DLOOP: A Flash Translation Layer Exploiting Plane-Level Parallelism

The 27th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2013 **Best Paper**)

Abdul Abdurrab, Tao Xie, and Wei Wang, San Diego State University, San Diego, CA 92182

Introduction

A flash translation layer (FTL) is a software layer running in the flash controller of a NAND flash memory solid-state disk (SSD) . It translates logical addresses received from a file system to physical addresses in SSD so that the linear flash memory appears to the system like a block storage device. Several recent research reports on SSD architecture reveal that SSD internal features such as advanced commands and multi-level parallelism could significantly impact the performance largely. By the insights provided in these research as well as our own investigation on how to effectively employ the multi-level parallelism present in flash SSD motivate us to develop an optimized page-mapping FTL named DLOOP (data log on one plane) that can exploit plane-level parallelism to achieve high performance while maintaining good durability. Experimental results show that DLOOP consistently outperforms a classical hybrid FTL named FAST and a modern page-mapping FTL called DFTL.

DLOOP employs the plane-level parallelism in two aspects:

- First of all, DLOOP always splits a multi-page sequential read/write request into multiple one-page requests and then disperses them across multiple planes to reduce waiting time by exploiting inter-plane parallelism;
- Secondly, it allocates logs (updates) onto the same plane where their associated original data resides so that intra-plane copy-back operation employed garbage collection can be carried out concurrently at plane level.

Illustrative examples

In order to spread requests evenly across all planes in a SSD without increasing the overhead of flash controller, following equation is used to calculate the plane number of an incoming request.

