Filling in the Missing Piece: Integrating Storage into CompOrg Courses

Xiangqun Zhang, Syracuse University

Xiangqun is a fifth-year Ph.D. student majoring in Computer & Information Science & Engineering. His research interests mostly focus on solid-state drives and their applications in modern use cases, but he is also passionate about teaching and pedagogy methods. He hopes his teaching could inspire more students to become researchers in the computer science field.

Dr. Ziyang Jiao, Syracuse University

Dr. Jiao receives his Ph.D. in Computer & Information Science & Engineering from Syracuse University. His research focuses on file and storage systems, solid-state drives (SSDs), flash memory, and sustainable computing. His work is driven by a research philosophy that seeks to imbue minimal yet meaningful knowledge into various layers of the I/O stack, thereby enabling a more efficient, synergistic, and adaptive storage ecosystem.

Dr. Farzana Rahman, Syracuse University

Dr. Rahman, an Associate Professor of Teaching in the EECS department of Syracuse University. Dr. Rahman has a strong history of being an excellent educator who is loved by undergraduate and graduate students of EECS. In recognition of her outstanding teaching performance and inclusive mentoring, she was awarded Syracuse University Laura J. and L. Douglas Meredith Program's Teaching Recognition Award in 2024, and College Educator of the Year award by Technology Alliance of Central New York in 2023. Central to her teaching approach is an active-learning style, which pairs hands-on programming exercises with challenging projects that demand students to cultivate skill in problem solving, debugging and software engineering in general. She is dedicated towards creating equitable education and learning experiences for all students by providing educational opportunities that are inclusive to–and supportive of– women, genderqueer, non-binary, and underrepresented and minority (URM) students.

As a diversity spokesperson of the department, Dr. Rahman spearheads various DEIA initiatives. One of her most impactful initiatives is Research Exposure on Socially Relevant Computing (RESORC), funded by Google Research, to increase both the exposure and visibility of undergraduate research at EECS. With more than 200 students participating in RESORC over the past 3 years, she has designed and facilitated multiple virtual workshops to help undergraduate students develop computing identity, research skills, practice teaching strategies, and explore research topics in computing and engineering domain.

Dr. Rahman's research and mentoring initiatives has been supported by many funding agencies, including the National Science Foundation, Google, NCWIT, Google TensorFlow, and American Association of Colleges and Universities to develop effective pedagogy in undergraduate computer science (CS) education. She is the winner of the NCWIT Extension Services (NCWIT ES-UP) award, ABI Systers PIO (Pass-It-On) award, Google ExploreCSR Award, and NCWIT educator award. She published numerous peer-reviewed articles in venues, including the Special Interest Group of the Association of Computing Machinery (ACM SIGCSE), IEEE RESPECT, and IEEE Frontiers in Engineering Education, American Society for Engineering Education (ASEE) conference. She has received funding from different funding agencies research and mentoring initiatives directed toward developing effective pedagogy in undergraduate computer education. She holds a Ph.D in computer science and specializes in a broad area of pervasive health technologies, and computer science education.

Prof. Bryan Kim, Syracuse University

Bryan S. Kim is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Syracuse University. His research interests center around building performant, reliable, and scalable memory and storage systems for data-intensive applications using emerging hardware technologies. His work has appeared in top computer systems venues such as FAST, ATC, OSDI, and EuroSys, and his research is supported through various projects, including the NSF CAREER award.

Filling in the Missing Piece: Integrating Storage into CompOrg Courses

Abstract

The Computer Organization (CompOrg) course is fundamental for students to learn how different computer components work as a whole. Different institutions may cover different topics, but they usually share similar scopes, which include C programming, binary representation, assembly, stack/heap, overflow, virtual memory, memory management, digital logic, and pipelining. On the other hand, storage devices are vital to computer systems. The storage device topic is designated a mandatory CompOrg topic in the latest Computer Science Curricula 2023, published in early 2024; still, it is usually overlooked by CompOrg courses, according to our review of existing course offerings spanning 29 reputable US universities.

To fill in the missing piece in CompOrg, we designed a new programming assignment, SSDLab, for our CompOrg course. SSDLab provides students with the opportunity to implement Flash Translation Layer (FTL), a key component of the Solid State Drive (SSD) firmware. Furthermore, we also provide written assignments, named StorageWrit, for students to learn how storage latencies are calculated given a set of storage devices and file characteristics, including storage type, latency, file size, etc. The goal is for students to learn the concept of modern storage organization and write better code for higher I/O throughput in the era of big data by understanding and identifying possible I/O bottlenecks in storage devices. We evaluated student performance by creating exam questions corresponding to the assignments and finding the correlation between assignments and exam questions. We then ask for student feedback on their experiences and what they have learned with SSDLab. Our results show that the students learned the new topic with similar effectiveness as those traditional topics in our course with our programming and written assignments. Students who finished SSDLab showed 32% higher grades in the corresponding exam question, and students enjoyed the hands-on experiences brought by SSDLab despite its relative difficulty. The SSDLab project is available on GitHub at `https://github.com/zhxq/SSDLab`.

Introduction

Computer Organization is a milestone for undergraduates: instead of learning how to use computers as a tool to program in CS1 and CS2, they start to learn how different computer components work as a whole. CompOrg courses are, therefore, designed to cover as many aspects of computer systems as possible. Common topics in CompOrg courses include binary

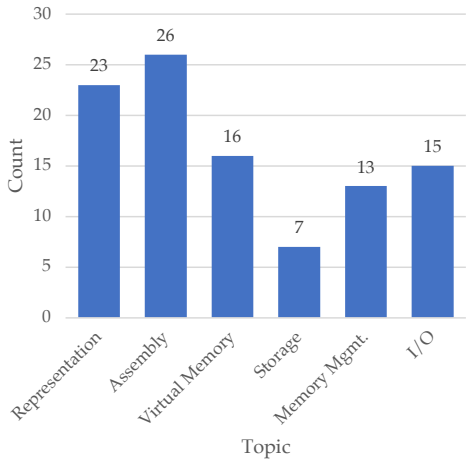

Figure 1: Topic distribution of the reviewed course offerings based on the recommended CompOrg course packaging suggestions in CSC2023. Only technical topics that also exist in CSC2013 are included. The memory hierarchy topic is split into VM and storage here.

representation, logic, assembly, CPU internals, memory hierarchy, virtual memory (VM[1]), and OS/hardware interface[1], [2].

Modern computers store programs and other data in secondary storage. When starting a program, it has to be loaded to the main memory from the secondary storage[3], and most applications will interact with storage devices with data reads/writes. The storage will become a bottleneck if it is slow, either inherently slow due to physical reasons (e.g., max disk head/platter movement speed) or due to application workload characteristics[3]. Therefore, it is essential to understand how storage devices work and make better use of them to improve application and system performance.

