Bulletin of the Technical Committee on

Data Engineering

June 2014 Vol. 37 No. 2

IEEE Computer Society

Letters

Special Issue on Adapting Database Systems to Flash Storage

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Changing Editors

One of the practices I inherited from Won Kim, when I became Bulletin Editor-in-Chief, was the practice of appointing four associate editors for two year terms. This provides a continuing string of issues with fresh content and enthusiastic editors. This means that every two years, I bid farewell to current editors and introduce new editors. Selecting editors is the most important part of what I do. And I am proud of my success in being able to enlist distinguished members of the database community as editors.

This issue marks the end of the terms of current editors, as each has now contributed two issues. I want to thank all of them for continuing the Bulletin's success in producing high quality, insightful, and timely issues on a compelling set of diverse topics. So thank you Juliana Freir, Paul Larson, Sharad Mehrotra, and Sudarshan for your very successful issues.

Let me now introduce the new associate editors who will be carrying the "Bulletin torch" for the next two years. They are Chris Jermaine of Rice University, Betina Kemma of McGill University, David Maier of Portland State University, and Xiaofang Zhou of the University of Queensland. I am delighted to be working with this new set of great editors- who I am confident will ensure that the Bulletin, and its readers, continue to prosper.

The Current Issue

It is now widely recognized that the underlying hardware upon which database systems are built has changed radically over the past few years. Modern processors are very different from earlier generations. And, the subject of the current issue, modern secondary storage devices are increasingly based on flash storage.

Flash is, in some respects, very different from hard disks, but in some respects it returns us to an earlier architectural age. Access latency and access rates are very substantially better than provided by hard disks. Interestingly, when juxtaposed with processor speed, flash's enhanced performance brings us back to the "relative" performance of hard disks vs older processors.

Because flash has different characteristics than hard disks, however, dropping flash into a current database system is very wide of the mark in using flash in a most effective manner. Flash writes are expensive compared to reads, and particularly random writes are very expensive. Flash also wears out, and requires strategies to extend its effective lifetime. Further, there is performance to be gained if the internal characteristics of flash solid state disks (SSDs) can be exploited.

The current issue provides a cross section of techniques being explored within the data management research community to exploit the unique properties of flash storage. It provides a very nice introduction to this increasingly important area. I want to thank Paul Larson for assembling this very timely issue, timely because flash storage will only get more important and pervasive as time goes on, whether deployed on user premises or in the cloud.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Relational database systems were designed assuming data would be stored on hard disks (HDD) which, at the time, was the only realistic choice. Over the years database systems have been optimized for HDD storage. Solid state disks (SSD) based on flash storage have now become an attractive alternative, offering much higher performance albeit at higher prise per GB. Simply replacing HDDs by faster SSDs can produce impressive performance improvements but a database system optimized for HDDs may not fully exploit the capabilities of SSDs. SSDs have different characteristics and capabilities than HDDs. For example, on SSDs writes are much more expensive than reads. So how should database systems adapt to fully exploit the capabilities of flash SSDs while also avoiding some of their limitations? The papers in this special issue explore different aspects of this challenge.

The first three papers explore ways to exploit the substantial internal parallelism of a flash SSD. The paper by Roh et al on MPSearch proposes a way to speed up individual B-tree operations by exploiting the fact that a single device can process multiple I/O request concurrently. The key idea is to submit a batch of read or write requests at once instead of one at a time. Their experimental results show significant speed-up on range searches and inserts.

In the second paper, Fan et al propose a striped layout of pages such that pages from different stripes are accessed through different internal channels of the SSD. This layout makes it possible to scan different stripes in parallel. They then show how this can be exploited to speed up scans, aggregation, joins, and sorts.

Smart SSDs with a programmable processor inside the device are on the horizon. This makes it possible to push some processing down into the device, leverage its internal parallelism, and reduce the amount of data returned to the host. The paper by Park et al explores the opportunities and challenges arising from this functionality. Their results show significant speedups for highly selective queries.

Flash SSDs do not perform updates in place; a modified page is always written to a new physical page. This property has primarily been seen as a challenge to overcome but this too provides opportunities for improvement. A flash device essentially implements shadow paging - after a write, the old copy of the page still remains. The paper by Kang et al proposes to take advantage of this property by delegating the responsibility for ensuring transactional atomicity to the device. Their experimental results show greatly reduced transaction latency compared with the traditional approaches currently used by SQLite.

SSDs are increasingly being used as a persistent second-tier cache, that is, as an extension of the buffer pool. This raises the question of how to most efficiently manage such a two-level cache, which is the issue explored in the paper by Liu and Salem. An easy solution is to manage the two caches completely independently of each other but, as shown in the paper, this is not best strategy. They introduce an integrated strategy and experimentally show that this improves the overall cache performance.

I would like to extend a heartfelt thanks to the authors for their contributions to this issue.

Happy reading! I trust that you will find the papers as interesting as I did.

Per-Ake Larson
Microsoft Corporation

MPSearch: Multi-Path Search for Tree-based Indexes to Exploit Internal Parallelism of Flash SSDs

Hongchan Roh, Sanghyun Park, Mincheol Shin, and Sang-Won Lee
{fallsmal, sanghyun, smanioso}@cs.yonsei.ac.kr, swlee@skku.in

Abstract

Big data real-time processing aims for faster retrieval of data and analysis. Lately, in order to accelerate real-time processing, big data platforms are trying to exploit NAND flash based storage devices, especially SSDs. NoSQL DBMSs have been used for real-time management of big data which significantly depends on index structures to efficiently manage data. Previous research about flash-aware index structures addressed the potential problems of hard-disk oriented designs. In this paper, we focus on exploiting potential benefits of flash SSDs. First, we examine the internal parallelism of flash SSDs by benchmarking several flash SSDs. Then we present a new I/O request concept, called psync I/O, that can exploit the internal parallelism of flash SSDs in a single process, and we propose a new search method (MPSearch) that enables tree based indexs to exploit the internal parallelism of flash SSDs. Based on MPSearch, we present a B+-tree variant, PIO B-tree (Parallel I/O B-tree). PIO B-tree enhanced B+-trees insert performance by a factor of up to 16.3, while improving point-search performance by a factor of 1.2. The range search of PIO B-tree was up to 5 times faster than that of the B+-tree. Moreover, PIO B-tree outperformed other flash-aware indexes in various synthetic workloads. In order to enhance NoSQL DBMS performance on flash SSDs, PIO B-tree can be adopted or MPSearch can be applied to other tree-based index structures adopted in NoSQL DBMSs.

1 Introduction

Big data real-time processing aims for faster retrieval of data and analysis than previous MapReduce based batch processing. Lately, in order to accelerate real-time processing, big data platforms are trying to exploit flash SSDs. NoSQL DBMSs such as HBase [1], Cassandra [2], MongoDB [3], CouchDB [4] have been used for real-time data management of big-data which significantly depends on index structures to efficiently manage data. The NoSQL DBMSs rely on efficient index structures such as B+-tree and LSM tree [5] (LSM tree for HBase and Cassandra, B+-tree for CouchDB and MongoDB).

The excellent IOPS performance of flash SSDs is due to their internal parallel architecture. Since flash SSDs embed multiple flash memory packages, it is possible for a flash SSD to achieve much higher IOPS (Input/Output Operations Per Second) than a flash memory package. However, the outstanding random I/O performance of flash SSDs will remain only a potential performance specification unless DBMSs take advantage of the internal parallelism and fully utilize the high IOPS.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Several flash-aware (flash-memory aware) B+-tree variants have been proposed. The B+-tree index is a good example of how to resolve DBMS issues with regard to flash-based storage devices. Flash-aware B+-tree variants and flash-aware indexes mostly focused on reducing write operations caused by index-insert operations [6][7] or utilizing sequential pattern benefits [8].

The purpose of this paper is to examine principles of taking advantage of the internal parallelism of flash SSDs and to optimize the B+-tree index by applying these principles. In this paper, we first present benchmark results on various flash SSDs and known characteristics of the flashSSD parallel architecture. Then we propose a new search method (MPSearch) that enables tree-based indexes to exploit the internal parallelism of flash SSDs. Based on MPSearch we introduce new algorithms and present a B+-tree variant, PIO B-tree (Parallel I/O B-tree), that integrates the new algorithms into the B+-tree (This paper is an extended version of [9]. In this paper we more focus on MPSearch algorithm and discuss its potential usage in other indexes).