$$plane\_no = LPN(request)\%total\_num\_of\_plane$$

- **new write requests**

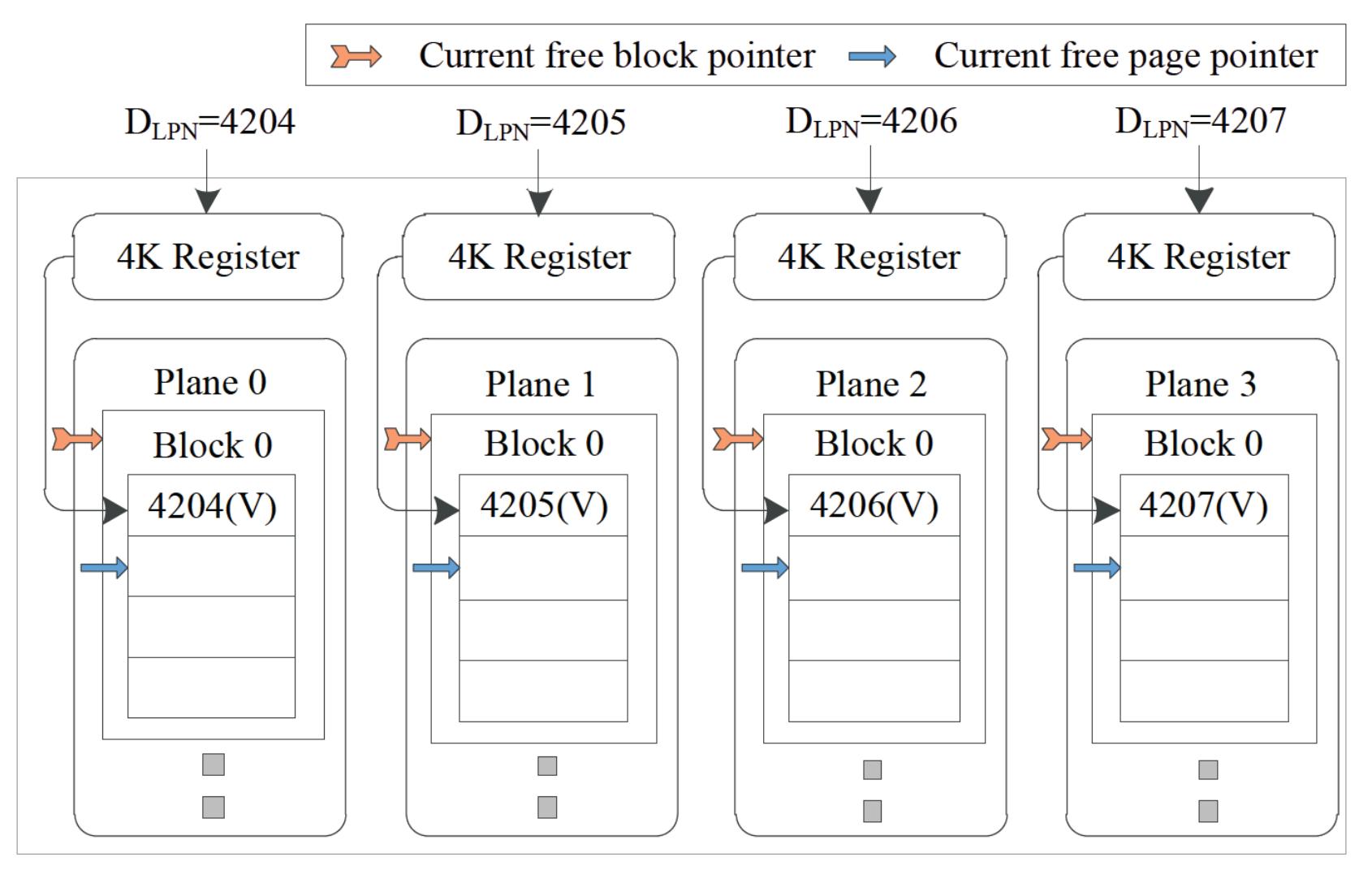


Figure 1. An illustrative example of four new writes.

We assume that an SSD only has four planes and the page size is 4KB. Fig. 1 illustrates how DLOOP processes a 14-KB new write request with starting *logical page number* (LPN) 4204. Since DLOOP always aligns each request on page boundary, the request will be divided into four individual one-page write requests: D_{LPN} = 4204, D_{LPN} = 4205, D_{LPN} =4206, and D_{LPN} = 4207. Also, assume that the SSD has never been accessed before this 14-KB write request comes. Based on the equation, DLOOP directs these four individual one-page write requests onto the data registers of plane 0, plane 1, plane 2, and plane 3, respectively.