However, storage devices, as the lowest level yet the most fundamental part of the memory hierarchy[4], are often ignored in CompOrg courses in reality, at least in the USA. We reviewed CompOrg course offerings from 29 reputable universities in the USA by extending the scope from the work of Almansoori, et al.[5]. We chose these offerings because their course websites/syllabi are available online. The course topics are listed in Table 1, with the topic distribution shown in Figure 1. We find 7 offerings that briefly mentioned storage devices, ranging from a few sentences to a few slides. On the other hand, 15 offerings mentioned I/O of any kind, 6 of which mentioned storage-related I/O (i.e., file I/O, but not mentioning storage). No course offerings provided any detailed discussion or assignments on storage devices to the best of our knowledge.

The previous ACM/IEEE Computer Science Curricula (CSC) published in 2013[1] listed storage and VM with the same level of expected outcome in the same section, but storage was not introduced at all for most course offerings, whereas VM enjoyed more attention, often as a main topic, with lectures devoted to the topic. Looking forward, storage is listed as a *CS Core* topic for CompOrg in the ACM/IEEE/AAAI CSC 2023 (released early 2024)[2]. This means future computer science students *must* know storage. However, CompOrg is the only course where storage is considered a CS Core topic; storage is also listed in the Operating Systems section, but

[1]VM indicates **virtual memory** (not virtual machine) in this paper.

Table 1: A review on CompOrg courses. In the *Stor?* column, × indicates no storage-related topic is provided. — indicates the course mentions file I/O but is not directly related to secondary storage devices. ✓ indicates there are discussions about secondary storage devices.

Institution	Course No.	Course Name	Term	Textbook	Stor?	Some Topics Covered besides Storage
Boston U	CS 472	CompArch	FA 23	[6]	×	binary, ASM, pipelining, parallelism, VM, digital logic
CalTech	CS 24	Intro to CompSys	FA 22	[3], [4]	×	binary, ASM, process, malloc, VM, threads
Columbia	CSEE 3827	Fund. of CompSys	SU 23	[7]–[9]	×	logic, FSM, ASM, pipelining, cache
Cornell	CS 3410	CompSys Org. & Prog.	SP 19	[10]	✓	logic gates, FSM, exception, concurrency, pipelining, VM, I/O
CMU	15-213	Intro to CompSys	SU 23	[11]	—	binary, ASM, malloc, VM, linking, exception, I/O, network, concurrency
FIU	CDA 3102	CompArch	FA 19	[7]	✓	binary, digital logic, ASM, pipelining, I/O
GA Tech	CS 2200	Systems & Networks	SP 16	[12]	—	ASM, control unit, pipelining, VM, parallelism, I/O, filesystems, network
Harvard	CS 61	SysProg & Machine Org.	FA 22	[11]	✓	binary, ASM, VM, mem hierarchy, concurrency
MIT	6.033	CompSys Eng.	SP 22	[13]	✓	VM, network, reliability, distributed systems, DB, security, virtual machine
NYU	CSCI-UA 0201	CompSys Org.	SP 23	[11]	×	binary, C, ASM, linker, processor, VM, malloc, concurrency
Ohio State	CSE 2421	System I	FA 23	[11]	×	C, I/O
Penn State	CMPSC 312	CompOrg & Arch.	SP 22	[14]	✓	binary, digital circuit, ASM, I/O, architectures, VM
Purdue	CS 252	Systems Programming	SP 24	[15]	—	C/C++, linker, bash, I/O, concurrency, SQL, socket, src ctrl
SD State	COMPE 271	CompOrg	FA 22	[16]	×	binary, ASM, C, pipelining, parallelism
Stanford	CS 107	CompOrg & Sys.	WI 23	[11]	×	binary, C, ASM, malloc, stack/heap, optimization
U of Rochester	CSC 252	CompOrg	FA 22	[11]	—	binary, ASM, pipelining, I/O, VM, concurrency, parallelism, syscall
UCB	CS 61C	Great Ideas in CompArch	SU 23	[10]	×	binary, C, ASM, FSM, logic, VM, concurrency, compiler
UCI	ICS 10	How Computers Work	SP 18	[17]	✓	binary, network, concurrency, ASM, HCI
UCLA	CS 33	Intro to CompOrg	FA 22	[11]	×	binary, ASM, compiler, exception, VM, concurrency, linking
UCSB	CS 64	CompOrg & Dig. Logic	WI 20	[7]	×	binary, ASM, digital logic, FSM, logic
UCSD	CSE 30	Intro to CompSys	WI 23	[4]	×	binary, C, ASM, C/ASM security
UMD	CMSC 216	Intro to CompSys	SU 23	[11]	—	C, malloc, binary, ASM, concurrency, I/O, signals
UMich	EECS 370	Intro To CompOrg	WI 23	[18]	×	C, binary, ASM, linking, FSM, digital logic, pipelining, VM
UPenn	CIS 2400	Intro to CompSys	FA 22	[4], [19]	—	binary, logic, ASM, processor, I/O, C, malloc
UT Austin	CS 429	CompOrg & Arch.	FA 23	[11], [18]	✓	C, binary, ASM, digital logic, malloc, pipelining, linking, I/O, DMA
UW Madison	CS 354	Intro to CompSys	SP 19	[11]	×	C, ASM, malloc, VM, concurrency
UWash	CSE 351	The HW/SW Interface	SU 23	[11]	×	binary, C, ASM, VM, malloc, process
Vassar	CMPU 224	CompOrg	FA 22	[11]	×	binary, ASM, logic, pipelining, malloc, cache
WUSTL	CSE 361S	Intro to Sys. Software	FA 23	[11]	×	binary, ASM, malloc, linking, VM, exception, signal

as a *KA Core* (recommended) topic. Furthermore, A recent news report shows that more high school students, including those who are taking CS courses, do not even understand the notion of *file* and *folder*; they have no idea of storage devices, while the older generation of people with exposure to hard disk and floppy find these concepts intuitive[20]. Students only know that their *data* are stored somewhere, but they do not know where and how their *data* are stored. This further shows the necessity of integrating the storage topic into CompOrg courses, which, again, is the only course students are guaranteed to learn about storage as defined by the latest CSC.

To bring students' attention to storage, we decided to add storage topics to our CompOrg course. We add six hours of lectures on storage-related topics, including hard disk drives (HDD) and solid state drives (SSD). The first half focuses on the physical architecture of those storage devices, while the second half focuses on the SSD firmware components. To evaluate students' understanding of the new material, We designed a new programming assignment on a pedagogical SSD simulator we developed. The assignment enables students to learn how SSDs work by implementing the main component, flash translation layer (FTL), of the SSD firmware, which allows them to understand how SSDs handle host I/O requests internally. We also design a new written assignment, StorageWrit, for students to learn the latency calculation of storage devices. Together, these assignments cover the most important aspects of storage devices.