This paper is organized as follows. Section 2 describes the internal parallelism and principles to utilize it together with the benchmark results on various flash SSDs. We present the MPSearch algorithm and introduce PIO B-tree in Section 3. Section 4 describes experimental results and we conclude this paper in Section 5.

2 Internal Parallelism of SSD

2.1 Understandings of Internal Parallelism

Figure 1 shows the internal architecture of a flash SSD. Flash SSDs are configured with multiple flash memory packages. Flash SSDs implement the internal parallelism by adopting multiple channels each of which is connected to a group of flash memory packages. There exist two types of internal parallelism such as channel-level parallelism between multiple channels and package-level parallelism between ganged flash memory packages in a group [10]. The performance enhancement can be estimated by the factors of channel-level and package-level parallelism. If there are m channels each connected to a gang of n flash memory packages, the performance gain can be up to mn times compared to the performance of a flash memory package.

If the host I/F (host interface) requests I/Os targeting different flash memory packages spanning several channels, channel-level parallelism is achieved by transferring the associated data through multiple channels at the same time. In this process, command queuing mechanisms (NCQ, TCQ) of the host I/F involve in producing favorable I/O patterns to the channel-level parallelism. The host I/F swaps the queued I/O operations and adjusts the orders of the I/Os in order to make the I/O requests target flash memory pages spanning multi channels. Based on the understandings of channel-level parallelism, it is a reasonable inference that the flash SSD performance can be enhanced by requesting multiple I/Os simultaneously.

We examined the performance effects of the internal parallelism through benchmark tests on six different flash SSDs. All the benchmarks were conducted in direct I/O mode. We carefully chose these flash SSDs to examine parallelism issues with as many SSD internal architectures as possible. For the tests we used micro-benchmarks with varying number of random I/Os requested at once (outstanding I/O level) created by using Linux-native asynchronous I/O API (libaio). Figure 2 (a) and (b) present the benchmark results when read and write operations are separately requested with I/O size fixed at 4KB. The read and write bandwidth was gradually enhanced with increasing outstanding I/O level (in short, OutStd level). We confirmed more than ten-fold bandwidth enhancement by increasing OutStd level in both read and write operations compared to the read and write bandwidth with the OutStd level of 1.

2.2 How to Utilize Internal Parallelism

In order to utilize channel-level parallelism, multiple I/O requests should be submitted to flash SSDs at once. Parallel processing is a traditional method to separate a large job into sub-jobs and distribute them into multiple

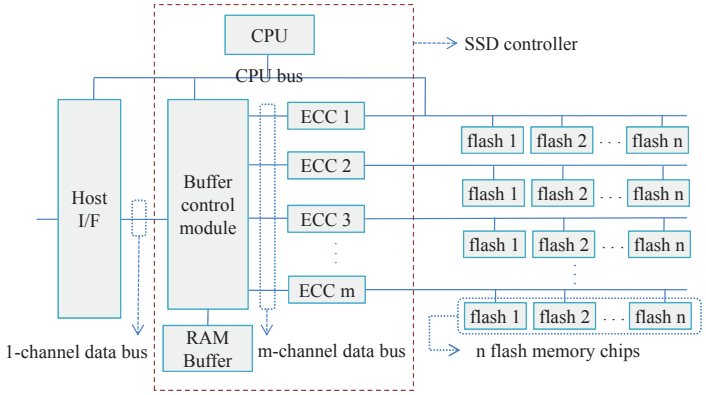


Figure 1: Internal architecture of flash SSDs

CPUs. Since I/Os of each process can be independently requested to OS at the same time, outstanding I/Os (multiple parallel I/Os) can be delivered to a flash SSD at the same moment.

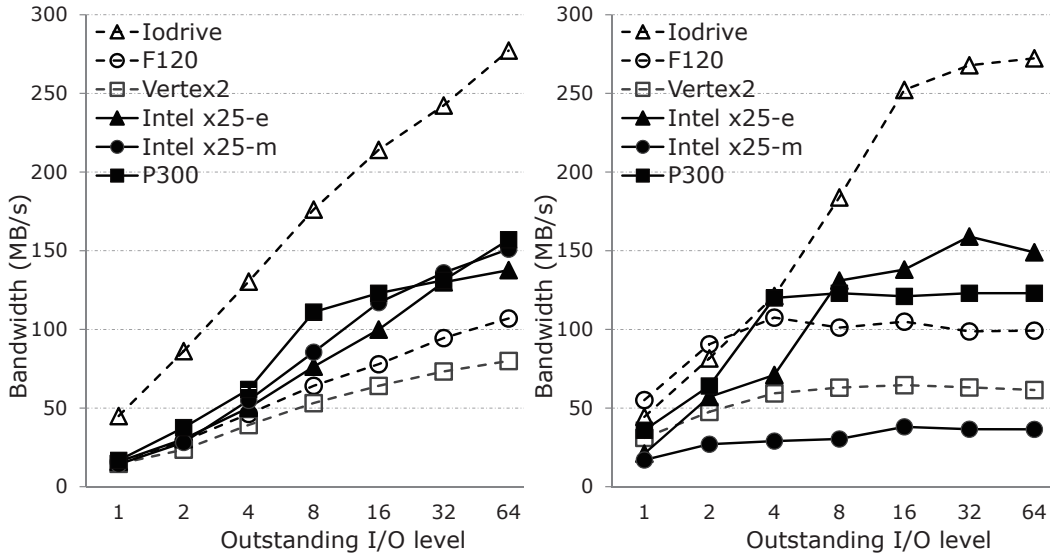
Due to the high performance gain from channel-level parallelism on a single flash SSD, we need to treat I/O parallelism as a top priority for optimizing I/O performance. In order to achieve it, a more lightweight method is needed since parallel processing (or mutlithreading) cannot be applied in every application programs involving I/Os. In order to best utilize channel-level parallelism, it is also required to deliver the outstanding I/Os to the flash SSDs minimizing the interval of consecutive I/O requests since inside flash SSDs they can batch-process only the I/O requests gathered in its own request queue (a part of NCQ technology) within a very narrow time span. Therefore, we suggest a new I/O request method, psync I/O, that creates outstanding I/Os and minimize the interval between consecutive I/O requests within a single process.

2.3 Psync I/O: Parallel Synchronous I/O

Two different types of I/O request methods exist in current operating systems. One is synchronous I/O (sync I/O) which waits until the I/O request is completely processed. The other is asynchronous I/O (async I/O) which immediately returns even if the I/O processing is still in progress, thus making it possible for the process to execute next command. Async I/O requires a special routine for handling later notification of I/O completion.

We hope that future OS kernel versions will include system calls for the still conceptual psync I/O. Psync I/O synchronously operates in the same way as traditional sync I/O except that the unit of operation is an array of I/O requests. We define the three requirements of psync I/O as follows. 1) It delivers the set of I/Os to the flash SSDs and retrieves request results at once. Another set of I/O requests can be submitted in sequence only after the results of the previous set are retrieved. 2) The I/Os are requested as a group in the OS user space and the group needs to be sustained until they are delivered to I/O schedulers in the OS kernel space, thereby minimizing the request interval between consecutive I/Os upon I/O schedulers 3) No special routine is required to handle I/O completion events since the process is blocked until the set of I/Os are completely handled.

Since no I/O requesting method that satisfies the psync I/O requirements exists in current OSs, we designed a wrapper function that emulates psync I/O by using Linux-native asynchronous I/O API. The wrapper function delivers a group of I/O requests to the 'io_submit' system call by containing them in Linux async I/O data structures (struct iocb), and it waits until all the results are returned, executing 'io_getevents' system call.



(a) Read (b) Write

Figure 2: Bandwidths with increasing OutStd I/O level

3 MPSearch

In this section, we introduce MPSearch that enables for tree-based indexes to exploit the channel-level parallelism of flash SSDs via Psync I/O. PIO B-tree is the integrated result of the optimized B+-tree index operations based on MPSearch.

Searches to next-level nodes cannot be performed without obtaining the search results of current-level nodes since index records of the current-level nodes contain the locations of next level nodes. The only way to exploit the channel level parallelism is to search multiple nodes at the same level. However, this is possible only if a set of search requests are provided at once. Under the assumption that the set of search requests is given, we design a Multi Path Search (MPSearch) algorithm that processes a set of requests at once while searching multiple nodes level by level. The method to acquire the set of requests is explained later with specific index operations.