- **update requests**

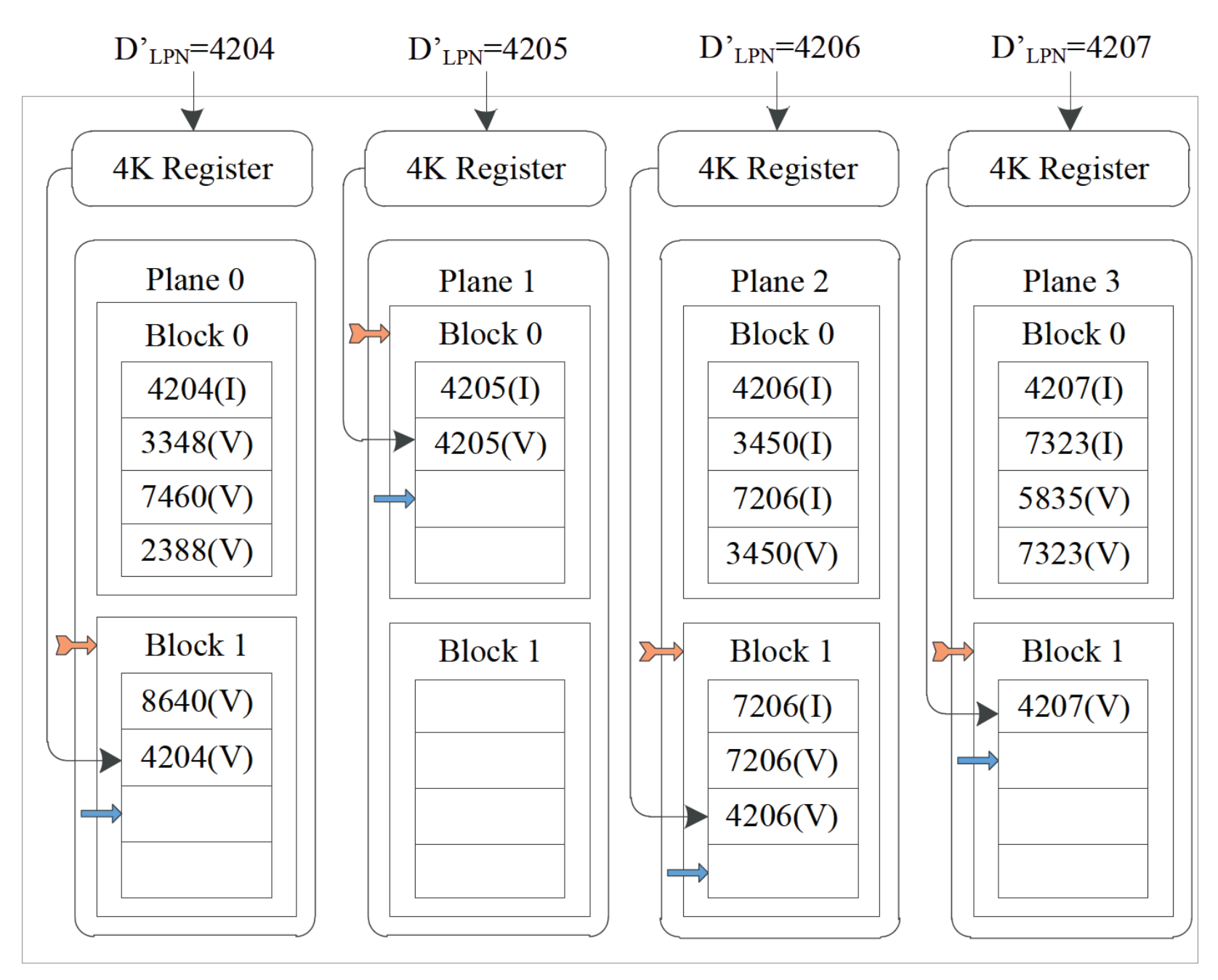


Figure 2. An illustrative example of four updates.

After a period of time, assume that the 14-KB write request with starting logical page number 4204 comes again. DLOOP splits it into four individual one-page update requests and then directs them to their destination plane where their associated original data resides. DLOOP first writes the new data into the page where is pointed by the current page pointer, and then invalidates the original page in the same block (see Fig. 2).

- **copy-back employed plane-level garbage collection**

Instead of global garbage collection, DLOOP uses plane-level copy-back employed garbage collection. It exploits plane-level parallelism, because of which overall performance is improved. Fig. 3 demonstrates this idea. Garbage collection is processing on plane1 and plane2. Meanwhile, plane0 and plane3 are still accommodating the upcoming request since copy-back operation doesn't occupy the bus and controller.

