# VST: A Virtual Stress Testing Framework for Discovering Bugs in SSD Flash-Translation Layers

Ren-Shuo Liu
*Department of Electrical Engineering*
*National Tsing Hua University*
Hsinchu, Taiwan
renshuo@ee.nthu.edu.tw

Yun-Sheng Chang
*Department of Electrical Engineering*
*National Tsing Hua University*
Hsinchu, Taiwan
yschang@gapp.nthu.edu.tw

Chih-Wen Hung
*Department of Electrical Engineering*
*National Tsing Hua University*
Hsinchu, Taiwan
s102061630@m102.nthu.edu.tw

***Abstract*—Flash translation layers (FTLs) are the core embedded software (also known as firmware) of NAND flash-based solid-state drives (SSDs). The relentless pursuit of high-performance SSDs renders FTLs increasingly complex and intricate. Therefore, testing and validating FTLs are crucial and challenging tasks. Directly testing and validating FTLs on SSD hardware are common practices though, they are time-consuming and cumbersome because 1) the testing speed is limited by the hardware speed of SSDs and 2) just reproducing bugs can be challenging, let alone locating and root causing the bugs.**

**This work presents virtual stress testing (VST), a simulation framework to enable executing SSD FTLs on PCs or servers against virtual SRAM, DRAM, and flash emulated by host-side main memory. FTL function calls, such as moving data from flash to DRAM, are served by the VST framework. Therefore, VST can test FTLs without SSD hardware requirements nor SSD speed limitations, and root causing bugs becomes manageable tasks. We apply VST to representative SSD design, OpenSSD, which is actively utilized and maintained by SSD and FTL communities. Experimental results show that VST can test FTLs at a speed up to 375 GB/s, which is several hundred times faster than directly testing FTLs on SSD hardware. Moreover, we successfully discover seven new FTL bugs in the OpenSSD design using VST, which is a solid evidence of VST's bug-discovering effectiveness.**

***Index Terms*—Embedded software, software testing, software debugging, systems simulation, data storage systems, disk drives, flash memories**

## I. Introduction

Flash translation layers (FTLs) are the core embedded software (also known as firmware) of NAND flash-based solid-state drives (SSDs). Because SSDs are designed to expose to a host computer logical address space that accepts reads and writes like hard disk drives (HDDs) do, FTLs are responsible for relocating flash data, erasing outdated flash data, and calculating flash addresses for every host read or write request. Academia and industry have been continuously innovating new strategies to enhance the performance of FTLs, including (by a broad FTL definition) address mapping, wear leveling [13], [30], hot-cold data separation [15], request scheduling [12], [18], dynamic write allocation [6], data migration [6], flash mode switching [11], [20], [27], refreshing [16], [21], WOM-coding [17], [28], and advanced error control and handling [25], [26], [32]. For example, the address mapping schemes have evolved from block-based, page-based [7], hybrid [24], to demand-based [19] design.

The relentless pursuit of high-performance FTLs renders FTL firmware increasingly complex and intricate; therefore, testing and validating FTLs are crucial and challenging tasks. In common practices, FTL developers perform real SSD-based stress tests to discover FTL bugs. For example, by executing stress-testing software (e.g., [3]), a computer can generate intensive read and write traffics to stress an SSD. The FTL is considered probably buggy if data corruption, abnormal SSD disconnection, or request timeout occurs during tests. This real SSD-based testing methodology is (and will still be) useful and required; however, it has two fundamental drawbacks and limitations:

- First, SSDs (especially flash memory) exhibit a quite limited access speed, which poses speed limitations to stress tests. For example, the fastest reported mainstream SSD in April 2017 is an NVMe PCIe SSD with a write speed of 0.73 GB/s on average (0.14 GB/s for random writes, 0.36 GB/s for queued random writes, and 1.7 GB/s for sequential writes) [4]. That means, it takes 23 minutes for one to stress test an FTL with 1 TB mixed writes ($1000/0.73/60 = 23$) even on this fastest NVMe PCIe SSD, let alone other relatively slower SSDs including SATA SSDs, USB sticks, eMMC chips, and SD cards, which also demand FTL tests.
- Second, investigating a failure occurring during real SSD-based FTL tests is a nightmare for FTL developers because it involves complicated factors including host OS (e.g., Windows or Linux) compatibility, mother board compatibility, cable signal quality, flash memory quality, power supply stability, FTL firmware, and non-FTL firmware. Just reproducing the failure can be very challenging, let alone locating and root-causing the bugs, which require experiences, patience, expensive equipment (e.g., protocol analyzers, logic analyzers, and JTAG debuggers), and sometimes good luck.

