragments to remain valid using a survival function of heads in the log list. Since FAST uses a single dedicated log block for sequential writes that start on a block boundary, such as middles and tails, the only fragments stored in the log list are heads. When a write to a previously mapped logical page occurs, the old copy becomes invalid. If the write is long enough to hit all pages in a valid fragment, then the entire fragment becomes invalid. Middles deliver the highest invalidation throughput, as they cover an entire data block; thus all fragments pertaining to the respective data block are invalidated. Heads and tails may in some cases invalidate entire fragments as well, depending on the logical address range covered.
Considering a number $L_v$ of valid fragments per log (all fragments assumed of length $H_l$), then the number $P_i$ of invalid pages is:

$$P_i = \frac{L_f - L_v}{L_f} \cdot F_h \cdot H_l \quad (4.13)$$

The number $P_v$ of valid pages that will be invalidated by middles and heads (here, tails ignored) is:

$$P_v = L_v \cdot f \cdot (F_m \cdot B + F_h \cdot H_l) \quad (4.14)$$

where the chance of a middle or head to hit a valid fragment is $L_v \cdot f$, since the device has a fraction $f$ of log blocks, each containing $L_v$ valid fragments.

From $P_i = P_v$ and using Equation 4.10 we determine the number of valid fragments per log:

$$L_v = \frac{F_h}{f (L_f \cdot F_m + F_h) + \frac{F_v}{L_f}} \quad (4.15)$$

Since the valid fragments correspond to data blocks containing fragments in the current log to be erased, the number of erasures resulting from full merges is computed from the number of valid fragments per log, adding one erasure for the log block:

$$M_f = (L_v + 1) \cdot Logs \quad (4.16)$$

where $Logs$ represents the total number of log blocks utilizes to process this workload. As logs may have been cleaned and reused, $Logs \geq V_p - V_l$, where $V_p - V_l$ is the actual number of blocks in the log list, see Table 4.1). Since the logs are filled with head fragments, it results that:

$$Logs = \frac{F_h \cdot H_l}{B} \quad (4.17)$$

From Equations 4.10, 4.16, and 4.17 we compute the number of full merges as:

$$M_f = \frac{L_v + 1}{L_f} \cdot \frac{T}{w} \quad (4.18)$$
We substitute $L_v$ and $L_f$ in Equation 4.18 to obtain full merges as a function of the workload parameters: free space percentage $p$, mean write length $w$ and total writes $T$. Finally, the total number of erasures is:

$$E = M_s + M_p + M_f$$ (4.19)

The analytic model estimates the number of erasures based on the probability of write requests in a workload to sequentially fill blocks (resulting in single erasure), or cause fragmentation (resulting in multiple erasures due to full merges), depending on the specific implementation details of the two algorithms studied, FAST and BAST.

Other metrics of interest, such as write amplification $A$, can be modeled in a similar way. For FAST, modeling write amplification is equivalent to modeling erasures. Thus $A$ can be expressed as the total number of storage writes (including both external writes and internal writes – i.e. valid pages in all logs which are copied to a clean block before erasure) divided by the number of writes in the workload (external writes):

$$A = \frac{T + H_l \cdot L_v \cdot \text{Logs}}{T}$$ (4.20)

with $H_l, L_v, \text{Logs}$ derived before.

For BAST, the erasures model is not equivalent to the write amplification model, although it can work well for workloads with high locality. Since partially-filled log blocks may be erased and evicted after a full merge, computing the number of valid pages to be copied by internal writes is more difficult.

The performance metric analyzed here is the “cleaning throughput”, which is the equivalent throughput $N_{\text{min}}/N_{\text{obs}}$ for the operation “erasure”, where $N_{\text{obs}}$ is the observed (measured or predicted) operation count and $N_{\text{min}} = W/B$ is the minimum operation count, with $W$ = total writes (pages) and $B$ = block size (pages). Performance is modeled as a function of the reserved space (also referred to as free space or log list) ratio and the mean write length. Figure 4.2(a) illustrates this 3D performance model for the FAST flash translation layer.
Figure 4.2: (a) Analytic model for FAST, estimating the "cleaning throughput" as a function of write length and reserved space. (b) Validation against simulation results. The 'cleaning throughput' relates full performance to erasure cost: it is the equivalent throughput \( N_{\text{min}}/N_{\text{obs}} \) for the operation "erasure", where \( N_{\text{obs}} \) is the observed operation count and \( N_{\text{min}} = W/B \) is the minimum operation count, with \( W = \) total writes (pages) and \( B = \) block size (pages).

Validation: We validate the analytic model against simulation results. FlashSim from Pennsylvania State University [KTGU09] was used to simulate BAST and FAST algorithms. FlashSim is integrated with DiskSim [BSGC08], and implements several FTL algorithms [Fla], including the FAST scheme analyzed here (along with page-mapped FTL and DFTL [GKU09a]). As BAST was not implemented in FlashSim, we extended the sim-
ulator to implement BAST as well.

Figure 4.2(b) shows the accuracy of the FAST and BAST analytic models for uniform writes. Tests are carried out for a volume of $W = 10^7$ 2K-pages written on 1GB storage capacity with 7% free space, and block size $B = 64$. Results are shown for various mean write lengths, and are in terms of equivalent throughput $r = \frac{N_{\text{min}}}{N_{\text{obs}}}$, where $N_{\text{min}}, N_{\text{obs}}$ are the minimum and observed operation counts. Here the examined operation is erasure, and $N_{\text{min}} = W/B = 10^7/64 = 156250$. As we see in Figure 4.2(b), the predicted values approximate the measured results obtained from simulation with good accuracy.

4.2 Black-box Models for Uniform Traffic

The approach used in the previous section relies on the availability of an analytical expression for the FTL performance, but it is possible to approximate this function by measurement of performance for different values of write length and free space fraction, and interpolating from these points. In simulation, where these parameters may be easily modified, this is straightforward. We use FlashSim to generate multiple interpolation points from synthetic data with uniformly distributed start addresses and exponentially distributed write lengths. Interpolation points relate performance (e.g. erasures-to-writes ratios) to various mean write lengths and free space fractions. We use Matlab to generate a “linear” interpolation function from multiple data points.

The interpolation-based model for FAST is illustrated in Figure 4.3, and consists in the two mentioned steps: 1) first collect multiple 3D-values using the FlashSim simulator with synthetic workloads, in which we vary the free space ratio and the mean write length, and observe the number of erasures (Figure 4.3(a)); 2) next, generate any other 3D-values in the model by interpolating from the observed values (Figure 4.3(b)).
Figure 4.3: (a) Interpolation points measured (from synthetic traffic) with FlashSim simulator for FAST. (b) Interpolation-based prediction model for FAST, derived from the measured interpolation points, estimating the “cleaning throughput” as a function of write length and reserved space. The ‘cleaning throughput’ relates full performance to erasure cost: it is the equivalent throughput $N_{\text{min}}/N_{\text{obs}}$ for the operation “erasure”, where $N_{\text{obs}}$ is the observed operation count and $N_{\text{min}} = W/B$ is the minimum operation count, with $W = \text{total writes (pages)}$ and $B = \text{block size (pages)}$. 

4.3 Model Extension to Real Workloads

Performance modeling for real workloads is hard, mainly due to non-stationarity of workloads in time and space (i.e. address space). To correct for this, we divide each workload in small
segments across the time and space dimensions, assuming quasi-stationary traffic within each segment, and analyzing each such segment independently. We start by discussing prediction of write performance (as writes are more challenging to predict), and follow up with accounting for reads in the next sections.

As seen above, performance is strongly affected by the amount of free space available; concentrating writes onto a smaller section of the volume has the effect of increasing the effective free space. To understand this, consider a logical volume of size $V_L$ (Table 4.1), with $k$ free blocks, for a free space ratio of $\frac{k}{V_L}$. If write activity is restricted to the first half of the volume, behavior will be that of a volume with $\frac{V_L}{2}$ logical blocks, but still $k$ free blocks, or double the free space ratio. In our model, we assume that each sub-range of the address space consumes a share of free space proportional to the arrival rate to that address range; this assumption provides an approximate answer.

Figure 4.4 illustrates the method to predict performance for real workloads. We divide the workload in small segments across time and storage space, and approximate each workload segment with uniform traffic, i.e. uniformly distributed requests and exponentially distributed write lengths. One implication of this approximation is that we assume free space is rigidly divided across intervals, rather than flexibly; however results show the error from
this assumption is small. For each segment, we first compute the two statistics which give the parameters of the model: sequentiality, i.e. mean request length, and free space utilization. Next, for each segment we estimate performance using one of the previously described methods, i.e. either analytically, or from pre-determined interpolation values. Finally, we integrate over all the segments to obtain the estimated performance for the entire workload.

In practice, we approximate the integral with a summation; thus, the total system performance $P_{total}$ for non-uniform traffic is computed with a double summation across time and address space:

$$P_{total} = \sum_{i}^{M} \sum_{j=1}^{\infty} P(n(i,j), w(i,j), f(i,j)),$$

where $M$ is the number of LBA space intervals, each interval of size $\frac{V_L}{M}$ over a logical volume of size $V_L$;

$n(i,j)$ is the number of writes in the $(i,j)$ interval;

$w(i,j)$ is the mean write length in the $(i,j)$ interval;

$f(i,j)$ is the percentage of free space in the $(i,j)$ interval, given by:

$$f(i,j) = r(i,j) \cdot f \cdot M$$

$r(i,j) = n(i,j)/n(i)$ is the access rate to the $(i,j)$ interval, and $f$ is the percentage of free space for the whole device. The rest of the section will discuss how these general formulas (Equations 4.22 and 4.21) are applied to specific FTLs (Pagemap, FAST, BAST).

To summarize, the method to predict performance for real workloads is comprised of the following steps (Algorithm 1):

Step 3 (predicting performance for uniform traffic) has been presented in Sections 4.1 and 4.2, and Step 4 is straightforward. We will next discuss Steps 1 and 2 – computing workload statistics. For that, we first observe that reads and writes behave differently on flash. Computing workload statistics requires understanding what parameters are relevant
**ALGORITHM 1:** Performance prediction method for real workloads.

**Step 1:** Divide the workload in multiple smaller segments, across time and LBA space.

**Step 2:** Compute statistics, i.e., free space distribution and sequentiality over the workload segments.

**Step 3:** Assume the traffic within each segment to be uniform, and estimate performance for each segment, using either the analytical method (Section 4.1) or the interpolation-based method (Section 4.2).

**Step 4:** Integrate over all workload segments to obtain total estimated performance.

for each type of requests (reads/writes), and incorporating their contribution in a meaningful way. We discuss these aspects next.

### 4.3.1 Read-write workload prediction: contribution of reads, contribution of writes

For writes, we showed that efficiency is determined by both mean write length and available free space (Figures 4.2(a) and 4.3(a)). In the case of reads, the performance is only affected by the mean read length, and not by the free space, as the next experiments indicated: we measured the throughput for multiple read-only synthetic workloads, which were evaluated on the Microsoft Research (MSR) SSD simulator [APW+08] for different mean read lengths and effective LBA space read from. We used the MSR simulator due to its capability of collecting time-related statistics (among other statistic types). The FlashSim simulator used to model FAST and BAST in Section 4.1 above provides only garbage collection statistics, and was thus not an option here, since reads do not trigger garbage collection (Conversely, the MSR simulator implements the Pagemap FTL only; it does not implement the FAST or BAST FTLs, and was thus not an option to evaluate the analytic models presented in Section 4.1).

We evaluated performance based on read throughput, which can be directly derived from
the data volume and time measured by the MSR simulator. Figure 4.5(a) shows the variation of the read throughput depending on the two parameters (mean read lengths and effective LBA space), illustrating that the effective LBA space has no effect on the throughput, while the mean read length has.

![Graph showing read throughput variation](image)

**Figure 4.5:** (a) Variation of read throughput depending on the mean read length, and size of the LBA space read from. We observe that read throughput is not affected by the LBA size. (b) $T_{rw} \approx T_r + T_w$, where $T_{rw}$ is the time to run a workload containing both read and writes, $T_r$ is the time to run the read-only subset of the same workload (with writes removed), and $T_w$ is the time to run the write-only subset of the same workload (with reads removed). The reported times were measured with the Microsoft Research SSD simulator.
This behavior is expected, since the free space is used in the context of writes to accommodate out-of-place updates, while it plays no role for reads. The read throughput is not affected by the size of the area accessed for reading. For multi-channel devices, such as the SSD emulated in Figure 4.5(a), page reads are striped across multiple channels or planes, with longer reads taking more advantage of parallelism; thus the throughput may vary depending on the mean read length.

In order to study the read/write interference in read-write workloads, we experimented with a set of real workloads shown in Figure 4.5(b), x-axis. More information about these workloads is given in Section 4.4. We split each workload in two independent parts, one containing only the read requests, and the other one containing only the write requests. We ran the original read-write workloads, as well as the read-only and write-only workloads separately in the MSR simulator and compared the timing results.

Figure 4.5(b) shows that running a workload containing both reads and writes takes about as much time as running the read-only and the write-only subsets of the same workload separately, and summing their individual times:

\[ T_{rw} = T_r + T_w \]  

(4.23)

where \( T_{rw} \) is the effective time to run a workload containing both read and writes, \( T_r \) is the effective time to run the read-only subset of the same workload (with writes removed), and \( T_w \) is the effective time to run the write-only subset of the same workload (with reads removed). *Effective time* denotes \( S - I \), where \( S \) is the total simulation time, and \( I \) represents the time when the emulated device is idle.

As we see from these results, the read/write interference is very small. Thus read-write workloads are predicted by first estimating the read time and the write time individually, and then adding them up to obtain the total predicted time for the original read-write workload. The prediction method is throughput-based (not latency-based): it interpolates
from previously collected synthetic data points, which measure read throughput for various read lengths, and write throughput for various write lengths and free space ratios. Having predicted the read throughput $H_r$ and write throughput $H_w$, we can directly compute the times $T_r = R/H_r$ and $T_w = W/H_w$, where $R$ and $W$ represent the volume of reads and writes, respectively.