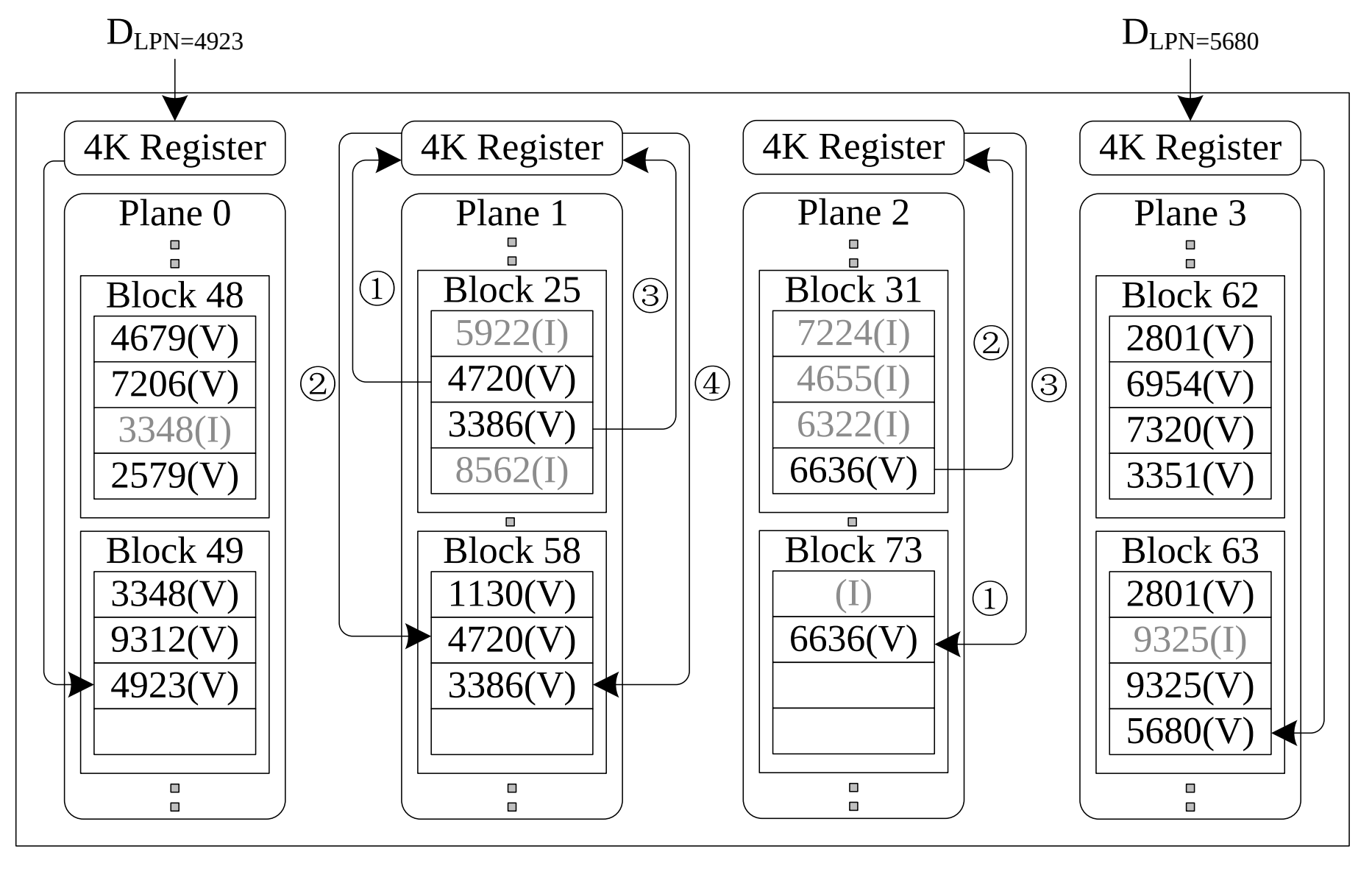


Figure 3. An illustrative example of garbage collection.

There are three situations when a GC occurs: 1) no data move is needed if the victim block has no valid page at all, 2) current free block still has enough room to accommodate valid pages from the victim block (shown in Fig. 3 plane1), and 3) DLOOP has to use a new free block as illustrated in Fig. 3 plane 2. Meanwhile, it also shows that DLOOP wastes a free page in the current free block in order to make sure the source page address and the destination page address share the same parity.

The simulator

We wrote almost 2000 lines of code to largely extend a validated open-source flash SSD simulator named FlashSim, which was built by extending DiskSim3.0. Since FlashSim has a modular architecture, newly developed FTLs can be readily integrated into it. Fig. 4 shows a simple view of the simulation architecture, which explains how requests are processed in our new DiskSim/FlashSim simulation environment.