To address the above shortcomings, this work proposes a virtual stress testing (VST) framework, a functional simulation methodology focusing on testing and validating FTLs (**source code available at http://ssdlab.ee.nthu.edu.tw/vst**). VST en-

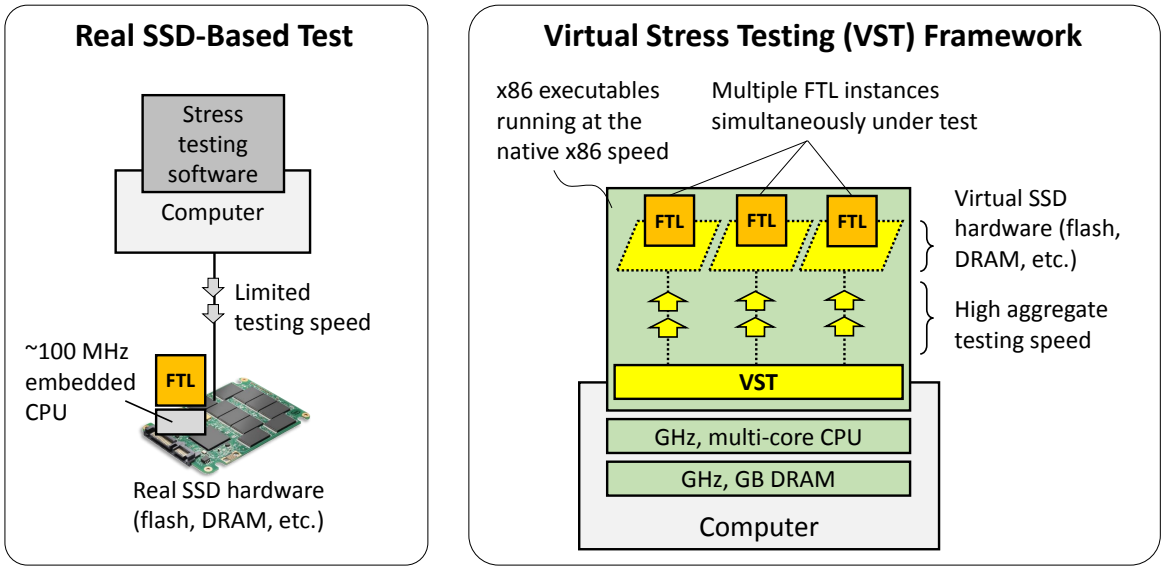


Fig. 1. Comparing the VST framework with real SSD-based tests

ables FTL developers to compile and execute native SSD FTL code on PCs or servers without the need of SSD hardware. VST utilizes a PC's or server's main memory to emulate various types of memory in SSDs (e.g., SRAM, DRAM, and flash). An SSD has a few GB to TB of capacity, which is usually much larger than PC or server main memory. To tackle this issue, we design a data structure that can efficiently represent large flash capacity.

We apply VST to well-known SSD design, OpenSSD [1]. OpenSSD is actively utilized and maintained by SSD and FTL communities; therefore, its FTLs are representative. Our evaluation shows that VST can perform stress tests on FTLs at a high speed up to 111 GB/s using single host CPU core and up to 375 GB/s using four host CPU cores, i.e., several hundred times faster than directly testing FTLs on SSD hardware. We successfully discover seven new FTL bugs in the OpenSSD design using VST, which is a solid evidence of VST's bug-discovering effectiveness. It would be very hard to discover and investigate these bugs without VST.

Several things are worth noting here. First, this work does not mean to abandon traditional real SSD-based tests, which can discover non-FTL bugs and hardware-related bugs that VST cannot detect. Second, upgrading VST is needed to support new flash or new SSD interfaces. For example, emerging 3D flash supports sub-block operations [14], and future SSDs can adopt key-value [22] or de-indirection interfaces [8], [31] instead of the conventional block-device interface. Modifications for these features would be modest and a one-time effort. Lastly, although VST speeds up FTL tests and ill FTL behavior detection, it remains engineers' responsibility (in this paper, ours) to pinpoint the buggy code in FTL firmware. Fortunately, the outcomes of VST can provide informative clues about in which function the bug is detected and what kind of ill behaviors the bug exhibits.

This paper is organized as follows. Section II presents related SSD simulator works. Section III presents VST design. Section IV elaborates on the bugs we discover in OpenSSD using VST and demonstrates the achievable testing speed of VST. Section V concludes this paper.

## II. Related Simulator Works

Simulators [2], [5], [6], [9], [23] are widely used in SSD researches. Two most representative simulators are Microsoft Research SSDSim [6] and Linux NANDsim [2]. However, as described below, they are not designed and optimized for *testing real-world FTLs at a high speed*, which VST is designed for and focuses on.

SSDSim [6] is one of the most widely used simulators by SSD and FTL researchers (including us) to model and compare the performance of different SSD or FTL designs. Like most computer system simulators targeting accurate performance analysis, SSDSim adopts an discrete-event simulation paradigm instead of a procedural paradigm, which is the nature of FTLs. Therefore, although SSDSim does simulate a few FTL policies, the FTL policies are tightly coupled with the discrete-event simulation framework, and porting another real-world FTL to SSDsim is extremely difficult. For an apparent example, SSDSim updates an FTL's logical-to-physical mapping table (*lpn_table[]*) and active block pointers in a C program file named *ssd_timing.c*, which as the file name suggests, also calculates the timing of various events such as writing flash pages. Another example is that in addition to performing an FTL's garbage collection (GC), *ssd_clean.c* also needs to calculate the latency of GC for performance simulation.

