Adaptive DRAM Cache Division for Computational Solid-state Drives

Shuaiwen Yu, Zhibing Sha, Chengyong Tang, Zhigang Cai, Peng Tang, Min Huang, Jun Li, Jianwei Liao*
College of Computer and Information Science, Southwest University, Chongqing, China

***Abstract*—High computational capabilities enable modern solid-state drives (SSDs) to be computing nodes, not just faster storage devices, and the SSD having such capability is generally called as the computational SSD (*CompSSD*). Then, the DRAM data cache of *CompSSD* should hold not only the output data of the tasks running at the host side, but also the input data of the tasks executed at the SSD side. To boost the use efficiency of the cache inside *CompSSD*, this paper proposes an adaptive cache division scheme, to dynamically split the cache space for separately buffering the output data running at the host and the input data running at the *CompSSD*. Specifically, we construct a mathematical model running at flash translation layer of *CompSSD*, to periodically determine the cache proportion of the workloads running at the host side and the *CompSSD* side, by considering the factors of the ratios of read/write data amount, the cache hits, and the overhead of data transfer between the storage device and the host. Then, both the output data and the input data can be buffered in their own private cache parts, so that the overall I/O performance can be enhanced. Trace-driven simulation experiments show that our proposal can reduce the overall I/O latency by `27.5%` on average, in contrast to existing cache management schemes.**

***Index Terms*—Solid-state Drives (SSDs), Computational SSDs, DRAM Cache, Adaptive Cache Division, I/O Performance.**

I. Introduction

NAND flash-based solid-state drives (SSDs) have been widely employed in both consumer digital devices, and high performance computing environments, thanks to their advances of high data throughput and low power consumption [1], [2]. In conventional computing systems, SSDs are treated as storage-only units, indicating data is stored on SSDs, while computation is performed at the host CPU [3]. For example, if the application aims to examine a piece of data to make a decision, it must pull the data from SSDs through a (relatively) slow interface (such as PCIe, SATA, and SAS) into the main memory where the host CPU has access to it [4].

However, it has been proven that such paradigm of "*move data closer to code*", cannot make full use of SSDs [4], [5]. On the one hand, SSDs are usually manufactured with a multiple-time higher internal bandwidth than the host interface bandwidth. On the other hand, modern SSDs have more computing power and memory capacity in micro-controllers [6], [7], to process only flash translation layer (FTL) tasks, implying powerful computing resources are generally ignored by conventional computing systems. To fully exploit SSD potential, the architecture of in-/near-storage computing had been proposed in the modern SSD device [8], [9], and this paper terms it as the computational SSD (abbreviated as *CompSSD*). The main idea is to treat a such SSD device as a small compute node with ARM or other embedded processors to execute data-intensive tasks on the storage device.

The SSD device consists of an on-board DRAM buffer, and a portion of this memory buffer is used as the data cache of SSD devices [10]. Cache management mainly focuses on the replacement strategy, that evicts a number of buffered data pages for making space for the new data. It is true that existing data cache management policies for native SSD devices can be naturally used to manage the cached data in *CompSSDs*, but they may not work efficiently for the fairness of cache use and the overall I/O performance. This is because the embedded cache of *CompSSD* devices should hold not only the output data of user tasks running at the host side, but also the input data of tasks executing at the SSD side.

To address the issue of cache management in *CompSSD*, this paper proposes an adaptive cache division scheme. It can dynamically split the cache space and separately buffer the output data and the input data while both the host and the *CompSSD* device are dealing with user tasks. In brief, this paper makes the following contributions:

- We propose an adaptive division scheme of data cache, for enhancing cache use efficiency and then accelerating the execution of user tasks in the scenarios of *CompSSDs*. Our approach dynamically splits the cache space for separately keeping the output data of the tasks running at the host side and the input data of the tasks running at the SSD side.
- We construct a mathematical model that runs at FTL of *CompSSD*, to periodically decide the private cache space for the workloads running at the host and *CompSSD*, by considering the factors of ratios of read/write and cache hits of currently running tasks, as well as the overhead of data transfer between the host and *CompSSD*.
- We perform simulation based experiments by replaying the grouped task sets that are coupled with commonly used block I/O traces of real world applications, in the scenarios of *CompSSD*. As our measurements indicate, the proposed cache division scheme can improve the overall I/O latency by between `0.9%` and `84.5%`, in contrast to existing cache management schemes.