4.3.2 Computing workload statistics: free space distribution and sequentiality

Algorithms 2 and 3 give a high-level description of how free space distribution and sequentiality is computed for a given workload. For clarity, the segments are represented as 2D arrays to illustrate the time and LBA segment indexes. This would require a large storage space, given that the possible LBA segment indexes may be high (up to the number of device blocks). In an actual implementation, a solution is to utilize a hash or map for (key, LBA segment #) pairs, and thus store only the effective number of LBA segments processed in a time interval, which is much smaller than the total possible number of LBA segments. The mapping is saved when a LBA segment is accessed for the first time, with the key reflecting the order of first-time access. In addition, the 2D array can be reduced to a 1D array by processing each time interval, saving the information to file and resetting and reusing the same variables for the next time interval.

LBA space intervals determination: The LBA interval size was experimentally determined, based on the observation that predictions converge to the real values when the number of LBA space intervals is increased. Figure 4.6 shows this convergence with the number of LBA space intervals, for the analytical and interpolation-based models, respectively (trace “fin2” and the FlashSim simulator were used).

Generally, finer granularity of the LBA space intervals better captures locality in real
workloads. Most of the experimental results presented next use LBA intervals of one logical block in size. Algorithms 2 and 3 use an LBA interval size initialized to the same value of one block: $S = 1$.

**Time intervals determination:** Division in time intervals depends on how the shared resource – the free list – is split between workload segments. The free list is utilized by all workload segments in a certain proportion, depending on the FTL implementation details. In FAST and Pagemap, the log list is filled sequentially, regardless of what data block the write is directed to. In BAST, each data block has an associated log, such that writes to a
particular data block go to the same log block; for many workloads, log blocks containing only a few writes are being evicted to clean space. Therefore the effective utilization of the free space in BAST is often under 100%.

Since we are interested in finding out how the log list is shared (split) between segments over each time period, we considered the time interval to be the time to fill in the log list. The time metric is represented by the number of writes currently processed from the workload, and is not an actual time in seconds.

For FAST/Pagemap, where the log list is filled up sequentially and completely, the time interval size is simply the number of pages in the log list (Algorithm 2):

$$W(i_w) = F \times B$$

(4.24)

where $B$ is the block size in number of pages, and $F$ is the free (reserved) space in number of blocks.

For BAST, the time interval is more difficult to compute, since the logs may not be filled completely. Assuming the average log utilization (number of written pages) in the steady state to be $avg$, then the time interval size is (Algorithm 3):

$$W(i_w) = avg \times Logs(i_w)$$

(4.25)

where $Logs(i_w)$ represent the number of log blocks filled (partially or entirely) by writes to interval $i_w$, and the condition is that all logs have been touched: $Logs(i_w) \geq F$.

The parameter $avg$ is pre-computed during a first pass through the workload (BAST uses 2 passes, FAST/Pagemap – 1 pass), as presented in function $avg_logs_utilization$ (Algorithm 4): writes are acquired until all logs have been used, at which point a current $avg$ is computed and the log at the head at the list is evicted in round-robin fashion; $avg$ is computed for each log list snapshot (with all logs utilized), and averaged in the end. Ideally, for high-locality workloads, the average number of writes per log will be close to the log size.
**Per-segment utilization of free space:** In FAST/Pagemap, each LBA segment \( j \) utilizes a share of the log list represented by the fraction of writes to its LBA region. This is illustrated in Algorithm 2:

\[
u(i_w, j) = \frac{N_w(i_w, j)}{W(i_w)}\] (4.26)

where \( N_w(i_w, j) \) is the number of page writes to segment \((i_w, j)\), and \( W(i_w) \) is the total number of page writes processed during time interval \( i_w \), for all \( j \) LBA segments.

In BAST, each LBA segment \( j \) utilizes a share of the log list represented by the fraction of logs containing writes to \( j \) (Algorithm 3):

\[
u(i_w, j) = \frac{Logs(i_w, j)}{Logs(i_w)}\] (4.27)

Algorithm 3 assumes that each LBA segment \( j \) corresponds to a single data block \((S = 1)\). Thus \( Logs(i_w, j) \) is computed as the number of logs filled by writes to \( j \), where the last log may be only partially filled:

\[
Logs(i_w, j) = \text{ceil} \left( \frac{N_w(i_w, j)}{B} \right) \] (4.28)

where \( N_w(i_w, j) \) is the number of page writes to segment \((i_w, j)\), and \( B \) is the log block size in number of pages.

**Free space distribution:** As discussed in the beginning of this section, concentrating writes onto a smaller section of the volume has the effect of increasing the effective free space.

We compute the effective free space for workload segment \((i_w,j)\) as the ratio “free space utilization – to – LBA segment size”:

\[
f(i_w, j) = \frac{u(i_w,j) \times F}{S}\] (4.29)

with \( u(i_w,j) \) determined either by Equation 4.26 (FAST/Pagemap) or Equation 4.27 (BAST).
**Sequentiality:** Sequentiality (mean request length) is computed as the average (arithmetic mean) length of read/write requests within a workload segment. For writes, we compute the mean write length for each 2D workload segment. For reads, the LBA segment size is irrelevant, thus we work with 1D workload segments, and we consider the segment size to be a pre-defined constant (e.g. \( M = 100000 \) page reads). We compute the mean read length for each 1D workload segment of specified size.
ALGORITHM 2: FAST and Pagemap FTLs.

Input: A workload specifying LBA and length of requests.

Output: Free space distribution and sequentiality over the workload segments.

/* parameters and some example values */

\( P_s = 8; \) /* page size in 512-byte sectors; here, 4K */

\( B = 64; \) /* block size in pages */

\( B_s = P_s \times B; \) /* block size in 512-byte sectors */

\( F = ...; \) /* free (reserved) space in blocks */

\( S = 1; \) /* LBA segment size in blocks */

\( M = 100000; \) /* max page reads in a time segment */

\( i_r, i_w, j; \) /* index of current read/write time segments, and LBA segment, respectively */

/* \( i_r \neq i_w \) since the time interval is decided differently for reads/writes */

\( R, W; \) /* number of page reads/writes processed in \( i_r \) or \( i_w \) */

\( N_r, N_w; \) /* number of page reads/writes in each \((i_r, j)\) or \((i_w, j)\) */

\( N_{sr}, N_{sw}; \) /* number of read/write sequences in each \((i_r, j)\) or \((i_w, j)\) */

/* continued on next page */
for each request = (Lba, Len, Op, ...) in workload do

    $j = \frac{Lba}{S \times B_S}$; /* find the LBA segment; Lba, Len specified in 512-byte sectors. */

    if Op eq 'read' then
        /* update number of page reads and read sequences */
        $R(i_r) + = \frac{Len}{P_S}; \quad N_r(i_r, j) + = \frac{Len}{P_S}; \quad N_{sr}(i_r, j) + +;
        
        if $R(i_r) \geq M$ then
            read_compute_stats();
            $i_r = i_r + 1;$
        end
    else
        /* update number of page writes and write sequences */
        $W(i_w) + = \frac{Len}{P_S}; \quad N_w(i_w, j) + = \frac{Len}{P_S}; \quad N_{sw}(i_w, j) + +;
        
        if $W(i_w) \geq F \times B$ /* if the reserved space is full */ then
            write_compute_stats();
            $i_w = i_w + 1;$
        end
    end
end

func read_compute_stats:

for each LBA segment $j$ do

    if $N_r(i_r, j) > 0$ then
        $L_r(i_r, j) = \frac{N_r(i_r, j)}{N_{sr}(i_r, j)}$; /* compute mean read length */
        print $L_r(i_r, j), N_r(i_r, j);$
    end
end

/* continued on next page */
func write_compute_stats:

for each LBA segment $j$ do

if $N_w(i_w,j) > 0$ then

$L_w(i_w,j) = \frac{N_w(i_w,j)}{N_{aw}(i_w,j)}$; /* compute mean write length */

$u(i_w,j) = \frac{N_w(i_w,j)}{W(i_w)}$; /* compute free space utilization for this segment */

$f(i_w,j) = \frac{u(i_w,j) \times F}{S}$; /* compute free space ratio */

print $f(i_w,j), L_w(i_w,j), N_w(i_w,j)$;

end

end

/* Algorithm 2 for FAST and Pagemap FTLs finished. */
ALGORITHM 3: BAST FTL.

Input: A workload specifying LBA and length of requests.

Output: Free space distribution and sequentiality over the workload segments.

... variables ...

avg = avg_logs_utilization();

for each request = (Lba, Len, Op, ...) in workload do

j = Lba / \text{S} \times B_S; /* find the LBA segment; Lba, Len specified in 512-byte sectors */;

if Op eq 'read' then
    ... handle reads ...

else
    write_record();

    if \text{W}(i_w) \geq \text{avg} \times \text{Logs}(i_w) \text{ and Logs}(i_w) \geq \text{F} /* if the reserved space is full */ then
        write_compute_stats();
        i_w = i_w + 1;
    end
end

func write_compute_stats:

for each LBA segment j do

    if \text{N}_w(i_w, j) > 0 then
        \text{L}_w(i_w, j) = \frac{\text{N}_w(i_w, j)}{\text{N}_s(i_w, j)}; /* compute mean write length */
        \text{u}(i_w, j) = \frac{\text{Logs}(i_w, j)}{\text{Logs}(i_w)}; /* compute free space utilization for this segment */
        \text{f}(i_w, j) = \frac{\text{u}(i_w, j) \times \text{F}}{\text{S}}; /* compute free space ratio */
        print \text{f}(i_w, j), \text{L}_w(i_w, j), \text{N}_w(i_w, j);
    end
end

/* continued on next page */
func write_record (i_w, j):

/* update number of page writes and write sequences */

W(i_w) = W(i_w) + \frac{Len}{\rho^\sigma};

N_w(i_w, j) = N_w(i_w, j) + \frac{Len}{\rho^\sigma};

N_{sw}(i_w, j) = N_{sw}(i_w, j) + 1;

\text{Logs}(i_w, j) = \text{ceil} \left( \frac{N_w(i_w,j)}{B} \right); /* logs filled partially or entirely by writes to (i, j) */

/* Algorithm 3 for BAST FTL finished. */
**Algorithm 4: Computing average log utilization.**

```python
func avg_logs_utilization ():
    front = 0, end = 0; /* front/end of the log list */
hash; /* a hash mapping LBA segments to the order in which they were added to the log list */
for each request = (Lba, Len, Op, ...) in workload do
    j = \frac{Lba}{S \times Bs}; /* find the LBA segment j */
    if Op eq 'write' then
        write_record(i, j);
        if not exists hash{j} then
            store pair (j, end) in hash;
            end = end + 1;
        end
    /* if all logs were used compute the current average log utilization */
    if Logs(i) ≥ F then
        a = \frac{W(i)}{Logs(i)}; /* average number of writes per log */
        sum = sum + a; /* save information at this step */
        i = i + 1;
        /* remove log(s) at the front of the list */
       jf = get(hash, front);
        W(i) = W(i) - Nw(i, jf);
        Nw(i, jf) = 0; Logs(i, jf) = 0;
        front = front + 1;
    end
end
avg = \frac{sum}{i}; /* compute an overall average */
reset W, Nw, Nsw, Logs to zero;
return avg;
```
4.4 Validation on Real Workloads

We validate this method on real workloads, chosen to cover a large number of classes of workload behavior – database, network, email, research. These workloads are publicly available and have been used in previous studies as well [NDR08, ELMS03, GKU09a]. They are represented by four traces from Microsoft Research (rsrch0, prxy0, src11, proj2) [NDR08], two database traces from the UMass Trace Repository (fin1, fin2) [uma07], and two from Harvard University (dea2, aka deasna2, and lair62b)\(^1\) [ELMS03].

Although flash devices exhibit history-dependent performance (i.e. the current state of the log list depends on previous states), performance (e.g. erasures-to-writes ratio, throughput, etc.) converges in time. Performance convergence is shown in Figure 4.7 for the cleaning throughput metric. All reported measurements were done after convergence, to ensure consistency. Smaller traces have been processed repeatedly in the same run to ensure convergence, and only the last processing reported.

Performance is estimated using the summation in Equation 4.21. As discussed in Section 4.3.2, we measure time in units of page writes, with a time interval equal to the number of pages in the log list, and LBA intervals of one logical block in size.

4.4.1 Validation by simulation

Part of the experiments were performed with FlashSim simulator [KTGU09], and part with the Microsoft Research (MSR) SSD simulator [APW+08], both of which are based on DiskSim [BSGC08]. Each of them has been used in several other studies, e.g. FlashSim [KGU+11, GKU09b, BOR11], and MSR SSD simulator [KOS+11, SXX+09, LKS+11].

FlashSim implements a single-channel SSD, with various possible FTLs (FAST, BAST, Pagemap and DFTL), from which we analyze FAST and BAST in this study. It provides

\(^1\)These are originally file system traces; they were converted to block traces by Harsh Kumar, by replay and capture of block level operations on the ext3 file system.
Figure 4.7: An example of performance convergence to the stable state (trace “prxy0”, with FlashSim). Workloads need to be run for a long enough time until performance had converged, and thus ensure consistency of reported results. The performance metric here is the “cleaning throughput”, which is the equivalent throughput $N_{\text{min}}/N_{\text{obs}}$ for the operation “erasure”, where $N_{\text{obs}}$ is the observed operation count and $N_{\text{min}} = W/B$ is the minimum operation count, with $W = \text{total writes (pages)}$ and $B = \text{block size (pages)}$.

garbage collection statistics, but no time-based statistics. Thus, with FlashSim we use a performance metric based on the number of erasures triggered during each workload run. Since reads have no effect on garbage collection, only the writes were considered in the eight workloads experimented with.