In addition, SSDSim abstracts away several SSD details that are irrelevant to performance simulation but necessary to realistic FTLs. For example, real-world FTLs usually store address mapping information (i.e., metadata) into flash memory and

TABLE I
VST APIs

| Category | API Name | Arguments |
|---|---|---|
| DRAM | vst_write_dram_32 | addr, val |
| | vst_read_dram_32 | addr |
| | vst_write_dram_16 | addr, val |
| | vst_read_dram_16 | addr |
| | vst_write_dram_8 | addr, val |
| | vst_read_dram_8 | addr |
| | vst_set_bit_dram | base_addr, offset |
| | vst_clr_bit_dram | base_addr, offset |
| | vst_tst_bit_dram | base_addr, offset |

| Category | API Name | Arguments |
|---|---|---|
| NAND Flash | vst_write_page | bank, blk, page, sect, n_sect, dram_addr, lpn, is_host_data |
| | vst_read_page | bank, blk, page, sect, n_sect, dram_addr, lpn, is_host_data |
| | vst_copyback_page | bank, blk_src, page_src, blk_dst, page_dst |
| | vst_erase_block | bank, blk |
| I/O | vst_write_sector | lba, len |
| | vst_read_sector | lba, len |
| Misc. | vst_memcpy | dst, src, len |
| | vst_memset | addr, value, len |
| | | |

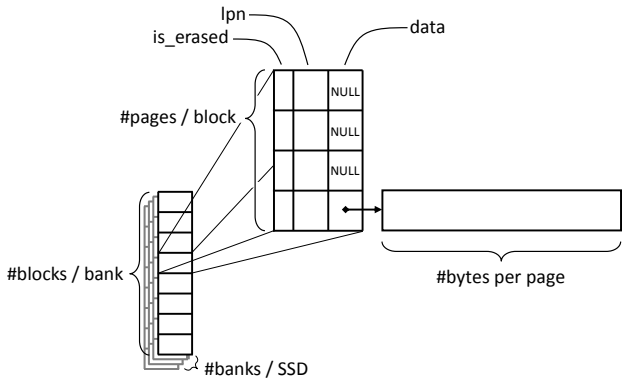


Fig. 2. Data structure for the emulated flash

retrieve them from the flash when necessary. SSDSim also models this strategy (i.e., storing a summary page at the end of each block); however, SSDSim only pays attention to the latency of doing so without indeed keeping the information. Another example is that when there are multiple requests queued to access multiple flash elements connected to a channel, SSDSim only calculates the overall latency assuming the requests are interleavingly completed (i.e., *ssd_collect_req-_in_gang()*) instead of invoking flash operations one by one like what a realistic FTL should do.

A particular type of simulators, specifically named emulators, use main memory or files to emulate the flash capacity of an SSD. NANDsim [2] in the Linux kernel is the most representative one among them. The advantages of this type of emulators are that the emulated flash appears as a block device in the OS and one really can execute applications (e.g., databases) on the emulated flash to experience the performance. However, emulators are not the optimal choice for FTL tests. If the flash capacity is emulated using main memory, which is typically few tens of GB, the size of the emulated SSDs is very limited. If the flash capacity is emulated using files, the testing speed cannot exceed the speed of the underlying storage, which turns out to be the same shortcoming as testing FTLs on real SSDs.

## III. VST Design

### A. Big Picture

Figure 1 compares the VST framework (on the right-hand side) with a real SSD-based test (on the left-hand side). In a real SSD-based test, an FTL is executed on the SSD controller, which typically consists of an embedded processor, on-chip SRAM main memory, and off-chip DRAM and flash that are not directly addressable. The FTL talks to on-chip DRAM and flash controllers to access DRAM and flash, respectively. PC- or server-side software utilities are used to generate intensive read and write requests to stress the FTL on the SSD and validate FTL's functionalities. The testing speed is limited because the embedded processor is typically a low-power processor, and in addition, each flash read, write, or erase operation must take 0.1 to few ms. If debugging the FTL is necessary, common practices utilize an in-circuit debugger to trace the firmware running on the SSD, a protocol analyzer to record the packets transmitted through the IO interface of the SSD, or a logic analyzer to capture the signals on the pins of the SSD controller.

In comparison, VST on the right-hand side can perform FTL tests without any SSD hardware requirements nor SSD speed limitations. FTLs are compiled as x86 dynamically linked shared objects (.so). VST emulates SSD hardware (virtual SSD hardware) such as SRAM, DRAM, and flash memories using the DRAM main memory of PCs or servers. VST also offers a set of generic application programming interfaces (APIs) such as reading an emulated DRAM word or erasing an emulated flash block to interface FTL objects with the virtual SSD hardware. Read and write requests are generated by the VST. With the above design, an FTL can be executed on the PC or server at a multi-GHz frequency like a native PC or server program does. Multiple FTL instances can be simultaneously under test, and the testing speed is limited by the memory system and x86 processors of PCs or servers instead of flash or a low-power processor. Once a failure occurs during FTL tests, engineers can easily reproduce the failure and analyze the failure in PC or server environments using generic debugging approaches such as the GNU debugger (GDB).