To summarize, our main contributions are:

- SSDLab, a programming assignment for students to implement a flash translation layer, which is a vital part of SSD firmware, on a pedagogical SSD simulator.
- StorageWrit, a written assignment for students to learn how HDD and SSD latencies are calculated.
- A detailed analysis of students' performance based on their assignment grades and exam outcomes.

Background and Related Work

Storage in CompEd: Though storage device exists in virtually every single computational device, the storage topic remains silent in the CompEd field. This is as true in 2025 as it was described by Desnoyers in 2011[21]. Our survey shows no papers focused on storage devices in CompOrg courses in ASEE, SIGCSE, and ICER conferences in the past 10 years. Our search with Google Scholar does not show any research on the effectiveness of using SSD simulators to teach SSD internals within the scope of CompOrg courses. Meanwhile, there are prior works covering most other CompOrg topics, including virtual memory[22], [23], security[5], [24], [25], memory management[26], [27], instruction set architecture[28], GPU[29], and textbook design[30]. This further shows storage is indeed a missing piece from CompOrg courses and CompEd in general.

Modern Storage Devices: HDD and SSD: HDD and SSD dominate the modern storage field. HDD is mechanical and stores information on magnetic disk platters. A disk head moves around to cooperate with rotating disk platters for data access. When overwriting data, the head can simply overwrite in place at the original physical location as long as it is not damaged (e.g., scratched)[31]. Each I/O request requires a time-consuming disk head seek. This means random

requests are significantly slower than sequential requests for HDDs[3]. Due to the movement of mechanical parts, HDD is considerably slower than SSD, which uses NAND flash chips to store data without any mechanical parts[3]. However, NAND flash chips do not support in-place updates like HDD platters. Therefore, SSDs require the flash translation layer, or FTL, in firmware to create an illusion for supporting in-place updates from the host perspective during data overwrites[32].

The responsibility of the FTL is two-fold: The first responsibility is to handle I/O requests by translating logical addresses from the OS to physical addresses on flash chips. The smallest unit for data writes is a page, whereas the smallest erase unit is an erase block, commonly containing hundreds of pages[33]. When handling a `WRITE` request to a logical page, the FTL has to assign the next free physical page in the current erase block[3]. The logical-to-physical (L2P) mapping table keeps the mapping. Overwrites and file deletions will invalidate the old mapping and the old physical page due to the out-of-place write behavior[34]. When an erase block is full, the FTL chooses a free erase block for future writes[35].

The second responsibility of the FTL is garbage collection (GC). Since the smallest erase unit is an erase block, when the number of free erase blocks falls under a certain threshold, the FTL has to choose a victim erase block to vacate, which has at least one invalid physical page. The L2P mapping will be updated when valid pages in the victim are relocated, and a physical-to-logical (P2L) mapping table is required to determine the logical page number of the to-be-relocated physical page from the perspective of the SSD controller[34]. The FTL then erases the victim, which is a lengthy process compared to normal read/writes, and finally marks the victim as free for future use[33]. Together, these two responsibilities create the illusion for the host that SSDs support in-place updates.

In summary, SSDs are faster than HDDs for both sequential and random writes due to the removal of mechanical parts[4]. However, SSDs require a more complex firmware than HDDs due to the hardware peculiarities, and the GC process could temporarily affect SSD performance. As for HDDs, they serve sequential requests faster than random ones because the head does not need to seek locations for each request[3]. We believe that it is important for students to learn these to write faster applications, and it is even more crucial if they are writing data-intensive or system applications.

Existing SSD Simulators/Emulators:

Since SSD is a computer device with specialized hardware, several SSD simulators and emulators exist to expedite academic research and prototype developments by overcoming the difficulty of implementing SSD firmware on real hardware. Table 2 shows a list of existing SSD simulators/emulators, where most are commonly used by the storage field. These tools, designed to mimic real SSDs as closely as possible, are often huge. They aim to provide realistic latency models for research and are well-acclaimed tools in the storage field. Many have been used for numerous SSD-related works throughout the years[33], [35], [45]–[48].

However, we believe these tools are not suitable for pedagogical purposes in our course because they are too complex for students. These tools are for experienced researchers in the storage field who are more familiar with systems development. Our students, on the other hand, first learned C in this course. They also know nothing about systems programming at this moment. Full-system

Table 2: A survey on existing SSD simulators and emulators. S denotes simulator, and E denotes emulator in the S/E column. Lang column shows the main development language.

Name	S/E	Lang	Side Notes
DiskSim[36]	S	C	Extended for SSD from the original work[37], which has an 87-page manual.
FlashSim[38]	S	C	Also extended from DiskSim.
SSDSim[39]	S	C	About 15,000 lines of code (LoC).
FTLSim[40]	S	C, Py 2	About 1,000 LoC (~700 lines of C and ~300 lines of Python 2).
MQSim[41]	S	C++	About 13,000 LoC.
Amber[42]	S	C++	Built on full-system emulator gem5.
OSTEP[3]	S	Py 3	Pedagogical simulator in Python 3 from the OSTEP textbook. About 500 LoC.
VSSIM[43]	E	C	Built on full-system emulator QEMU.
FEMU[34]	E	C	Built on full-system emulator QEMU.
NVMeVirt[44]	E	C	Kernel module, adopted from FEMU.

emulators are too difficult for students at this stage. Similarly, other large simulators with a codebase of tens of thousands of lines or a nearly 100-page manual are too large for students to handle. Two existing simulators are relatively feasible compared to others: The first is FTLSim, but it was released in 2012 and written in C and Python 2, the latter of which is now obsolete[49]. The other possible choice is the SSD simulator from the OSTEP textbook commonly used in operating system courses, but it is unsuitable for our course due to several reasons. The largest concern is that it does not come with a grading system, limiting its application in a pedagogical environment. On the other hand, the OSTEP SSD simulator does not take real or handcrafted traces; it only accepts synthetic workloads generated by the simulator itself. This limits the ability of students to craft their own traces to debug their code. Furthermore, it is written in Python 3, but our course focuses on C as the programming language. This means all existing SSD simulators and emulators are unsuitable for our needs.