The Microsoft Research SSD simulator implements a multi-channel SSD, keeping track of simulation time, as well as time-related and garbage collection statistics; however, it only implements a Pagemap FTL. With the MSR simulator we analyze time, throughput, and erasure-based performance metrics for each read-write workload experimented with.

**FAST and BAST FTLs in single-channel SSDs. Write-only workloads:** For each of the FAST and BAST FTLs we compare analytic predictions and interpolation-based predictions (from multiple synthetic interpolation points pre-determined with the FlashSim simulator) against measured (by simulation) results obtained by running the entire workload
in FlashSim (Figure 4.8).

![Diagram](image)

**Figure 4.8:** Analytical and interpolation-based predictions for FAST and BAST FTLs on real workloads, compared to measured (simulation) results. The “cleaning throughput” is the equivalent throughput \( N_{\text{min}} / N_{\text{obs}} \) for the operation “erasure”, where \( N_{\text{obs}} \) is the observed operation count and \( N_{\text{min}} = W / B \) is the minimum operation count, with \( W = \) total writes (pages) and \( B = \) block size (pages).

The tests reported here use the following configuration: 30 GB of storage, with 7% free space, 2K pages, 64-page blocks. We collected \( 5 \times 5 = 25 \) interpolation points from synthetic traffic, with mean write lengths of \( \{1, 16, 64, 128, 256\} \) pages and effective free space ratios of \( \{0.07, 0.5, 1, 10, 100\} \).
The results (Figure 4.8) generally show good correlation between the predicted and measured performance values for each workload. For both analytic and interpolation-based predictions, we observe the following error factors. A high number of block boundary accesses occur in trace lair62b; they are translated by FAST into partial merges. FAST attempts to exploit sequentiality by using a new log block every time a boundary write occurs, even if the same data block is accessed. For some workloads, this policy leads to a high number of partial merges, which was observed in the simulation of lair62b with FAST. Unlike the other traces, dea2 has very long write lengths of 100 2K-pages on average and high variance, ranging up to 2000 2K-pages. In this case, free space distribution may be harder to compute, which is shown by the higher error in the BAST predictions for dea2.

**Pagemap FTL in multi-channel SSDs. Read-write workloads:** We used the Microsoft Research SSD simulator to emulate a NAND flash SSD with the parameters described in Table 4.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total capacity</td>
<td>32 GB</td>
</tr>
<tr>
<td>Flash chip elements</td>
<td>8</td>
</tr>
<tr>
<td>Planes per element</td>
<td>8</td>
</tr>
<tr>
<td>Blocks per plane</td>
<td>2048</td>
</tr>
<tr>
<td>Pages per block</td>
<td>64</td>
</tr>
<tr>
<td>Page size</td>
<td>4 KB</td>
</tr>
<tr>
<td>Reserved free blocks</td>
<td>8 %</td>
</tr>
<tr>
<td>Minimum free blocks</td>
<td>1 %</td>
</tr>
<tr>
<td>FTL mapping scheme</td>
<td>Page-level</td>
</tr>
<tr>
<td>Allocation pool logic</td>
<td>Per channel (element)</td>
</tr>
<tr>
<td>Cleaning policy</td>
<td>Greedy</td>
</tr>
<tr>
<td>Page read latency</td>
<td>0.025 ms</td>
</tr>
<tr>
<td>Page write latency</td>
<td>0.2 ms</td>
</tr>
<tr>
<td>Block erase latency</td>
<td>1.5 ms</td>
</tr>
<tr>
<td>Chip transfer latency per byte</td>
<td>25 ns</td>
</tr>
</tbody>
</table>

The workloads represent the original UMASS (fin1, fin2) and Microsoft (rsrch0, prxy0, src11, proj2) traces, containing reads and writes in various proportions, and Harvard (dea2,
traces for which only the write-only subset (no reads) was available during the experiments. The requests were aligned to 4 K boundary, where 4 K represents the page size used in the simulator.

We used the interpolation-based prediction method to estimate efficiency of reads and writes in the tested real-world workloads. First, we measured the interpolation points from synthetic workloads, using the MSR simulator. We collected $5 \times 5 = 25$ interpolation points from synthetic write-only workloads, varying the mean write length and the size of the area written to: mean write length of \{1, 16, 64, 128, 256\} pages, and size of area written to of \{91, 916, 9168, 18336, 114440\} blocks. The emulated SSD has an effective free space of 7\%, which represents about $F = 9168$ blocks (Table 4.2). The effective free space for an area of size $S$ is: $\frac{F}{S}$, which results in a free space percentage of \{0.08, 0.5, 1, 10, 100\} for the area sizes simulated.

To predict the reads, we collected 5 interpolation points from synthetic read-only workloads with various mean read length of \{1, 16, 64, 128, 256\} pages. As discussed in Section 4.3.1, the read performance is only affected by the mean read length, and not by the free space.

Next, we applied Algorithm 2 to divide the real-world workload in segments and compute the parameters of each segment, i.e. free space distribution and mean read/write length. We derived the performance values for each workload segment by interpolation using the method described in Section 4.2 for uniform traffic, and, finally, summed over all the segment results to obtain the total workload performance.

The measured versus predicted time, throughput, and cleaning throughput are illustrated in Figures 4.9, 4.10(a), and 4.10(b), respectively. In Figure 4.9, “time” represents the effective running time spent in reads/writes and any FTL operations (excluding idle time). In Figure 4.10(a), “throughput” represents the volume of data read and written, divided by the effective running time. In Figure 4.10(b), “cleaning throughput” is the equivalent through-
put $N_{\text{min}}/N_{\text{obs}}$ for the operation “erasure”, where $N_{\text{obs}}$ is the observed operation count and $N_{\text{min}} = W/B$ is the minimum operation count, with $W =$ total writes (pages) and $B =$ block size (pages).

Figure 4.9: Measured (simulation) versus predicted running time for each read-write workload. The read portion and the write portion were predicted independently and their individual timings were summed to obtain the estimated running time for the entire workload.

Generally, we observe that the prediction results closely approximate the results obtained by simulation. Less accurate estimations are obtained for the trace dea2. The difficulty to predict SSD efficiency on this trace may be related to its specific access patterns. We observed that many requests are long: assuming 4K page size, 15% of requests are longer than a 64-page block, more specifically they are about 300 page long on average; the longest requests go up to 1000 pages. As discussed next, multi-channel (or plane) striping breaks locality, and its effect may be harder to capture and predict. (This is different from the FAST and BAST FTLs studied before, which present a single-channel implementation, thus eliminating the striping effects. FAST also attempts to take maximum advantage of sequentiality in workloads by using the sequential block). The experiments showed that dea2 predictions improve when Algorithm 2 is used with larger LBA segment sizes. With 30
Figure 4.10: Measured (simulation) versus predicted throughput (a), and cleaning throughput (b) for each read-write workload. The read portion and the write portion were predicted independently and their individual timings were summed to obtain the estimated running time for the entire workload.

LBA segments (i.e. in Algorithm 2 set $S = 3815$ instead of $S = 1$ blocks per segment), the read-write throughput and cleaning throughput prediction errors are as low as 0.2% and 1%, respectively. Although usually finer-grained LBA space intervals better capture locality in real workloads, increasing estimation accuracy, as illustrated in Figure 4.6, this may not be the case for the pagemapped multi-channel processing of dea2, where the effect of the particular access patterns (in conjunction with striping) appears to be better captured with
larger LBA segments.

4.4.2 Validation on real SSDs

Performance modeling of real SSDs is more challenging, because we have no knowledge about the FTL implementation inside the SSDs. Therefore we use the interpolation-based prediction method, in which the interpolation points have been previously measured for each SSD, representing “(free space ratio, mean write length, throughput)” 3D-values.

Application of this approach to real devices is similar, except that variation of the free space parameter is more difficult to estimate. We do this by testing on varying fractions of the device LBA space, assuming that, as described above, the entire free space of the device is available even when operating on only a fraction. This does not allow testing of free space ratios less than that of the device itself; however in practice this does not appear to introduce significant error. (These data points would only apply to a small fraction of arrivals, thus diminishing the effect of any errors.)

Tests were performed on two mass market SSDs (Kingston SNV425 64 GB and Plextor 64 GB), and are shown in Figure 4.11.

We considered the amount of free space to be the difference between the total internal device capacity of $64 \cdot 2^{30}$ B and the size exposed to the host (approx. 7%). From other tests on the same devices we inferred an 8 K page size, and 64-page block size. We used the same set of real workloads that cover a wide range of access patterns. We initially processed these traces by aligning the writes to 8 K boundary and scaling up the write addresses (LBAs), since they originally touched a very small portion of the 64 GB devices, to obtain an access distribution spread out over the entire device. The scaling factor is $F = V_L/\max_\text{lb}$, where $V_L$ represents the total number of logical blocks of the device, and $\max_\text{lb}$ is the highest logical block number accessed in the workload.

In order to collect interpolation points, write throughput was measured for uniformly
Figure 4.11: SSD performance predictions on real workloads. Method: interpolation-based predictions from measured values. Sustained sequential write throughputs for 64 GB Kingston and Plextor SSDs are 105 MB/s and 40 MB/s, respectively. In all but two cases (dea2 for Plextor, proj2 for Kingston) good correspondence between predicted and measured throughput was observed. In (b), “Error Factor” represents $E = \max\{p, m\} / \min\{p, m\}$, where $p$ is the predicted value, and $m$ is the measured value.

distributed memoryless-length bursts, as modeled above. Multiple measurements were made on each of the SSDs, with write lengths ranging from 1 to 256 pages, and free space ratios from 0.07 to 8000.

The prediction model uses the FAST-like distribution of free space presented in Algorithm
2, as other tests have indicated FAST-like behavior for these devices. Using the actual number of pages written to within each interval instead of the interval size, which differs especially for sparsely distributed small writes improves accuracy significantly for traces fin1 on Kingston and rsrch0 on Plextor within a factor of 1.1 (from a factor of 2.2 given by the basic Algorithm 2). The trace dea2 presents the converse problem, where many writes are larger than an LBA segment size. Using the actual LBA region size accessed in each integration interval instead of the LBA interval size improves accuracy for dea2 on Plextor to a factor of 1.9 (from a factor of 2.3 given by the basic Algorithm 2). The other 13 cases presented in Figure 4.11 use the general prediction model described in Algorithm 2, with no modifications (although improvements were observed in some cases when using variations of the model as described above).

Figure 4.11(a) compares the measured and predicted write throughput for each workload, while Figure 4.11(b) shows the error factor for these predictions. The error factor is computed as \( E = \max\{p, m\}/\min\{p, m\} \), where \( p \) is the predicted value, and \( m \) is the measured value.

We see that 14 out of the 16 experiments exhibit fairly accurate predictions, within a factor of 1.4. Moreover, in half of these cases, highly accurate estimations are obtained, within less than a factor of 1.1 (i.e. relative error less than 10%). Only two out of the 16 tests are less accurate, presenting errors within a factor of 2 (dea2 on Plextor, proj2 on Kingston).

From these results we see that a black-box prediction based on a very general model of FTL behavior results is fairly accurate for a considerable number of cases, spanning a wide range of performance.

4.5 Potential Inaccuracy Factors

Capturing the complex behavior of SSDs on real workloads is difficult. The predictions use approximations and simplifying assumptions to go from well-behaved traffic to non-
uniform workloads. Moreover, real SSDs present many unknowns related to their internal structure and functioning, which are usually known in a simulation. Some of these unknown parameters could be the free space size, zone organization, number of channels, or the flash translation layer algorithm. Assumptions and approximations are used in the prediction model to account for these parameters for real SSDs. Lastly, SSDs often employ optimization tweaks to increase efficiency on specific traffic patterns, which can not be captured with a general prediction model.

Inaccuracy factors can be related to several aspects, including:

A. Device parameters: page/block size, free space

B. Zone organization

C. Number of channels, allocation policies, striping

D. Flash translation layer algorithm

F. Specific optimization tweaks in SSDs, buffering policies

A. Device parameters: page/block size, free space. The prediction model uses several device parameters, such as page size, block size, free space size. These values are often not known, but sometimes they can be inferred.

For the SSDs studied, we estimated the total free space as the difference between the total internal device capacity and the size exposed to the host. The actual free space is probably smaller, because flash devices may use some pages within a block to store metadata information. The first page in every log block is often used for metadata (e.g. the House and Memorex USB flash drives from Chapter 3).

B. Zone organization. Flash devices may divide the physical space in regions called zones. Each zone has its own mapping table, which is maintained in memory to service
requests for pages in that region. Smaller zones reduce memory requirements and work well for workloads with high locality, but random requests incur high overhead since the in-memory mapping table needs to be replaced often.

Each zone manages its own free list and runs an instance of the flash translation layer. Thus the prediction model will not be able to accurately estimate the workload statistics (such as free space distribution over workload segments) for writes that span zone boundaries. Inaccuracies are further introduced if writes span boundaries during interpolation points collection. Let us consider writes of $2N$ pages long, where the first $N$ page writes fall within one zone, and the next $N$ fall within the neighboring zone. Since each zone has a separate free space, this scenario would result in twice as much efficiency compared to the case when all $2N$ writes are directed to the same zone.

Zoning is used in many devices to reduce memory requirements. Our reverse engineering work (Chapter 3) revealed zone organization of various USB flash drives. Accounting for multiple zones and writes spanning over more than one zone is difficult, increasing the complexity of the prediction model significantly. The prediction results presented here do not account for zoning. We modeled the real SSDs (Section 4.4.2) as single zone devices (the real number of zones was not known).

C. Number of channels, allocation policies, striping. The number of channels, allocation policies, and parallelism implemented in an SSD may lead to performance aspects hard to capture in the prediction model. Request patterns are sensitive to the allocation policy, with an important effect on performance [JK12]. Specifically, writing patterns in conjunction with the allocation policy determine the state of the log list.