### B. APIs

Table I lists the key APIs supported by VST. The APIs are in four groups, I/O, DRAM, flash, and miscellaneous. The I/O API group generates reads or writes to an FTL under test. The arguments for the two APIs are the starting logical block address (LBA) and the length of the requested data. Please note that host data are not one of the arguments here because an FTL should work irrelevantly to the actual host data. The DRAM API group includes reading and writing to different sizes of data in the emulated DRAM. The miscellaneous API group includes moving and filling data within DRAM or SRAM. Emulating the I/O requests is done by synthesizing requests according to trace files or a random sequence. Emulating the DRAM and SRAM is done by creating raw arrays in the DRAM main memory of PCs or servers.

The flash API group supports four fundamental flash operations, i.e., reading a page, writing a page, copying a page internally (i.e., copyback), and erasing a block. Flash memory behaves differently from raw arrays, and thus the emulation is more challenging than that of DRAM and SRAM. Detailed design of flash APIs is presented in subsection III-D.

### C. FTL-Aware VST Design and Optimizations

We observe that naively simulating FTL codes on PCs or servers would cause excessive memory usage and redundant memory accesses, which dramatically slow down stress tests. By profiling the execution time and inspecting FTL source code, we identify two FTL-aware design and optimizations for VST regarding flash and RAM (an SSD's SRAM or DRAM).

First, we propose that flash data need to be handled separately according to their impacts to FTL's correctness. By doing so, VST can achieve a high effective testing speed up to several hundreds of GB/s. For example, VST keeps the summary page that seals each flash block in the emulated flash capacity but maintains a tag of logical page number (lpn) for each flash page of host data, which are not directly used by FTLs. Differentiating host data from other data significantly reduces the memory traffics and footprint for simulating a large-capacity SSD and boosts the testing speed. The footprint reduction ratio is approximately equal to the number of flash pages per flash block (depending on the design of each FTL). We anticipate FTL developers can annotate this data differentiating information in FTLs when flash APIs are invoked. Such annotation is straightforward and a one-time effort.

Second, we observe that FTLs frequently perform RAM fills and RAM copies, but a significant portion of them are in fact irrelevant to FTL tests. A lot of redundant operations result from 1) filling a RAM buffer with 0xFF bytes when an erased page is read and 2) copying data between RAM buffers when a write request triggers a read-modify-write procedure, but it turns out that the FTL does not need these RAM data. Similar to handling flash data, we anticipate FTL developers can annotate in FTLs whether each RAM fill or RAM copy is omittable during FTL tests.

### D. Flash Emulating Schemes

Figure 2 depicts the data structure we design to emulate flash using host-side DRAM main memory. The data structure consists of multiple arrays of subarrays. Each array corresponds to a bank (also called a channel or a gang) of flash elements, each subarray represents a block of multiple flash pages, and each page consists of three fields, is_erased, lpn, and a data pointer. The is_erased field stands for whether the page is erased, the lpn field is a tag recording the logical page number of the page, and the data pointer is allocated memory space if the data of the page are not omittable.

**Algorithm 1** vst_write_page

```
1:  if (page > 0) then
2:      prev_page ≡ emulated_flash[bank][blk][page-1]
3:      if (prev_page.is_erased) then
4:          exit("ill behavior: non-sequential write in a block")
5:      end if
6:  end if
7:  page ≡ emulated_flash[bank][blk][page]
8:  if (! page.is_erased) then
9:      exit("ill behavior: in-place write")
10: else
11:     page.is_erased = false
12:     if (is_host_data) then
13:         page.lpn = lpn
14:     else
15:         page.data = malloc(bytes_per_page)
16:         for (i = 0; i < len; i++) do
17:             page.data[start+i] = dram_addr[i]
18:         end for
19:     end if
20: end if
```

*vst_write_page:* Algorithm 1 lists the pseudocode of the vst_write_page API. Lines 1 to 6 check whether all pages in a block are written according to a strict sequential order, i.e., page 0, 1, ..., to the last page [10]. This constraint is needed for multiple-level cell (MLC) flash to keep the error rate of write disturbs under a predefined level. Lines 8 and 9 check whether a page is in-place written without an erasure in advance. Lines 11 and 13 update the is_erased and lpn fields.

Please note that we do not mean that violating the above sequential-write and out-place-write rules is always a bug. Some advanced FTLs [16]–[18] do exploit the advantages of doing so in the presence of certain safety net mechanisms. Because OpenSSD FTLs are not designed this way, reporting all violations of these rules is necessary (Section IV).

It is challenging to emulate the full capacity of an SSD using the DRAM main memory of PCs or servers because of the large capacity of SSDs. To address this challenge, we let VST omit to store the host data, which are irrelevant to FTL stress tests. Lines 12 to 19 implement this strategy.