Therefore, we have to create a new pedagogical SSD simulator for this course. The simulator should be in user space and have an appropriate difficulty so the students can learn SSD internals without putting too much effort into debugging[23]. Instead of focusing on data structures and other implementation details, the simulator should abstract SSD features into intuitive helper functions, and the students should be able to assemble their FTL using these functions. The students should not need to read the whole code base to learn how to use the simulator; they should be able to easily do so by reading the short list of helper functions and their descriptions, and they should be able to write their C code as simple as pseudocode. The simulator should also be able to perform autograding for the ease of the instructors. We keep these requirements in mind when developing our simulator and assignment.

Design

SSDLab: SSDLab is the programming assignment for the storage topic in our course. It is based on the pedagogical SSD simulator we developed since existing SSD simulators/emulators are for academic and industrial use instead of pedagogical as discussed in § . We provide predefined data structures, including the logical-to-physical (L2P) table, the physical-to-logical (P2L) table, and table entries, to the students. These two tables will be initialized with empty entries automatically

Table 3: SSDLab function signatures. Table contents are from the assignment handout.

Helper Functions (Signature)	**Description**
`bool isFrontierBlockFull()`	Checks if the current frontier block is full. Returns `true` if it is full and `false` otherwise.
`int getAvailPPN()`	Gets the next available physical page number (PPN) from the current open block. Returns a page number. Returns -1 if there is no free page available.
`bool isValidLPNMapping(int lpn)`	Returns `true` if this logical page number (LPN) is in use (having a valid mapping to a physical page number in the mapping table). Returns `false` otherwise.
`int getL2PMapping(int lpn)`	Returns the PPN given the LPN.
`void setL2PMapping(int lpn, int ppn)`	Sets the logical to physical page number mapping if PPN ≥ 0. Invalidates the mapping table entry for this LPN if PPN is -1.
`bool isValidPPNMapping(int ppn)`	Returns `true` if this PPN is in use (having a valid mapping to a logical page number in the reverse mapping table). Returns `false` otherwise.
`int getP2LMapping(int ppn)`	Returns the LPN given the PPN.
`void setP2LMapping(int lpn, int ppn)`	Sets the physical to logical page number mapping in the reverse mapping table, then marks the NAND page as valid if LPN ≥ 0. Invalidates the reverse mapping table entry for this LPN, and sets the NAND page given the PPN if LPN is -1.
`void useNextFreeBlock()`	Sets the frontier block to the next available free block. Returns `false` if there are no available free blocks; returns `true` if the operation is successful.
`void markBlockFree(int block)`	Erases the block; marks the given NAND block as free and all NAND pages inside the block as free for future use when a new block is needed.
`int ppnFromBlockPageNum(int block, int page)`	Returns the physical page number by giving the block number and the page number inside the block.
`int getBlockStatus(int block)`	Check the block status. Returns the block status using an `enum` value described below. • `BLOCK_FREE`: indicates the block is free. • `BLOCK_FULL`: indicates the block is full, and data in the block cannot be updated. • `BLOCK_FRONTIER`: indicates the block is currently the frontier block.
`bool shouldGC()`	Returns true if the number of remaining free blocks falls below a certain percentage. You should call your GC function when the return value is true.
`int countBlockInvalidPages(int block)`	Returns the number of invalid NAND pages in the given NAND block.
Functions to Implement (Signature)	**Description**
`int findVictim()`	Finds a victim block and returns the victim block number. If there are no eligible victim blocks (e.g., there are no full blocks, or all pages in all full blocks are valid), return -1.
`void gc(int victimBlock)`	Do GC on a given victim block.
`void handleWriteRequests(int addr, int size)`	handles the mapping table lookup by utilizing functions above based on the memory request given as arguments.

without student intervention. We also provide helper functions (Table 3) so students can implement the following three functions in a total of 50 to 60 lines of code.

Function `handleWriteRequests()`: SSDLab only accepts write requests since read requests do not change any mapping information. When calling `handleWriteRequests()` to handle a host write request to a logical page, the SSD FTL should assign a physical page using `getAvailPPN()` if the frontier (i.e., currently being used) erase block is not full, which can be checked by `isFrontierBlockFull()`; a new, free frontier block should be used by `useNextFreeBlock()` if the current one is full. The FTL then sets the new L2P and P2L mappings by using L2P/P2L setter functions with the new physical page number and invalidates the P2L mapping of the old physical page if the request overwrites an existing page; this is achievable by checking `isValidLPNMapping()` and `getL2PMapping()`.

Subsequent functions `gc()` and `findVictim()`: Garbage collection, `gc()`, should be triggered if the number of free erase block falls under a certain watermark[3], which can be checked by the `shouldGC()` function. `gc()` chooses an erase block using a victim selection algorithm, commonly greedy[40], defined by `findVictim()`. The greedy algorithm chooses the full erase blocks with the largest number of invalidated pages by checking `countBlockInvalidPages()`. All valid physical pages in the victim should be copied to the current write frontier, which is similar to the algorithm of handling normal host writes, but requires checking the P2L table for the logical page numbers of the physical pages using `getP2LMapping()`. The victim erase block will be erased using `markBlockFree()`.

Debugging & Grading: To help students debug, we provide compiled object files of sample `findVictim()` and `gc()` answers. Students can check their implementation correctness using the autograder after implementing `handleWriteRequests()` but before implementing their own version of those two functions using these object files. The autograder can automatically compile and grade with pre-compiled and/or student-written `findVictim()`/`gc()`, so students can debug accordingly by knowing which of the three functions caused the error. A compiled executable with the sample answer of all three functions is also provided, which the autograder uses as a reference to grade a student's implementation as a whole. We adopt real storage device traces as workloads for grading purposes from the work of Lee, et al. published at SYSTOR[50] and process them for the 1GiB SSDLab logical address space. We also provide a short trace with around 100 write requests for students to get started. Students are also encouraged to craft their own small traces to build their code ground-up and to debug their code in a controllable manner.

80% of the grade is awarded for overall correctness: We check if all translations from P2L→L2P and L2P→P2L match within a student's implementation since different victim selection algorithms and GC trigger policies may lead to different yet valid mapping table results. A mismatch indicates data corruption. We also check if the number of student mapping table entries matches with the sample executable to prevent manipulation and cheating. The rest 20% of the grade is for code efficiency. A good victim selection algorithm and GC trigger policy can lead to

lesser write amplification, which is measured by write amplification factor (WAF)[2]:

$$\text{WAF} = \frac{\text{Host Write Pages + GC Write Pages}}{\text{Host Write Pages}}$$

We use WAF as the efficiency metric and consider the student's implementation with the best WAF as 100% credit for this part. The grades for other students are linearly proportional to the best WAF. We also created an automatic grading system to collect and grade all student implementations based on the above grading rules.

