BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs

Sungjin Lee[1], Kermin Fleming[2], Jihoon Park[1], Keonsoo Ha[1], Adrian Caulfield[3]
Steven Swanson[3], Arvind[2], and Jihong Kim[1]

School of CSE[1] CSAIL[2] Department of CSE[3]

Seoul National University Massachusetts Institute of Technology University of California, San Diego

{chamdoo,promar2,air21c,jihong}@davinci.snu.ac.kr {kfleming,arvind}@csail.mit.edu {acaulfie,swanson}@cs.ucsd.edu

Abstract: *In this paper we describe BlueSSD, an open platform for exploring hardware and software for NAND flash-based SSD architectures. We introduce the overall architecture of BlueSSD from a hardware and software perspective and briefly explain our design methodology. Preliminary evaluation shows that BlueSSD delivers performance comparable to commercially available SSDs.*

1. Introduction

With continuing improvements in both capacity and price, NAND flash-based solid-state drives (SSDs) are becoming increasingly popular in a variety of systems ranging from embedded devices to enterprise servers. Over the past few years, a considerable number of studies have been conducted on designing storage architectures and firmware techniques [1]. However, the evaluation and verification of such designs has typically been performed in software simulation, raising the usual simulation issues of modelling speed and fidelity.

The picture for SSD prototyping grows even more cloudy in the future. Flash devices have already reached 20nm, and manufacturers have already begun to suggest that, to meet disk-scaling estimates, multi-level flash cells (MLC), which permit the encoding of several bits per cell, will have to be exploited to provide higher density. However, the performance degradation encountered in using multi-level flash is well-known: MLC flash is both slower and has a higher bit-error rate than single level cell (SLC) flash. Thus, developers of flash disks are faced with a somewhat difficult challenge – how to improve performance and reliability of SSDs while dealing with an increasingly unreliable substrate.

It is likely that the solutions to the density problems will require some hardware acceleration. Compression techniques reduce the raw data stored on the disks, improving density. Error correction seeks to increase disk reliability, at the cost of decreasing density. The interplay of these and other similar operations is not clear. However, it is likely that hardware acceleration, by itself, will not provide a complete solution. The file system layer may contain information which can improve the performance of the hardware layer. For example, the file system may flag files that it believes will compress well. Therefore, careful attention must be paid to the co-design problem to enable the development of cross-layer protocols.

In this paper, we present an open development platform for flash-based SSDs, called *BlueSSD*, which is based on an FPGA prototyping system. Our primary goal is to develop a generic infrastructure for designing, implementing, and evaluating hardware/software components efficiently. BlueSSD provides a software framework for developing flash firmware on top of a user-modifiable hardware platform. Hardware components of BlueSSD are developed using Bluespec [2], a high-level hardware description language that supports fast prototyping, especially as compared to Verilog and VHDL. BlueSSD provides the first open-source, flash-enabled infrastructure to the community.

2. Solid State Drives

SSDs seek to emulate traditional hard disk drive (HDD) technology while providing higher performance at a lower operating power. Like its predecessor, the solid state drive provides traditional read and write operations on logical pages of data, typically a few kilobytes in size. SSDs however, have a few key operational differences which require the introduction of a separate, specialized software, known as the flash translation layer (FTL), to maintain the abstraction of the HDD. This layer allows production file systems to seamlessly interface with the underlying SSD substrate.

The major difference between flash and disk resides in the write operation. Flash technology does not support re-writes in-situ. Rather, due to the physical properties of flash cells, a block in flash must be erased before it can be rewritten. Although the SSD write operation is relatively fast, the erase operation takes a long time and operates over a block comprised of many pages, rather than on a single page. Thus, to erase a block, all pages with live data within a block must be copied somewhere else. These properties of the SSD yield the two most important functions of the FTL. First, because logical blocks in the SSD change physical location frequently, a mapping problem exists, which must be solved by the FTL. Second, due to the coarse granularity of the erase operation, erasure-on-write is impractical. Thus, the FTL must take care of producing writable pages off-line, so that a pool of pages is always available for writing. This process, which involves erasing pages and copying their live content elsewhere, is known as garbage collection, and exposes many interesting architectural trade-offs, particularly since flash chips have a finite number of erase-write cycles in their useful lifetime.