*Corresponding author: J. Liao is with the College of Computer and Information Science, Southwest University of China, Chongqing, China, 400715. E-mail: *liaotoad@gmail.com*.

979-8-3503-4859-0/DATE24/© 2024 EDAA

The rest of paper is organized as follows: Section II depicts background on *CompSSDs* and cache management for conventional SSDs, as well as our objectives. The specifications on our approach are presented in Section III. Section IV describes the evaluation experiments and relevant discussions. At last, the paper is concluded in Section V.

II. Background and Objectives

A. Computational SSDs

The computational SSD model was firstly introduced (originally named as *SmartSSD* in the paper) by Kang et al. [11], which allows host systems to fully exploit the performance of SSDs, through offloading I/O-intensive tasks from the host to the SSD device. Specifically, the *CompSSD* devices have an internal execution engine for processing locally-stored data, thus eliminating the burden of data transfer from the interconnect between the host and the storage system, and reducing the execution time of a batch of user applications in the end. Consequently, a number of in-/near-storage accelerators have been proposed on the top of *CompSSD*, for speeding up various kinds of application scenarios, such as database sort [12], search engines [3], and graph-based processing [13].

B. Cache management in SSDs

Prevalent cache management schemes, including first in first out (*FIFO*), least recently used (*LRU*), least frequently used (*LFU*) [14], and their advanced variations, have been widely deployed in various computing systems. The data cache management policies that especially targeting at conventional SSDs, employ the information on requests' recency or frequency to identify valuable data pages or blocks for cache replacement [15], [16]. In general, existing cache management schemes of SSDs can be classified as block-based and page-based types, by referring to how the cache pages are managed [17]. Specifically, all the pages associating with the same logical block are grouped together in a block while carrying out cache replacement [18], whereas all cached pages are treated individually and each page is evicted on a replacement in the case of using a page-based policy [19], [20].

C. Objectives

Existing cache management schemes in conventional SSDs can be directly applied in the contexts of *CompSSD*, but they are not purposely designed for *CompSSD* and fail to consider the difference between the cached output data and the cached input data belonging to the tasks running at the host side and the SSD side. As a result, the use efficiency of cache in *CompSSD* becomes limited, thus impacting the completion of all user tasks. To address this issue, we propose a cache management scheme for *CompSSD* devices, to separately buffer the output data and the input data, with the supports of dynamic cache division for both types of data according to I/O workload characteristics of running tasks.

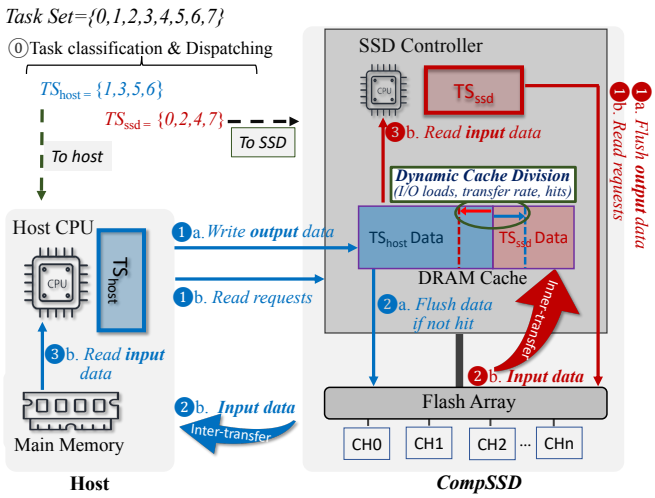

Fig. 1. Architectural overview of adaptive cache division for *CompSSD*. Note that identifying the type of tasks and scheduling them to either the host or the *CompSSD* device are not the aim of this paper, as we manage the DRAM cache of *CompSSD* after the tasks have been already dispatched. Besides, we assume that both the host and *CompSSD* do have required resources for computations, and I/O is the bottleneck of the system.