Figure 4.12 shows the number of valid pages in the log selected for garbage collection (averaged per workload run) for the multi-channel SSD emulated in Section 4.4.1 using the Microsoft Research SSD simulator. The greedy policy used by the Pagemap FTL always
selects the log with the minimum number of valid pages to be cleaned. Before log erasure, its valid pages are moved to a different log, contributing to the write amplification.

![Figure 4.12: Number of valid pages in the log selected for cleaning. Greedy selection method: the log with minimum number of valid pages is selected from the log list. These pages need to be moved before the selected log is erased, contributing to the write amplification. The average value over all cleaned logs during the entire workload run is shown.](image)

We observe that the allocation policy leads to highly different log list states for the proj2 and dea2 workloads (Figure 4.12): with per-plane allocation, garbage collection has to be invoked while most pages are still valid (have not yet become stale), which explains the drop in throughput shown in Figures 4.13 (a) and (b) for the same workloads (per-plane compared to per-element allocation). These two traces present larger write sequences compared to the other workloads, and the allocation-dependent striping of writes across channels and planes potentially breaks locality, with significant effect on performance.

D. Flash translation layer algorithm. The internal algorithms implemented inside SSDs are proprietary, however the state-of-the-art FTLs such as FAST, BAST, Pagemap represent core algorithms on which real implementations are based. In the case of uniform traffic, the accuracy of the analytic model depends on knowing the actual FTL design;
Figure 4.13: Effect of allocation policy. The dea2 and proj2 traces are sensitive to the allocation policy, i.e. for these two traces which present larger write sequences compared to the other workloads, striping (which depends on the allocation method) breaks locality having a visible effect: the per-plane allocation policy delivers lower throughput compared to the per-channel policy.

(a) Throughput (MB/s).

(b) Cleaning throughput, computed as the equivalent throughput $N_{\min}/N_{\text{obs}}$ for the operation “erasure”, where $N_{\text{obs}}$ is the observed operation count and $N_{\min} = W/B$ is the minimum operation count, with $W =$ total writes (pages) and $B =$ block size (pages).
this is not necessary for the black-box prediction model where estimations are derived from measured interpolation points.

However, for non-uniform traffic, knowledge of the FTL type is necessary for computing workload statistics, consisting in determination of workload segments and computation of free space distribution per segments. This is independent of which uniform traffic approximation method is subsequently used within each workload segment (analytical or black-box).

Section 4.3.2 presents two algorithms used to compute workload statistics: Algorithm 2 that assumes logs to be written sequentially and completely, as is the case of FAST and Pagemap FTLs, and Algorithm 3, customized for BAST-like behavior, where logs may be only partially filled at the time when they are evicted from the log list.

While a general prediction model that works without any knowledge of the underlying FTL is desirable, that is difficult to achieve. We experimented with using Algorithm 2 (which does not account for partially filled logs) to predict BAST, to see how prediction accuracy is affected. We note that Algorithm 2 is the more general and simpler of the two, because it computes free space distribution per workload segments simply by using the access rate to each segment. As Figure 4.14(a) shows, this algorithm is able to provide predictions within a factor of 2 (“bast using fast distribution”). While these estimations may be acceptable in some cases, the BAST-customized Algorithm 3 gives better predictions (“bast using bast distribution”), demonstrating that knowledge of how the FTL works improves predictions (here, the knowledge used is minimal).

With BAST, log list utilization plays an important role, as logs may be evicted before being filled. Figure 4.14(b) illustrates average log utilization for each workload. The average log utilization is computed with Algorithm 4, and represents the mean number of written pages per log (rounded to the next integer to avoid underestimation of the time interval size for small average log utilization).

We see the correlation between low log utilization in Figure 4.14(b) and low accuracy
Figure 4.14: (a) Effect of workload statistics computation algorithm on prediction results. Predicting BAST using Algorithm 3 (customized for BAST) – “bast using bast distribution”, and Algorithm 2 (customized for FAST) – “bast using fast distribution”, respectively.
(b) Average log utilization (in pages, out of 64 pages per block) with BAST. The average log utilization is small (logs are evicted before being filled) in the case of workloads which give less accurate results in (a) “bast using fast distribution”.
of “bast using fast distribution” in Figure 4.14(a) (traces “fin1”, “fin2”, “proj2”), due to the implicit and incorrect assumption of Algorithm 2 that logs are fully utilized. (For tests in Figures 4.8 and 4.14, we scaled the write addresses in traces “fin1” and “fin2”, which originally touched a very small portion of the emulated device, to obtain an access distribution spread out over the entire device. This resulted in a sparse log list utilization with BAST, and offered insight about workloads with less spatial locality).

Often real-world workloads exhibit high locality, leading to high log utilization, as is the case with most workloads in these experiments. For these, Algorithm 2 (designed for FAST/Pagemap) gives fairly accurate predictions in the case of the BAST FTL as well. Thus high-locality workloads could be predicted with a single general prediction method which does not require knowledge of the implemented FTL.

E. Specific optimization tweaks in SSDs, buffering policies. Often flash devices implement optimizations to improve efficiency on specific access patterns. For example, the House USB flash drive reverse-engineered in Chapter 3 handles multi-page writes differently than single-page writes. Multi-page updates are written directly, together with the valid pages from the same data block, in order, in a new data block, avoiding an intermediary out-of-order storage step to another log block. This results in 2 block erasures (one for each of the odd/even planes) and less write amplification for multi-page updates, as opposed to 4 block erasures and higher write amplification for single-page updates.

Buffering requests to increase write efficiency is definitely possible with current SSDs, which present 64-128 MB or more of onboard DRAM. Interestingly, none of the two SSDs tested in Section 4.4.2 does consistently better than the other, on the same set of traces. From Figure 4.11 we see that the measured throughput of Kingston SSD is higher on four of the workloads, while Plextor SSD performs better on the other four workloads. This indicates that each SSD implements its own internal optimizations, which favor some writing patterns.
more than others, making the task of constructing a general performance prediction model particularly hard.

4.6 Concluding Remarks

Capturing the complex behavior of SSDs in many real-world applications is difficult, requiring knowledge of their internal algorithms and behavior for various workloads. We have developed analytic models of the performance of two commonly-used flash translation layers, along with black-box performance models, and a methodology to apply these models to real workloads. In these models, performance is estimated as a function of free space utilization and traffic sequentiality. We validate this methodology by measurements both in simulation and on real devices. In addition, we identify and investigate potential inaccuracy factors that may affect the predictions. One important factor is the allocation policy across channels, as the striping method may strongly affect performance on various request patterns. Other factors include unknown device parameters, zone organization, FTL implementation details, and various optimization tweaks used in SSDs to increase efficiency on specific traffic patterns.
Chapter 5

Active Flash: Data Analytics on Flash Storage

Scientific discovery today is becoming increasingly driven by extreme-scale computer simulations—a class of applications characterized by long-running computations across a large cluster of compute nodes, generating huge amounts of data. As these systems scale, however, I/O performance has failed to keep pace: the Jaguar system at Oak Ridge National Laboratory (ORNL) incorporates over 250,000 cores and 1.2 GB of memory per core, but with a 240 GB/s parallel storage system has a peak I/O rate of less than 1 MB/s per core (Figure 5.1). With current scaling trends, as systems grow to larger numbers of more powerful cores, less and less storage bandwidth will be available for the computational output of these cores.

Massively parallel simulations are only part of the scientific workflow. To derive knowledge from the volumes of data created in such simulations, it is necessary to analyze this data, performing varied tasks such as data reduction, feature extraction, statistical processing, and visualization. Current workflows typically involve repeated steps of reading and writing data stored on a shared data store, further straining the I/O capacity of the system.

Current approaches to avoid bottleneck to a central storage either use some fraction of the cores on the compute nodes to execute analysis routines, or dedicate a percentage of compute nodes to storing and analyzing output of the simulation application before storage
of final results to the parallel file system [AKRV08, LVB⁺10, PVK⁺11, ZAD⁺10]. However, such approaches may adversely impact the main simulation by competing for compute cycles and DRAM. In today’s HPC systems, memory is becoming a critical resource as the memory-to-FLOP ratio has been steadily declining, from 0.85 for the No. 1 machine on Top500 in 1997 to 0.13 for Jaguar and 0.01 for the projected exaflop machine in 2018 [TOP, oE09]. Moreover, for a substantial class of HPC applications characterized by close, fine-grained synchronization between computation on different nodes, non-determinacy resulting from competing CPU usage can result in severe performance impairment [TEFK05], as such jitter causes additional waiting for “stragglers” at each communication step.

In addition, power has become a constraint in the scaling of today’s HPC systems, reaching megawatt values for the largest systems. Thus, the average power consumption of the top 10 HPC systems today is 4.56 MW [TOP], with the No. 1 system drawing 12.66 MW.
To address these concerns, we propose Active Flash, a model which moves data analysis from the host CPU to the more power-efficient storage controller, thus avoiding many rounds of redundant I/Os and reducing the energy consumption significantly. Figures 5.2(a) and (b) show possible Active Flash scenarios: simulation data is analyzed on the SSD controller either after passing through the parallel file system (a), or before (b), in which case the compute nodes are provisioned with compute-node local flash. With node-local flash, simulation output may be accessed locally on the nodes where it is produced, reducing the vast amounts of data generated on many-core nodes before any centralized collection steps. Incorporating node-local storage to address bandwidth limitations of a central file system has been recently adopted by other systems such as Tsubame2 at the Tokyo Institute of Technology [Hat11] and Gordon [gor09] at the San Diego Supercomputing Center (SDSC).

The contributions of this work include:

- Proposed architecture for Active Flash and feasibility discussion,
• Exploration of performance-energy tradeoffs,

• Demonstration of the Active Flash model on realistic data analysis applications,

• Proposed scheduling policies of data analysis on the controller, and evaluation of the compute-IO tradeoffs of these policies by simulation

• Implementation of an Active Flash prototype using the OpenSSD experimental platform, demonstrating: host–controller communication, preemption of data analysis tasks when competing with I/O service tasks, and scientific data analysis applications running on the controller integrated with the FTL software.

5.1 Active Flash Architecture and Feasibility

The proposed Active Flash architecture uses the contemporary advancements in flash technology. Figure 5.3 describes the Active Flash architecture, with the HPC simulation (main application) running on the compute node (host CPU), and generating large volumes of data which are sent to the storage device, and processed on the storage controller, thus reducing the data movement costs. Flash storage offers multiple advantages that make Active Flash a feasible approach:

• High I/O bandwidth: SSDs offer high I/O bandwidth due to interleaving techniques over multiple channels (typically 8 on consumer devices to 16 or more on high-end ones) and flash chips. Typical read/write throughput values for contemporary SSDs are 150–250 MB/s, and up to 400-500 MB/s for PCIe-based devices such as the OCZ RevoDrive PCI-Express SSD [ocz].

• Availability of idle times in workloads: Although NAND flash management uses some CPU time on the embedded CPU, processor utilization on SSDs is highly dependent on
workloads. The processor is idle between I/O accesses, and as higher and higher-speed CPUs are used to reduce per-request latency, may even be substantially idle in the middle of requests as well. These idle periods may in turn be used to run tasks that are offloaded from the host.

- High-performance embedded processors: a number of fairly high-performance CPUs have been used in SSDs, as mentioned previously; however there are also many other ‘mobile-class’ processors which fit the cost and power budgets of mid-to-high end SSDs. (e.g. the ARM Cortex-A9 [corb] dual-core and quad-core CPUs, capable of operating at 2000 MHz). The comparatively low investment to develop a new SSD platform would make feasible an Active Flash device targeted to the HPC applications.
5.2 Performance–Energy Tradeoffs

We analyze the performance–energy tradeoffs of the Active Flash model. We generalize the study to a hybrid model, in which the SSD controller is used in conjunction with the host CPU to perform the data analysis. A fraction $f$ of the data analysis is carried out on controller, and the rest on the host CPU. Moving the entire analysis on the controller is a specific case of the hybrid model, when $f = 1$. The two scenarios compared are:

- **baseline**: the entire data analysis is performed on the host CPU.

- **hybrid**: a part of the data analysis is carried out on the SSD controller; the rest, if any, is running on the host CPU.

We consider two HPC scenarios, which may occur in the baseline model and the host-side of the hybrid model:

- **alternate**: data analysis alternates with other jobs (e.g. HPC simulation). When data analysis is performed, it fully utilizes the CPU.

- **concurrent**: data analysis runs concurrently with other jobs (e.g. HPC simulation). It utilizes only a fraction of the CPU compute power.

Performance and energy consumption of the data analysis task are determined chiefly by data transfer and computation. Assuming data transfer takes place over a low-powered bus such as SATA/SAS (developed with mobile use in mind) the contribution of data transfer to energy consumption should be negligible, and will be ignored. This data transfer, however, plays a significant role in performance, giving the hybrid model a significant advantage over the baseline model, as fewer transfers occur, with no data ever being transferred from the controller back to the host CPU.
Table 5.1: List of variables: device and data analysis parameters.