We represent an internal node of a B+-tree index as a set of key values (K_i) and pointer values (P_i) each pointing to the location of a child node as depicted in Figure 3, where F represents the fanout (the maximum number of pointers).

Let S denotes the set of search requests.

$$S = \{s \mid s\ is\ the\ key\ value\ for\ each\ search\ request\}$$

S includes all the key values of search requests, and $|S|$ represents the number of given search requests. The basic concept of MPSearch is described as follows.

First, the root node of the B+-tree index is retrieved. The key values of the root node are inspected, and the pointers to the next-level nodes designated by any of the search requests are extracted as (1), where P denotes the set of the extracted pointers.

$$\begin{aligned} P &= \{P_i \mid i \in I\} \\ I &= \{1 \le i \le F \mid K_{i-1} \le s < K_i, s \in S, K_0 = -\infty, K_F = \infty\} \end{aligned} \tag{1}$$

Second, the next-level nodes designated by the pointer set (P) are read at once through psync I/O. The entries

of the read internal nodes are examined node by node, and the pointer set for each node corresponding to S is extracted using (1). The extracted pointer sets create an array of the pointer sets as follows.

Third, all the child nodes designated by the pointers in the pointer sets of are read at once through psync I/O.

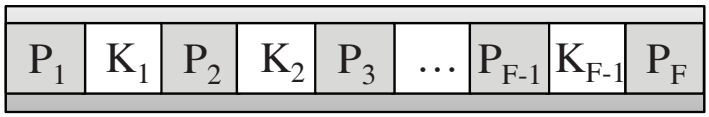


Figure 3: Internal node structure

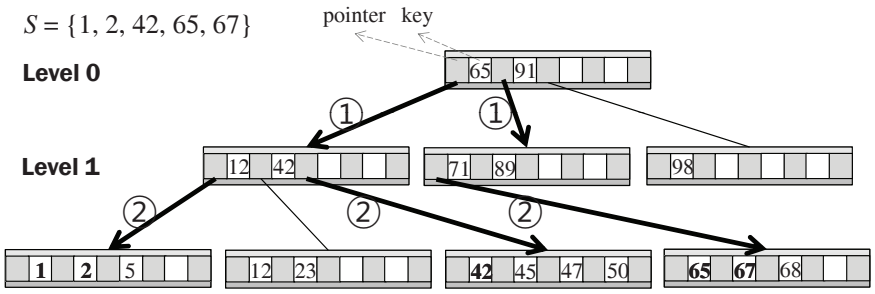


Figure 4: MPSearch with two psync I/Os

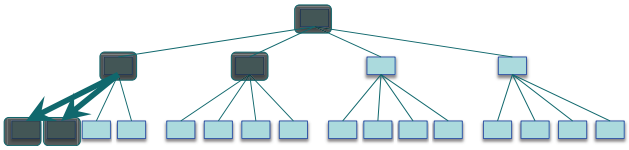


Figure 5: MPSearch with PioMax of 2

The entries of the read internal nodes are examined for every node, and for each node the pointer set to the next-level nodes is extracted, creating an array of the pointer sets, again. This process is repeated, until it reaches leaf nodes and retrieves all the leaf nodes corresponding to the search requests S. Figure 4 presents an example of MPSearch when the number of search requests is 5. With 2 psync I/O calls, the leaf nodes having the search keys are retrieved by MPSearch.

During this procedure, psync I/O is executed $(treeHeight - 1)$ times, at maximum processing $|S|$ read requests each time since psync I/O is executed once for every level except the root level. This indicates that MPSearch can achieve $|S|$ OutStd level at maximum. It also implies that $|S|$ in-memory buffer pages are required for every psync I/O call since one buffer page is needed for every node to be loaded into main memory. This might consume a significant amount of main memory space if $|S|$ is considerably large.

Therefore, we adopt a parameter called $PioMax$ that indicates the maximum number of I/Os submitted to a psync I/O call. By doing so, the maximum main memory space is limited to $(treeHeight - 1)PioMax$ pages.

MPSearch reads $PioMax$ nodes at a time analogously to DFS (Depth First Search) as depicted in Figure 3. In other words, MPSearch is a variant of DFS algorithm, differentiated by the number of nodes explored at a time from DFS (1 for DFS, PioMax for MPSearch). In terms of exploiting the internal parallelism of flash SSDs, PioMax is not necessary to be considerably large as demonstrated in the results of Figure 2. A moderate value (around 32) can utilize the saturated high bandwidth.

Not limited to B+-tree, MPSearch can be easily adopted in other tree-based index structures such as Fractal Tree [11], R-tree [12], LSM tree [5], and FD-tree [8], since MPSearch is basically a tree exploration method (a variant of DFS) associated with how the nodes are read from storage devices. This implies that MPSearch algorithm can be universally utilized for numerous data management methods to exploit the internal parallelism of storage devices. It is straightforward to apply MPSearch into the range search of B+-tree. This is simply

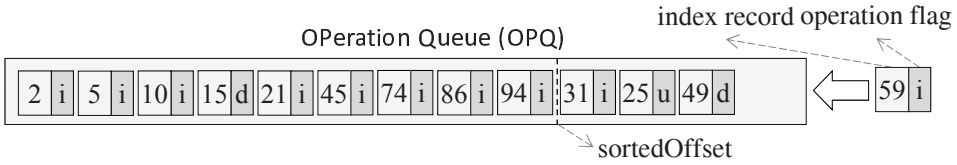


Figure 6: MPSearch with two psync I/Os

achieved by requesting the search request set S defined as (2) via MPSearch.

$$S = \{s \mid range.start \leq s < range.end\} \tag{2}$$

The MPSearch retrieves the leaf nodes including the entries with key values in the range. The traditional method to conduct a range search is reading the leaf nodes that are linked together one by one in sequence, after searching for the first leaf node containing an entry with the least key value of the range. The new range search, called parallel range (prange) search reads relevant internal nodes level by level via psync I/O until it reaches the leaf level.

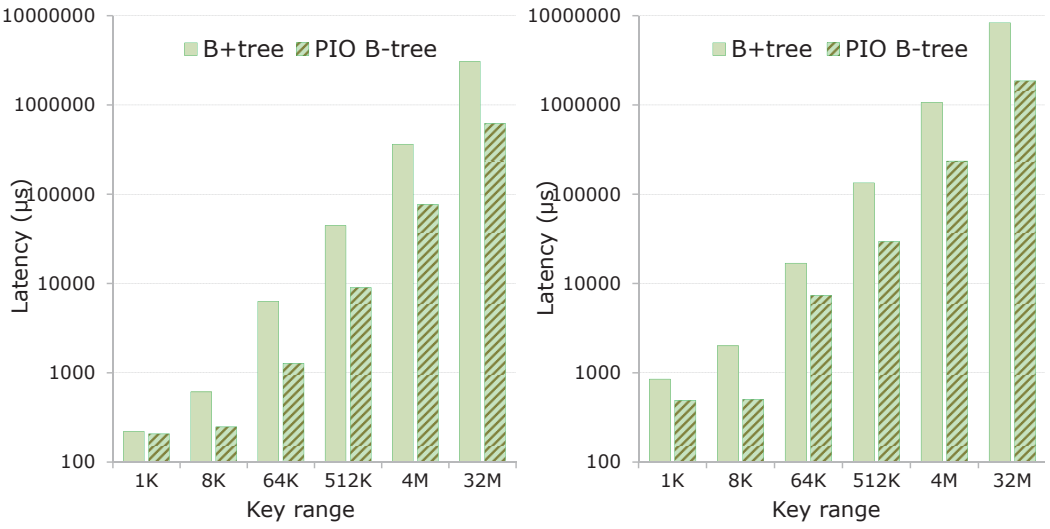
Likewise, multiple point-search operations can be queued in the expense of delaying response and simultaneously processed based on MPSearch algorithm.

For index update operations, PIO B-tree adopts in-memory structure called OPQ (OPeration Queue) to accumulate a group of update operations for later batch-updates based on MPSearch.