Beneath the FTL lies the SSD controller. Imple-

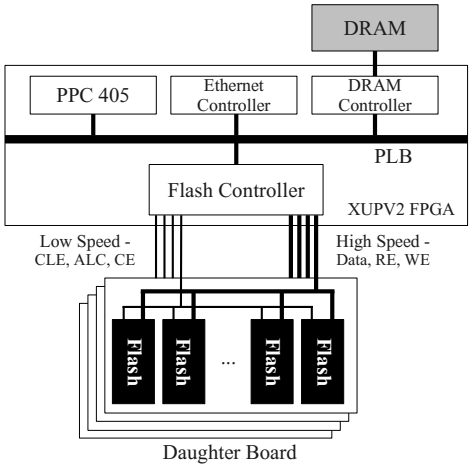


Figure 1: The overall architecture of BlueSSD.

mented in hardware, this layer translates the commands of the FTL into a series of operations, typically involving the transfer of data from memory to the flash, but possibly also involving other compute-intensive data transformations, like compression or error correction. This layer may also introduce some fine-grained reordering of operations according to their data dependencies. For example, the controller may allow reads to bypass outstanding erase and write operations, provided that no data dependencies are violated.

At the bottom of the SSD stack are the physical devices, an amalgamation of flash chips wired together on a set of shared buses. The buses are operated independently by separate, relatively simple finite state machines.

3. BlueSSD

3.1. BlueSSD Hardware

Figure 1 shows the overall architecture of BlueSSD. BlueSSD is based on the Zarkov system [3], an SSD prototype developed by UCSD, and is extended to provide more efficient and flexible hardware/software design environment. BlueSSD is built on top of the Xilinx XUPV2P FPGA board [4] and the custom flash storage daughter board shown in Figure 4 (see Appendix). The flash daughter board holds up to 32 flash packages and up to four identical buses. Each of the four buses on the daughter board consists of up to 8 packages and up to 16 chip-enables. The bus supports eight 1- or 2-chip enable packages or four 4-chip enable packages. Currently, the board is populated with 32 1GB MLC flash parts, each with a single chip enable.

The flash controller of BlueSSD is implemented in the FPGA fabric using a combination of a Bluespec wrapper and modified bus controllers from Zarkov. The PowerPC 405 (PPC 405) on the XUP board runs the Linux 2.6.25.3 kernel [5] that is used as a platform for firmware such as a flash translation layer (FTL). The DRAM memory is used to store data for both system software and also the hardware system. The 100Mb Ethernet controller allows us to transfer data from/to the PC host system.

3.2. Hardware Design Methodology

The major issue in developing extensible hardware is interface definition. This problem is particularly important in the context of BlueSSD because we expect users to extend its hardware and software functionality in parallel. In typical hardware designs, designers are free to choose whatever wire semantics and timing they like. For example, a wire involved in notifying a user of some status could be pulsed once or held at a level until the status changes. This diversity greatly increases the burden of modification to the system, since each interface must be understood in detail.

BlueSSD handles the problem of imprecise interface definition by relying on the Bluespec programming language. Bluespec automatically generates signals noting when interfaces may be activated. In BlueSSD, all interfaces generating state updates are request-response and rely on Bluespec to generate correct signalling. Thus, designers are freed from the task of understanding sub-interfaces completely – the burden of determining when an interface may be activated is placed upon the designer of the interface, not the user of the interface.

We further simplify the interfaces in BlueSSD by adopting a latency insensitive design style, which has been used to facilitate modular refinement in a several large systems [6] [7]. Our modules are not permitted to make timing assumptions about when their inputs will be ready. For example, our memory controller does not begin a write transaction until it has received the data for the transaction, because it cannot assume that streaming data will be provided at a sufficient rate. This enables users to develop code in an incremental manner, since any new module need only be *functionally* correct. Latency insensitivity is also critical in evaluating different SSDs architectures, since we may freely alter the performance of system peripherals, like memory, without affecting the functional correctness of BlueSSD.

