FlashBench: A Workbench for a Rapid Development of Flash-Based Storage Devices

Sungjin Lee, Jisung Park, and Jihong Kim

School of Computer Science and Engineering, Seoul National University, Korea
{chamdoo, jspark, jihong}@davinci.snu.ac.kr

***Abstract*—As the cell size of NAND flash memory is shrinking, its physical characteristics such as performance and lifetime are significantly degraded. As effective solutions of overcoming such poor physical characteristics, more cross-layer system-level approaches (such as compression and deduplication techniques) are expected to be developed. These system-level techniques typically employ intelligent software algorithms supported by specialized hardware accelerators. Using hardware accelerators combined with sophisticated software algorithms greatly increases the design complexity of flash-based storage devices. However, existing storage design environments are not adequate enough to handle this increased design complexity in a timely and efficient manner. To address this new challenge, we propose a novel storage development environment, called FlashBench, that helps developers to build high-complexity storage solutions quickly. FlashBench is designed to provide a generic framework for the rapid development and validation of storage software/hardware algorithms by supporting multi-level design environments, specifically optimized for seamless hardware/software cross-layer integrations. Our case study demonstrates that FlashBench enables developers to implement high-complexity flash devices with specialized optimization functions in a shorter development time over traditional design environments.**

I. Introduction

NAND flash-based storage devices have been widely used in mobile embedded systems because of its low-power consumption and high mobility. Recently, thanks to the continued scale-down of a NAND memory cell size combined with the use of the multi-level cell (MLC) technology, NAND flash-based solid-state drives (SSDs) have emerged as an attractive storage solution, replacing hard disk drives (HDDs).

Unfortunately, as the semiconductor process is scaled down and the multi-level cell (MLC) technology is commonly used, the performance and lifetime of NAND flash memory deteriorates significantly. For example, the number of program/erase (P/E) cycles of single-level cell (SLC) flash memory fabricated in a 7x nm process is 100K P/E cycles. On the other hand, for 2-bit MLC flash memory fabricated in a 3x nm process, the number of P/E cycles is reduced to 10K [1]. The performance of MLC flash memory is also seriously degraded with an increase of the NAND flash memory density. The read and write latencies of SLC flash memory are about 20 μs and 200 μs, respectively, but these numbers are increased to more than 40 μs and 800 μs in MLC flash memory [1].

In order to overcome the poor physical characteristics of the semiconductor substrate, a cross-layer system-level approach, which employs intelligent software algorithms supported by specialized hardware accelerators, is emerging as a promising solution [2]–[4]. Data compression is a representative cross-layer approach that takes advantage of a high-speed hardware compression accelerator [2], [3]. The endurance of flash memory depends on the amount of data written to it. Thus, if the amount of written data is reduced by data compression, the lifetime of flash devices can be proportionally improved. Moreover, since hardware-assisted compression reduces data physically transferred from/to flash memory with a small computational overhead, the I/O performance can be improved as well. Data deduplication [4] also reduces the number of bytes written to flash devices by storing only unique data blocks by identifying duplicate blocks using cryptographic functions, e.g., SHA-1. The identification of the unique data block can be accomplished with a minimal performance penalty if a cryptographic function is supported by hardware.

Flash software must be carefully redesigned and be integrated with hardware modules to fully utilize the capabilities of hardware acceleration. For example, if hardware-accelerated compression is employed, requested data are to be stored in flash memory in a compressed form. To effectively deal with such compressed data, several flash software modules, including address mapping and garbage collection, must be redesigned properly; otherwise, the benefits of using hardware compression cannot be maximally exploited [3]. The similar design issues are also raised when data deduplication is used. With data deduplication, different logical blocks could share the same physical block if their contents are identical. If these identical blocks are improperly managed, serious data reliability/integrity problem cannot be avoided.

Even though a cross-layer system-level solution is very effective in overcoming many challenges of newer NAND flash memory technologies, it greatly increases the design complexity of flash-based storage devices. Unfortunately, existing storage design environments (such as ones based on functional simulation [5], [6] and ones based on custom storage prototyping [7], [8]) are ill-prepared to deal with such complicated hardware/software design issues effectively. Since most existing environments focus on a single design level, when multiple design levels are necessary ranging from a highly software-oriented level to a detailed hardware-software interaction level, developers often need to move a different design environment by manually modifying their designs, which is a very time-consuming process. In order to build a more-

978-1-4673-2789-3/12/$31.00

optimized flash storage device that satisfies challenging performance and lifetime requirements of new NAND flash memory in a timely and efficient manner, therefore, we strongly believe that a more efficient storage design environment is necessary.