Figure 6 shows an example of OPQ. Each update operation is completed immediately after its index-record is inserted into the Operation Queue (OPQ) as an OPQ entry. OPQ entries are not written to the flash SSDs until the OPQ entries are batch processed. Since update operations are not immediately reflected to flash SSDs and reside on the in-memory structure for a while, this method requires additional features to the traditional DBMS recovery scheme for avoiding data-loss during system crashes. For detailed description of index update operation and crash recovery of PIO B-tree, please refer to Section 3.1.3 of [9].

4 Experimental Results

In this section, we present the experimental results of PIO B-tree. First, we evaluate the effectiveness of the proposed index operations by comparing the performance of PIO B-tree with B+-tree. Then, we compare PIO B-tree with other flash-aware indexes such as BFTL [7], and FD-tree [8] in synthetic workloads. The synthetic workloads were used for a better control of the workload configuration. We conducted the experiments on a Linux machine with 8-core CPU (2.0 GHZ), and 16GB main memory. We used two flash SSDs: Iodrive, a high-priced enterprise-class SSD, and P300, an enterprise-class SSD.



(a) Iodrive (b) P300

Figure 7: Range search time with different ranges in log scale

B+-tree, BFTL, and PIO B-tree were implemented based on the description in their original papers. The indexes were initially built with 1 billion entries by using a bulk loader, thus occupying more than 8GB of storage space. The LRU buffer manager was employed for the indexes. The maximum available main memory space was fixed at 16MB.

4.1 Parallel Range Search

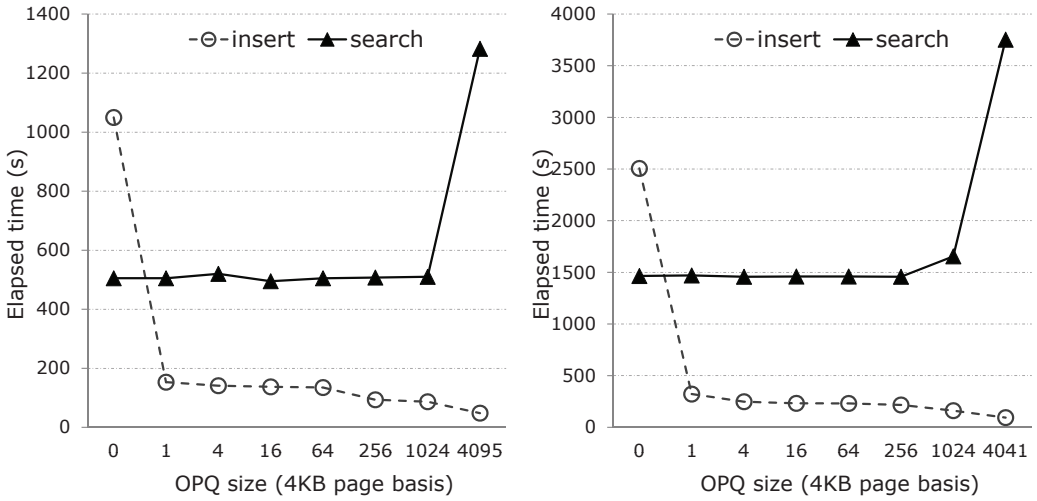
We evaluated the range search performance by requesting queries having various key ranges. The buffer pool size was fixed at 16MB. The gap between the start key and end key in the range was increased from 1024 (1K) to 33554432 (32M) by a factor of 8 at a time.

One hundred range searches were performed. Figure 7 represents the average elapsed time of a range search in log-scale with respect to the key ranges. PIO B-tree outperformed B+-tree with any given range on every flash SSD. The prange search of PIO B-tree was 5 times faster than the range search of B+-tree when the key range was greater than or equal to 64K on Iodrive. With the range greater than or equal to 512K, PIO B-tree was 3.5 times faster than B+-tree on P300.

4.2 Update Operations

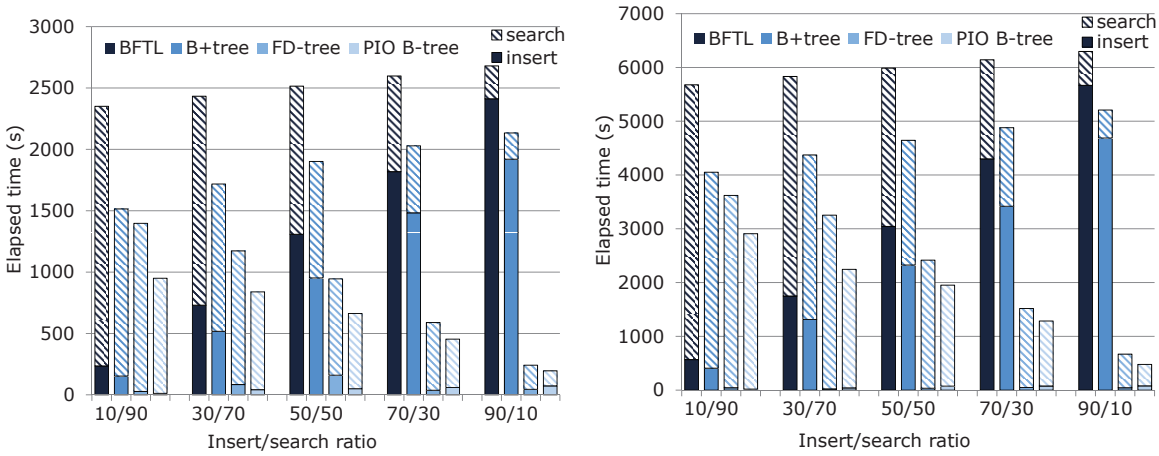
We evaluated the performance of update operations by using update-only workloads. Since index-insert, index-delete, index-update operations demonstrated almost the same performance, we only report the insert workload result. After allocating the main memory to OPQ, the rest of main memory space was allocated to the buffer pool. We measured the elapsed time of five million insert operations, increasing the OPQ size on a 4KB page basis. In order to assess the point-search performance degradation by the reduced buffer pool, we measured the elapsed time of five million point-search requests. Figure 8 presents this result.

Compared to the B+-tree performance, only with the OPQ size of one (4KB), PIO B-tree has demonstrated remarkable insert performance. The PIO B-tree was 7.2, and 8.2 times faster than B+-tree on Iodrive, and p300, respectively. With a large OPQ size, it was up to 28 times faster than B+-tree (on P300 with OPQ size 4041). Note that PIO B-tree was 16.3 times faster than B+-tree in insert operations and at the same time 1.2 times faster than B+-tree in search operation, with the OPQ size of 1024 on P300.



(a) Iodrive (b) P300

Figure 8: Insert/search time of PIO B-tree with different OPQ sizes



(a) Iodrive (b) P300

Figure 9: Overall elapsed time in mixed workloads

4.3 Comparison with Flash-aware Indexes

We compared four indexes such as BFTL, B+-tree, FD-tree, and PIO B-tree in five workloads. The workloads were differentiated by the insert ratio and search ratio. We configured the workloads to have randomly chosen 10 million operations with the specified insert and search ratio. Since the flash-aware indexes take advantage of trade-offs between insert and search performance based on parameters, to be fair, we chose the best parameter according to the workload feature for each index.

As shown in Figure 9, the PIO B-tree was 2.5 to 13.7, 2 to 13 faster than the BFTL on Iodrive, and P300, respectively. The PIO B-tree was 1.6 to 11, and 1.4 to 10.9 times faster than the B+-tree on Iodrive, and P300, while the PIO B-tree was 1.23 to 1.47, and 1.24 to 1.45 times faster than the FD-tree on Iodrive, and P300, respectively. In the graphs we differentiated insert and point-search time. The FD-trees insert time was similar to that of PIO B-tree. The performance gap between PIO B-tree and FD-tree was mainly due to the PIO B-trees faster search performance.

5 Conclusions

Due to the embedded flash memory packages, internal parallelism is an inherent feature of flash SSDs. In this paper, we found an efficient way (psync I/O) to generate parallel I/Os for exploiting the internal parallelism. We presented MPSearch algorithm and PIO B-tree that exploit the internal parallelism of flash SSDs. PIO B-tree outperformed B+-tree in search and update pe operations.