<table>
<thead>
<tr>
<th><strong>Data analysis parameters:</strong></th>
<th><strong>Device parameters:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>baseline:</strong></td>
<td><strong>baseline:</strong></td>
</tr>
<tr>
<td>( t_b ) total computation time (CPU time)</td>
<td>( s_{h}, s_{c} ) host CPU speed, controller speed</td>
</tr>
<tr>
<td>( u_b ) host CPU utilization</td>
<td>( s ) ( s_{h}/s_{c} )</td>
</tr>
<tr>
<td>( \Delta E_b ) host CPU energy consumption</td>
<td>( H_{idle}, H_{load} ) host CPU idle and load power consumption</td>
</tr>
<tr>
<td><strong>hybrid:</strong></td>
<td>( C_{idle}, C_{load} ) controller idle and load power consumption</td>
</tr>
<tr>
<td>( t_c ) computation time on controller (CPU time)</td>
<td>( \Delta P_h ) ( H_{load} - H_{idle} )</td>
</tr>
<tr>
<td>( u_c, u_h ) controller and host CPU utilization</td>
<td>( \Delta P_c ) ( C_{load} - C_{idle} )</td>
</tr>
<tr>
<td>( f ) fraction of data analysis carried out on controller</td>
<td>( p ) ( \Delta P_c/\Delta P_h )</td>
</tr>
<tr>
<td>( S_e ) effective slowdown (CPU time ratio)</td>
<td><strong>Device parameters:</strong></td>
</tr>
<tr>
<td>( S ) visible slowdown (wall clock time ratio)</td>
<td><strong>Device parameters:</strong></td>
</tr>
<tr>
<td>( \Delta E_c, \Delta E_h ) controller, host CPU energy consumption</td>
<td>( p ) ( \Delta P_c/\Delta P_h )</td>
</tr>
<tr>
<td>( \Delta E ) energy savings</td>
<td><strong>Device parameters:</strong></td>
</tr>
</tbody>
</table>

### 5.2.1 Performance study

Table 5.1 gives a complete list of variables used in this study. They address time, speed, energy, and CPU utilization.

Working alone (i.e. the baseline model), the host CPU takes time \( t_b \) to finish the entire computation. The controller is \( s \) times slower than the host CPU. Thus it finishes its share \( f \) of the data analysis in:

\[
t_c = f \cdot s \cdot t_b
\]  \( (5.1) \)

The effective slowdown of the computation in the hybrid model compared to the baseline model is:

\[
S_e = \frac{t_c}{t_b} = f \cdot s
\]  \( (5.2) \)

For alternate data analysis, the effective slowdown (CPU time ratio) equals the visible
slowdown (wall clock time ratio).

For \textit{concurrent} data analysis, the visible slowdown may be smaller than the effective slowdown (due to task parallelism on the host CPU), and depends on $u_b$, the fraction of time the data analysis job uses the CPU (i.e. the CPU utilization due to data analysis):

$$S = \frac{t_c}{t_b/u_b} = f \cdot s \cdot u_b$$ \hspace{1cm} (5.3)

The fraction $f$ of data analysis performed on the controller determines the visible slowdown. Moreover, for every $u_b$ we can determine the fraction $f$ to move on the controller such that the data analysis job incurs no visible slowdown cost, or can finish even faster than in the baseline approach.

A fraction $f$ of the data analysis job running on an $s$ times slower processor uses: $u_c = f \cdot s \cdot u_b$. We consider that the controller invests all its computation cycles in the data analysis: $u_c = 1$. Thus $f = 1/(s \cdot u_b)$. The entire work is done on the controller ($f = 1$) at $u_b = 1/s$.

To summarize, if $u_b \leq 1/s$ (e.g. due to competing load on the host CPU from the data-generating application), we can move the entire data analysis on the controller with no visible slowdown. (If $u_b < 1/s$ there is actually a speedup). If $u_b > 1/s$, we can move a fraction $f = 1/(s \cdot u_b)$ of the data analysis on the controller with no visible slowdown.

$$f = \begin{cases} 1, & \text{for } u_b \in [0, 1/s) \\ 1/(s \cdot u_b), & \text{for } u_b \in [1/s, 1] \end{cases}$$ \hspace{1cm} (5.4)

In addition, a few host CPU cycles have become available for other jobs. The remaining host CPU utilization due to data analysis is:

$$u_h = (1 - f) \cdot u_b = \begin{cases} 0, & \text{for } u_b \in [0, 1/s) \\ u_b - 1/s, & \text{for } u_b \in [1/s, 1] \end{cases}$$ \hspace{1cm} (5.5)
5.2.2 Energy study

The energy savings in the hybrid model compared to the baseline model are:

\[ \Delta E = 1 - \frac{\Delta E_h + \Delta E_c}{\Delta E_b} \]  \hspace{1cm} (5.6)

The energy consumption of the host CPU in the hybrid model decreases with the fraction of work transferred to the controller: \( \Delta E_h = (1 - f) \cdot \Delta E_b \). Thus \( \Delta E = f - \Delta E_c/\Delta E_b \).

The energy consumption over a time interval \( \Delta t \) at a power rate \( P \) is: \( E = \Delta t \times P \). Considering a \( \Delta P \) increase in power consumption between the idle and load processor states, the equation becomes:

\[ \Delta E = f - \frac{t_c}{t_b} \cdot \frac{\Delta P_c}{\Delta P_h} \]  \hspace{1cm} (5.7)

Finally, using the \( t_c \) formula determined in Equation (5.1), the energy savings are:

\[ \Delta E = (1 - sp) \cdot f \]  \hspace{1cm} (5.8)

5.2.3 Experimental study

In this section, we present a concrete estimation of energy savings compared to job slowdown. We conducted power usage and timing measurements on the following platforms to obtain realistic results:

- **Embedded CPU (controller):** We measured an example of a high-end 32-bit controller, the 1 GHz ARM Cortex-A9 MPCore dual-core CPU running on the Pandaboard [pan] development system.

- **Host CPU:** The Intel Core 2 Quad CPU Q6600 at 2.4 GHz.

We benchmarked the speed of the two processors for a single internal core, with Dhrystone [dhr]. Although Dhrystone is not necessarily an accurate predictor of application performance, results in later sections show that it gives a fairly good approximation for the
ones we will look at (Section 5.3). We measured whole-system idle and load power consumption in each case. Table 5.2 presents the measured values and the resulting parameters $s$ and $p$.

**Table 5.2:** Measured parameters for speed and power consumption, and derived parameters $s$ and $p$.

<table>
<thead>
<tr>
<th>$s_h$ (DMIPS)</th>
<th>$s_c$ (DMIPS)</th>
<th>$\Delta P_h$ (W)</th>
<th>$\Delta P_c$ (W)</th>
<th>$s$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>16215</td>
<td>2200</td>
<td>21</td>
<td>0.8</td>
<td>7.3</td>
<td>0.038</td>
</tr>
</tbody>
</table>

![Figure 5.4](image.png)

**Figure 5.4:** Energy savings versus slowdown in the Hybrid Model. The controller is used in conjunction with the Host CPU to carry out part of the computation. No job slowdown occurs for $S = 1$.

Figure 5.4 illustrates the performance-energy tradeoffs of the hybrid model, specifically the energy savings and slowdown depending on the fraction of data analysis carried out on the controller. The x-axis shows the fraction $f$ of data analysis performed on the controller. In the specific case of $f = 1$, i.e. the entire data analysis is performed on the controller, the energy savings reach the peak value of 0.72 at the effective slowdown cost of 7.3 (the $S_e$ line at $f = 1$). However, if the data analysis job utilizes only half of the CPU time in the baseline model by sharing it with other concurrent jobs (the “$S$ for $u_b = 0.5$” line), its
visible job slowdown in the hybrid model is proportionally less (e.g. for $f = 1$ the visible slowdown is about 3.6).

**Figure 5.5:** Data analysis split without slowdown, in the Hybrid Model. The controller is used in conjunction with the Host CPU to carry out part of the computation. No job slowdown occurs for $S = 1$.

**Figure 5.6:** Energy savings without slowdown, in the Hybrid Model. The controller is used in conjunction with the Host CPU to carry out part of the computation. No job slowdown occurs for $S = 1$.

Figures 5.5 and 5.6 investigate how to split data analysis in the hybrid model to obtain energy savings without any visible job slowdown. Figure 5.5 shows the fraction of data
analysis on each processor in this case. If the baseline host CPU utilization is smaller than 0.13 \((u_b < 0.13 \text { on the x-axis})\), the entire data analysis can be accommodate on the controller without a visible slowdown. Moreover, Figure 5.6 shows that in this case \((u_b < 0.13)\), performing the entire data analysis on the controller gives a speedup \((S < 1, \text { y2-axis})\) and peak energy savings of 0.72. Even in the worst case (at \(u_b = 1 \text { on the x-axis}\)), i.e. full host CPU utilization due to data analysis (baseline model, alternate), the controller is able to free the host CPU by about 0.13 (Figure 5.5), while saving about 0.1 of the baseline energy consumption (Figure 5.6) without slowing down the data analysis.

**Discussion:** These results indicate that moving the entire data analysis on to the controller, or even just a fraction of it, can give significant energy savings. Moreover, the fraction of data analysis to be carried out on the controller can be tuned (based on the baseline host CPU utilization due to the data analysis job) to control the job slowdown cost. In some cases, energy savings can be obtained without slowing down the data analysis.

### 5.3 Data Analysis Applications

To demonstrate the feasibility of the Active Flash model in real-world cases, we examine four data analysis applications from representative domains of high performance computing. The post-processing performed in these examples is driven by the contrast between the large volumes of high-precision data which may be needed to represent the state of a simulation closely enough for it to evolve accurately over time, as compared to the lesser amount of detail which may be needed in order to examine the state of the simulated system at a single point in time. *Data reduction* encompasses a range of application-agnostic methods of lossy (e.g. decimation, precision reduction, etc.) or lossless compression; our analysis examines simple lossless data compression on scientific data in several formats. In contrast, *feature detection* refers to more application-aware computations, tailored to extract relevant
data in a particular domain. We investigate two fairly general-purpose feature-detection algorithms—edge and extrema detection—on meteorological and medical data, as well as a specialized algorithm for heartbeat detection.

In estimating performance requirements for an Active Flash implementation, we note that HPC simulations do not necessarily output data at a constant rate. In particular, an alternate output behavior is that of checkpointing, where the computational state on all nodes is periodically written to storage, allowing recovery to the latest such checkpoint in the case of interruption due to e.g. hardware failure. Although transparent mechanisms for performing checkpointing exist [AAC09], application-implemented checkpoints using files which may serve as the final output of the simulation are in fact common. We consider the case where an application writes a checkpoint to local SSD at regular intervals; the checkpoint is then post-processed on the Active Flash device for e.g. central collection and visualization. In this scenario we have a deadline for Active Storage computation; this processing must complete in the window before the next checkpoint is written. The size of these checkpoints is bounded by the node memory size, and their frequency is limited by the duration of the checkpoint write process, and the desire to minimize the overhead of these periodic halts on the progress of the main computation.

5.3.1 Description of applications

**Edge detection:** Edge detection is an example of feature extraction applied to image processing, in which specific portions (i.e. the edges) of a digital image are detected and isolated by identifying sudden changes in image brightness. We use SUSAN, an open source, low-level image processing application [SB97] examined in earlier studies of active storage [RGF98], to detect edges in a set of weather images collected with GMS (Japan’s Geostationary Meteorological Satellite system), which are publicly available on the Weather Home, Kochi University
Figure 5.7: (a), (b) – Edge detection applied to an image rendering weather information from June 11, 2011, provided by GMS-5 (Japan Meteorological Agency) and the Kochi University Weather Home. The edges were detected with a brightness threshold of 40.

Finding local extrema: Finding the local maxima and minima in a noisy signal is a problem which appears in several fields, typically as one of the steps in peak detection algorithms, along with signal smoothing, baseline correction, and others. A comprehensive study
of public peak detection algorithms on simulation and real data can be found in [YHY09].

We use the open source implementation available at [Xu11] to detect local extrema in a wave signal, using a method [Bil11] which looks for peaks above their surroundings on both sides by some threshold distance, and valleys below by a corresponding threshold. We apply this application to a set of ECG (electrocardiogram) signals from the MIT-BIH Arrhythmia Database [MM01]; example results may be seen in Figure 5.8(a), using a threshold distance (delta) of 0.1.

**Heartbeat detection:** Heartbeat detection is a signal processing application with great
impact in medical fields; although typically used with real patient data, the rise of computational science in medical fields [PFP+11] leads to applications of such algorithms in the HPC simulation environments targeted by Active Flash. We evaluate the performance of the SQRS heartbeat detection algorithm [EZ79], which approximates the slope of an ECG signal by applying a convolution filter on a window of 10 values. It then compares this filtered signal against a variable threshold to decide whether a normal beat was identified; sample output is shown in Figure 5.8(b). We evaluate an open source implementation of the SQRS algorithm from PhysioNet [GA+00], applied to ECG signals from the MIT-BIH Arrhythmia Database.

**Data compression:** Data compression is used in many scientific domains to reduce the storage footprint and increase the effective network bandwidth. In a recent study, Welton et al. [WKC+11] point out the advantages of decoupling data compression from the HPC software to provide portable and transparent data compression services. With Active Flash, we propose to take the decoupling idea one step further, and harness the idle controller resources to carry out data compression on the SSD.

We use the LZO (Lempel-Ziv-Oberhumer) lossless compression method which favors speed over compression ratio [lzo]. Experiments were conducted using two common HPC data formats encountered in scientific fields: NetCDF (binary) data and text-encoded data. The data sets are extracted from freely available scientific data sources for atmospheric and geosciences research (NetCDF format) [CORa], and bioinformatics (text format) [Eur].

5.3.2 **Experimental setup**

The experimental platforms used are the same as in Section 5.2.3: the Pandaboard development system featuring a dual-core 1 GHz ARM Cortex-A9 MPCore CPU (controller), 1 GB of DDR2 SDRAM, and running Linux kernel 3.0, and a host machine featuring an Intel Core
2 Quad CPU Q6600 at 2.4 GHz, 4 GB of DDR2 SDRAM, and running Linux kernel 2.6.32. The applications chosen were all platform-independent C language programs; to reduce the effect of compiler differences GCC 4.6.1 was used on both platforms for all tests. Measurements were made of the computation phase of each program (i.e. excluding any time taken by input or output) running on a single core with no competing processes; measurements were made on both the host CPU and the controller.

### 5.3.3 Results

A summary of measured timings and data reduction values is given in Table 5.3.