Finally, we provide a set of abstractions for interfacing with system resources. For example, in our current system, the controller hardware accesses system memory via a multiplexed DMA engine. Components of BlueSSD which utilize the memory are presented with one of two abstractions, either a streaming burst memory interface or a BRAM-like single word memory interface; the plumbing for these interfaces is generated automatically by the compiler. Users of the memory abstractions are unaware of the underlying multiplexing, beyond observing performance degradation in the case of contention. We provide similar abstractions for inter-hardware communication and for debugging.

3.3. Debugging

The full BlueSSD system makes use of a production Linux file system and a full FTL. However, de-

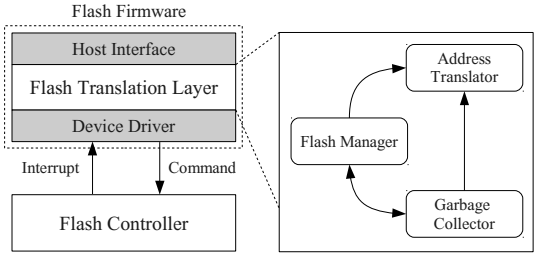


Figure 2: The overall structure of the flash firmware.

bugging a hardware system in the context of complex software system is difficult. To ease the task of hardware designer, we provide multiple debugging systems of varying functionality and complexity. For general debugging, we provide a direct, independent, and extensible interface to the hardware controller from a host PC. To facilitate this interface, we have ported the AWB [8] hardware development system to the XUPV2. We make use of the debugging facilities offered in that tool, such as soft connections, to add debug logic at various points in the controller, without modifying the external interface of the controller. We have found this particularly useful in collecting run-time statistics and in debugging deadlocks.

Although the host debugging interface is general, it is impossible to achieve 100MB/s bandwidth over a serial link. For high-speed testing, we provide a processor-based driver with a light-weight library for exercising the SSD. Since this configuration lacks an OS interface, it cannot be used to evaluate real workloads, but it can be used to collect performance statistics on a wide range of synthetic performance benchmarks. We find that data on the raw performance of the SSD hardware collected from this system is more accurate, as interference from the software system is almost completely eliminated.

3.4. SSD Software

In designing BlueSSD, one of our goals is to provide a flexible software infrastructure so that several flash management schemes can be employed with minimal modifications to the existing software system. To achieve this goal, the software of BlueSSD is constructed with a layered design emphasizing modularity at each level.

Figure 2 shows the overall structure of the flash firmware, which is composed of three layers: a device driver for the flash controller, a host interface, and a flash translation layer (FTL). The device driver provides the interface for the FTL to access the flash controller and provides the upper layer with several low-level functions that perform page read, page write, and block erase operations. The driver layer also notifies the FTL of interrupts from the flash controller, signifying the completion of operations. This interface permits the implementation of a wide range of FTL schemes.

The host interface layer is responsible for the communication between the host system (e.g., PC) and BlueSSD. More specifically, it receives I/O commands and data, and then transfers them to the FTL. After a request has been processed, the interface layer sends a response to the host system along with data read from flash memory, if it is a read request. The host interface layer is intended to abstract host communications, so that we may migrate freely between host communication protocols without modifying other firmware.

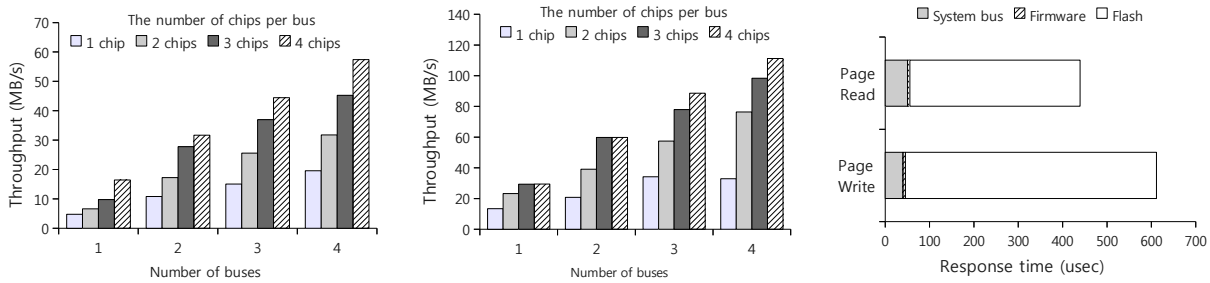
The FTL plays a major role in determining the overall performance of BlueSSD. The FTL has been developed with three components: a flash manager, an address translator, and a garbage collector. The flash manager processes I/O requests passed from the host interface layer, translating addresses and copying data to and from the hardware controller. The flash manager maintains semaphores for each flash chip, preventing concurrent requests to the same flash chip. These semaphores are updated by the device driver upon interrupts, thereby avoiding the overhead of polling.