We expect that exploiting the internal parallelism of flash SSDs will be more important since NAND flash based storage technologies are evolving toward providing extremely high IOPS. Beyond hundreds of thousands of IOPS supported by PCI-e based SSDs, enterprise storage vendors such as Nimbus Data, PureStorage, and EMC has recently released flash-array products providing more than millions of IOPS. It is evident that data management systems should request multiple I/Os in a single thread to exploit the extremely high IOPS of current flash array products unless they afford to manage millions of threads to handle the I/Os.

As future work, we plan to apply PIO B-tree in CouchDB and apply MPSearch algorithm in the LSM-tree of HBase in order for the NoSQL DBMSs to process multiple search requests faster in a single thread on flash SSDs.

References

[1] HBase, "Hbase," https://hbase.apache.org/.

[2] Cassandra, "Cassandra," http://cassandra.apache.org/.

[3] MongoDB, "Mongodb," https://www.mongodb.org/.

[4] CouchDB, "Couchdb," http://couchdb.apache.org/.

[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, 1996.

[6] G.-J. Na, S.-W. Lee, and B. Moon, "Dynamic in-page logging for flash-aware b-tree index," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09, 2009, pp. 1485–1488.

[7] C.-H. Wu, T.-W. Kuo, and L.-P. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, 2007.

[8] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1195–1206, Sep. 2010.

[9] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 286–297, Dec. 2011.

[10] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *HPCA*, 2011, pp. 266–277.

[11] Tokutek, "Fractal tree," http://www.tokutek.com/2012/12/fractal-tree-indexing-overview/.

[12] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, 1984, pp. 47–57.

Optimizing Database Operators by Exploiting Internal Parallelism of Solid State Drives

Yulei Fan, Wenyu Lai, and Xiaofeng Meng
School of Information, Renmin University of China, Beijing, China

{fyl815,xiaolai913,xfmeng}@ruc.edu.cn

Abstract

With the development of flash memory technology, flash-based solid state drives (SSDs) are gradually used in more and more devices and applications. In addition to characteristics of flash memory itself, a unique characteristic of SSDs, namely internal parallelism, should also be considered to improve performance of SSDs-based DBMSs, especially query processing. In this paper, we first describe the internal architecture of SSDs and the resulting internal parallelism of SSDs. In the second part, we present a parallel table scan operator, ParaScan, that exploits the internal parallelism of SSDs. Based on ParaScan, we then propose a parallel hash join operator, ParaHashJoin, and a parallel aggregation model, ParaAggr. Experimental results show that ParaScan, ParaHashJoin and ParaAggr on SSDs significantly outperform traditional table scan, hash join and aggregation. Furthermore, sort, as an important basic operator for other complex operators, can also be redesigned based on ParaScan. We design a parallel sort algorithm, ParaSort and then present a parallel ParaSort operator in the third part. Looking forward, database query processing by exploiting internal parallelism of SSDs, can be generalized to other kinds of SSDs with similar internal parallel characteristics.

1 Introduction

As flash memory-based solid state drives (SSDs) improve in price, capacity, reliability and performance, more and more devices and applications gradually adopt SSDs as secondary storage media. Not only do SSDs provide faster access speed, lower power consumption, lighter weight, smaller size and better shock resistance than HDDs, they also have rich internal parallelism that can be used to improve I/O bandwidth[1, 2]. Query processing of HDDs-based DBMSs are mainly designed according to HDDs' mechanical limitations, so they may benefit less or even nothing when SSDs simply replace HDDs for OLAP and OLTP[3]. Beside the characteristics of flash memory, the internal parallelism of SSDs need to be considered when designing query processing algorithms for SSDs-based DBMSs including scan, join, aggregation, sort, etc[4].

This paper explores how to optimize scan, join, aggregation and sort operators by exploiting the internal parallelism of SSDs. We review related work in section 2. In section 3, we outline how to detect the internal parallel architecture of SSDs. And then in section 4, we present parallel table scan ParaScan, parallel hash join ParaHashJoin and parallel aggregation ParaAgg and verify their efficiency by running them on HDD and SSDs. And then we propose a parallel sort ParaSort in section 5. Finally, we offer conclusions in section 6.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Related Work

To achieve higher capacity and better I/O performance, most SSDs adopt a multi-channel and multi-way architecture for connecting flash chips with a flash memory controller. The flash memory controller can access flash chips in parallel[5]. Therefore, it is necessary to understand the impact of this parallelism inside SSDs on the performance of SSDs-based DBMSs. Chen[2] studied how to uncover internal parallelism features of SSDs by detailing an abstract parallel model of SSDs and methods of detecting SSDs. Roh[6] created a new I/O request method in the OS that can be used to generate parallel I/Os to access SSDs by applications. Based on this, the authors designed a new B+-tree variant called PIO B-tree. Hu[7] analyzed the performance impact and interplay of advanced commands, physical-page allocation and data granularity for internal parallelism of SSDs. According to internal parallelism architecture of SSDs, Park[8] designed parallel-aware request processing including request rescheduling and dynamic write request mapping.

Myers[9] and Lee[10] measured the performance of various traditional database algorithms on SSDs. For flash-based DBMSs, a number of query processing algorithms have been designed for flash characteristics, especially excellent random read algorithms[11, 12, 13, 14, 15, 16, 17]. Myers[9] improved the sort-merge join algorithm by optimizing the storage of intermediate results. Graefe[12, 13] focused on speeding up select, project and join operators based on a new page layout, PAX, for SSDs-based DBMSs. Simultaneously, they proposed FlashScan, FlashJoin and RARE-join algorithms and implemented them in PostgreSQL. Liang[14] optimized a join algorithm for column-based DBMSs by reading only needed columns. Li[15, 16, 17] improved no-index join algorithms by reducing intermediate results and optimizing the fetching strategy. Different from previous studies, we try to optimize scan, aggregation, join and sort operators by exploiting the internal parallelism of SSDs.

3 Internal Parallelism of SSD

3.1 Internal Parallel Architecture

The internal architecture of SSDs includes the host interface, SSD controller, RAM buffer and flash memory packages[8]. SSDs connect to the host by a host interface that can be SATA, SAS or PCIe. The SSD controller executes I/O requests and issues commands to flash memory packages via a flash controller. The SSD controller is also in charge of managing the RAM buffer which holds the address mapping table of the flash translation layer (FTL) and other metadata such as ECC etc. SSDs implement internal parallelism by adopting multiple channels that can operate independently and simultaneously. Each channel is shared by a set of flash memory packages, on which operations can also be interleaved, so the bus utilization can be optimized[5, 18]. As shown in[8], channel-level parallelism and package-level parallelism are two typical levels of parallelism. Such rich internal parallelism provides us an opportunity to improve the performance of applications on SSDs.

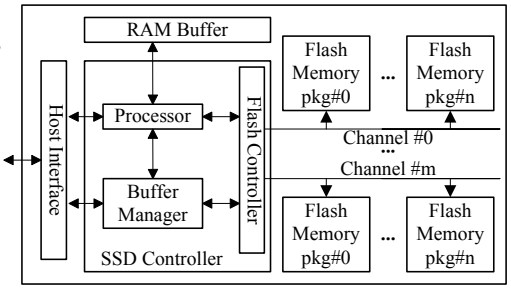


Figure 1: Inside SSD

3.2 Detecting SSD Internals

Before detecting and utilizing the internal parallelism, we must ensure that SSDs have native command queuing (NCQ) mechanisms and AHCI (Advanced Host Controller Interface) mode must be enabled for the host[19]. Multi-threaded processing can be used to produce multiple parallel I/Os. Furthermore, it is necessary to know some key architectural features of SSDs such as the number of channels for setting a proper concurrency level

and avoiding over-parallelization. However, it is hard to obtain such information because these details are often regarded as critical intellectual property of SSD manufacturers. Therefore, we try to detect two representative SSDs as shown in Table 1. Base on publicly available documents, Chen[2] characterized internal organization of SSD as an abstract model including chunk size and the number of domains. A chunk is a unit of data that is continuously allocated within one domain. A domain is a set of flash memories that share a specific set of resources. Guided by the model and the detecting method[2], we obtain chunk size and the number of domains of SSD-S and SSD-M as shown in Table 1 (detailed detecting results are in [21]).