StorageWrit: StorageWrit is a written assignment for students to calculate HDD and SSD latency under different workloads. Although HDD and SSD are both storage devices, they have very different underlying physical designs. HDD suffers from random workloads due to the repositioning of the disk head for each random I/O request but performs better under sequential workloads since the head only needs to be repositioned once per request. SSD, on the other hand, has better performance for both workloads, but it is still slower with random workloads due to extra metadata overheads[3]. The StorageWrit is aligned to these properties of common storage devices: Students are required to calculate the HDD/SSD device latency under different workloads given a device performance model.

The details of the assignment is as follows: For HDD, we ask for the time needed to fetch a 4MiB file, given the HDD rotational rate, average seek time, and average capacity per track. The question contains two different cases: the file being fully sequential (i.e., no need to seek every time) and fully scattered on the disk (i.e., need to seek every time after reading a sector). For SSD, we ask for the time needed to fetch a 4MiB file, given the SSD characteristics like the bandwidth per chip, the number of chips working in parallel, flash memory page size, and flash memory page latency. The question is also split into two cases where the file is sequentially or randomly placed on SSD.

Implementation

Assignment Handout: For SSDLab, we provide a detailed handout covering the goal of the programming assignment and all the information students may need. Students should be able to complete their programming assignments without attending lab sessions by reading the handout. We also strive to encourage students to read the handout by providing background stories. For example, the SSDLab background is set at a fictional semiconductor company in the 2000s, which is historically correct[51]. StorageWrit, on the other hand, is distributed as a part of a usual biweekly written assignment.

Lecture, Lab Session, and Office Hours: There are three sessions per week for our CompOrg course: two 1.5-hour lectures, in which we focus on the new knowledge points led by the Professor, and a 1-hour lab session in which students work on their programming assignments led by the TA. Besides lectures and lab sessions, we also provide at least one hour of office hours every workday, which translates to at least five hours of office hours per week. We chose to devote two weeks to storage-related materials because we found most other course offerings

[2]The only write amplification source in SSDLab is GC, so we only model GC-induced write amplification in SSDLab. Real SSDs may have other sources of write amplification like wear leveling[3], [46].

shown in Table 1 provide none or only a few lines of information for storage. The first week is for HDD/SSD physical internals, which is related to the latency calculation. The second week is for SSD and FTL details. Our planned student workload per programming assignment is 50 to 60 lines of code in a duration of two weeks. Thus, we also provide two weeks of lab sessions to discuss the programming assignment.

In the labs, we review the overall structure of the topics (i.e., how FTL handles requests) and make sure students understand how to implement the assignments from a high level by asking and answering questions about the key steps to implement the programming assignments. These questions and answers are also tied to the function signatures we provide in Table 3: we do not tell exactly which function to use when answering each question, but we describe the steps in human language as close to the function signatures as possible. We also introduce the autograders during lab sessions so the students can debug better with the autograder.

During office hours, students usually ask for answers to specific questions, including why their implementations cannot compile, why their implementation is partially incorrect, how to run the autograder, and how to improve the WAF for a better grade. We provide more tailored instructions during office hours than during lectures and lab sessions. The usual approach is to guide them to read the manual (i.e., the handout) so they will understand the expectations and autograder usage better. For questions related to the code, we first discuss the expected behavior as we do in the lab sessions and let the students discover the problem. If the student is stuck, we will provide more explicit hints (e.g., which exact step they should be working on, what does the step do, etc.) so the student can continue their work. We also teach them to use the autograder wisely so that they can focus on one function at a time by using the pre-compiled version of `findVictim()` and `gc()` when implementing their `handleWriteRequests()`.

To make way for the storage topic, we removed the CPU pipelining topic from our course. The CPU pipelining topic is covered by the Computer Architecture course in our department's curriculum. On the other hand, the Operating Systems course in our department's curriculum does not include the topic of storage. Again, CSC2023 suggests covering the topic of storage in either the CompOrg course or the OS course[2]. By implementing storage in our course, we now cover the missing storage topic in the department's curriculum. Lastly, we want to emphasize again that storage is considered mandatory in CompOrg while considered optional in OS as suggested by CSC2023; thus, it is more favorable to implement the storage topic in our course than in the OS course.

Evaluation

In this section, we evaluate our effort of integrating storage into our CompOrg course at Syracuse University, NY, USA, during the Fall 2023 semester. A total of $N = 37$ students are enrolled in this course at the end of the semester. We will analyze our efforts from different perspectives, including student performance and feedback.

Assignment Analysis: When designing the SSDLab, we expect students to finish it by writing 50-60 lines of code (LoC) in two weeks, and we prove the feasibility by writing a sample implementation with LoC within that range. Figure 2a shows the students' LoC distribution for

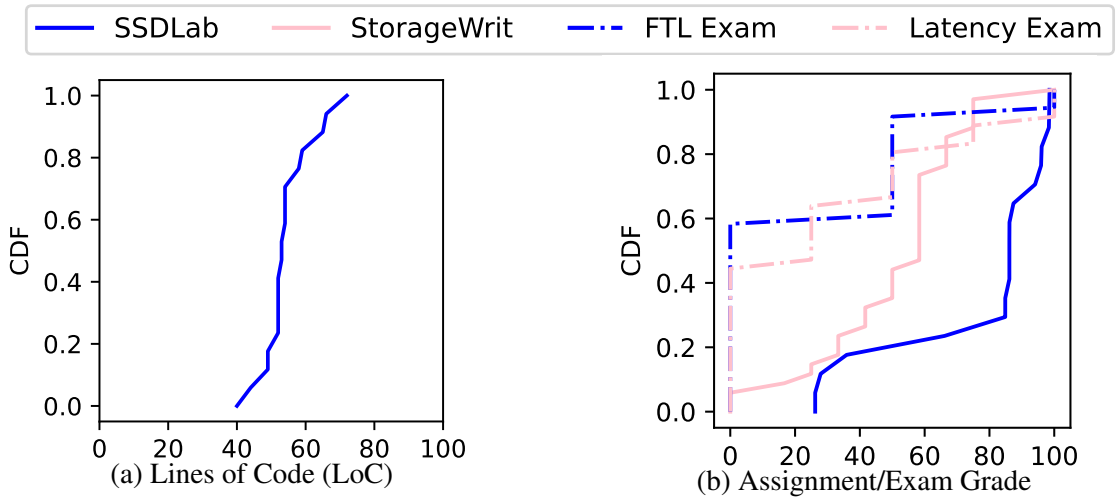

Figure 2: CDF of LoC and grades. Note: exam grade CDFs contain grades of empty responses, whereas assignment grade CDFs only contain the grades of those who submitted.

the 18 students who finish SSDLab[3,4]. All students are able to finish SSDLab between 40-72 LoC, i.e., ± 12 LoC around our expectation.