*vst_read_page:* Algorithm 2 displays the pseudocode of the vst_read_page API. For a read request to host data, it is sufficient to only check whether lpn remains consistent (Lines

**Algorithm 2** vst_read_page

```
1: page ≡ emulated_flash[bank][blk][page]
2: if (is_host_data) then
3:    if (page.lpn != requested lpn) then
4:       exit("ill behavior: inconsistent lpn");
5:    end if
6: else
7:    if (page.data) then
8:       for (i = 0; i < len; i++) do
9:          dram_addr[i] = page.data[i];
10:      end for
11:   else
12:      for (i = 0; i < len; i++) do
13:         dram_addr[i] = 0xff;
14:      end for
15:   end if
16: end if
```

**Algorithm 3** vst_erase_block

```
1: for (p = 0; p < pages_per_blk; p++) do
2:    page ≡ emulated_flash[bank][blk][p])
3:    page.is_erased = true
4:    page.lpn = UNKNOWN_LPN
5:    free(page.data)
6:    page.data = NULL
7: end for
```

2 to 5). If an FTL tries to retrieve its own data (i.e., FTL metadata), the data pointer is de-referenced to provide the data (Lines 7 to 10). Lines 11 to 15 return data with all 0xff bytes (the erased state of flash cells) if an erased page is read.

*vst_erase_block:* Algorithm 3 shows the pseudocode of the vst_erase_block API. A loop iterates over all pages in a block. The is_erased field of each page is set, and the data pointer of each page is freed.

## IV. Evaluation

We apply VST to OpenSSD [1] to demonstrate the effectiveness of VST. Currently, the firmware of OpenSSD has evolved from version 1.0.0 to the latest stable version of 1.1.0. OpenSSD offers three FTLs: Greedy, DAC, and FASTer. Greedy FTL implements a page-based FTL with a greedy victim selection policy during GC. Based on Greedy FTL, DAC FTL further adopts the DAC hot-cold data separation scheme [15] and a cost-benefit victim selection policy during GC. FASTer is a hybrid FTL, which adopts page-based mapping for the log area and block-based mapping for the data area [24].

We execute VST on a PC having an Intel Core i7-4790 3.6 GHz quad-core processor, 32 GB DDR3 DRAM main memory, a 1 TB 7200-RPM HDD, and the Ubuntu 16.04 OS. The latest FTL code of OpenSSD (i.e., ftl.c) and VST code are compiled using gcc 5.4.0 with the -O3 optimization flag. We investigate the FTL bugs manifesting during VST execution using GNU GDB and common software debugging techniques.

```
1  // the latest (v1.1.0) DAC ftl.c
2  assign_new_write_vpn(...)
3  {
4    ...
5    if(write_vpn is the last page of a blk){
6      ...
7      seal the blk;
8      while(free_blk_cnt() <= NUM_REGIONS){ // bug
9        garbage_collection();
10     }
11
12     if(write_vpn is no longer the last page of a
       blk){
13       increment write_vpn;
14       return write_vpn;
15     } // bug
16
17     do{
18       blk = next_blk();
19     while(is_not_free(blk));
20   }
21   // write page -> next block
22   if(blk != write_vpn/PAGES_PER_BLK){ // bug
23     set write_vpn as (blk * PAGES_PER_BLK);
24     ...
25   }else{
26     increment write_vpn; // bug
27   }
28   return write_vpn;
29 }
```

Fig. 3. Code snippet of DAC FTL

```
1  // the latest (v1.1.0) FASTer ftl.c
2  full_merge(...)
3  {
4    ...
5    if(victim blk happens to be a sequential write
       log blk){
6      partial_merge();
7      if(...){
8        return; // bug
9      }
10   }
11   ...
12   if(reaching the last page of the victim blk){
13     allocating a new log block;
14   }
15 }
```

Fig. 4. Code snippet of FASTer FTL

We find that all the three FTLs of the latest OpenSSD firmware have bugs. Without using VST, it would be difficult for engineers (including us) to find out these bugs because 1) the firmware is complex, 2) the bugs are related to subtle boundary conditions, and 3) some of the bugs only happen during GC, which needs a large number of write requests to trigger. In addition, although these bugs can also fail real SSD-based stress tests, without VST, reproducing and locating the bugs in real SSDs are still extremely challenging and time-consuming tasks.

```
1  // the latest (v1.1.0) Greedy ftl.c
2  assign_new_write_vpn(...)
3  {
4    if(write_vpn is the last page of a blk){
5      ...
6      seal the blk;
7      if(free_blk_cnt() == 1){
8        garbage_collection();
9      }
10   }
11   if(blk != write_vpn/PAGES_PER_BLK){
12     set write_vpn as (blk * PAGES_PER_BLK);
13     ...
14   }else{
15     increment write_vpn; // bug
16   }
17 }
```

Fig. 5. Code snippet of Greedy FTL

```
1  // the latest (v1.1.0) Greedy ftl.c
2  logging_pmap_table(...)
3  {
4    for(bank=0; bank < NUM_BANKS; bank++){
5      ...
6      inc_mapblk_vpn(bank, mapblk_lbn); // bug
7      ...
8      // store metadata to flash
9      write_metadata(mapblk_vpn);
10   }
11 }
```

Fig. 6. Code snippet of Greedy FTL

### A. FTL Bug Discovery Results

*DAC FTL:* Four bugs are found in the *assign_new_write_vpn()* function, which is responsible for allocating a free flash page to store data. For brevity, Figure 3 lists the pseudocode. If the active block (the block to be written to) is full (Line 5), Line 7 seals the block, and Lines 8 to 10 keep performing GC if the number of free blocks is below a watermark. In Lines 12 to 15, if the active block is no longer full (due to the GC in Lines 8 to 10), a free page of the active block is returned; otherwise, Lines 17 to 19 search for a new free block. After a new free block is obtained, the *if* code block (Lines 22 to 24) selects a free page of the new block; otherwise, the next free page in the current active block is selected (Line 26).