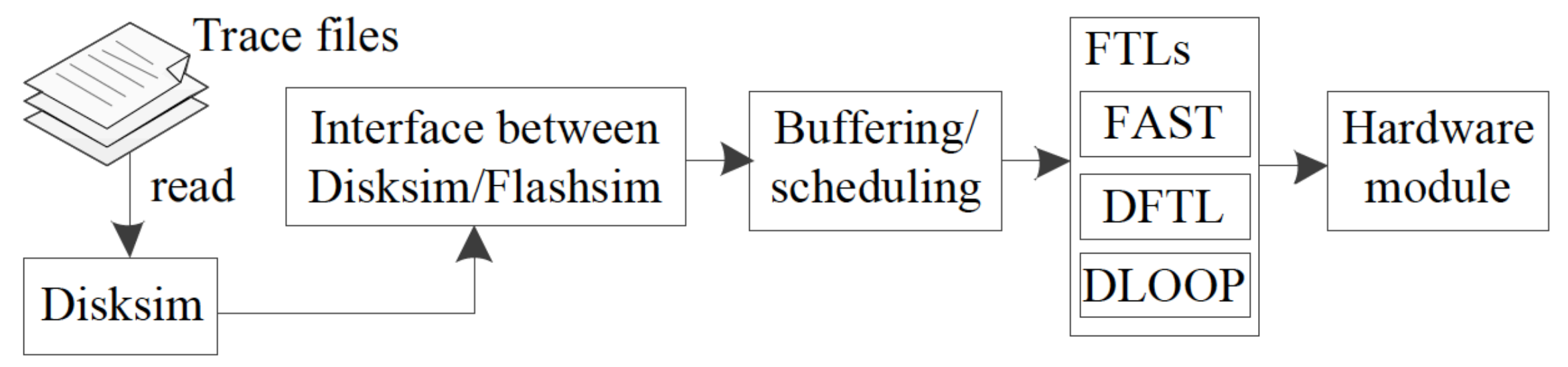
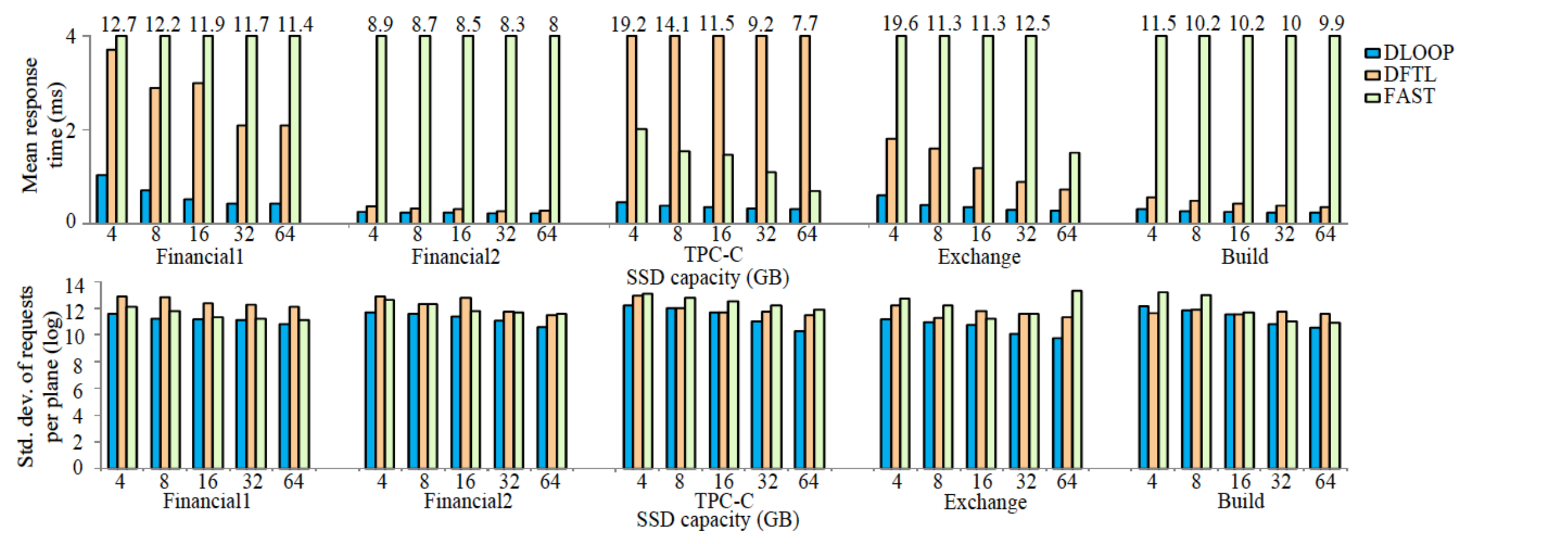