Figure 2b shows the assignment grades (solid lines) for SSDLab (blue) and StorageWrit (pink). SSDLab has a strict grading rule: Any incorrect mapping table entry results in a 50% penalty. About 70% of the students implement the mapping tables correctly. For those who get more than 80% of the score, the distribution can be categorized into two clusters in the CDF graph. The first cluster receives around 85% of the total score, whereas the second is around 95%. This is caused by different GC trigger conditions, which results in different WAFs. Those who receive around 85% trigger GC whenever the number of available erase blocks falls under a specific range, even if the victim has only a few invalid pages; all valid pages in the victim have to be relocated to another erase block, causing high write amplification. Modern SSDs have enough over-provisioned space up to 28%[52], which means SSDs have more physical space than logical capacity, and GC can be delayed if the victim only has a few invalid pages. A widely-used SSD emulator, FEMU[34], skips GC when less than 1/8 of the total pages in the victim are invalid[53]. Those who receive around 95% employ similar lazy GC techniques.

The grades for StorageWrit are more uniformly distributed compared to SSDLab. The overall grade for StorageWrit is lower than SSDLab because we only provided examples to calculate HDD and SSD latencies in StorageWrit instead of explicitly discussing the topic step-by-step in our lecture; the final grade CDF shows that we should discuss latency calculation in detail in future course offerings. The students are also not able to confirm their answer correctness automatically for StorageWrit, whereas the programming assignment autograder allows them to check their implementation correctness easily. We discussed the solution in detail during the lecture after the grades of StorageWrit were released as remediation.

[3]This does not include a student's implementation since it has only one line of code.

[4]The submission rate for SSDLab is about 50% because we announced before SSDLab, the last programming assignment (PA), that the lowest PA grade would be dropped.

Exam Results: The final exam contains two questions, one corresponding to SSDLab (mimic FTL to handle `WRITE` and `GC`) and one to StorageWrit (calculate latency). Figure 2b shows the exam question score CDFs in dashdotted lines. Interestingly, although students perform well overall on SSDLab, their exam performance is lower than expected. Only 40% of students receive points for the FTL question in the final exam. Although the course rule of skipping the lowest programming assignment score causes a low submission rate of the SSDLab, it inadvertently creates a perfect control group to compare the exam performance between those who submit (18 students, 48.6%) and skip (19 students, 51.4%) SSDLab. The average FTL question grade for all students is 29% lower than the overall final exam grade average. However, we see higher FTL question grades for those who finished SSDLab. The average FTL question score of the 18 students who submitted SSDLab is 19% lower than the overall final exam average, while the 19 students who skipped show 39% lower. We also observe a Pearson Correlation Coefficient (PCC) of 0.55 between the FTL question and SSDLab, further proving the help of SSDLab on understanding SSD internals. We further find a significant correlation between SSDLab and FTL question grades for those who submitted SSDLab using ordinary least squares (OLS) regression, yielding a coefficient of 0.720, t-value of 2.631, and p-value of 0.018 (≤ 0.05), showing the effectiveness of SSDLab.

There are also gaps between assignments and the exam. The exam question asks the students to manually place the data during incoming write requests and perform GC under given conditions. In SSDLab, students implement the FTL without trying to place/relocate data manually. They may understand the high-level structure by utilizing the helper functions, but they may fail to follow every detailed step in the exam question. To reduce the gap between exam questions and homework, the instructor should provide more written practice like StorageWrit, which shows a better CDF in the final exam question despite students performing worse than the SSDLab, as StorageWrit provides practice on paper, similar to its corresponding exam question.

Last but not least, in this semester's final exam, we introduce an optional make-up midterm for students to have a second chance for a better midterm grade. However, we find that the make-up midterm negatively impacts the performance of the FTL question with a PCC of -0.36 between the make-up midterm and the FTL question grade, showing that students may have to choose between the FTL question and the make-up midterm due to time constraints. The FTL question score of the students who skip SSDLab shows an even stronger negative correlation with the make-up midterm, resulting in a PCC of -0.56, showing that they are more likely to bet on the make-up midterm for a higher overall course grade. We also perform an OLS regression using the grade of the FTL question as y, the grade of SSDLab as x_1, and the grade of the make-up midterm as x_2. We see a coefficient of -0.5151 for the make-up midterm grade with a t of -2.430 and p of 0.021 (≤ 0.05), showing a significant negative correlation between the grade of the FTL question and the make-up midterm, which further proves the side effects caused by the make-up midterm.

Student Feedback: We distribute surveys for student feedback on the programming assignments after the IRB request is approved and the assignment grades are out. Student feedback is mostly positive on SSDLab. They find hands-on assignments interesting and enjoy the implementation process, although SSDLab is considered the most difficult programming assignment in the course. A student mentioned, "[SSDLab is the m]ost challenging project all year, but never felt unobtainable." Background stories in the assignment handouts lighten students' moods and

motivate them to finish reading the handout in detail. Students thank the autograder and object files for their help in debugging. The comments in the project template help students learn and think about the steps required in the assignment. The code of the helper functions, along with their comments, also shows their value in helping students to succeed. One student wrote, "For SSDLab, I can also say that students should absolutely read the code comments and the assignment description before/while writing the functions. They basically tell you directly what the functions need to do and the steps involved to make them do that." After the assignment, students feel the assignment helped them to understand the topic; as one student wrote, "Go to office hours for it, the solution isn't that complicated, but getting to the solution requires a solid understanding."

The students also provide potential future improvements. The most valuable suggestion is to talk about the lazy garbage collection technique for SSDLab explicitly; we send an email to students regarding lazy GC and discuss this during the lab session, but we never explicitly discuss it during the lecture. Students also ask for more resources and tips on debugging, which is reasonable because the students only have two programming assignments in C before SSDLab. C is famous for its steep learning curve, and we should provide more help with programming and debugging.

Potential Improvements

From the comparison between the assignment grades and final question grades, we believe the integration of storage into our CompOrg course is mostly successful, but we should improve the integration with more possibilities for SSDLab and better course organization to further reflect student achievements in the exam as well as the assignments.

Modern SSDs are far more complex than the model we provide in class. A real SSD has multiple flash chips that work in parallel for faster throughput[33], but our simulator and assignment only consider the simplest SSD without any internal parallelism because we think this is too complex for students to understand and implement within a short time period. Data consistency and integrity should also be covered since they are the requirements of a storage device, but unfortunately, we did not have the time to cover them in our course; we will consider discussing them at a high level (e.g., the definition and the necessity) in the future. We should also integrate SSDLab with other related topics in the course (e.g. error correction code) to create connections between different topics. Meanwhile, more types of SSDs are now available on the market. For example, traditional SSD purely depends on the decision of the FTL when selecting a physical page to write to, but some recent SSD types allow host-SSD coordination on data placement to reduce write amplification[32], [52]. Due to the student workload we have expected (50-60 LoC in two weeks), we did not include the scopes above because it will be too much burden for the students. However, we hope to provide these features in the future so the instructors may have a more diverse range of possible assignments for the students and the students have more flexibility and design choices during implementation.