<table>
<thead>
<tr>
<th>Application</th>
<th>Computation throughput (MB/s)</th>
<th>Data reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>controller</td>
<td>host CPU</td>
</tr>
<tr>
<td>Edge detection</td>
<td>7.5</td>
<td>53.5</td>
</tr>
<tr>
<td>Local extrema</td>
<td>339</td>
<td>2375</td>
</tr>
<tr>
<td>Heartbeat detection</td>
<td>6.3</td>
<td>38</td>
</tr>
<tr>
<td>Compression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>average bin&amp;txt</td>
<td>41</td>
<td>358</td>
</tr>
<tr>
<td>binary (netcdf)</td>
<td>49.5</td>
<td>495</td>
</tr>
<tr>
<td>text</td>
<td>32.5</td>
<td>222</td>
</tr>
</tbody>
</table>

Figure 5.9 gives a comparative view of computation speeds on controller and host CPU for the four data analysis applications. Edge detection and heartbeat detection are more computation intensive, compared to the other two applications. We used the detailed log-scale to display these speed values of about 7 MB/s on the controller. Data compression is next in terms of computation speed, averaging at about 41 MB/s. Unlike the previous applications, local extrema detection is less computationally intensive. Current interfaces commonly transfer data at about 200 MB/s. Thus this application is I/O bound instead of computation speed limited.

Figure 5.10 illustrates the time needed to process 30 GB of data (half the memory size per node of the Gordon system) at the measured speeds. Compression and local extrema
Figure 5.9: Computation throughput of the data analysis applications running on the SSD controller, and on the host CPU, respectively. The bottom part of the figure uses a log-scale to observe low y-axis values in detail.

detection are very fast even on the controller (under 15 minutes). Edge and heartbeat detection are slower, but still deliver acceptable timings (70-80 minutes) for a realistic scientific computation. We note that these measurements use fairly compact input data formats; it is likely that actual simulation data will be less dense due to the need for higher precision for each data point. Since the runtime of these algorithms is typically a function of the input size in data points, throughput in practice is likely to be higher.

Figure 5.11 shows how many times longer it takes to run these applications on controller, instead of the host CPU. On average, the slowdown is about 7.2, which confirms the benchmarking results from Section 5.2. The same figure shows energy savings of about 0.75. The measured computation speeds for each application, and the measured power values of each test platform (Section 5.2.3) are used in Equation (5.7) to derive the fraction of saved energy.

In all cases, the output can be represented much more compactly than the original data (it contains less information). The feature extraction applications delivered a substantial data reduction of over 90%, while compression averaged at about 50% for the two input
Figure 5.10: Computation time for 30 GB input data on the controller and on the host CPU, respectively. The bottom part of the figure uses a log-scale to observe low y-axis values in detail.

Figure 5.11: Slowdown and energy savings estimated with Equation (5.7) using $f = 1$, and the measured times and power values.

data formats studied. We observe that compressing binary data is faster than compressing text, at the expense of a smaller compression ratio (the text format is more compressible than binary NetCDF). The input and output are in binary format for heartbeats and edge detection, and text format for local extrema.

Discussion: These results indicate that off-loading data analysis to a storage device
based on a high-end controller has the potential to deliver acceptable performance in a high performance scientific computing environment. Using heartbeat detection as an example, the rate at which the ECGSYN electrocardiogram signal simulator [ecg] generates output data on our Intel host CPU is 3.2 MB/s of text data, equivalent to 0.26 MB/s in the binary format assumed in Figure 5.9. Even assuming 16 cores per host node, each producing simulation output at this rate, the total output is comfortably less than the 6.3 MB/s which could be handled on the controller. Alternately, assuming a checkpoint post-processing model, we see that the worst-case time for processing a volume of data equal to a realistic node checkpoint size is roughly an hour, making it realistic to consider flash-based processing of checkpoints in parallel with the main computation. Best suited for Active Flash are applications with minimal to no data dependencies, such as the ones illustrated here. Subsets of weather/ECG simulation data can be analyzed independently, without the need to exchange partial results among the compute and storage nodes. Also, we assume that jobs run without contention, as nodes are typically dedicated for an HPC application at a time.

5.4 Scheduling Data Analysis on Flash

In this section, we propose several policies to schedule both data analysis on the flash device and flash management tasks, i.e. garbage collection (GC). GC is typically triggered when the number of free blocks drops below a pre-defined threshold, suspending host I/O requests until completion; it is therefore important to schedule analysis and GC in a way that optimizes both analysis as well as overall device I/O performance.

5.4.1 Scheduling policies

The scheduling policies examined are as follows:

**On-the-fly data analysis** – Data written to the flash device is analyzed while it is
still in the controller’s DRAM, before being written to flash. The primary advantage of this approach is that it has the potential to significantly reduce the I/O traffic within the device by obviating the need to re-read (and thus re-write) data from the SSD flash. However, the success of this approach is dependent on factors such as the rate at which data is output by the main application, the computation throughput on the controller and the size of the controller DRAM. If data cannot be analyzed as fast as it is produced by the host-resident application, then the application must be throttled until the analysis can catch up.

**Data analysis during idle times** – In *idle-time* data analysis, controller-resident computation is scheduled only during idle times, when the main application is not performing I/O. Most HPC I/O workloads are bursty, with distinct periods of intense I/O and computation [KGS+10]; for these workloads, it is possible to accurately predict idle times [MRZ+09, MRL+09], and we exploit these idle times to schedule data analysis on the controller. This increases the I/O traffic inside the flash device, as data must be read from flash back into DRAM, and after computation written back to flash. However, our results indicate that, in many cases (i.e. computation bound data analysis), the additional background I/O does not hurt overall I/O performance.

**Idle-time data analysis plus GC management** – With *idle-time-GC* scheduling, optimistic garbage collection tasks as well as data analysis are controlled by the scheduler. Since GC will occur when absolutely necessary regardless of scheduling, data analysis is given priority: if an idle time is detected, but there is no data to be processed for data analysis, then, GC is scheduled to run instead. This complements the default GC policy, where GC is invoked when the amount of available space drops below a minimum threshold. Pushing GC earlier during idle times may incur additional write amplification than if GC were triggered later, because fewer pages are stale by the time GC is invoked. However, this early GC does not affect performance since it happens only when the device is idle.
5.4.2 Simulator implementation and setup

We have used the Microsoft Research SSD simulator [APW+08], which is based on DiskSim [BSGC08] and has been used in several other studies [SXX+09, KOS+11, LKS+11]. We have simulated a NAND flash SSD, with the parameters described in Table 5.4. We have extended the event-driven SSD simulator to evaluate the three scheduling policies. In addition to the default parameters for SSD simulation, the Active Flash simulator needs additional parameters, which are shown in Table 5.5.

Table 5.4: SSD parameters in the MSR simulator.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total capacity</td>
<td>64 GB</td>
</tr>
<tr>
<td>Flash chip elements</td>
<td>16</td>
</tr>
<tr>
<td>Planes per element</td>
<td>8</td>
</tr>
<tr>
<td>Blocks per plane</td>
<td>2048</td>
</tr>
<tr>
<td>Pages per block</td>
<td>64</td>
</tr>
<tr>
<td>Page size</td>
<td>4 KB</td>
</tr>
<tr>
<td>Reserved free blocks</td>
<td>15 %</td>
</tr>
<tr>
<td>Minimum free blocks</td>
<td>5 %</td>
</tr>
<tr>
<td>FTL mapping scheme</td>
<td>Page-level</td>
</tr>
<tr>
<td>Cleaning policy</td>
<td>Greedy</td>
</tr>
<tr>
<td>Page read latency</td>
<td>0.025 ms</td>
</tr>
<tr>
<td>Page write latency</td>
<td>0.2 ms</td>
</tr>
<tr>
<td>Block erase latency</td>
<td>1.5 ms</td>
</tr>
<tr>
<td>Chip transfer latency</td>
<td>25 ns</td>
</tr>
</tbody>
</table>

The MSR SSD implementation captures the I/O traffic parallelism over flash chip elements. However, the controller is a resource shared by all of the flash chips. While I/O requests to different flash chips may be scheduled simultaneously, computation (i.e. processing a new data unit on the controller) can only start after the previous one has ended. Our
extension to the simulator accounts for this fundamental difference between handling I/O streams and computation.

We implemented the idle-time scheduling policy, wherein data analysis is triggered when the I/O queue is empty and there are no incoming requests. A typical GC policy such as that implemented in this simulator will invoke GC when the amount of free space drops below a minimum threshold (Table 5.4 - 5% of the storage space, equivalent to 0.33 of the reserved space). In order to implement the idle-time-GC data analysis policy, we introduced an additional GC threshold, the GC-idle threshold, set to a high value (0.9 of the reserved space, equivalent to 13.5% of the storage space) to allow additional dirty space to be reclaimed during idle times.

While we expect a high degree of sequentiality in HPC data, we have experimented with worst-case conditions. We have simulated a synthetic write workload, consisting of small random writes, to represent the data generated by the main application on the host CPU. The request lengths are exponentially distributed, with a 4K mean value, and the inter-arrival rate (modeled by a uniform distribution) is set accordingly in each experiment to simulate different data generation rates of the scientific application running on the host CPU. The volume of write requests issued is 1 GB in every test.

5.4.3 Results

Figure 5.12 illustrates the potential bottlenecks in the Active Flash model: the computation throughput of the analysis on the controller, the flash management activity, in particular GC, and the I/O bandwidth of flash. In our results, we evaluated the scheduling policies by studying the effects of these limiting factors.

To ensure high internal activity, the entire logical space of the SSD is initially filled with valid data. As the state of the reserved block list plays a critical role in SSD performance, we considered the following two experimental conditions for free block list:
• **free**: All reserved blocks are initially free. A previous GC process has already reclaimed the space.

• **full**: The reserved blocks are initially filled with invalid data, resulting from previous write requests (updates). We maintained only a very small number of free blocks in order for the GC process to work. Victim blocks selected for cleaning contain mostly invalid pages, and possibly a few valid pages. Before the block is erased, the valid pages are moved in a new free block. A minimal number of free blocks (in our experiments, 5 blocks per flash chip) ensures that there is space to save the valid data during cleaning.

In the following experiments, we refer to the data generation rate of the main application running on the host CPU as *data generation rate*, and to the computation throughput of the data analysis running on the SSD controller as *computation throughput*.

**I/O bound data analysis**: We evaluated an I/O bound data analysis, in which the I/O bandwidth of flash represents the main bottleneck (bottleneck 3 in Figure 5.12). With contemporary SSDs, featuring high I/O bandwidth of 150-250 MB/s, and even higher for some PCIe-based SSDs (400-500 MB/s), the case of I/O bound data analysis is expected to be less common.

In these experiments, we compare on-the-fly and idle-time data analysis scheduling policies, when all analysis was performed on the controller (case \( f = 1 \) in the hybrid model from...
Figure 5.13: I/O-bound data analysis. Maximum sustained data generation rate of the scientific application for “on-the-fly” and “idle-time” data analysis running entirely on the controller at 390 MB/s. I/O bandwidth of simulated SSD = 145 MB/s (bottleneck). Analysis starts either with reserved space “free” (no GC), or “full” (intensive GC). GC becomes the bottleneck in case “full”, for data generation rates higher than 25 MB/s. $r$ = data reduction ratio (values of “$r$” experimented with: 0, 0.5, 0.9).

Section 5.2). A very high throughput (390 MB/s) was set for controller-based data analysis, so that the maximum write bandwidth (145 MB/s) of the flash array in our simulated SSD would be the bottleneck; results are presented in Figure 5.13.

Case ‘free’ (no GC): If the entire reserved space of the emulated SSD is initially free, the SSD can accommodate 1 GB of write requests without the need to invoke GC. In this case, the controller-resident data analysis can keep up with a maximum data generation rate which highly depends on the data reduction resulting from analysis.

First we present the results for the on-the-fly policy. If no data reduction was obtained after data analysis ($r = 0$), then the same volume of data is written to the SSD, and the maximum sustained data rate is limited by the I/O bandwidth of the SSD (145 MB/s). If the data analysis resulted in $r = 0.5$ data reduction, then only half of the input data size is written to the SSD, resulting in a higher data generation rate of about 260 MB/s which can be sustained. If the data analysis reduced the data considerably, by $r = 0.9$, the I/O traffic
is much decreased, and the computation part of the data analysis on the SSD becomes the limiting factor (bottleneck 1 in Figure 5.12), at 390 MB/s (i.e. the computation throughput of the data analysis job running on the controller).

For the idle-time data analysis policy, the maximum sustained data generation rate ranges from 75 MB/s to 123 MB/s, increasing with data reduction (Figure 5.13). However, with this scheduling policy, the entire data is first written to the SSD, which reduces the maximum sustained rate below the I/O bandwidth of the SSD. Other factors that contribute to the smaller sustained data generation rate of the idle-time policy compared to the on-the-fly policy (for I/O bound data analysis) are: additional background I/O traffic necessary to read the data back to the controller and then write the results of data analysis, and restricting data analysis to idle times only.

*Case ‘full’ (intensive GC)*: If we start without any free space (Figure 5.13), intensive space cleaning is required to bring the minimum number of free blocks above the minimum limit. Due to garbage collection, the maximum sustained data generation rate for 1 GB of data drops to 25 MB/s regardless of the value of data reduction ratio (bottleneck 2 in Figure 5.12).

**Computation bound data analysis:** We studied the maximum sustained data generation rate of the scientific application of computation bound analysis (bottleneck 1 in Figure 5.12), for on-the-fly and idle-time scheduling policies. The entire data analysis was performed on the SSD controller (*f* = 1 in the hybrid model discussed in Section 5.2) by the time the host-resident application finished generating data. Since in this case the computation was the limiting factor, data reduction did not have a significant effect on the results and was set to 0.5.