III. Adaptive Cache Division for *CompSSDs*

A. Architectural Overview

Figure 1 illustrates a high-level overview of *CompSSD* with cache division, in which 8 user tasks are needed to be processed. As seen, some compute-intensive tasks (i.e., the set of TS_{host}) have been dispatched at the host, whereas certain I/O-intensive tasks (i.e., the set of TS_{ssd}) are scheduled on the *CompSSD* device, and the goal is to complete these two batches of tasks with small time overhead. On the one hand, compute-intensive tasks read the input data the main memory from the SSD device, and the computation outputs will be buffered in the cache inside SSD. Once the cache becomes full, cache replacement is triggered and the replaced data is flushed onto SSD to make room for the new write data. On the other hand, I/O intensive tasks can quickly read the input data with inner transfer, and buffer them in the cache locally, while the output data can be directly flushed to the flash array.

Our proposed scheme of adaptive cache division is included in Figure 1, and our basic idea is to adaptively split the DRAM cache space into two parts. Then, we can separately buffer the output data of compute-intensive tasks, and the input data of I/O-intensive tasks, to speed up I/O processing for completing all the tasks. Considering both the tasks in both sets may have varied I/O characteristics, our approach supports adaptively adjusting cache division, by referring to the factors of the I/O workloads of tasks, the recent cache hits, and the data transfer rate, for enhancing cache use efficiency.

B. Cache Division Model

To adaptively determine how much cache space is dedicated for buffering either the output data or the input data associating with user tasks, we construct a mathematical model for cache

TABLE I
NOTATION DESCRIPTIONS IN THE MODEL

Notation	Explanation Description
T_{sys}	Overall time of completing all system tasks
T_h/T_s	Time cost of completing tasks running on **h**ost or *CompSSD*
R_h/R_s	The amount of data being read on host or *CompSSD*
W_h/W_s	The amount of writes on host or *CompSSD*
H_h/H_s	Cache hit ratio if all cache is allocated to host or *CompSSD*
T_{tr}	Time cost for inter-transferring a page from *CompSSD* to host
T_{rd}	Time cost for reading a page from *CompSSD*
T_{wr}	Time cost for writing a page to *CompSSD*
K	The ratio of cache occupied by tasks running on host

division. By referring to the notations reported in Table I, this section depicts the proposed cache division model in details.

The overall time cost of system tasks can be defined as the maximum value of the time cost of the tasks that are scheduled at the host side and the time cost of tasks that are dispatched running on *CompSSD*, as illustrated in Equation 1.

$$T_{sys} = MAX\{T_h, T_s\} \tag{1}$$

The time cost of completing the tasks running at the host side is defined in Equation 2, that consists of three parts. It includes the data transfer time between *CompSSD* and the host, the time needed for reading data from *CompSSD*, and the time required for writing data to *CompSSD*. Specifically, the amount of data written to the flash array determines the write time overhead, which is related to the size of allocated cache space (i.e., K) and the hit ratio for the host size if all the cache (i.e., H_h).

$$T_h = T_{tr} \times (R_h + W_h) + T_{rd} \times R_h + T_{wr} \times \frac{1}{K} \times (1 - H_h) \times W_h \tag{2}$$

Equation 3 defines the time overhead of completing the tasks running on *CompSSD*, which does not require performing data transfers between the host and *CompSSD*.

$$T_s = T_{rd} \times \frac{1}{(1-K)} \times (1 - H_s) \times R_s + T_{wr} \times W_s \tag{3}$$

Because we have a fixed size of data cache, T_h will become smaller if more cache space is allocated for the task running at the host, meanwhile T_s will increase. By referring to Equation 1, we can yield the minimum value of T_{sys} *if-and-only-if* $T_h = T_s$. Then, if a value of K^* can be found for satisfying $T_h = T_s$, and we suggest that K^* is the optimal solution to yield the minimal value of overall completion time. Consequently, we build a function in Equation 4 to help obtaining the optimal ratio of cache division.

$$f(K) = T_h - T_s \tag{4}$$

As seen in Equation 4, $K \to 0^+$, $f(K) \to +\infty$, meanwhile $K \to 1^-$, $f(K) \to -\infty$, and the function of $f(K)$ is monotonically decreasing within the interval of $(0, 1)$. Thus, there must exist a unique value of K^* within the interval of $(0, 1)$ to satisfy $f(K^*) = 0$, and which can be decided if the relationship between the different values of K and the cache hits (i.e., H_h and H_s) is obtained in practical applications.