Table 1: Detecting Results

	Manufacturer	Flash	Capacity	Page Size	Interface	NCQ	Chunk Size	Domains No.
SSD-S	Intel	SLC	32 GB	4 KB	SATA	32	4 KB	20
SSD-M	Intel	MLC	160 GB	4 KB	SATA	32	16 KB	20

4 ParaScan, ParaHashJoin and ParaAggr

4.1 ParaScan: Parallel Table Scan

Most SSDs adopt RAID-0 data storage as shown in Fig. 2. In SSDs-based database systems, records within a chunk are organized according to traditional row-based page layout. One chunk is one logical data page with an unique logical address. According to logical addresses, all chunks of a relational table are uniformly placed in all domains. All striped chunks of a relational table are mostly placed in logical consecutive regions in a file. On the assumption of 20 domains, the chunk whose logical address is 20*n (n=0,1,2...), will be in 1st domain, the chunk whose logical address is 20*n+1 (n=0,1,2...), will be in 2nd domain, and so on. Striped domains provide interleaved accesses to reduce sharing of data-bus channels. For example, chunk 0 and chunk 20 in domain 0 cannot be accessed simultaneously, but chunk 0 in domain 0 and chunk 1 in domain 1 can be read in parallel. This provides an opportunity to distribute different data accesses to different domains.

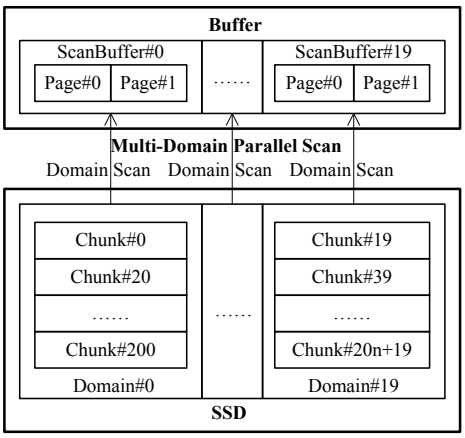


Figure 2: ParaScan

Based on this, we propose a parallel table scan called ParaScan to improve the efficiency of table scan. As shown in Fig. 2, every domain scan reads data chunks one by one from corresponding domain and then puts them into its own ScanBuffer. Multiple domain scan operations can be executed independently and simultaneously. All domain scan and all ScanBuffers compose multi-domain parallel scan named ParaScan. Because chunks are mostly placed in logical consecutive regions, we suggest that all data pages of a relational table should be uniformly distributed into different domains as much as possible to make full use of the internal parallelism of SSDs. Data size in one domain usually exceed ScanBuffer size, so it is necessary to replace processed data pages in ScanBuffer with unprocessed data pages when executing domain scan.

We implement ParaScan by a multi-thread processing technique. Each domain scan corresponds to one thread. The performance of ParaScan depends not only on the number of domains but also on the number of physical threads supported by CPU and NCQ supported by SSDs. We ran ParaScan on an HP PC with Ubuntu 12.10, 8G DDR3 memory, a 500G 7400rpm SATA3 Seagate HDD, Intel Core i5-2400 @ 3.10GHz processor with four cores supporting four physical threads and two SSDs as shown in Table 1. The data set was the ORDERS table(150 million rows, 2G) taken from TPC-H benchmark. Fig. 3 compares the performance of scans while varying the number of threads. Traditional table scan is ParaScan implemented by a single thread.

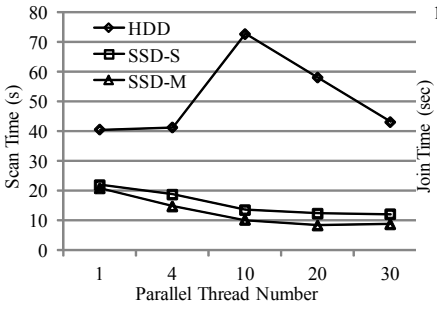


Figure 3: ParaScan

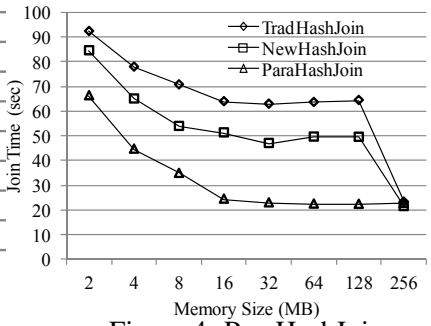


Figure 4: ParaHashJoin

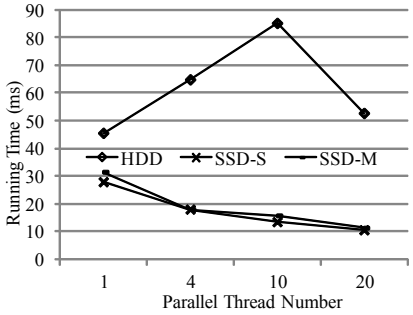


Figure 5: ParaAggr

Experimental results show that ParaScan isn't well suited for HDD but can exploit the internal parallelism of SSDs (detailed experimental results are in [20, 21]).

4.2 ParaHashJoin: Parallel Hash Join

Based on ParaScan, we present a parallel hash join operator called ParaHashJoin. It consists of three phases, ParaHash, MiniJoin and Fetch, as shown in Fig. 6. ParaHash hashes records of different domains in parallel to exploit the internal parallelism of SSDs. The output of ParaHash only contains the row-id (RID) and the join attribute of every record to reduce the amount of data processed in MiniJoin. A RID consists of page id and in-page offset of a record. Finally, Fetch retrieves the needed attributes by RIDs. Because the output of ParaHash and MiniJoin are incomplete results, ParaHashJoin uses less non-volatile storage to materialize intermediate results. However, multi-threaded processing and random reads in ParaHashJoin will cost more CPU. But experimental results show that this tradeoff is worthwhile.

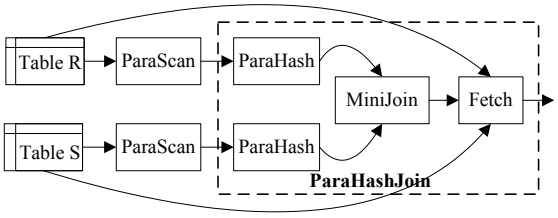


Figure 6: ParaHashJoin

As shown in Fig. 7, each ParaHash instance calculates hash values of records in its ScanBuffer and then adds their join attribute values and RIDs into specific hash buckets. Each hash bucket must maintain a lightweight lock to solve conflicts between hash threads. Hash function is $hash\_value = join\_attr\&(B-1)$. If the results of ParaHash on table R can be maintained in memory, we only need to do ParaScan on table S after ParaScan and ParaHash on table R. And then according to join attribute value of each record of table S in ScanBuffers, MiniJoin directly probe hash buckets of table R and produce join results in the form $\{join\_attr, RID_R, RID_S\}$. Otherwise, MiniJoin gets one hash buckets of table R and corresponding hash bucket of table S into memory to generate join results, and processes other hash buckets in this way.

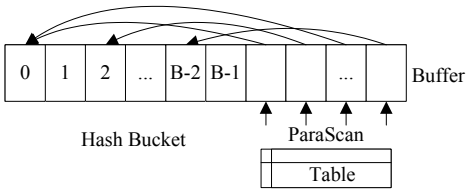


Figure 7: ParaHash

Because results of MiniJoin are incomplete join results, according to RID in join results, Fetch randomly reads necessary attribute values to generate final join results. Designing efficient fetching strategy is very important for minimizing I/Os of reading data pages, but we adopt a sort-based fetching strategy inspired by DigestJoin[15, 16, 17] to avoid reloading pages as much as possible as shown in [20]. Before fetching data pages, we sort results of MiniJoin, so needed data pages can be loaded in order. The sort will cost more CPU but we show that this tradeoff is worthwhile in experimental results.

Implementation technique and running platform of ParaHashJoin are the same as ParaScan. In addition to the ORDERS table, the input data set includes the CUSTOMER table(1.5 million rows, 256M) obtained from TPC-H benchmark. In Fig. 4, TradHashJoin adopts traditional hash join and table scan. NewHashJoin adopts

traditional hash join and ParaScan. We mainly consider equi-join of CUSTOMER and ORDERS on a single attribute. Join selectivity is 1%. The number of threads in ParaScan and ParaHash is fixed at 20. In this paper, we only present experimental results on SSD-S. Fig. 4 evaluate the impact of memory size on three join algorithms as memory size varied from 2MB to 256MB. Both TradHashJoin and NewHashJoin require 256M to execute in one pass while ParaHashJoin only need 16M. Fig. 4 shows that, in the best case, ParaHashJoin is 1-1.5 x faster than NewHashJoin and TradHashJoin (detailed experimental results are in [20]). Our experimental results show that it is worthwhile to pay the extra CPU cost for multi-threaded processing and random reads.