The address translator is responsible for translating logical addresses from the host system to the physical address space of the flash controller. Currently, the translator uses a page-level mapping between logical and physical memory. Although page-level mapping requires a huge mapping table, it is highly flexible, making it easy for us to evaluate the maximum performance BlueSSD achieves under a variety of hardware configurations. To maximize the parallelism of multiple buses and chips, the address translator assigns a physical page address in zigzag order. For instance, if there exist n buses and k chips per bus, the address translator allocates physical pages to incoming logical pages from 0th chip of 0th bus to (k-1)th chip of (n-1)th bus. This approach spreads requests evenly among buses and chips, improving performance and assisting in wear-leveling.

Because flash pages are write-once, each time a page of data is overwritten in the SSD, the old page becomes invalid since it contains stale data. As invalid pages accumulate, they must be reclaimed by the garbage collector to provide free space for writing new data. In BlueSSD, the default garbage collector is invoked when a certain flash chip has no free space to store incoming data. Upon invocation, the garbage collector selects the block with the largest number of invalid pages in the target flash chip and copies any valid pages in that block to a specially reserved free block in the same chip. After copying these pages, the garbage block is erased and replaces the reserved free block for the next garbage collection iteration. The empty pages of the former reserved free block are now available for writing new data. Finally, the mapping table is updated to reflect changes in flash memory.

4. Preliminary Results

We have measured the overall throughput of BlueSSD while varying the number of buses and chips under a simple synthetic benchmark, which sends se-



(a) Write throughput (MB/s) (b) Read throughput (MB/s) (c) Breakdown of response time (usec)

Figure 3: I/O performance of BlueSSD.

quential read and write requests to the flash daughter board. The current version of BlueSSD interacts with the host system (e.g., PC) through an Ethernet-based interface. Due to the narrow bandwidth of this connection, it is difficult to measure the maximum performance that BlueSSD accomplishes. To solve this problem, our evaluations have been performed using modified flash firmware, which generates FTL I/O requests directly. Finally, a special programming method [9] was used for writing data to the MLC chips to achieve SLC-like performance.

Figure 3 shows the throughput of BlueSSD using various combinations of buses and chips, ranging from 1 to 4 buses and 1 to 4 chips. The performance of BlueSSD increased in proportion to the number of buses and chips, allowing BlueSSD to handle more I/O requests in parallel. When four buses and four flash chips per bus were used, BlueSSD achieved throughput close to 60MB/s write bandwidth and 110MB/s read bandwidth. This performance is comparable to that of Samsung's 1.8-inch SLC SSD (MCCOE64G8MPP-0VA), which exhibits 80MB/s for a write and 100MB/s for a read.

To analyze the performance in detail, we also examined the breakdown of the response time taken for reading or writing a single page from or to the flash daughter board when four buses and four chips are used. In Figure 3(c), the system bus is the time consumed by transferring data through PLB and the firmware represents the time taken to execute the firmware. The flash is the time for data to be read from or written to the flash board by the flash controller. The firmware and the system bus, which cannot process more than one I/O request simultaneously, accounted for a small portion of the total elapsed time. The flash controller consumes the majority of time, but this overhead can be mitigated by executing multiple requests in parallel.

5. Conclusion

This work presents BlueSSD, an open platform for NAND flash-based SSDs, which provides a complete hardware/software environment for flash research. Preliminary results show that BlueSSD delivers sufficient performance for academic research. We are currently researching dynamic, hardware-accelerated compression schemes and conducting a comparative study of FTL algorithms.

We also plan to improve the BlueSSD hardware. To obtain a larger FPGA and higher host bandwidth, we plan to migrate the current BlueSSD to a newer board based on the Xilinx Virtex 5 LX110T [10], which supports the PCI-E interface.