Figure 5.14 shows the maximum sustained data generation rate depending on the computation throughput of controller-resident data analysis. As concrete examples, we pinpoint
Figure 5.14: Computation-bound data analysis. Maximum sustained data generation rate of the scientific application depending on the computation throughput (bottleneck) of in-storage data analysis. Analysis starts either with reserved space “free” (no GC), or “full” (intensive GC). GC becomes the bottleneck in case “full”, for data generation rates higher than 25 MB/s.

on the figure the computation bound data analysis applications whose performance was measured in Section 5.3 (feature extraction, i.e. edge/heartbeat detection running at 7 MB/s, and compression running at about 41 MB/s on the controller).

Case ‘free’ (no GC): When data analysis has the entire reserved space free initially, no garbage collection is required to process the 1 GB of data. The maximum data generation rate is dictated by the computation throughput of the data analysis. Both on-the-fly and idle-time strategies show a linear increase with the computation throughput on the controller.

Case ‘full’ (intensive GC): High GC activity was required when data analysis was started with no free reserved space. The maximum data generation rate increased linearly with the computation throughput of in-storage data analysis up to 20 MB/s, after which the background GC and related I/O activity become the limiting factor (bottleneck 2 in Figure 5.12).

Data analysis in hybrid schemes: In the results above, we investigated the maximum sustained data generation rate of the scientific application to accomplish the entire data
analysis on the controller \( f = 1 \) in the hybrid model from Section 5.2). Here we examine the case where only a fraction \( f < 1 \) of the data analysis is offloaded to the storage controller with the rest carried out on the host CPU.

[(a) Fraction of data analysis run on controller, start state “free” (no GC)]]

![Graph](image)

[(b) Fraction of data analysis run on controller, start state “full” (intensive GC)]

![Graph](image)

**Figure 5.15:** In-storage data analysis during idle times in hybrid schemes. Fraction of the data analysis which can be accommodated on the controller, while being able to sustain a specific data generation rate. In (b), intensive GC saturates the SSD at about 25 MB/s.

The hybrid model works best with the idle-time scheduling policy, based on the following considerations. For the on-the-fly policy, generated data is stored in the DRAM residing on
the SSD and the data analysis job running on the controller reads the data from DRAM for processing. The size of the DRAM incorporated in the SSD restricts the amount of data that can be stored for analysis. Once the DRAM becomes full, the data analysis needs to keep up with the main application. With the idle-times policy, the generated data is stored on the SSD, and, depending on the availability of idle times, a portion of the data is processed by the data analysis job running on the controller. Thus, higher data generation rates (having fewer idle time periods) can also be sustained, however, in that case, only a part of the data analysis can be accommodated on the controller, and the rest will be carried out on the host CPU. Scheduling analysis during idle times allows for trading part of the active data analysis for a higher data generation rate.

Figure 5.15 examines this tradeoff—the fraction of data analysis possible on the controller versus the host-resident application data generation rate—for different data analysis speeds, ranging from 2 MB/s up to 43 MB/s. Since these values are smaller than the maximum sustained data generation rate for I/O bound data analysis (‘idle’) illustrated in Figure 5.13 (i.e. 75-125 MB/s depending on data reduction), the data analysis in these experiments is computation bound (bottleneck 1 in Figure 5.12). Next we discuss the results illustrated in Figure 5.15 for the data analysis examples described in Section 5.3, i.e. compression and feature extraction applications.

Case ‘free’ (no GC): The entire data analysis can be carried out on the controller at data generation rates smaller or equal to the data analysis computation throughput on controller (i.e. 41 MB/s for compression and about 7 MB/s for feature extraction), as was previously discussed (see the computation bound data analysis section). For feature extraction, when the data generation rate is increased from 7 MB/s to 25 MB/s, the controller can still handle 0.3 of the entire data analysis during idle times, and a further increase to 60 MB/s of data generation rate drops this fraction to 0.1. For compression, when the data generation rate is increased from 41 MB/s to 60 MB/s, the controller can still handle 0.7 of the data analysis.
**Case ‘full’ (intensive GC):** The impact of intensive GC is shown in Figure 5.15(b). At 25 MB/s data generation rates, the controller can handle a fraction of 0.28 for feature extraction analysis, while compression is able to run to completion. For data generation rates higher than 25 MB/s, intensive cleaning eventually saturates the SSD (bottleneck 2 in Figure 5.12).

**Data analysis with Garbage Collection management:** Previous results showed the high impact of GC on SSD performance. The third scheduling policy proposed here addresses this concern by tuning GC to take advantage of idle time availability in application workloads.

The default GC mechanism implemented in the SSD simulator triggers GC when the number of free reserved blocks drops under a hard (low) threshold (Table 5.4). We introduced an additional soft (high) threshold (Table 5.5) to coordinate the idle-time GC activity. Thus, when the number of free blocks is between the two thresholds, we push the GC earlier during idle times, given that there is no data to be processed on the controller at that moment.

The experiments started without any free space (**Case “full”**), to trigger intensive GC (bottleneck 2 in Figure 5.12), and Figure 5.16 shows the fraction of reserved blocks that are clean at the end of the experiments. The bars in the figure are labeled with the fraction of data analysis that the controller was able to accommodate during idle times. The results are illustrated for different computation throughputs of data analysis, and various application data generation rates.

In all cases, GC needs to raise the number of free blocks at least up to the low threshold (cleaning 0.33 of the entire reserved space). For slow data analysis (1 MB/s), this is the most it can do. Since the computation takes long, data analysis on the controller monopolizes idle times. For faster data analysis, such as 7 MB/s in case of feature extraction applications, and small data generation rates (1 MB/s, 5 MB/s), GC is able to raise the number of free blocks up to the high threshold (cleaning 0.9 of the reserved space), while performing the entire
Figure 5.16: In-storage data analysis during idle times in hybrid schemes. Data analysis with Garbage Collection management, “full” start state (intensive GC). Garbage collection is pushed earlier during extra available idle times. The bar labels represent the fraction of data analysis accommodated on the controller. The y-axis shows fraction of reserved blocks that were clean at the end of each experiment (which started with no reserved blocks free), for different data generation rates “s”.

Data analysis on the controller. Sustaining higher data generation rates is possible with faster data analysis/reduction, e.g. GC cleans 0.9 of the reserved blocks during compression (running at 41 MB/s) of data generated at a rate of 20 MB/s.

Discussion: These results indicate that the on-the-fly policy offers the advantage of significantly reducing the I/O traffic, while the idle-time policy maximizes storage resource utilization by carrying out data analysis during low-activity periods. Also, idle-time scheduling offers flexibility: it permits sustaining the desired (high) application data generation rate, when only part of the data analysis is performed on the controller, and the rest on the host CPU (the hybrid Active Flash model).

Multiple factors affect the maximum sustained data generation rate, depending on the type of data analysis. For I/O bound data analysis, the data reduction size obtained from running the analysis has a major impact. This is not the case for computation bound data analysis, where the computation throughput of data analysis determines the maximum
sustained data generation rate.

Garbage collection activity significantly affects performance. *GC tuning during idle times* proposed in the third policy can be a valuable resource. Consider a sequence of application workloads that generate data at different rates. We can take advantage of the extra idle times in slower applications, to clean the invalid blocks on the SSD and thus be able to accommodate the other faster applications in the workload as well.

## 5.5 Active Flash Prototype

We implemented an Active Flash prototype using the Jasmine OpenSSD development platform [Jas], with the parameters shown in Table 5.6. OpenSSD is based on the Barefoot™ controller from Indilinx — an ARM-based SATA controller used in numerous high-performance SSDs such as Corsair Memory’s Extreme/Nova, Crucial Technology’s M225, G.Skill’s Falcon, A-RAM’s Pro series, OCZ’s Vertex/Vertex [Jas].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>ARM Indilinx Barefoot™ at 87.5 MHz</td>
</tr>
<tr>
<td>Host Interface</td>
<td>SATA 2 at 3 Gbps</td>
</tr>
<tr>
<td>SDRAM</td>
<td>64 MB</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>64 GB</td>
</tr>
</tbody>
</table>

The OpenSSD platform can be used to develop open source SSD firmware; it provides basic source code for sample FTLs and for interfacing with SATA controller and SSD controller. We implemented Active Flash on top of OpenSSD, adding code to facilitate running data analysis kernels on the SSD controller.

The Active Flash mechanism is illustrated in Figure 5.17. Two types of requests are recognized:
Figure 5.17: Active Flash prototype.

- normal read/write requests;

- data analysis requests.

Data analysis requests are transmitted over an out-of-band channel which is not used for normal read/write requests. A data analysis request is essentially a write to a reserved LBA location which is intercepted, interpreted and resolved by the controller.

Thus, in-storage data analysis consists of the following two steps:

1. **request:** Host CPU sends data analysis request to the SSD over SATA, in the form of a write command to a reserved LBA location.

2. **resolve:** SSD controller interprets the command and runs the requested data analysis kernel. The result is either returned to the host, or written to the SSD.

### 5.5.1 Data analysis commands

The request message format is illustrated in Figure 5.18. The main fields (Type, LBA and Length) indicate the desired kernel, and the location and size of input data. We use a 32-bit representation for each field.

There are two possible methods of data storage on the SSD: as raw data or as a file within the host file system. Data location depends on the storage method as follows:
raw data: In this case, the data has previously been written to a pre-established location on the SSD using direct I/O operations. In Linux, the device is mapped to the /dev directory and can be written either using existing utilities (e.g. the “dd” command) or the C/C++ API for file access (open, read, write). Given a start address and a length, raw data is written as a single contiguous block. The data analysis request format from Figure 5.18 reduces to $N = 1$ and a single (LBA, Length) pair.

file: In this case, the SSD has previously been partitioned, formatted to a new file system (e.g. ext3), which is mounted to a directory on the host. The file system manages the address mappings on the device, and thus the data allocation may be fragmented, depending on the available logical space. The data location on the SSD is represented as a set of $N$ fragments (intervals), each being identified by a (LBA, Length) pair, denoting the start address and the size of one contiguous interval. In Linux, the file system block numbers (LBAs) of a file can be retrieved using FIBMAP ioctl() requests or the hdparm utility.

The basic command fields – Type, LBA and Length – suffice to encode several data anal-
ysis requests, such as the statistical kernels demonstrated later in this section, i.e. aggregates (max, mean), standard deviation, linear regression, etc.

The Options field is used when the data analysis kernel needs additional specific parameters. For example, a K-means clustering kernel may require knowledge of some variables defining the current problem instance, such as number of vectors, dimension of each vector, number of clusters, etc. Based on the data analysis type, the controller expects and interprets the Options field, extracting the associated parameters.

Scientific data usually uses a multi-dimensional array layout to permit efficient subset processing. The array dimensions can be included in the Options field, otherwise they default to specific or implicitly known values. For example, aggregate functions (min, max, mean) implicitly treat data layout as a one dimensional array of numbers, and linear regression implicitly works on 2-column 2D-array of numbers.

Alternatively, data can be stored on the SSD in standard self-describing scientific formats such as NetCDF [RD90] or HDF5 [HDF]. A self-describing format consists of two parts: a header which describes the layout of the data arrays (and other metadata in the form of name/value attributes), and the actual data arrays. While not currently integrated in the Active Flash prototype, self-describing data complies with our prototype design, thus the implementation could be extended to handle it as well.

5.5.2 Scheduling I/O and data analysis tasks

SSD controllers perform multiple functions, including the normal read/write operations and other additional internal operations (e.g. garbage collection, wear leveling, bad block management, error correction, etc.) triggered by I/O service tasks. Thus the I/O service tasks and data analysis tasks compete for computing time on the SSD controller. In order to minimize the impact of data analysis on SSD performance, we implemented a preemption-based scheme in which data analysis is given low-priority, being interrupted when I/O requests...
Figure 5.19: Competing I/O and data analysis tasks. Preemption of data analysis.

Consider the sequence of events from Figure 5.19. The HPC application sends I/O requests $IO_1$ and $IO_2$ (e.g. writes) to the SSD at two different times, each followed by a data analysis request, denoted in the figure by $DA_1$ and $DA_2$. Since $IO_2$ arrives while $DA_1$ is in progress, the lower priority task $DA_1$ is preempted and resumed after $IO_2$ finished. After $DA_1$ completes, the second data analysis request, $DA_2$, is processed next. Data analysis requests are serviced in the order they are received; this is implemented using a FIFO queue. To ensure persistency in case the system goes down unexpectedly, this queue can be stored on flash in the region reserved for data analysis commands.

Any interleaved sequence of I/O service tasks and data analysis tasks is handled as described above, by preempting data analysis when an overlap occurs. This requires saving the state of the interrupted task, to be able to resume it later. Since the data analysis kernels process the input data in a streaming fashion, it is necessary to maintain information about the current position in the input stream. Thus the current data analysis state is defined by the following parameters:

- next logical page address to be processed
- sector offset
- number of sectors remaining
The data analysis preemption mechanism is implemented as follows. Data analysis tasks progress in small granularity units, by essentially processing a segment of the input data at a time. After each data segment is processed, the controller checks for any pending I/O tasks that may have been received in the meantime. In case there are pending I/O requests, the current data analysis task is interrupted, its state is saved, and the controller switches to the I/O tasks; otherwise, the data analysis continues on the next segment of input data.

I/O requests arriving while a data analysis task is in progress have to wait until the current data segment is processed. On average, this delay represents about half the time to process a segment and depends on the data analysis kernel.

\[
\text{Delay} \ (x) = \frac{1}{2} \frac{S}{H(x)}
\]  

(5.9)

where \( S \) is the input data segment size, and \( H(x) \) is the throughput of the data analysis kernel \( x \) on the controller. In our implementation, the input data segment is \( S = 32 \) K, since this is also the read size from flash to controller’s DRAM used by the firmware. For the data analysis kernels experimented with (see Table 5.7), running at about 2–4.5 MB/s on the SSD controller, the average delay is about 3 to 8 ms. This shows that the interference of data analysis tasks with I/O service requests is relatively small.