Algorithm 1 Cache management with adaptive division

Require: *En-queued I/Os (#0, #1, ...), cur_k, CACHE_SIZE.*
Ensure: *All requests have been serviced.*

```
1:  data_amount ← 0;
2:  while reqId = 0, 1, 2, ... do
3:      data_amount+=get_size(reqId);
4:    /*periodically calculate the optimal division ratio*/
5:      if data_amount ≥ CACHE_SIZE then
6:          H_h ← get_global_host_hits();
7:          H_s ← get_global_SSD_hits();
8:          K* ← get_kstar(H_h, H_s);
9:          data_amount ← 0;
10:     end if
11:     if cache hit then
12:         move_node_to_head(); //with LRU manner
13:     else
14:       /*cache replacement to approach the optimal ratio*/
15:         if cache is full then
16:             if cur_k < K* then
17:                 evict_input_data();
18:             else
19:                 evict_output_data();
20:             end if
21:           /*update the current division ratio*/
22:             update_cur_k();
23:         end if
24:         insert_new_node();
25:     end if
26: end while
```

C. Implementation Details

We periodically collect features of I/O workloads running at the host and the *CompSSD* in history, and then use the proposed model to calculate the optimal division ratio of DRAM cache, which will be used to direct cache replacement in the following time window. Algorithm 1 depicts the implementation details on cache management with adaptive cache division. In the process of servicing I/O requests from the host and *CompSSD*, we periodically calculate the optimal ratio of cache division in cache management, when the total amount of accessed data becomes larger than the size of data cache since the last time window, by referring to [20].

As read, *Lines* `6-7` demonstrate the function of calculating the hit ratios at the host and the *CompSSD*, assuming all cache space is allocated for their tasks. To this end, we employ two ghost cache lists to obtain the cache hit numbers on the condition of all cache space is used for buffering the data of tasks running on either the host or *CompSSD*, even though less data pages are actually buffered in the cache, by referring to [21]. After that, *Line* `8` shows deciding the optimal division ratio of the data cache on *CompSSD*, equaling to the condition of $f(K)$ in Equation 4 becomes `0`. *Lines* `13-25` present the

TABLE II
EXPERIMENTAL SETTINGS OF *MQSim*

SSD parameters	
SSD Host Interface	PCIe-3.0 (NVMe-1.2)
Inter Transfer Rate (PCIe)	1GB/s
(Channel, Chip)	(8,2)
(FTL, GC Threshold)	(Page-level, 10%)
DRAM Capacity	128M
DRAM Access Latency	1us
Chip parameters	
(Die, Plane, Block, Page)	(1, 1, 1024, 1024)
(Page Size, Cell Density)	(8KB, QLC)
Read Latency (LSB-...-MSB)	(90, 120, 150, 180)us
(Program Latency, Erase Latency)	(1.3ms, 10ms)

TABLE III
SPECIFICATIONS ON BENCHMARKS

Task Type	I/O Trace	Write Ratio	Write Amount	Read Amount
Computing Tasks ($TS_{\underline{h}ost}$)	prxy_0 (*h0*)	0.97	2.48GB	0.11GB
	prn_0 (*h1*)	0.89	2.37GB	0.73GB
	wdev_0 (*h2*)	0.79	0.29GB	0.12GB
	web_0 (*h3*)	0.70	0.51GB	1.01GB
I/O Tasks ($TS_{\underline{s}sd}$)	lun0 (*s0*)	0.49	1.01GB	2.27GB
	lun1 (*s1*)	0.40	1.14GB	2.62GB
	lun2 (*s2*)	0.38	1.20GB	3.62GB
	lun3 (*s3*)	0.31	1.31GB	3.91GB

Note: *Task sets* used in evaluation are mixed with two types of tasks: TS_0={*h0, h1, s0, s1*}, TS_1={*h1, h2, s1, s2*}, TS_2={*h2, h3, s2, s3*}, TS_3={*h3, h0, s3, s0*}, TS_4={*h0, h1, h2, s0, s1, s2*}, TS_5={*h1, h2, h3, s1, s2, s3*}, TS_6={*h0, h1, h3, s0, s1, s3*}, TS_7={*h0, h2, h3, s0, s2, s3*}, TS_8={*h0, h1, h2, h3, s0, s1, s2, s3*}.

information on cache replacement, by referring to the optimal ratio and the current ratio of cache division.