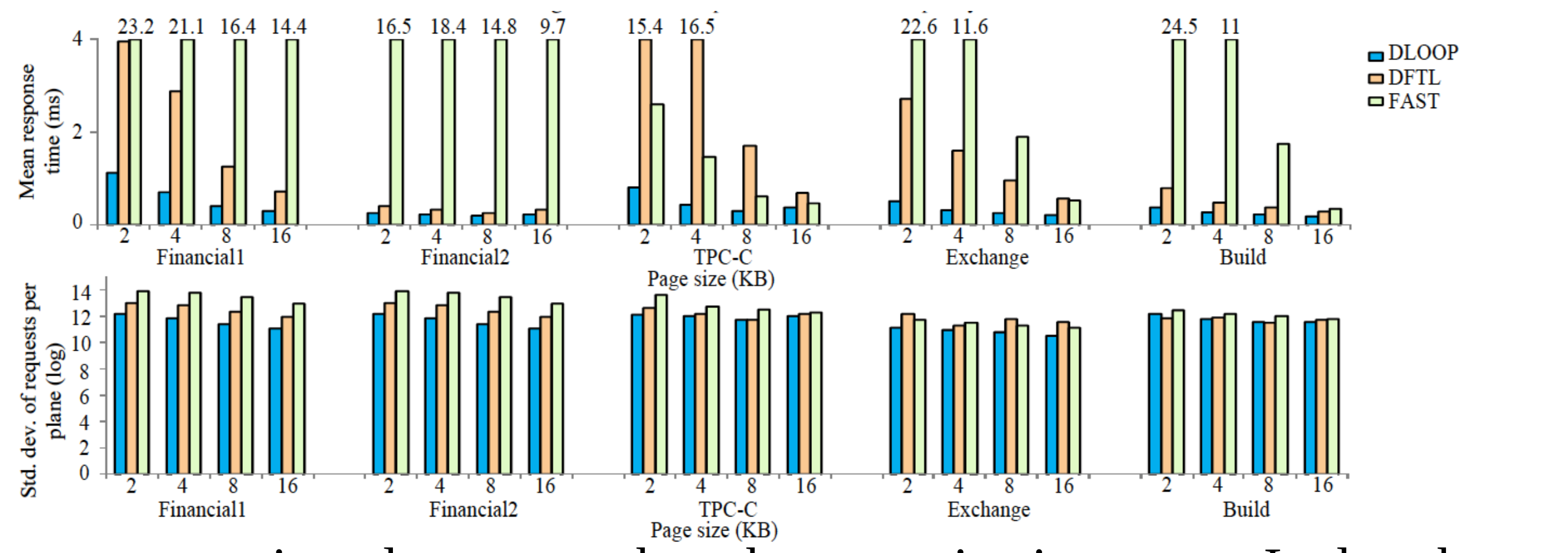