Lines 8, 15, 22, and 26 (Lines 1129, 1136, 1148, and 1157 of the original code, respectively) contain the four bugs we newly discover using VST. In Line 8, the watermark is set too low (i.e., equal to the number of hot-cold regions). Each region can hold one free block as its active block, and thus DAC FTL can get stuck in the infinite loop (Lines 17 to 19) because of a shortage of available free blocks. The second bug is at Line 15, where the *else* code block is missing. When the active flash block happens to be full after garbage collection (Lines 8 to 10), there should be an *else* code block at Line 15 to seal the flash block (like what Line 7 does). The third bug is at Line 22, which expects that the free block found by Lines 17 to 19 must be a different block and cause a change of the physical address. However, in a rare case, during several times of GC (Lines 8 to 10), an active block can become full, then be erased, and again be chosen as the active block. Please note that this bug correlates with the first one: one reason why the FTL can run out of available free blocks is that the abovementioned FTL implementation does not allow the current active block to become the next active block right after GC. Lastly, when an SSD is powered on for the very first time, Line 26 incorrectly causes the first page of the active block to be skipped. This situation is a bug for DAC FTL because the FTL is not designed to exploit the benefits of violating the rule that all pages in an MLC flash block must be written in a strictly sequential order (as mentioned in Subsection III-D).

TABLE II
STRESS TESTING TRACES

| Name | Write | | Read | |
|---|---|---|---|---|
| | #Requests (M) | Tot. Size (TB) | #Requests (M) | Tot. Size (TB) |
| hm_0 | 78 | 1 | 43 | 0.5 |
| mds_0 | 87 | 1 | 12 | 0.3 |
| prn_0 | 67 | 1 | 8 | 0.2 |
| proj_0 | 23 | 1 | 3 | 0.1 |
| prxy_0 | 91 | 1 | 3 | 0.0 |
| prxy_1 | 60 | 1 | 113 | 1.8 |
| rsrch_0 | 80 | 1 | 8 | 0.1 |
| src1_0 | 18 | 1 | 28 | 1.1 |
| src1_2 | 28 | 1 | 9 | 0.2 |
| src2_0 | 90 | 1 | 11 | 0.1 |
| src2_2 | 18 | 1 | 8 | 0.6 |
| stg_0 | 77 | 1 | 14 | 0.4 |
| ts_0 | 85 | 1 | 18 | 0.3 |
| usr_0 | 69 | 1 | 47 | 2.0 |
| wdev_0 | 84 | 1 | 21 | 0.3 |
| web_0 | 79 | 1 | 34 | 1.1 |

*FASTer FTL:* One bug is found in the *full_merge()* procedure, which is responsible for collecting data from a victim log block to a data block (Figure 4). Under some conditions, the victim block happens to be a sequentially-written block, and the full merge task can be completed by a partial merge (Lines 5 to 10). At the end of the full merge procedure, a new log block is allocated (Lines 12 to 14). FASTer FTL is buggy because, at Line 8 (Line 1558 of the original code), the *full_merge()* procedure returns and misses to check whether allocating a new block is needed like what Lines 12 to 14 do.

*Greedy FTL:* Two bugs are found in Greedy FTL. Both result in skipping the first page of a block when an SSD is powered on for the very first time. This situation is a bug because Greedy FTL is not designed to exploit the benefits of violating this rule (as mentioned in Subsection III-D). The first bug is in the *assign_new_write_vpn()* procedure (Line 15 of Figure 5 or Line 598 of the original code). The other is in the *logging_pmap_table()* function (Line 6 of Figure 6 or Line 910 of the original code).