IV. EXPERIMENTS AND DISCUSSIONS

A. Experimental Settings

We evaluate our proposal using the trace-driven SSD simulator of *MQSim*, that supports modeling the performance of nowadays multi-queue SSDs [22]. Table II lists our experiment settings. The DRAM cache size is typically 0.1% of the SSD capacity [23], indicating 128MB cache in our configuration.

We select eight block I/O traces of real applications from the Microsoft Research Cambridge collection [24] and the VDI trace collection [25], for carrying out trace-driven experiments. Specifically, the used VDI traces are *additional-01-2016021715-LUN0&LUN1*, *additional-01-2016021716-LUN0&LUN1*, *additional-01-2016021717-LUN0&LUN1*, and *additional-01-2016021714-LUN0&LUN1*, labelling as *lun0* to *lun3* respectively. We then classify these traces into two types of computing-intensive tasks and I/O-intensive tasks according to the applications' features. Then, we group a number of two-type tasks into two sub-sets, and schedule them running at the host and *CompSSD* separately in each scheduling round.

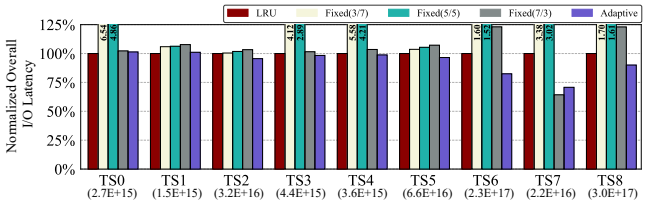

Fig. 2. The results of overall I/O latency after running the selected grouped task sets. *Note that the numbers underlying X-axis are the absolute values with the baseline of LRU.*

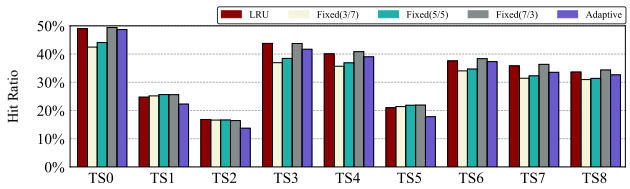

Fig. 3. Cache hits after running the grouped task sets.

Table III reports the information on the selected I/O traces and the grouped cases of task sets in evaluation. As seen, a *write ratio* of 0.97 indicates write requests are 97% of the total number of I/Os, and a *write/read amount* of 2.48GB/0.11GB respectively represents the footprint of write/read operations.

Apart from the proposed *Adaptive* scheme, another **two** methods are also implemented in comparison evaluation:

- *LRU*, which is the baseline cache management approach. It manages all cached data with same rules, regardless of their properties of output/input data.
- *Fixed Cache Division*, which fixedly divides the cache space into two parts, for separately buffering the output data and the input data. In order to check the effectiveness of varied division ratios, we set the percentage of cache space used for the output data of host tasks, as 30.0%, 50.0%, and 70.0% of the total cache space, indicating the rest cache space is dedicated for the input data, labelling as *Fixed(3/7)*, *Fixed(5/5)*, *Fixed(7/3)*.

Note that our proposal can be deployed in page- and block-based replacement schemes, and our evaluation configured all selected schemes using page-based replacement by default.

B. Performance Results

1) Overall I/O Latency: The overall I/O latency is the primary performance indicator in the context of *CompSSD*, which has been defined in Equation 1. We recorded the overall I/O latency after running the grouped task sets, and Figure 2 shows the normalized results. As seen, our proposed *Adaptive* method can yield the best I/O performance in all grouped task sets. More exactly, *Adaptive* can decrease the overall I/O latency by between 0.9% and 84.5% (27.5% on average), when comparing to *LRU*, *Fixed(3/7)*, *Fixed(5/5)*, *Fixed(7/3)*. This is because the objective of *Adaptive* is to expedite the completion of all tasks running on both the host side and the *CompSSD* side, by adaptively enlarging the cache space for the tasks that currently are slowly served. As a result, our