5.5.3 Results

We implemented the host-controller communication mechanism described above, along with four data analysis kernels, as part of the SSD firmware. The data analysis kernels experimented with represent statistical computations common for scientific data processing: max, mean, standard deviation and linear regression. These kernels are implemented in a streaming fashion: data is read from flash to DRAM in 32 K segments and processed one segment at a time. The input for all experiments consisted of 100 MB of data (32-bit unsigned integers, in binary format), stored on the SSD as raw data.
We ran each of the data analysis kernels in the Active Flash prototype by sending data analysis request commands from a Linux host and running the computation on the SSD controller. For comparison, we also ran the same data analysis kernels on a Linux Host machine, carrying out the computation on the Host CPU. The two environments present the following configurations:

- **Active Flash**: implemented within the OpenSSD platform, with the specifications from Table 5.6.

- **Host**: features an AMD Phenom (tm) 9550 Quad Core Processor at 2200 MHz, 2 GB of DRAM, 64 GB of flash memory, and SATA 2 host interface at 3 Gbps.

**Throughput**: The throughput results are shown in Table 5.7. We see that the statistical data analysis kernels achieve about 2–4.5 MB/s on the SSD controller (second column), and about 7–14 times more on the Host CPU (third column). However, these results are obtained on an 87.5 MHz controller (see Table 5.6), and are expected to scale up with the speedier cores integrated in higher-end SSDs (e.g. OCZ RevoDrive X2 presents four 780 MHz Tensilica cores).

**Table 5.7**: Throughput of data analysis kernels running on the SSD controller within the Active Flash prototype, compared to the throughput of running the same kernels on the Host CPU.

<table>
<thead>
<tr>
<th>Data analysis kernel</th>
<th>Throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Active Flash</td>
</tr>
<tr>
<td></td>
<td>ARM 87.5 MHz</td>
</tr>
<tr>
<td>Max</td>
<td>4.5</td>
</tr>
<tr>
<td>Mean</td>
<td>4.0</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>3.3</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>1.9</td>
</tr>
</tbody>
</table>

**Division of I/O and computation time**: The total data analysis run time consists of
two parts – I/O and computation:

\[ T = T_{io} + T_{comp} \]  

(5.10)

In Active Flash, \( T_{io} = T_{io}^{AF} \) represents the time to read the data from flash to controller’s DRAM:

\[ T_{io}^{AF} = T_{FLASH->DRAM_ONBOARD} \]  

(5.11)

In the case of Host–resident data analysis, \( T_{io} = T_{io}^{H} \) is the time to read the data from flash to the DRAM on host:

\[ T_{io}^{H} = T_{FLASH->HOST} \]  

(5.12)

Table 5.8 shows the splitting between I/O time (i.e. data read from flash to DRAM) and computation time in each of the two cases: Active Flash prototype and Host CPU–resident data analysis, for 100 MB of input data. As expected, in Active Flash, \( T_{comp} \) is dominant \( (T_{comp} \gg T_{io}) \), while with Host–resident data analysis, \( T_{io} \) is dominant \( (T_{io} \gg T_{comp}) \).

We used a 32 K read size from flash to DRAM, which results in comparable time spent in reading for the two cases (i.e. 2.4 s in Active Flash and 2.6 s for Host). With smaller read size, the Host total read time increases (e.g. at 4 K read size, \( T_{io} \) is 9 s).

Table 5.8: Run time splitting between I/O (i.e. data read from flash to DRAM) and computation, of data analysis kernels running on the SSD controller within the Active Flash prototype, compared to the case of Host CPU-resident data analysis. Input size = 100 MB.

<table>
<thead>
<tr>
<th>Data analysis kernel</th>
<th>Time (s)</th>
<th>Active Flash</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ARM 87.5 MHz</td>
<td>AMD 2200 MHz</td>
</tr>
<tr>
<td>I/O</td>
<td>compute</td>
<td>I/O</td>
<td>compute</td>
</tr>
<tr>
<td>Max</td>
<td>2.4</td>
<td>19.8</td>
<td>2.6</td>
</tr>
<tr>
<td>Mean</td>
<td>2.4</td>
<td>22.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>2.4</td>
<td>27.7</td>
<td>2.6</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>2.4</td>
<td>49.6</td>
<td>2.6</td>
</tr>
</tbody>
</table>

121
**Power and energy consumption:** We measured the power rate for both the Active Flash prototype and the Host during idle time, sustained writing to flash, and data analysis. These results are shown in Table 5.9. We observe that, with the prototype, the power rate during data analysis is close to the power in idle state; this is due to the OpenSSD firmware not implementing any power savings features to reduce power consumption while not under load; instead, it busy waits, continuously checking for new tasks to execute.

**Table 5.9:** Power rate in Active Flash versus Host during idle time, sustained writing to flash, and data analysis.

<table>
<thead>
<tr>
<th>Power rate (W)</th>
<th>idle</th>
<th>writes to flash</th>
<th>data analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Flash</td>
<td>1.35</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>Host</td>
<td>79</td>
<td>80</td>
<td>96</td>
</tr>
</tbody>
</table>

We computed the energy consumption during data analysis as follows:

\[
E = P \times T
\]

(5.13)

where \(P\) is the power rate during data analysis (Table 5.9), and \(T\) is the total run time (see Equation 5.10 and Table 5.8).

**Table 5.10:** Energy consumption when data analysis kernels are running on the SSD controller within the Active Flash prototype, compared to the energy consumption when running the same kernels on the Host CPU.

<table>
<thead>
<tr>
<th>Data analysis kernel</th>
<th>Energy consumption (Ws)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Active Flash</td>
</tr>
<tr>
<td>ARM 87.5 MHz</td>
<td>AMD 2200 MHz</td>
</tr>
<tr>
<td>Max</td>
<td>31</td>
</tr>
<tr>
<td>Mean</td>
<td>34</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>42</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 5.10 presents the energy consumption \(E\) measured in watt-seconds (i.e. joules).
We see that the Active Flash prototype consumes 5 to 9 times less energy than the Host while running the data analysis kernels to completion.

**Discussion:** While active flash requires careful consideration with respect to design choices, the prototype presented here demonstrates that the active computation capability can be integrated in today’s SSDs without much implementation effort or major changes to current interfaces. The results indicate important reduction in energy consumption, but low throughputs. However, these measurements were performed with the less powerful 87.5 MHz controller from OpenSSD. The achievable throughput is expected to be significantly higher with current SSDs that operate at hundreds of MHz and feature multiple cores. Piggybacking the computation on the SSD controller in periods of inactivity also comes with a gratuitous advantage: it frees the host CPU for other tasks and maximizes resource utilization.

### 5.6 Concluding Remarks

We propose an active storage model on flash, called Active Flash, which addresses I/O bandwidth and power constraints in high-performance computing. Typically, scientific HPC applications consist of both simulation and post-processing tasks, which require many rounds of I/Os between compute nodes and storage. Active Flash reduces the data transfer costs by moving the post-processing step closer to where data already resides, from the host CPU onto the storage controller. We explore energy and performance trade-offs in moving computation from host to storage, demonstrate the ability of embedded controllers to perform data analysis tasks at acceptable speeds, and present a simulation study of the possibilities of scheduling controller-resident computation along with host I/Os and internal flash management tasks. In addition, we implement an active flash prototype and evaluate it on four data analysis kernels running on the controller.
Chapter 6

Conclusions

As NAND flash becomes widely used in storage systems, behavior of flash and flash-specific algorithms becomes ever more important to the storage community. Various factors have a major effect on the potential performance of flash devices. In this thesis we study write endurance, throughput efficiency, and active computation capability, providing new models to capture their effect on device performance.

We have investigated write endurance on a small scale on USB drives and on flash chips themselves due to their accessibility; however the values we have measured and approaches we have developed are applicable across devices of all sizes. We have shown how reverse-engineered details of flash translation algorithms from actual devices in combination with chip-level measurements may be used to predict device endurance, with close correspondence between those predictions and measured results. We have also presented non-intrusive timing-based methods for determining many of these parameters.

In addition, we have developed analytic models of the performance of two commonly-used flash translation layers, along with black-box models that do not require knowledge of underlying algorithms. We present a methodology to apply these models to real-world workloads by capturing free space utilization and traffic sequentiality, and validate this methodology by measurements both in simulation and on real devices.
As HPC clusters continue to grow, relative performance of centralized storage subsystems has fallen behind, with state-of-the-art computers providing an aggregate I/O bandwidth of 1 MB/s per CPU core. We propose a new model, Active Flash, which utilizes energy-efficient storage controllers to perform data analysis, thus addressing both I/O bandwidth and system power constraints which limit the scalability of today’s HPC systems. We examine the energy-performance trade-offs of the Active Flash approach, deriving models that describe the regimes in which Active Flash may provide improvements in energy, performance, or both. Measurements of data analysis throughput and corresponding power consumption for actual HPC algorithms show that in-storage computation using Active Flash could significantly reduce total energy with little performance degradation, while simulation of I/O-compute trade-offs demonstrates that internal scheduling may be used to allow Active Flash to perform data analysis without impact on I/O performance. As a proof-of-concept, we have implemented an Active Flash prototype featuring a host–controller communication protocol to send and intercept data analysis commands, and a preemption mechanism to interrupt data analysis during I/O tasks. The prototype is demonstrated on four data analysis applications from the field of statistical analysis.

One could argue that low-power processors as in FAWN [AFK+09] coupled to local flash storage would benefit from power savings similar to Active Flash, without the need to extend the current firmware in SSDs. However, the Active Flash approach has an additional essential advantage: a more efficient interconnect between the flash chips and the compute engine. The internal bandwidth in SSDs often exceeds their external bandwidth by a factor of $2-4\times$, and this gap is expected to grow due to increased internal parallelism [CPO+11]. This high internal bandwidth in today’s and projected future SSDs recommends them as a feasible technology for large-scale data-intensive computing, especially for streaming data processing such as searching, filtering, aggregation. Additional advantages of in-storage processing over an array of low-power CPUs coupled to local SSDs are: energy savings from not having
to transfer the potentially large amount of data to the CPU, and cost savings by using a compute resource that is already available in SSDs, and thus obviating the need to buy low-power CPUs.

The research work presented in this thesis has sparked several open problems, that constitute promising future work directions.

First, a more in-depth understanding of current flash translation layer algorithms in SSDs, and of their interaction with workload patterns is necessary to be able to better quantify and predict both endurance and performance of flash devices. Our work described in Chapter 3 provides valuable insight especially for low-end devices. However, devices at the other end of the spectrum, i.e. high-end devices, may employ additional optimizations such as multi-channel striping and possibly buffering policies that significantly increase endurance and performance (as discussed in Section 4.5). Thus, understanding the intricacies of SSDs and how they ultimately affect endurance and performance on various workload patterns is yet an open issue, and methods as the ones described in Chapter 3 (reverse-engineering, timing analysis) can be used to tackle this problem.

Second, designing scheduling algorithms that improve SSD performance and prolong the life of flash devices on various request patterns is another important future research direction. As discussed in Section 3.4, the behavior of SSDs largely depends on the workloads they are presented with. While for disks there already exist efficient scheduling algorithms (e.g. the elevator algorithm), request scheduling algorithms for flash have not yet been implemented in practice, leaving space for much improvement in this area. FTLs are built to take advantage of temporal locality; thus a significant performance increase can be obtained by reordering data streams to maximize this advantage. For flash, most benefit is gained with a scheduling policy in which the same data blocks are written successively, in order to reduce garbage collection activity, consisting of data movement, block merges and erasures (unlike for disks, for flash devices there is no reason to reschedule reads).
Third, since many real-world applications can potentially benefit from Active Flash, identifying these applications and integrating them with active flash techniques is another promising future work direction. In this thesis, we experimented with Supercomputing scenarios, however we envision applicability of Active Flash in the fields of Cloud Computing and Big Data as well. Active Flash is particularly useful for non-indexed search applications, where it can reduce I/O bottlenecks (along with energy consumption) by processing large amounts of data locally, on the storage device, providing an output which is generally much smaller compared to the input. Examples of such real-world applications include: 1) Data de-duplication, with the goal of identifying redundant data sections: with Active Flash, splitting the input in chunks and computing the hash of each chunk can be done locally, on the SSD controller. 2) Error checking, since transferring data over the network may introduce errors, thus a checksum or hash of the data can be computed and verified on the SSD controller after the data is received (or computed and appended to the data before the data is sent). 3) Computing statistics on collected data, e.g. from monitoring logs, is also a frequent real-world activity. These large logs residing on persistent storage can be directly processed in Active Flash, by extracting the relevant information and computing the statistics locally. Examples of statistics that are often needed by actual applications include: occurrence distribution of some event, data aggregation, etc. Some of these statistics have been implemented in our prototype, as presented in Section 5.5, however many others are frequently used in practice. Thus, a future research direction with strong real-world impact consists of exploring applicability of Active Flash in today’s computing landscape, shaped by Big Data requirements and large-scale use of computing resources, identifying the applications that can gain a realistic benefit from Active Flash, quantifying this benefit and understanding potential tradeoffs.

Fourth, using dedicated hardware logic to speed up active flash computations is another potential research direction. While the SSD controller can be responsible for examining
the data stream read from flash, additional hardware inside the SSD such as compare logic and aggregation logic can be used to increase efficiency of specific functions, depending on the problem requirements. For example, the compare logic may apply proper filtering conditions based on matching values stored in registers. Hardware acceleration has been previously proposed by Kim et al. [KOP+11] to speed up database scan operations on SSDs. Also, a number of SSDs on the market such as the Intel Solid-State Drive 520 Series [Dat] use dedicated hardware to implement on-board data compression and thus automatically compress data sent to the SSD. Similarly, applications running on the controller, in Active Flash, can be accelerated with specifically designed hardware. This constitutes a future research direction with significant potential for Active Flash optimizations and efficiency increase.
Bibliography


dhr ECL Dhrystone Benchmark. White paper.


[GKU09a] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address