In this paper, we present a flash storage development environment, called *FlashBench*. The FlashBench provides multiple design environments at different design levels so that storage developers can work on the most appropriate design level for a given development stage. In particular, FlashBench satisfies three important requirements in designing high-complexity flash devices. First, FlashBench enables a *rapid development* and *validation* of tightly coupled software/hardware modules by providing high-level software and hardware development platforms. Thus, developers evaluate the feasibility and correctness of new algorithms at an earlier stage of development. Second, FlashBench supports a *seamless* and *easy cross-layer integration* of software/hardware modules. With a well-defined and easy-to-extend interface, FlashBench makes it easy for developers to seamlessly integrate software/hardware modules and to easily extend the hardware/software interface. Third, FlashBench helps to implement *highly flexible* and *portable* software modules. This allows developers to implement, evaluate, or optimize software algorithms at the most appropriate design level without modifying software modules themselves. Besides satisfying the requirements above, FlashBench offers a set of useful libraries, reference software designs, and a high-level debugging facility.

In order to evaluate the effectiveness of FlashBench, we performed a case study that designs and implements flash-based SSDs with hardware compression and specialized software algorithms. By leveraging multiple design environments (at different design levels) offered by FlashBench, the algorithms were designed and implemented quickly in an incremental fashion without greatly increasing the design complexity. The integration of software modules into hardware modules was accomplished smoothly with no great difficulty. The result is a highly-optimized SSD prototype developed during a relatively short time. Our resulting SSD prototype improved the performance and lifetime by 28% and 43% respectively, over the existing SSD without hardware compression.

The remainder of this paper is organized as follows. In Section 2, we describe existing design environments for developing flash devices. Section 3 presents the detailed descriptions of FlashBench and our storage design flow based on FlashBench. In Section 4, we illustrate our case study of building an SSD prototype based on hardware-accelerated data compression. Section 5 concludes with a summary and directions for future extensions to FlashBench.

II. Related Work

Over the past few years, functional simulation [5], [6] and custom storage prototyping [7], [8] have been typically used in designing flash-based storage devices. Functional simulation environments can be grouped into two types, one based on an abstract device model and the other based on a detailed device model. The abstract model-based simulation uses a simple and abstract storage device model, which functionally models the primitive I/O behaviors of NAND flash memory, but actual data are ignored [5], [6], It is useful to explore and evaluate the design of new algorithms quickly using various I/O traces. Unlike the abstract model-based simulation, the detailed model-based simulation provides a realistic storage environment by simulating/emulating a real storage device accurately. It helps developers to evaluate new algorithms while executing real applications. Furthermore, it enables developers to verify the correctness of new algorithms, e.g., data consistency and data integrity, which cannot be assured by the abstract model-based simulation. The custom storage prototyping environment provides a real storage platform where hardware/software modules are to be implemented. It thus has an advantage of providing more accurate and correct assessment of algorithms over the functional simulation environment [7], [8].

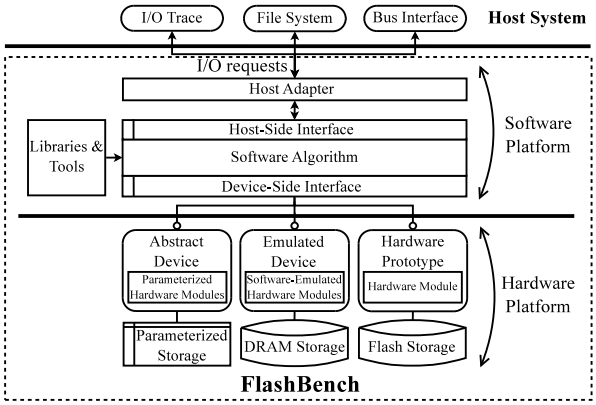


Fig. 1: An overall architecture of FlashBench.