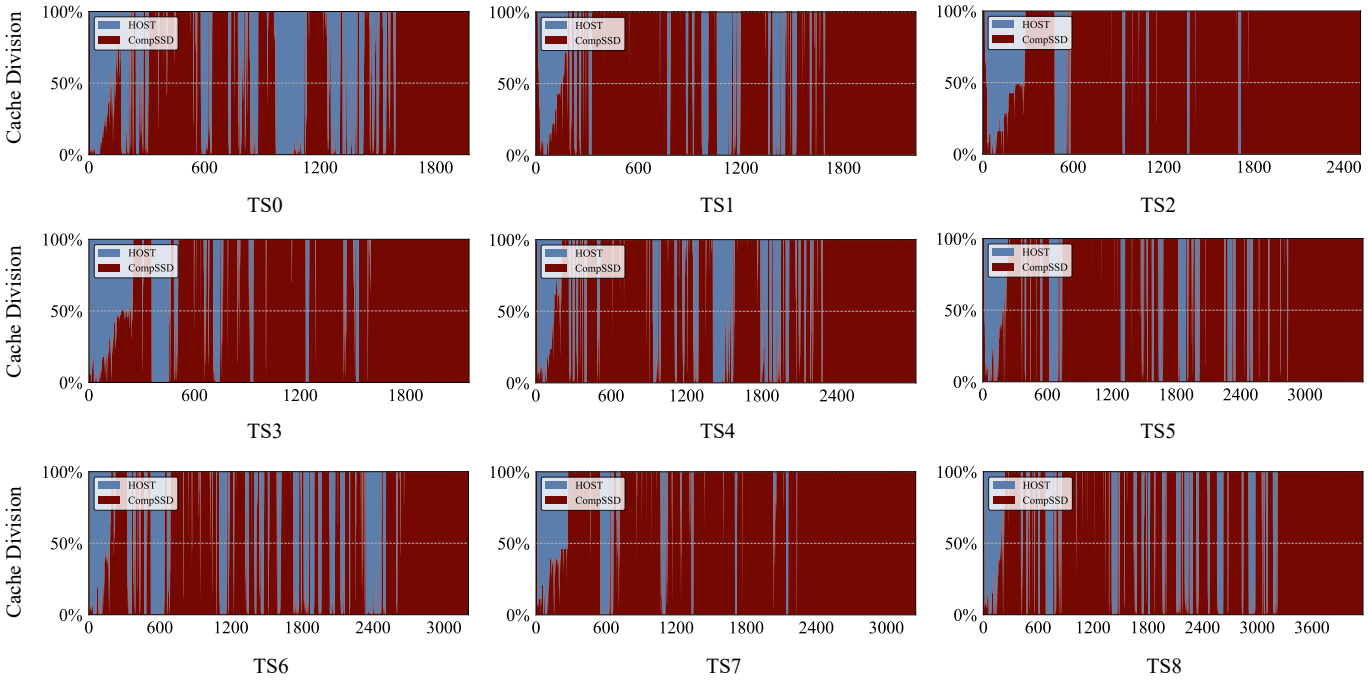

Fig. 4. Cache division status for both types of tasks running at the host and *CompSSD*, with the proposed *Adaptive* scheme.

approach can yield a small value of the overall completion time of all tasks in evaluation.

Another noticeable information is that, the related schemes of *Fixed(3/7)*, *Fixed(5/5)* and *Fixed(7/3)* reveal worse overall I/O latency, in contrast to the baseline of *LRU*. This is because the fixed cache division policies do not consider differences of I/O workloads of user tasks, and fail to efficiently utilize the cache space in an adaptive manner, meanwhile *LRU* aims to keep the frequently accessed data in the cache no matter the output data or the input data.

2) Cache Hits: The measure of cache hit defines the ratio of the pages from I/O requests that can be absorbed by the cache, without accessing to underlying flash memory. In the scenario of conventional SSD devices, a higher cache hit ratio means less access to underlying flash arrays, which generally contribute to better I/O performance. This section will check whether such principles exist in the context of *CompSSD*.