4.3 ParaAggr: Parallel Aggregation

Based on ParaScan, we propose a parallel aggregation algorithm called ParaAggr for implementing some simple aggregation such as *sum*, *max*, *min*, *count*, *average* etc. ParaAggr consists of two phase, SubAggr and TolAggr, as shown in Fig. 8. Each SubAggr instance only processes records in its ScanBuffer and all SubAggr run in parallel. Either every computing result can be forwardly put to TolAggr, or TolAggr forwardly get all computing reseults produced in SubAggr. And then TolAggr calculates all results of SubAggr. The calculations in SubAggr phase and TolAggr phase may be same or different. For example, *count* operator can be divided into count in SubAggr and summation in TolAggr(others are shown in [21]).

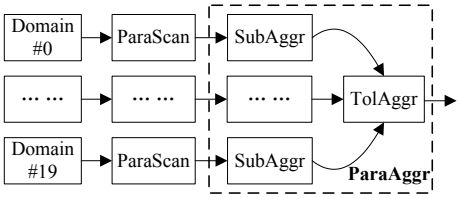


Figure 8: ParaAggr

By exploiting multi-threaded processing we implemented ParaAggr including *sum*, *max*, *min*, *count*, *average*. We ran ParaAggr on a Lenovo K46A with Fedora 14, 2G DDR3 memory, a 500G 5400rpm SATA3 Seagate HDD, Intel Core i5 @450MHZ processor with double cores supporting four physical threads, two SSDs as shown in Table 1 and another SSD as shown in [21]. The data set is the CUSTOMER table(870 thousand rows, 1GB) taken from the TPC-C benchmark. Fig. 5 compares the performance of aggregations while varying the number of SubAggr threads. Traditional aggregation is ParaAggr with 1 SubAggr thread. As shown in Fig. 5, in the best case, ParaAggr is 3-4 x faster than traditional aggregation. Other experimental results are detailed in [21].

5 ParaSort: Parallel Sort for Up-level Operators

Sort is an important operator for up-level operators such as join, group by, having, limit, sub-query etc. In this section, we proposed Parallel Sort Model called ParaSort and present Parallel ParaSort.

5.1 Parallel Sort Model

We now sketch a parallel sort algorithm called ParaSort based on ParaScan. As show in Fig. 9, ParaSort consists of two phases, SubSort and Fetch. SubSort is in charge of getting sorted attribute values and RIDs of all records and then sorting them according to the sort key. SubSort outputs ordered results the sort key and RID and then Fetch randomly reads required attribute values required by up-level operators. The fetching strategy can utilize the strategy mentioned in 4.2. As the critical component of ParaSort, SubSort can be implemented according to traditional sort method to sort any size data, but SubSort can't run until the end of ParaScan if ScanBuffers can store all records; otherwise, SubSort will carry out as long as ScanBuffers is full. What's more, SubSort can be implemented in parallel when sorted attribute values and RIDs of all records fit in memory.

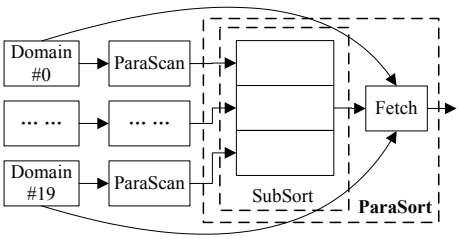


Figure 9: ParaSort

5.2 Parallel ParaSort

When memory is enough, we discuss a parallel ParaSort by implementing SubSort in parallel. As shown in Fig. 10. Every SubSort gets sorted keys and RIDs of data records from corresponding ScanBuffer and then sorts them according to sort key. All SubSorts run in parallel. Results of one SubSort is ordered but results of all SubSorts is semi-ordered. TolSort merges semi-ordered results of all SubSorts to obtain ordered results. Fetch retrieves the necessary attribute values for up-level operators by ordered results of TolSort. The fetching strategy can also use the strategy mentioned in 4.2.

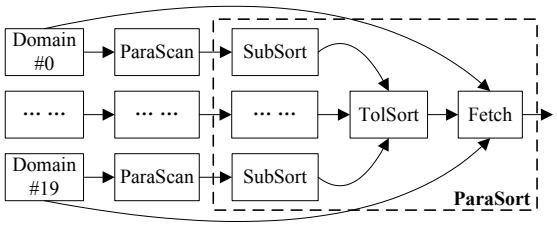


Figure 10: Parallel ParaSort

6 Conclusions

In addition to excellent performance characteristics of flash memory, there is rich parallelism inside SSDs. Previous researchers mainly focus on optimizing algorithms by exploiting properties of flash memory, but the internal parallelism of SSDs also needs to be considered when optimizing SSDs-based DBMSs, especially query processing.

By exploiting the rich internal parallelism of SSDs, we presented ParaScan and then proposed ParaHashJoin and ParaAggr based on ParaScan. Experimental results showed that we were able to speed up performance of ParaScan, ParaHashJoin and ParaAggr by $>$ 1x. And for complex operators, we designed ParaSort. In other words, many database operators can be sped up by exploiting the internal parallelism of SSD. Future research will show whether this is feasible, effective and efficient and how it affect industrial OLAP and OLTP systems.

Acknowledgments

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61379050, 91224008); the National 863 High-tech Program (No. 2013AA013204); Specialized Research Fund for the Doctoral Program of Higher Education(No. 20130004130001), and the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010).

References

[1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In SIGMETRICS, pages 181-192, 2009.

[2] F. Chen, R. Lee and X. Zhang. Essential Roles of Exploiting Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In HPCA, pages 266-277, 2011.

[3] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In SIGMOD, pages 55-66, 2007.

[4] R. Ramakrishnan and J. Gehrke. Database Management Systems (3rd Ed.). McGraw Hill, 2002.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In USENIX, pages 57-70, 2008.

[6] H. Roh, S. Park, S. Kim, M. Shin and S.-W. Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proc. VLDB, 5(4): 286-297, 2012.

[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo and S. P. Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granu-larity. In ICS, pages 96-107, 2011.

[8] S. Y. Park, E. Seo, J. Y. Shin, S. Maeng and J. Lee. Exploiting internal parallelism of flash-based SSDs. Computer Architecture Letters, 9(1):9-12, 2010.

[9] D. Myers. On the use of NAND flash memory in high-performance relational databases. MIT Msc Thesis, 2008.

[10] W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In SIGMOD, pages 1075C1086, 2008.

[11] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In DaMoN, pages 1-8, 2009.

[12] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In DaMoN, pages 17-24, 2008.

[13] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In SIGMOD, pages 59-72, 2009.

[14] Zh. Liang, D. Zhou, X. Meng. Sub-Join: Query Optimization Algorithm for Flash-based Database. Journal of Frontiers of Computer Science and Technology, 2010, 4(5): 401-409.

[15] Y. Li, S. T. On, J. Xu, B. Choi, H. Hu. DigestJoin: Exploiting Fast Random Reads for Flash-based Joins. In MDM, pages 152-161, 2009.

[16] Sh. Gao, Y. Li, J. Xu, B. Choi, H. Hu. DigestJoin: Expediting Joins on Solid-State Drives. In DASFAA pages 428-431, 2010.

[17] Y. Li, S. On, J. Xu, B. Choi, H. Hu. Optimizing Nonindexed Join Processing in Flash Storage-Based Systems. IEEE Transactions on Computers, 62(7):1417-1431, 2013.

[18] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, achitecture, and system organization. In ISCA, pages 279-289, 2009.

[19] S.-W. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In SIGMOD, pages 863-870, 2009.

[20] W. Lai, Y. Fan, X. Meng. Scan and Join Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. In WAIM, pages 381-392, 2013.

[21] Y. Fan, W. Lai, X. Meng. Database Table Scan and Aggregation by Exploiting Internal Parallelism of SSDs. Chinese Journal of Computers, Vol 35(11):2327-2336, 2012.