Each storage design environment has a distinctive advantage over the others in terms of the design exploration speed, algorithm validation, and evaluation accuracy. However, since the existing design environments are not vertically integrated from the abstract design level to the most detailed level, the advantage of each environment is rarely realized in practice because they require significantly different software/hardware interface. FlashBench overcomes such limitations by providing multiple design environments that enable developers to produce highly flexible and portable software algorithms running on various hardware device models. This makes it possible for developers to easily obtain the benefits of functional simulation and custom storage prototyping.

III. Design and Implementation of FlashBench

We start by describing an overall architecture of FlashBench and presenting our design flow. We then explain the important components of FlashBench in details.

A. Overview of FlashBench

Figure 1 shows an overall architecture of FlashBench, which is composed of two main platforms: a hardware platform and a software platform. The hardware platform provides various storage hardware device models at different design abstraction levels from an abstract device model to a hardware prototype. The software platform provides a framework for the design

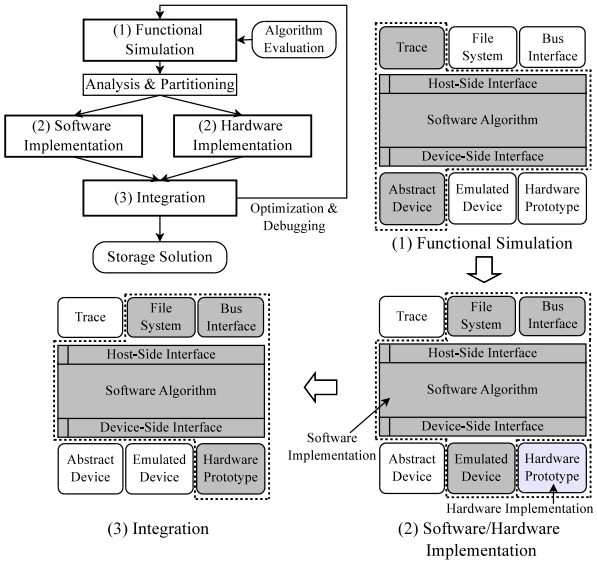


Fig. 2: A design flow and use cases with FlashBench.

and implementation of flash software algorithms, in addition to offering several libraries and tools for fast development and easy evaluation. Input requests for FlashBench can be in different formats. FlashBench can work with 1) I/O request traces collected from various host systems, 2) I/O requests issued from a file system, 3) I/O requests transmitted through the bus interface, e.g., Ethernet, SATA, and PCIe, from a host system running on a different machine.

The hardware platform supports three kinds of hardware device models, an abstract device, an emulated device, and a flash hardware prototype, depending on their modeling accuracy. The abstract device model is similar to the device models used in abstract model-based functional simulators [5], [6] in that it models the high-level behaviors of both NAND flash memory and hardware modules with simple parameter values. As expected, the abstract device is useful to develop new algorithms quickly. The emulated device model is similar to detailed model-based functional simulators in that it emulates a real storage device using DRAM chips or external DRAM disks. In the emulated device model, custom/special hardware modules are emulated in software as well. Real applications thus can be executed in the emulated device, making it possible to verify the correctness of algorithms without a real hardware prototype. Finally, the flash hardware prototype model is an FPGA-based hardware prototyping system, which provides a more accurate development framework for developing hardware modules using real NAND flash chips.

The software platform is designed to implement highly flexible and portable software modules that can be executed on various hardware device models offered by the hardware platform. This high-level flexibility and portability can be realized by adopting a device-side interface which provides a common and extensible interface for flash software to access a hardware device. Moreover, in order to support various host systems, the software platform employs a host-side interface and a host adapter. The host-side interface defines a set of functions for accessing flash software. The host adapter is an intermediate layer between a host system and flash software that receives I/O requests from a host system and then delivers them to flash software through the host-side interface. The host adapter is capable of handling I/O requests issued from various file systems or transmitted via the bus interface[1]. The host adapter also makes it possible to evaluate new algorithms by replaying I/O traces.

Currently, the hardware and software platforms are implemented in the Linux operating system using the ANSI-C language, except for the flash hardware prototype implemented in an Xilinx FPGA board using Bluespec SystemVerilog (BSV) [9]. Due to its high flexibility and portability, software algorithms can be directly executed in a block device driver of the Linux kernel or can be easily ported to firmware inside a flash-based storage device.

B. Design Flow Based on FlashBench

In this section, we describe our design approach for developing flash-based storage devices using FlashBench. As shown in Figure 2, the overall design flow is very similar to the approach typically used in the conventional software/hardware codesign. The main difference is that FlashBench is more suitable for designing flash-based storage devices by taking into account the software/hardware hierarchy of a storage subsystem.