Figure 3 presents the results of cache hits after replaying the selected task sets in *CompSSD*. The interesting clue shown in the figure is about, *LRU* yields the best cache hits in all cases, since it always keeps the hot accessed data in the cache, but it fails to achieve the smallest values of I/O latency, implying a larger value of cache hits does not result in better I/O performance. We suggest that this is because the write hit and the read hit in the cache will make varied levels of impacts on I/O performance when running different batches of task sets in the context of *CompSSD*. Thus, **we suggest** that the metric of cache hit should not be treated as the primary indicator of I/O performance when running a part of user tasks in *CompSSD*.

3) Cache Division Statistics: As discussed, the fixed cache division schemes, including *Fixed(3/7)*, *Fixed(5/5)* and *Fixed(7/3)* employ unchanged cache division during the whole lifetime of running the grouped task sets. Thus, they may fail to make advantage of cache space for speeding up the execution of user tasks that may have varied requirement of cache space at different execution stages. On the other side, our proposal intends to divide cache space into two parts in a periodical manner according to the particular features of current I/O workloads, for separately buffering the output data of the host side tasks and the input data of the *CompSSD* side tasks.

Figure 4 presents the results of cache division status when running the grouped task sets at different time windows. In the figure, the X-axis represents the sequence of time window, and Y-axis means the cache distribution in percentage. Because different benchmarks have varied total execution time, the length of time windows varies from case to case. As shown, our *Adaptive* method dynamically adjusts cache space division for both types of user tasks. Specifically, one type of user tasks, taking the tasks running at the host side for example, may expect cache space in a concentrated manner at a specific time slot, resulting in a major part of cache space is used by such tasks, whereas the cache space is mostly allocated for the tasks running at the *CompSSD* side at another time slot. In other words, our approach can adaptively split the cache space for both types of user tasks, according to the I/O features of the currently running tasks at host and *CompSSD*. As a result, the overall I/O latency can be noticeably reduced, which had been described in Section IV-B1.

C. Overhead Analysis

With respect to the time overhead, the proposed *Adaptive* method does require to additionally compute the division ratio of cache space for different types of task sets. We emphasize that it brings about negligible time cost, as our model decides the ratio with a time complexity of $O(1)$.

The main memory overhead of our proposal is due to the additional storage space required for keeping the parameters used by the space division model. In details, our proposal requires maintaining two ghost linked lists to estimate the cache hit numbers assuming all cache space is dedicated for either the tasks running at the host side or the tasks running at the *CompSSD* side. In other words, it consumes 80KB= 5B/node$\times$(64MB/8KB nodes)$\times 2$, for holding all nodes of two ghost lists, with the cache configuration of 64MB. In addition, the model has to record the amount of read and write data of user tasks on both the host side and the SSD side, which translates to 64B = 8 channels$\times$(2 counts$\times 4$B). Then, we summarize that the space overhead caused by our proposal accounts for a very small part of the storage capacity of SSD, resulting in an acceptable amount of memory space in SSDs.

V. Conclusion

This paper has proposed an adaptive DRAM cache division scheme for *CompSSDs*. It can dynamically split the cache space for separately buffering the output data and the input data associating with the tasks running at the host side and the SSD side. To this end, we have built a mathematical model that runs at the flash translation layer of SSDs, to periodically determine the cache proportion of the host side tasks and the SSD side tasks, by considering the ratios of read/write and the cache hits of applications, as well as the overhead of data transfer between the storage device the host. In other words, both output data and input data will be buffered in their own private cache parts, thus ensuring the fairness of cache use, to eventually reduce the overall I/O latency.

Through a series of simulation tests based on several real-world disk traces, we have illustrated that our proposal can noticeably cut down the overall I/O latency by up to 84.5% after running the grouped task sets, comparing to the state-of-art cache management schemes for SSD devices.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments. This work was partially supported by "National Natural Science Foundation of China (No. 61872299, No. 62032019)", "Natural Science Foundation Project of CQ CSTC (No. cstc2021ycjh-bgzxm0199, 2022NSCQ-MSX0789)"

References

[1] Micheloni R. Solid-state drive (SSD): A nonvolatile storage system. *Proceedings of the IEEE **(PIEEE)***, 2017.