Figure 4. Architecture of the extended simulator.

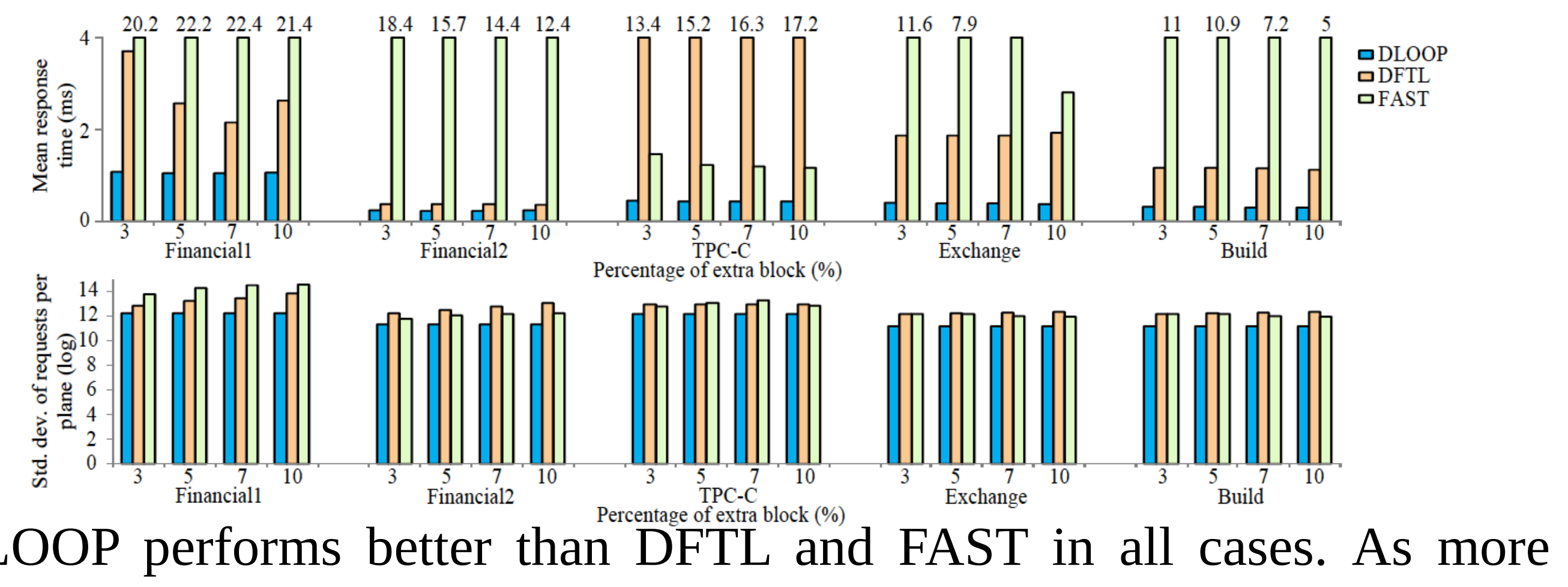
Performance evaluation



In terms of mean response time, on average, DLOOP performs 70% and 90% better than DFTL and FAST, respectively. As SSD capacity increases, we see as their mean response times all reduce noticeably.



Mean response time decreases when the page size increases. It also shows that for every page size DLOOP performs better than DFTL and FAST.



DLOOP performs better than DFTL and FAST in all cases. As more extra blocks are available, FAST improves performance but DLOOP remains almost the same.

Conclusions

In this research we propose an optimized page-mapping FTL called DLOOP, which fully exploits plane-level parallelism including the fast intra-plane copy-back operations to achieve high performance while maintaining good durability by evenly distributing requests across all planes. Although FTLs developed by flash SSD manufacturers might also exploit the internal parallelism, they are normally commercial secrets, and thus, are unknown to the public domain. Therefore, developing a high-performance FTL exploiting plane-level parallelism and the quantified analysis of the promising experimental results remain valuable to research communities.

Acknowledgement

This work was supported by the US National Science Foundation under Grant CNS-0834466 and CNS(CAREER)-0845105.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in Proc. USENIX Annual Technical Conference, pp.57-70, 2008.
[2] 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 Proc. Int'l Conf. Supercomputing (ICS'11), pp. 96-107, 2011.