In the initial stage of developing flash storage devices, we often explore several design candidates and evaluate the feasibility of various algorithms. Since quick explorations of design alternatives are required, abstract model-based functional simulation is commonly used, which can be easily supported by utilizing the I/O trace replaying and abstract device facilities of FlashBench. Once a promising design solution is decided, we perform software/hardware partitioning based on the result of our simulation analysis and define the key interface between partitioned software and hardware modules. We implement the proposed software modules using the emulated device and verify the correctness of the algorithms while executing real applications. As with most designs, the software modules will be refined multiple times. At the same time, hardware acceleration modules are implemented in the flash hardware prototype. The next step is to integrate software modules into hardware modules. Since software and hardware modules are implemented using the same interface defined at the functional simulation step, the actual integration step proceeds very smoothly. Finally, we iterate the previous steps whenever necessary to improve the current design solution.

The high flexibility and portability of FlashBench helps us to perform the optimization and debugging steps in a more efficient manner. For example, if some bugs were detected during the integration step of software and hardware modules, we can easily go back to the emulated device mode without changing the algorithms themselves so that we can quickly identify whether the bugs are from the software modules or

[1]Currently, FlashBench supports only the Ethernet interface. But, it can be easily extended for other bus interfaces, including SATA and PCIe.

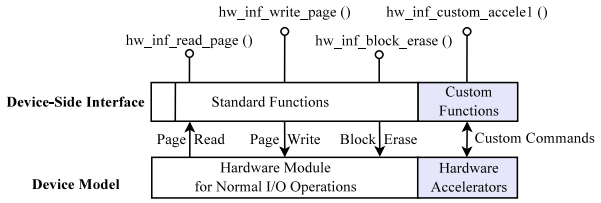


Fig. 3: Standard and custom functions of a device-side interface.

hardware modules. Since the emulated device model represents an error-free hardware, it is relatively straightforward to check if the software models have errors. The optimization and tuning of software algorithms can be conducted easily as well. We can switch to the functional simulation mode with the abstract device model, and then optimize and re-evaluate the algorithms quickly with various I/O traces. Furthermore, the easy-to-extend hardware/software interface of FlashBench allows us to change the current interface easily even if the need for a better hardware/software interface arises at the late stage of the design cycle.

C. Software Platform

We describe four main components that compose the software platform of FlashBench in details. Note that the software platform itself is a standalone software development environment that can be exploited to evaluate the effect of various software algorithms and modules (such as block device drivers, address mapping, garbage collection, and wear-leveling) on performance and lifetime while varying several parameters under a fixed hardware design.

Host-Side Interface: The host-side interface defines functions that are required for a host system to communicate with flash software. The prototypes of some important functions in the host-side interface are listed in Code 1.

```
bool host_inf_open_disk(struct* param);
void host_inf_close_disk();
bool host_inf_make_request(int request_type,
      int offset, int length, char* buffer);
```

Code 1: Prototypes of host-side interface functions

`host_inf_open_disk()` is invoked when the host system attempts to open a flash device. The host system sends an argument containing some parameters needed for the initialization of a flash device. `host_inf_close_disk()` is called when the host system closes a flash device. The host system sends I/O requests to flash software using `host_inf_make_request()` with some parameters, including a type of I/O operations, i.e, read or write, the offset of a page, the number of pages to be read or to write starting from the offset, and the pointer to the buffer memory that holds data to be written or the buffer memory where data read are to be stored. The actual behaviors of the host-side interface have to be implemented by software developers according to the algorithms they develop.

Device-Side Interface: The device-side interface is a set of functions that offer access to a hardware device, which are

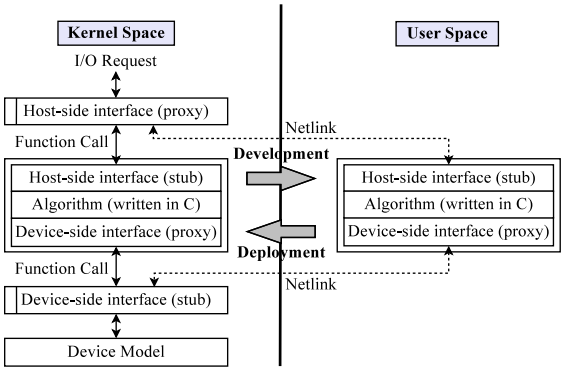


Fig. 4: Debugging support in FlashBench.