[2] Jaffer S., Maneas S., Hwang A., and Schroeder B. Evaluating file system reliability on solid state drives. *In Proceedings of USENIX Annual Technical Conference **(USENIX ATC)***, 2019.

[3] Wang J., Park D., Papakonstantinou Y., and Swanson S. SSD in-storage computing for search engines. *IEEE Transactions on Computers **(TC)***, 2016.

[4] Balasubramonian R., Chang J., and Manning T., et al. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro **(Micro)***, 2014.

[5] Do J., Sengupta S., and Swanson S. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM **(CACM)***, 2019.

[6] Cosmos OpenSSD Platform. *http://www.openssd-project.org.*

[7] Xu X., Cai Z., Liao J., and Ishiakwa Y. Frequent access pattern-based prefetching inside of solid-state drives. *In Proceedings of Design, Automation, and Test in Europe **(DATE)***, 2020.

[8] Do J., Kee Y. S., and Patel J. M., et al. Query processing on smart SSDs: Opportunities and challenges. *In Proceedings of ACM SIGMOD Conference **(SIGMOD)***, 2013.

[9] Lee J. H., Zhang H., and Lagrange V., et al. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Computer Architecture Letters **(CAL)***, 2020.

[10] Tripathy S., and Satpathy M. SSD internal cache management policies: A survey. *Journal of Systems Architecture **(JSA)***, 2022.

[11] Kang Y., Kee Y. S., Miller E. L., and Park C. Enabling cost-effective data processing with smart SSD. *In Proceedings of IEEE Conference on Mass Storage Systems and Technologies **(MSST)***, 2013.

[12] Salamat S., Zhang H., Ki Y., and Rosing T. NASCENT2: Generic near-storage sort accelerator for data analytics on SmartSSD. *ACM Transactions on Reconfigurable Technology and Systems **(TRETS)***, 2022.

[13] Kim J. H., Park Y. R., and Do J., et al. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Transactions on Computers **(TC)***, 2023.

[14] Lee D., Choi J., and Kim J., et al. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *In Proceedings of International Conference on Measurement and Modeling of Computer Systems **(SIGMETRICS)***, 1999.

[15] Lee D., Choi J., and Kim J., et al. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers **(TC)***, 2001.

[16] Chai Y., Du Z., Qin X., and Bader D. A. WEC: Improving durability of SSD cache drives by caching write-efficient data. *IEEE Transactions on Computers **(TC)***, 2015.

[17] Lin H., Li J., and Sha Z., et al. Adaptive management with request granularity for DRAM cache inside NAND-based SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **(TCAD)***, 2023.

[18] Wang M., and Li Z. A spatial and temporal locality-aware adaptive cache design with network optimization for tiled many-core architectures. *IEEE Transactions on Very Large Scale Integration Systems **(VLSI)***, 2017.

[19] Wu S., Lin Y., and Mao B., et al. GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs. *In Proceedings of ACM International Conference on Supercomputing **(ICS)***, 2016.

[20] Sha Z., Cai Z., and Trahay F., et al. Unifying temporal and spatial locality for cache management inside SSDs. *In Proceedings of Design, Automation, and Test in Europe **(DATE)***, 2022.

[21] Megiddo N., and Modha D. ARC: A Self-Tuning, low overhead replacement cache. *In Proceedings of USENIX Conference on File and Storage Technologies **(FAST)***, 2003.

[22] Tavakkol A., Gómez-Luna J., and Sadrosadati M., et al. MQSim: A framework for enabling realistic studies of modern Multi-QueueSSD devices. *In Proceedings of USENIX Conference on File and Storage Technologies **(FAST)***, 2018.

[23] Liu K., Wu K., and Wang H., et al. SLAP: An adaptive, learned admission policy for content delivery network caching. *IEEE International Parallel and Distributed Processing Symposium **(IPDPS)***, 2023.

[24] Narayanan D., Thereska E., and Donnelly A., et al. Migrating server storage to SSDs: analysis of tradeoffs. *In Proceedings of European Conference on Computer Systems **(EuroSys)***, 2009.

[25] Lee C., and Matsuki T., et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. *In Proceedings of International Systems and Storage Conference **(SYSTOR)***, 2017.