Acknowledgments: This work was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. R0A-2007-000-20116-0 and R33-2009-000-10095-0). The ICT at Seoul National University and IDEC provided research facilities for this study. This work was also supported in part by the Brain Korea 21 Project in 2010. The MIT portion of this work was funded by the National Science Foundation (Grant #CCF-0541164). Research at USCD was funded by the National Science Foundation (Grants NSF0811794 and NSF0643880).

References

[1] N. Agrawal et al., "Design Tradeoffs for SSD Performance," In *Proceedings of the USENIX Annual Technical Conference*, pp. 57-70, 2008.

[2] Bluespec Inc., http://www.bluespec.com.

[3] A. M. Caulfield et al., "Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications," In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 217-228, 2009.

[4] Xilinx Inc., "Virtex-II Pro Development System," http://www.xilinx.com/products/devkits/XUPV2P.htm.

[5] Linux From Scratch, http://www.linuxfromscratch.org/lfs/.

[6] M. C. Ng et al., "From WiFi to WiMAX: Techniques for High-Level IP Reuse across Different OFDM Protocols," In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, pp. 71-80, 2007.

[7] K. Fleming et al., "H.264 Decoder: A Case Study in Multiple Design Points," In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, pp. 165-174, 2008.

[8] J. S. Emer et al., "Asim: A Performance Model Framework," *IEEE Computer*, Vol. 35, No. 2, 2002.

[9] S. Lee et al., "FlexFS: A Flexible Flash File System for MLC NAND Flash Memory," In *Proceedings of the USENIX Annual Technical Conference*, pp. 115-128, 2009.

[10] Xilinx Inc., "XUPV5-LX110T Development System," http://www.xilinx.com/univ/xupv5-lx110t.htm.

APPENDIX

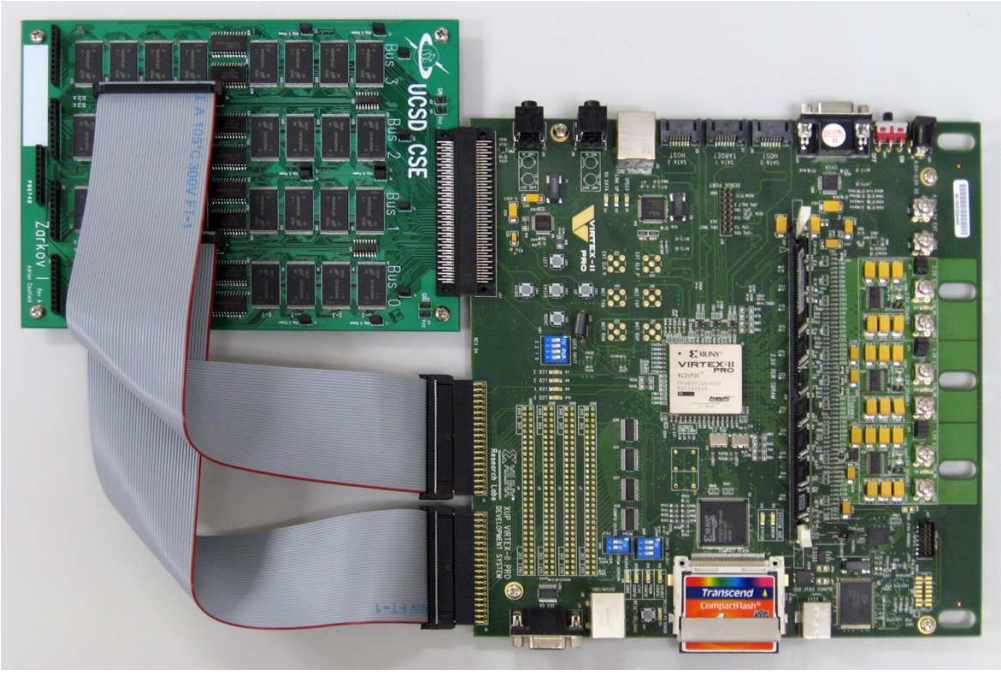


Figure 4: The Zarkov hardware platform. The flash daughter board is on the left with the XUP on the right.