categorized into two types: standard and custom functions. The standard functions are used for handling the normal I/O operations of flash memory, such as read, write, and erase operations, and the custom functions are used for the access of hardware accelerators. Code 2 below shows the prototypes of some representative functions in the device-side interface.

```
/* standard functions */
bool hw_inf_read_page(int bus, int chip, int block,
      int page, char* buffer);
bool hw_inf_write_page(int bus, int chip, int block,
      int page, char* buffer);
bool hw_inf_erase_block(int bus, int chip,
      int block);
/* custom function */
bool hw_inf_custom_accele1(/*some arguments*/);
```

Code 2: Prototypes of device-side interface functions.

As depicted in Figure 3, `hw_inf_read_page()` and `hw_inf_read_write()` are invoked when software algorithms need to read or write some data from or to a page of flash memory. `hw_inf_erase_block()` is used to erase a block in flash memory. The custom functions, e.g., `hw_inf_custom_accele1()` in Code 2, are manually added by developers after hardware and software are partitioned.

All of the functions in the device-side interface must be implemented by software developers so that I/O commands issued from flash software can be delivered to the underlying hardware device. The standard functions for three different hardware devices are already implemented. The custom functions have to be implemented by developers later.

Debugging Support: The flash software is typically running in the OS kernel, especially in a block device driver, or it is implemented as firmware inside a flash-based storage device. Therefore, debugging flash software is difficult and challenging. FlashBench mitigates this difficulty by allowing software algorithms to be implemented in the user space of an operating system. Therefore, lots of useful debugging and profiling tools, `gdb` and `gprof`, can be used when developing software algorithms. For its easy deployment, the software algorithms implemented in the user space can be ported to a block device driver or to firmware with no or minimal changes.

To realize this benefit, we must carefully address the following two technical issues. The first issue is to keep interface-

level compatibility. The interface and its parameters of the software algorithms have to remain the same, regardless of where software algorithms run, i.e., user space or kernel space. As shown in Figure 4, we solve this problem by employing a proxy/stub model widely used in a remote procedure call (RPC) system. The host-side and device-side interfaces are divided into the proxy and the stub each. If the software algorithms are implemented at the user level, the proxy and the stub communicate with one another using a netlink protocol, a socket-like mechanism for IPC between the kernel and user space processes in Linux. If the software algorithms are implemented in the kernel, the proxy and the stub communicate with one another by a direct function call. The second issue is code-level compatibility. The algorithms implemented in the user space must be able to be run in a block device driver or firmware without any changes in source codes. Since software modules are written in the ANSI-C language compatible with a variety of platforms, FlashBench is able to achieve high-level source-code compatibility.

Libraries & Tools: FlashBench offers ready-to-use libraries which manage the status of several flash storage elements, such as chips, blocks, and pages. This helps developers to implement their own algorithms rapidly. FlashBench provides two well-known flash software designs, page-level mapping and block-level mapping FTLs [5], as a reference design for software developers. In addition, FlashBench includes several benchmark tools and a performance monitoring unit that help to evaluate the performance and lifetime of flash devices.

D. Hardware Platform

In this subsection, we explain three device models in the hardware platform of FlashBench in details. For these device models, the hardware modules (corresponding to standard functions in the device-side interface) that handle normal I/O operations are implemented already and provided to developers by default. On the other hand, the acceleration modules have to be implemented by developers.

Abstract Device: The abstract device is implemented in software in FlashBench. In the abstract device, the high-level behaviors of NAND flash memory and hardware accelerators are characterized by parameter values, which are taken from the datasheets of real devices or are estimated by simulation. For example, the performance parameters of NAND flash memory, e.g., read and write latencies, can be taken from the datasheet of flash memory parts. As another example, if a hardware compression module is required as a hardware accelerator, the average compression ratio of a given compression algorithm and the time spent for data compression can be used as these parameter values.