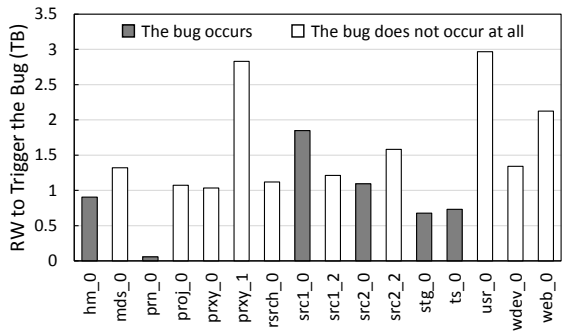


Fig. 7. RW amount to trigger the bug of FASTer

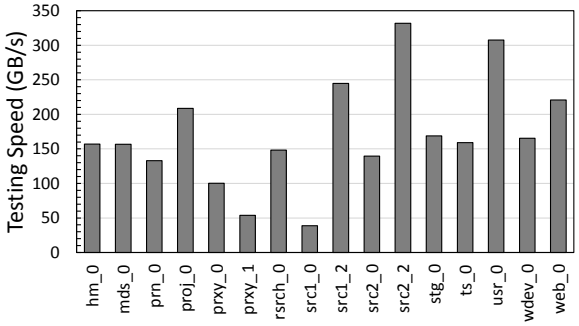


Fig. 9. Testing speed for Greedy FTL

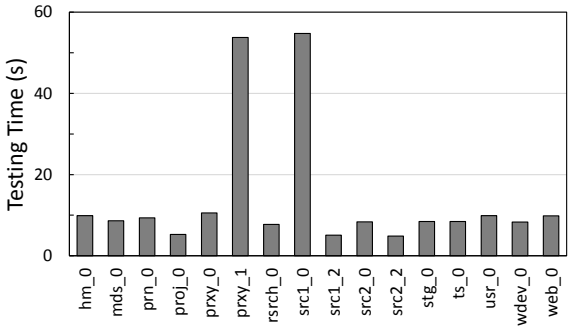


Fig. 8. Testing time for Greedy FTL

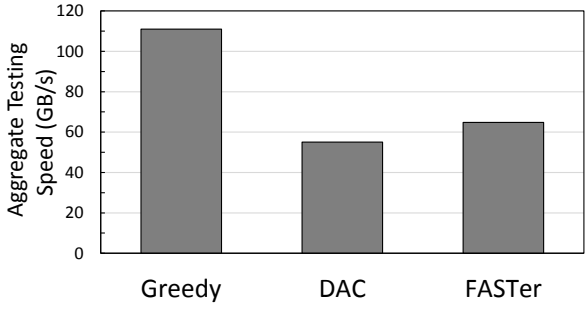


Fig. 10. Testing speed for three FTLs

### B. FTL Testing Speed

Table II lists 16 disk IO traces [29] we use to drive VST. We select traces with a sufficient write amount ($\geq 3\ GB$ writes and a write-to-read ratio of $\geq 20\%$) to stress test SSDs. We align traces to 8 KB address boundaries and repeat each trace with an LBA offset till the write amount reaches 1 TB. Please note that traces are not a must for VST because one can let VST synthesize random requests internally instead.

Figure 7 shows the amount of read and write requests required before the bug of FASTer is triggered. Although these numbers are obtained using VST, they can estimate the number for real SSD stress tests, too. Out of the 16 traces, ten traces do not trigger the bug to appear at all. For the other six traces, up to 1.8 TB of accesses are needed before the bug occurs. These results confirm that the bug is hard to find without a sufficiently large stress test scale.

Greedy and DAC FTLs are immediately detected buggy by VST for all 16 traces due to the existence of the non-sequential page write bugs. Even though, we want to point out that the non-sequential page write bugs (and the consequences of violating the MLC flash rule) are in fact hard to reproduce or debug using real SSDs because the bugs only occur once when an SSD is powered on for the very first time.

We fix the seven bugs founded in the three FTLs and conduct the following tests. Figure 8 demonstrates the testing speed advantage of VST. It takes only 4.8 to 55 seconds for VST to complete Greedy FTL tests using one trace. The differences in the required testing time result from different write randomness and different read-to-write ratios among traces. In comparison, as mentioned in Section I, if one performs the tests on a real SSD with 0.73 GB/s write performance [4], each trace would take more than 23 minutes to complete because each trace contains 1 TB writes and additional reads.

Figure 9 shows that the effective throughput of VST with the Greedy FTL ranges from 39 GB/s to 332 GB/s. Figure 10 shows that the effective throughput for VST to test the three FTLs is 55 GB/s to 111 GB/s over 16 traces.

VST is superior in scalability to conventional real SSD-based tests. For instance, by leveraging servers or cloud computing services, a massive number of VST tests can be instantiated in parallel. Figure 11 shows that just one four-core PC can offer up to 375 GB/s of aggregate testing throughput, which is several hundred times higher than a real mainstream SSD. Without VST, the cost and difficulties of setting up and maintaining such massive number of real SSDs and their associated testing equipment would be significant.

## V. Conclusions

This work proposes VST, a framework for FTL developers to execute, test, and debug SSD FTL firmware on PCs or servers. VST addresses the challenges that 1) FTL firmware is increasingly complex and intricate, and testing and validating FTLs are crucial and challenging tasks, 2) real SSD-based tests have limitations in speed, 3) it is hard to root-cause failures occurring during real SSD-based FTL tests, and 4) existing