The implementation of course rules may have repercussions. We identify that announcing the lowest grade would be dropped before the last programming assignment, SSDLab, is very likely the cause of its low submission rate. The make-up midterm during the final exam further

amplifies the negative effect and causes the low overall average score on the FTL question, as students prefer abandoning the question for the make-up exam. We will be more cautious in the future when carrying course rules with new topics.

Conclusion

In this paper, we show the gap between the high importance of the storage topic and its low adoption rate in CompOrg courses. To help solve this problem, we integrate storage topics in our CompOrg course with lectures, SSDLab, and StorageWrit. However, we still see room for future improvements, which we will address in later course offerings. Our pilot experience shows that integrating storage into CompOrg courses requires non-trivial efforts and cautious designs. Although the SSDLab programming assignment is harder than all other programming assignments in the course, as students reported, students mentioned they have learned a great amount from the SSDLab. Students who finished SSDLab also showed a 32% higher score in the corresponding final exam question. We hope this paper sets a foundation and encourages future research on integrating storage in CompOrg courses.

Acknowledgement

This research was supported in part by the National Science Foundation (CNS-2008453) and has been approved for IRB exemption by Syracuse University (IRB #23-394).

References

[1] The Joint Task Force on Computing Curricula ACM & IEEE Computer Society, *Computer science curricula 2013*, `https://ieeecs-media.computer.org/assets/pdf/CS2013-final-report.pdf`, (Accessed on 02/08/2024), Dec. 2013.

[2] A. N. Kumar *et al.*, *Computer Science Curricula 2023*. New York, NY: ACM, 2024.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00. Arpaci-Dusseau Books, Aug. 2018.

[4] S. J. Matthews, T. Newhall, and K. C. Webb, *Dive into systems*, `https://diveintosystems.org/`, (Accessed on 01/31/2024), 2019.

[5] M. Almansoori, J. Lam, E. Fang, K. Mulligan, A. G. Soosai Raj, and R. Chatterjee, "How secure are our computer systems courses?" In *2020 ACM Conference on International Computing Education Research*, ser. ICER '20, New York, NY: ACM, 2020, pp. 271–281. [Online]. Available: `https://doi.org/10.1145/3372782.3406266`.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition* (The Morgan Kaufmann Series in Computer Architecture and Design), 6th ed. Oxford, England: Morgan Kaufmann, Dec. 2020.

[7] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition* (The Morgan Kaufmann Series in Computer Architecture and Design), en, 5th ed. Oxford, England: Morgan Kaufmann, Sep. 2013.
[8] M. Morris Mano, C. R. Kime, and T. Martin, *Logic & Computer Design Fundamentals*, 5th ed. Upper Saddle River, NJ: Pearson, Mar. 2015.
[9] D. Harris and S. Harris, *Digital Design and Computer Architecture*. Elsevier, 2013.
[10] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The hardware software interface*, en. Oxford, England: Morgan Kaufmann, May 2017.
[11] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Pearson, 2016.
[12] U. Ramachandran and W. D. Leahy, *Computer Systems: An Integrated Approach to Architecture and Operating Systems*. Upper Saddle River, NJ: Pearson, Jul. 2010.
[13] J. H. Saltzer and F. Kaashoek, *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.
[14] L. Null, *Essentials of Computer Organization and Architecture*, en, 5th ed. Sudbury, MA: Jones and Bartlett, Feb. 2018.
[15] G. A. J. Rodriguez-Rivera and J. Ennen, *Class notes*, `https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/`, (Accessed on 01/31/2024), Oct. 2014.
[16] F. Vahid and R. Lysecky, *Introduction to computer systems and assembly programming - zybooks*, `https://www.zybooks.com/catalog/computer-systems-and-assembly-programming/`, (Accessed on 01/31/2024).
[17] H. Abelson, K. Ledeen, H. Lewis, and W. Seltzer, *Blown to Bits*, en, 2nd ed. Boston, MA: Addison Wesley, Nov. 2019.
[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition* (The Morgan Kaufmann Series in Computer Architecture and Design). Oxford, England: Morgan Kaufmann, Mar. 2016.
[19] Y. Patt and S. Patel, *Introduction to Computing Systems: From Bits & Gates to C/C++ & Beyond*, 3rd ed. Columbus, OH: McGraw-Hill Education, Oct. 2019.
[20] M. Chin, *Kids who grew up with search engines could change STEM education forever - The Verge*, `https://www.theverge.com/22684730/students-file-folder-directory-structure-education-gen-z`, (Accessed on 01/31/2024), Sep. 2021.
[21] P. J. Desnoyers, "Teaching operating systems as how computers work," in *42nd Technical Symposium on Computer Science Education*, ser. SIGCSE '11, New York, NY: ACM, 2011, pp. 281–286. [Online]. Available: `https://doi.org/10.1145/1953163.1953249`.
[22] A. Qasem, "Yoda: A pedagogical tool for teaching systems concepts," in *53rd Technical Symposium on Computer Science Education - Volume 1*, ser. SIGCSE 2022, New York, NY: ACM, 2022, pp. 613–618. [Online]. Available: `https://doi.org/10.1145/3478431.3499322`.
[23] S. Silvestro, T. T. Yuen, C. Crosser, D. Zhu, T. Korkmaz, and T. Liu, "A user space-based project for practicing core memory management concepts," in *49th Technical Symposium on Computer Science Education*, ser. SIGCSE '18, New York, NY: ACM, 2018,

pp. 98–103. [Online]. Available: https://doi.org/10.1145/3159450.3159581.

[24] M. Almansoori, J. Lam, E. Fang, A. G. Soosai Raj, and R. Chatterjee, "Textbook underflow: Insufficient security discussions in textbooks used for computer systems courses," in *52nd Technical Symposium on Computer Science Education*, ser. SIGCSE '21, New York, NY: ACM, 2021, pp. 1212–1218. [Online]. Available: https://doi.org/10.1145/3408877.3432416.

[25] J. Walker, M. Wang, S. Carr, J. Mayo, and C.-K. Shene, "A system for visualizing the process address space in the context of teaching secure coding in c," in *51st Technical Symposium on Computer Science Education*, ser. SIGCSE '20, New York, NY: ACM, 2020, pp. 1033–1039. [Online]. Available: https://doi.org/10.1145/3328778.3366894.