Emulated Device: The emulated device models the key components of flash-based storage devices, including buses, chips, blocks, and pages, using system DRAM memory or external DRAM disks, and emulates the detailed I/O behaviors of NAND flash memory in software. For example, if data compression is necessary as a hardware-acceleration unit, a software compression algorithm is implemented in the emulated device to include a compression ability in a hardware side. More accurate evaluation is possible with the emulated device over the abstract device. In the example of data compression, a compression ratio of actual data under a realistic storage device can be used for the evaluation of algorithms.

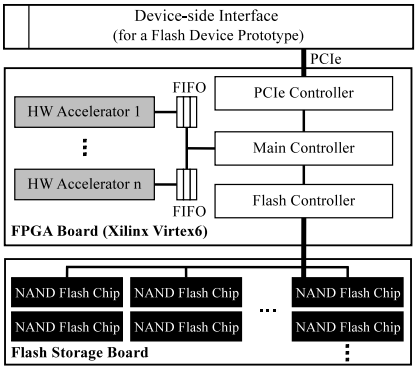


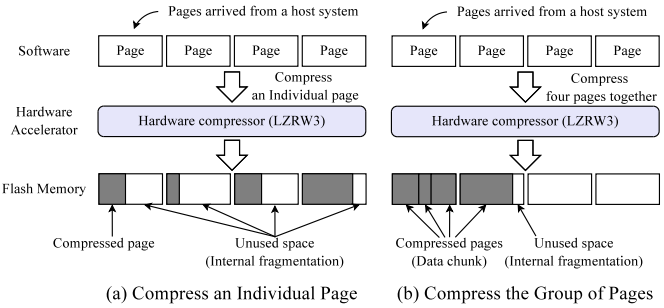
Fig. 5: An overall architecture of a flash hardware prototype.

Prototype Device: The flash hardware prototype is an FPGA-based development environment for implementing new hardware modules. Note that the flash hardware prototype is an extended version of our previous FPGA-based SSD prototype, called BlueSSD [10]. Therefore, several hardware configurations (such as the number of buses, the number of channels, and the use of DMA) can be easily changed according to the design requirements.

Figure 5 shows the overall architecture of our flash hardware prototype, which is composed of two main parts: an FPGA board and a flash storage board. The FPGA board is based on the Xilinx's Virtex6 FPGA development board which provides the reconfigurable fabric that developers use to implement various hardware modules, including hardware accelerators. All of the hardware modules in the FPGA board are written in BSV [9]. The flash storage board is a custom PCB board that holds several NAND flash memory chips.

The flash hardware prototype includes basic hardware modules, a PCIe controller, a main controller, and a flash controller. The PCIe controller accepts I/O commands (along with data if they are present) from flash software through the PCIe interface, and then delivers them to the main controller. The main controller is responsible of handling I/O commands sent from the PCIe controller. If the I/O command is a normal flash I/O operation, e.g., a read or write operation, the main controller forwards it to the flash controller, so as to read or write data from/to flash memory. If the I/O command requires hardware acceleration, the main controller sends data to the hardware accelerator so that the data are processed by the accelerator. As depicted in Figure 5, hardware acceleration modules are connected to the main controller through FIFOs in a latency-insensitive style [10], [11]. This approach allows various hardware accelerators to be easily inserted into or removed from the main controller.

The device-side interface receives commands from flash



(a) Compress an Individual Page (b) Compress the Group of Pages

Fig. 6: An internal fragmentation problem and our solution.

software specifying the parameters for read, write, erase, or custom commands, as depicted in Code 2. Then, it makes a request for the hardware using the received parameters, and sends it through the PCIe interface. Once the request has finished processing on the hardware, the device-side interface returns results to flash software.

IV. Case Study: Hardware-Accelerated Compression for Flash-Based SSDs

In this section, we describe our case study using FlashBench in building flash-based solid-state drives (SSDs) with hardware-accelerated compression coupled by specialized software algorithms. Our SSD prototype is designed and implemented according to the design flow shown in Figure 2. First of all, we analyze several compression algorithms to choose a suitable one and then investigate important design issues by leveraging the abstract device model of FlashBench. After partitioning hardware and software based on the result of our analysis, we define the hardware/software interface and then implement hardware and software algorithms simultaneously in the emulated device and in the flash hardware prototype, respectively. Finally, we perform an initial evaluation with the resulting SSD prototype and then conduct optimization to further improve the performance and lifetime of the SSD[2].