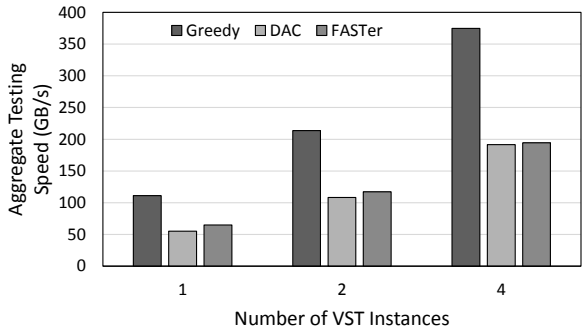


Fig. 11. Parallel testing speeds using multiple cores

SSD simulators are not designed and optimized to test real-world FTLs at a high speed. We apply VST to OpenSSD [1] to evaluate the speed and effectiveness of VST. Experimental results show that VST can perform FTL tests at a speed of 111 GB/s using single host CPU core and up to 375 GB/s using four host CPU cores. We successfully discover seven new bugs in the latest OpenSSD FTLs, which is a solid evidence of VST's bug-discovering effectiveness.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful feedback. This work is supported in part by the Ministry of Science and Technology (MOST) of Taiwan under grants 105-2218-E-007-023-, 106-2218-E-007-002-, and 106-2221-E-007-125-.

## REFERENCES

[1] Jasmine OpenSSD platform. http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.
[2] NANDsim. http://manpages.ubuntu.com/manpages/xenial/man4/nandsim.4freebsd.html.
[3] PassMark BurnInTest. http://www.passmark.com/products/bit.htm.
[4] Samsung SSD 960 PRO 512GB benchamarks. http://ssd.userbenchmark.com/SpeedTest/182182/Samsung-SSD-960-PRO-512GB.
[5] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating goliath storage systems with david. In *USENIX Conference on File and Storage Technologies (FAST)*, 2011.
[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (USENIX ATC)*, 2008.
[7] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41:88–93, April 2007.
[8] M. Bjørling, J. González, and P. Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017.
[9] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual reference manual (CMU-PDL-08-101). Technical report, Parallel Data Laboratory, 2008.
[10] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *International Conference on Computer Design (ICCD)*, 2013.
[11] C.-W. Chang, G.-Y. Chen, Y.-J. Chen, C.-W. Yeh, P.-Y. Eng, A. Cheung, and C.-L. Yang. Exploiting write heterogeneity of morphable MLC/SLC SSDs in datacenters with service-level objectives. *IEEE Trans. Comput.*, 66(8):1457–1463, Aug 2017.
[12] L.-P. Chang, C.-H. Cheng, and K.-H. Lin. A flash scheduling strategy for current capping in multi-power-mode ssds. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.
[13] Y.-M. Chang, Y.-H. Chang, J.-J. Chen, T.-W. Kuo, H.-P. Li, and H.-T. Lue. On trading wear-leveling with heal-leveling. In *Design Automation Conference (DAC)*, 2014.
[14] T.-Y. Chen, Y.-H. Chang, C.-C. Ho, and S.-H. Chen. Enabling sub-blocks erase management to boost the performance of 3D NAND flash memory. In *Design Automation Conference (DAC)*, 2016.
[15] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for plash memory. *Softw. Pract. Exper.*, 29(3):267–290, Mar. 1999.
[16] A. Cristal. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *International Conference on Computer Design (ICCD)*, 2012.
[17] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *International Symposium on Microarchitecture (MICRO)*, 2009.
[18] L. M. Grupp, J. D. Davis, and S. Swanson. The harey tortoise: Managing heterogeneous write performance in SSDs. In *USENIX Conference on Annual Technical Conference (ATC)*, 2013.
[19] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.
[20] S. Im and D. Shin. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.*, 56(12):641–653, Dec. 2010.
[21] J. Jeong, Y. Song, S. S. Hahn, S. Lee, and J. Kim. Dynamic erase voltage and time scaling for extending lifetime of NAND flash-based SSDs. *IEEE Trans. Comput.*, 66(4):616–630, April 2017.
[22] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. KAML: A flexible, high-performance key-value SSD. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
[23] M. Jung, E. H. Wilson, D. Donofrio, J. Shalf, and M. T. Kandemir. NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.
[24] S.-P. Lim, S.-W. Lee, and B. Moon. FASTer FTL for enterprise-class flash memory SSDs. In *Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2010.
[25] R.-S. Liu, M.-Y. Chuang, C.-L. Yang, C.-H. Li, K.-C. Ho, and H.-P. Li. EC-Cache: Exploiting error locality to optimize LDPC in NAND flash-based SSDs. In *Design Automation Conference (DAC)*, 2014.
[26] R.-S. Liu, M.-Y. Chuang, C.-L. Yang, C.-H. Li, K.-C. Ho, and H.-P. Li. Improving read performance of NAND flash SSDs by exploiting error locality. *IEEE Trans. Comput.*, 65(4):1090–1102, April 2016.
[27] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conference on File and Storage Technologies (FAST)*, 2012.
[28] F. Margaglia, G. Yadgar, E. Yaakobi, Y. Li, A. Schuster, and A. Brinkmann. The devil is in the details: Implementing flash page reuse with WOM codes. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[29] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4:10:1–10:23, November 2008.
[30] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and P.-C. Huang. New ERA: New efficient reliability-aware wear leveling for endurance enhancement of flash storage devices. In *Design Automation Conference (DAC)*, 2013.
[31] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *USENIX Conference on File and Storage Technologies (FAST)*, 2012.
[32] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *USENIX Conference on File and Storage Technologies (FAST)*, 2013.