[26] B. P. Railing and R. E. Bryant, "Implementing malloc: Students and systems programming," in *49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18, New York, NY: ACM, 2018, pp. 104–109. [Online]. Available: https://doi.org/10.1145/3159450.3159597.

[27] J. Clements and S. Krishnamurthi, "Towards a notional machine for runtime stacks and scope: When stacks don't stack up," in *2022 ACM Conference on International Computing Education Research - Volume 1*, ser. ICER '22, New York, NY: ACM, 2022, pp. 206–222. [Online]. Available: https://doi.org/10.1145/3501385.3543961.

[28] S. A. Zekany, J. Tan, J. A. Connelly, and R. G. Dreslinski, "RISC-V reward: Building out-of-order processors in a computer architecture design course with an open-source ISA," in *52nd Technical Symposium on Computer Science Education*, ser. SIGCSE '21, New York, NY: ACM, 2021, pp. 1096–1102. [Online]. Available: https://doi.org/10.1145/3408877.3432472.

[29] C. Corsi, R. Geist, and D. Lingerfelt, "A virtual graphics card for teaching device driver design," in *45th Technical Symposium on Computer Science Education*, ser. SIGCSE '14, New York, NY: ACM, 2014, pp. 555–560. [Online]. Available: https://doi.org/10.1145/2538862.2538895.

[30] S. J. Matthews, T. Newhall, and K. C. Webb, "Dive into systems: A free, online textbook for introducing computer systems," in *52nd Technical Symposium on Computer Science Education*, ser. SIGCSE '21, New York, NY: ACM, 2021, pp. 1110–1116. [Online]. Available: https://doi.org/10.1145/3408877.3432514.

[31] D. S. Clementsen and Z. He, "Vertical partitioning for flash and HDD database systems," *Journal of Systems and Software*, vol. 83, no. 11, pp. 2237–2250, 2010, ISSN: 0164-1212. [Online]. Available: https://doi.org/10.1016/j.jss.2010.06.044.

[32] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *6th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '14, Philadelphia, PA: USENIX Association, Jun. 2014. [Online]. Available: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang.

[33] S. Yan *et al.*, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," *ACM Trans. Storage*, vol. 13, no. 3, Oct. 2017, ISSN: 1553-3077. [Online]. Available: https://doi.org/10.1145/3121133.

[34] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The case of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies*, ser. FAST '18, USA: USENIX Association, 2018, pp. 83–90.

[35] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, "Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes," in *19th USENIX Conference on File and Storage Technologies*, ser. FAST '21, USENIX Association, Feb. 2021, pp. 187–202. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/zhou.

[36] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *2008 USENIX Annual Technical Conference*, ser. ATC '08, 2008.

[37] J. S. Bucy, G. R. Ganger, *et al.*, *The DiskSim simulation environment version 3.0 reference manual*. School of Computer Science, Carnegie Mellon University, 2003. [Online]. Available: https://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-CS-03-102.pdf.

[38] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," *SIGPLAN Not.*, vol. 44, no. 3, pp. 229–240, Mar. 2009, ISSN: 0362-1340. [Online]. Available: https://doi.org/10.1145/1508284.1508271.

[39] 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 *International Conference on Supercomputing*, ser. ICS '11, New York, NY: ACM, 2011, pp. 96–107. [Online]. Available: https://doi.org/10.1145/1995896.1995912.

[40] P. Desnoyers, "Analytic modeling of ssd write performance," in *5th Annual International Systems and Storage Conference*, ser. SYSTOR '12, New York, NY: ACM, 2012. [Online]. Available: https://doi.org/10.1145/2367589.2367603.

[41] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices," in *16th USENIX Conference on File and Storage Technologies*, ser. FAST '18, Oakland, CA: USENIX Association, Feb. 2018, pp. 49–66. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/tavakkol.

[42] D. Gouk *et al.*, "Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '18, IEEE, Oct. 2018, pp. 469–481. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00045.

[43] J. Yoo *et al.*, "VSSIM: Virtual machine based SSD simulator," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies*, ser. MSST '13, IEEE, May 2013, pp. 1–14. [Online]. Available: https://doi.org/10.1109/MSST.2013.6558443.

[44] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "NVMeVirt: A versatile software-defined virtual NVMe device," in *21st USENIX Conference on File and Storage Technologies*, ser. FAST '23, Santa Clara, CA: USENIX Association, Feb. 2023, pp. 379–394. [Online]. Available:

https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon.

[45] E. Lee, I. Son, and J.-S. Kim, "An efficient order-preserving recovery for F2FS with ZNS SSD," in *15th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '23, New York, NY: ACM, 2023, pp. 116–122. [Online]. Available: https://doi.org/10.1145/3599691.3603416.

[46] Z. Jiao, J. Bhimani, and B. S. Kim, "Wear leveling in SSDs considered harmful," in *14th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '22, New York, NY: ACM, 2022, pp. 72–78. [Online]. Available: https://doi.org/10.1145/3538643.3539750.

[47] X. Zhang, S. Pei, J. Choi, and B. S. Kim, "Excessive SSD-internal parallelism considered harmful," in *15th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '23, New York, NY: ACM, 2023, pp. 65–72. [Online]. Available: https://doi.org/10.1145/3599691.3603412.

[48] Z. Jiao, X. Zhang, H. Shin, J. Choi, and B. S. Kim, "The design and implementation of a capacity-variant storage system," in *22nd USENIX Conference on File and Storage Technologies*, ser. FAST '24, Santa Clara, CA: USENIX Association, Feb. 2024, pp. 159–176. [Online]. Available: https://www.usenix.org/conference/fast24/presentation/jiao.

[49] Python Software Foundation, *Sunsetting Python 2 | python.org*, https://www.python.org/doc/sunset-python-2/, (Accessed on 02/09/2024), Jan. 2020.

[50] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *10th ACM International Systems and Storage Conference*, ser. SYSTOR '17, New York, NY: ACM, 2017. [Online]. Available: https://doi.org/10.1145/3078468.3078479.

[51] Samsung, *Leading the transition from HDDs to SDDs | Samsung semiconductor global*, [Online; accessed 2025-01-11], Sep. 2022. [Online]. Available: https://semiconductor.samsung.com/consumer-storage/world-no1-flash-memory/episode2/.

[52] M. Bjørling *et al.*, "ZNS: Avoiding the block interface tax for flash-based SSDs," in *2021 USENIX Annual Technical Conference*, ser. ATC '21, USENIX Association, Jul. 2021, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/bjorling.

[53] H. Li, *Femu/hw/femu/bbssd/ftl.c at master · vtess/femu*, https://github.com/vtess/FEMU/blob/master/hw/femu/bbssd/ftl.c, (Accessed on 03/03/2024).