A. Design and Implementation

We first investigate three well-known compression algorithms, LZ77 [12], X-Match [13], and LZRW3 [14], so as to figure out which algorithm is suitable for being used in data storage. We investigate some important characteristics of the algorithms, including a compression ratio, computational overhead, and hardware complexity, by doing software simulation and analyzing the properties of the algorithms. After the analysis, LZRW3 is chosen as the best compression algorithm because of its high compression ratio and relatively low design complexity. As expected, for all the compression algorithms, software-based compression turns out to be infeasible because of its high computational overhead. Based on these results, we decide to implement the LZRW3 algorithm in hardware.

We then perform simulation using the abstract device model, so as to rapidly investigate technical issues raised by the use of compression. We use the page-level FTL, which is provided by FlashBench by default, as our reference software design under the assumption that a compression ratio is fixed to 45%, which is the average compression ratio of LZRW3 [14]. We find that there is an internal fragmentation problem when data compression is used in a flash-based storage device [3]. The unit of read and write operations in NAND flash memory is a page. As shown in Figure 6(a), if data compression is performed for an individual page, the size of compressed data does not fit into a unit of a page, wasting the rest of the page. Thus, the actual number of pages written to NAND flash memory is not reduced. To mitigate the internal fragmentation problem, we propose a chunk-based I/O strategy, which compresses several pages together and writes them to NAND flash memory in their entirety, as illustrated in Figure 6(b).

[2]For more detailed descriptions, refer to our previous work [3].

	Sensor	Linux	Document	MP3	Average
Compression Ratio	0.25	0.55	0.72	1.25	0.69

TABLE I: A summary of compression ratios of data files.

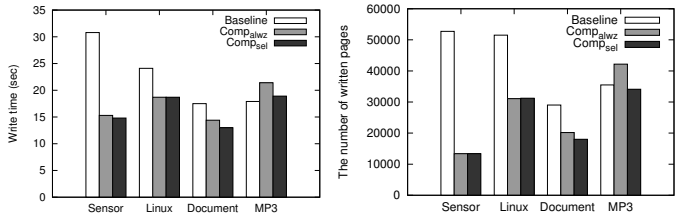
This chunk-based I/O strategy requires significant changes in the conventional software, hardware, and interface architectures. First, the flash software must be redesigned to carefully handle data chunks in flash memory, each of which is composed of multiple compressed pages. Second, the software/hardware interface must be changed to transfer the data of several pages immediately from/to the hardware. Finally, the flash hardware needs to be designed to compress the whole data chunk received from the software using the LZRW3 compression algorithm. Furthermore, it lets the software know the number of pages actually used for writing for the management of compressed data by the software.

Based on the design requirements derived from the simulation using the abstract device, we implement our software and hardware algorithms in the emulated device and in the flash hardware prototype, respectively. We also add some custom functions to the device-side interface so that reads and writes are carried out in a unit of several pages, instead of an individual page. In our case study, the integration of the software and the hardware can be done in a straight manner by just changing the configuration of FlashBench.

B. Evaluation and Optimization

We evaluate the effect of our flash-based SSD prototype on performance and lifetime using various data files, which exhibit different compression ratios. Table I summarizes the compression ratio of our data files. Figure 7 displays our evaluation results. Here, `Baseline` is the SSD prototype without hardware-assisted compression, and $\texttt{Comp}_{\text{alwz}}$ is the SSD prototype with hardware-assisted compression. The detailed descriptions of $\texttt{Comp}_{\text{sel}}$ will be presented later.

As shown in Figure 7(a), $\texttt{Comp}_{\text{alwz}}$ improves write performance by 23% on average. In addition, $\texttt{Comp}_{\text{alwz}}$ reduces the number of pages written to NAND flash memory by 37%, thus extending the lifetime of SSDs by the same amount. However, for the data file whose compression ratio is low, e.g., MP3 files, the performance and the lifetime of $\texttt{Comp}_{\text{alwz}}$ are worse than



(a) Write time (sec) (b) The number of written pages

Fig. 7: Evaluation results.

those of `Baseline`. Dictionary-based compression usually appends some metadata to compressed data for decompression. Therefore, if the compression ratio of input files is quite low, the size of compressed data could be larger than that of the original files due to metadata overhead. This is the reason why $\texttt{Comp}_{\text{alwz}}$ performs poorly, in terms of performance and lifetime, in comparison to `Baseline` for MP3 files.

In order to prevent the side effects of compression, we decide to further optimize our SSD prototype so that compression is performed selectively depending on the compression ratio of input files. For optimization, we simply change the configuration of FlashBench to use the emulated device, and then design and implement a selective compression policy in the software. Our selective compression policy determines whether to use compression or not by monitoring the compression ratio of previously written data. We also add an additional interface between the hardware and the software so that a decision of the software is delivered to the hardware. Finally, we implement hardware logic that bypasses the hardware compression step if it is not necessary. $\texttt{Comp}_{\text{sel}}$ in Figure 7 shows how selective compression affects the performance and lifetime of SSDs. As shown in Figure 7, $\texttt{Comp}_{\text{sel}}$ mitigates the performance penalty caused by compression and reduces the amount of data written to SSDs by filtering useless compression for incompressible data: $\texttt{Comp}_{\text{sel}}$ exhibits 28% higher performance and 43% longer lifetime compared with `Baseline`.

In our experience, the design and implementation of our SSD prototype can be eased by leveraging multiple design environments of FlashBench, which enable us to develop storage solutions incrementally from algorithm selection to hardware/software implementation without a significant increase of design complexity. The seamless integration support for hardware and software modules helps us to produce a high-quality storage solution by allowing optimization at the late stage of the design cycle. Our case study shows that FlashBench has great potential as a development environment for designing and implementing a well-optimized storage device in a reasonable time.

V. Conclusion

In this paper, we presented a workbench, called FlashBench, which aims to provide a generic software/hardware development framework for NAND flash-based storage devices. FlashBench allows storage developers both to assess the effectiveness of new algorithms and to verify their correctness at an earlier stage of a system design. For the seamless and easy cross-layer integration of hardware and software modules, FlashBench supports a standardized and extensible hardware/software interface infrastructure. From the result of our case study on a flash-based storage device with hardware-accelerated compression, FlashBench reduces the design time greatly in comparison with traditional design environments, improving 28% higher performance and 43% longer lifetime than that without hardware compression.

The proposed FlashBench environment can be improved in several directions. First, in the current version of FlashBench, a hardware/software interface code must be manually implemented or modified by system designers. For more efficient hardware/software co-design, the functionality of automatic interface synthesis, which is based on Bluespec Codesign Language (BCL) [15], will be added to FlashBench. Second, we will improve the host adapter so that it supports various bus interfaces, including PCIe and SATA, which are widely used in commercial SSD products.

Acknowledgement

We would like to thank anonymous referees for valuable suggestions that greatly improved the paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. R33-10095 and No. 2012-0006417).

References

[1] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[2] T. Park and J.-S. Kim, "Compression Support for Flash Translation Layer," in *Proceedings of the International Workshop on Software Support for Portable Storage*, 2010.

[3] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim "Improving Performance and Lifetime of Solid-State Drives Using Hardware-Accelerated Compression," *IEEE Transactions on Consumer Electronics*, 2011.

[4] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2011.

[5] N. Agrawal, V. Prabhakaran, and T. Wobber, "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX Annual Technical Conference*, 2008.

[6] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives," in *Proceedings of the International Conference on Advances in System Simulation*, 2009.

[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity," in *Proceedings of the International Conference on Supercomputing*, 2011.

[8] Y. Seong, E. Nam, J. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y. Bae, J. Lee, Y. Cho, and S.-L. Min, "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture", *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 905-921, 2010.

[9] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Proceedings of the International Conference on Formal Methods and Models for Co-Design*, 2004.

[10] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, Arvind, and J. Kim, "BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs," in *Proceedings of the International Workshop on Architectural Research Prototyping*, 2010.

[11] K. Fleming, C.-C. Lin, N. Dave, Arvind, G. Raghavan, J. Hicks, "H.264 Decoder: A Case Study in Multiple Design Points," in *Proceedings of the International Conference on Formal Methods and Models for Co-Design*, 2008.

[12] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, 1977.

[13] M. Kjelso, M. Gooch, and S. Jones, "Design and Performance of a Main Memory Hardware Data Compressor," in *Proceedings of the EUROMICRO Conference*, 1996.

[14] R. N. Williams, "An Extremely Fast Ziv-lempel Data Compression Algorithm," in *Proceedings of the Data Compression Conference*, 1991.

[15] M. King, N. Dave, and Arvind, "Automatic Generation of Hardware/Software Interfaces," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.