Query Processing on Smart SSDs

Kwanghyun Park[1], Yang-Suk Kee[2], Jignesh M. Patel[1],
Jaeyoung Do[3], Chanik Park[2], David J. Dewitt[3]
[1]University of Wisconsin – Madison, [2]Samsung Electronics Corp., [3]Microsoft Corp.

Abstract

Data storage devices are getting smarter. Smart flash storage devices (a.k.a. Smart SSDs) are on the horizon and package a small programmable computer inside the device. Thus, users can run code closer to the data right inside the SSD, on the "other" side of the I/O bus. The focus of this paper is on exploring the opportunities and challenges associated with exploiting this functionality of Smart SSDs for relational analytic query processing. We have implemented an initial prototype of Microsoft SQL Server running on a Samsung Smart SSDs. Our results demonstrate that significant performance and energy gains can be achieved by pushing selected query processing components inside the Smart SSD. We also identify various changes that SSD manufacturers can make to increase the benefits of using Smart SSDs for data processing applications, and suggest possible research opportunities for the database community.

1 Introduction

Modern SSDs pack CPU processing and DRAM storage components inside the SSD to carry out the routine functions (such as managing the FTL logic) for the SSD. Thus, there is a small programmable computer inside a SSD device, presenting an interesting opportunity to move computation closer to the storage.

The key reason for moving computation closer to the data is shown in Figure 1. This figure shows the current and projected internal I/O bandwidth that is inside a Samsung Smart SSD, and the I/O bandwidth of the host interfaces (e.g. SAS and SATA standards). The numbers shown in this figure are relative to the I/O interface speed in 2007 (375 MB/s). Data beyond 2012 are internal projections by Samsung. The key trend highlighted in Figure 1 is that the I/O bus standards (such as SATA and PCIe) evolve slower than the speed of the internal network that is used inside the SSD. Without mechanisms to push computation inside the Smart SSD, we are essentially doomed to be "drinking from a narrow straw" when using SSDs for data intensive workloads.

In this paper, we build on our recent work [7] in this area. The focus of this paper is on exploring how analytical database workloads can exploit Smart SSDs. We have built a prototype version of SQL Server that selectively pushes computation to the Smart SSD, and in this paper we describe the details of our implementation. We also present results that show how using the Smart SSD in this way improves not just the performance of some queries, but also reduces the energy that is required to execute the queries. As energy consumption is a critical factor for database appliance and cloud services, using Smart SSDs also provides an interesting opportunity to design energy-efficient data processing systems.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

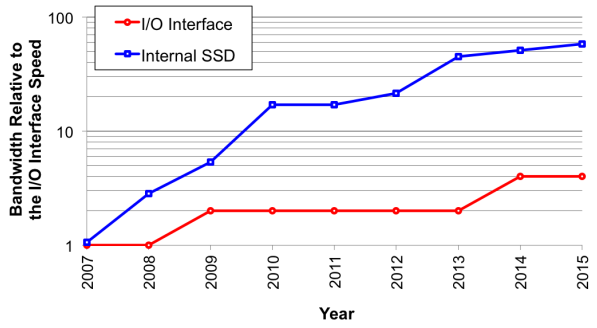


Figure 1: Bandwidth trends for the host I/O interface.

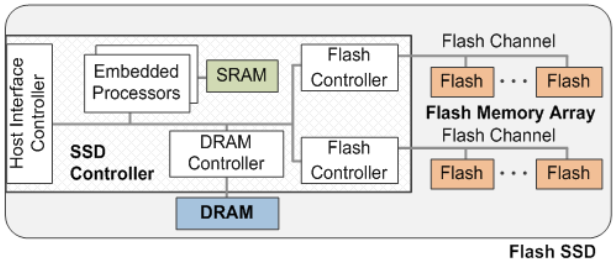


Figure 2: Internal architecture of a modern SSD

On a more subdued note, Smart SSDs are fairly hard to program and use today. In this paper, we also describe the challenges that we had in using the Smart SSDs, and point to potential opportunities to improve the Smart SSDs so that they are easier to use and deploy in the future.

2 Background: SSD Architecture

The SSD controller has four key subcomponents: a host interface controller, an embedded processor(s), a DRAM controller, and flash memory controllers. The host interface controller implements a bus interface protocol such as SATA, SAS, or PCI Express (PCIe). The embedded processors are used to execute the SSD firmware code that runs the host interface protocol, and also runs the Flash Translation Layer (FTL), which maps Logical Block Address (LBA) in the host OS to the Physical Block Address (PBA) in the flash memory. Time-critical data and program code are stored in the SRAM. Today, the processor of choice is typically a low-powered 32-bit RISC processor, like an ARM series processor, which typically has multiple cores. The controller also has on-board DRAM memory that has higher capacity (but also higher access latency) than the SRAM. The flash memory controller is in charge of data transfer between the flash memory and DRAM. Its key functions include running the Error Correction Code (ECC) logic, and the Direct Memory Access (DMA). To obtain higher I/O performance from the flash memory array, the flash controller uses chip-level and channel-level interleaving techniques. All the flash channels share access to the DRAM. Hence, data transfers from the flash channels to the DRAM (via DMA) are serialized. The NAND flash memory array is the persistent storage medium. Each flash chip has multiple blocks, each of which holds multiple pages. The unit of erasure is a block while the read and write operations in the firmware are done at the granularity of pages.

3 API for Query Processing

We have implemented a Smart SSD runtime framework that implements the common functionality that is needed to run user-defined programs in the Smart SSDs. We have developed a simple session-based protocol that is compatible with the standard SATA/SAS interfaces (but could be extended for PCIe). The protocol consists of three commands: OPEN, GET, and CLOSE.

- OPEN, CLOSE: A session starts with an OPEN command and terminates with a CLOSE command. Once the session starts, runtime resources including threads and memory that are required to run a user-defined program are granted, and a unique session id is then returned to the host.
- GET: The host can monitor the status of the program and retrieve results that the program generates via a GET command. This command is mainly designed for the traditional block devices (based on SATA/SAS

interfaces), in which case the storage device is a passive entity and responds only when the host initiates a request.

Using these extensions, we can now invoke specific database operator code that runs inside the Smart SSD. Essentially, the query operation to be performed is passed as parameters to the OPEN call, and corresponding result tuples are fetched into SQL Server using the GET call. Finally, the state information inside the Smart SSD is cleared using the CLOSE call.

4 Evaluation

In this section, we present selected results from an empirical evaluation of Smart SSD with Microsoft SQL Server. For our experiments, we push down scan, simple hash join, and aggregation operations into the Smart SSD. Note that in this paper we largely focus on presenting results that go beyond the initial results, which we presented in [7]. In other words, we refer the interested reader to [7] for additional results.

4.1 Experimental Setup

4.1.1 Workloads

For our experiments, we use the LINEITEM and the PART tables defined in the TPC-H benchmark [3]. Our modifications to the original LINEITEM and PART table specifications are as follows:

1. We use a fixed-length char string for the variable-length column,
2. All decimal numbers are multiplied by 100 and stored as integers,
3. All date values are converted to the number of days since the last epoch.

The LINEITEM and the PART tables are populated at a scale factor of 100. Thus, the LINEITEM table has 600M tuples (∼90GB), and the PART table has 20M tuples (∼3GB).

For our experiments, we also use two synthetic tables called Synthetic64_R and Synthetic64_S. Both these tables have 64 integer columns. The Synthetic64_R table has 1M tuples (∼300MB) and the Synthetic64_S table has 400M tuples (∼120GB). The first column of the Synthetic64_R (R.Col_1) is the primary key, and the second column of the Synthetic64_S (S.Col_2) is the foreign key pointing to R.Col_1.

To insert data into the table, we created a SQL Server heap table (without a clustered index). By default, tuples in these tables are stored in slotted pages using the traditional N-ary Storage Model (NSM). For the Smart SSDs, we also implemented the PAX layout [5] in which all the values of a column are grouped together within a page.

4.1.2 Hardware/Software Setup

All experiments were performed on a system running 64bit Windows 7 with 32GB of DRAM (24 GB of memory is dedicated to the DBMS). The system has two Intel Xeon E5430 2.66GHz quad core processors, each of which has a 32KB L1 cache, and two 6MB L2 caches shared by two cores. For the OS and the transactional log, we used two 7.5K RPM SATA HDDs, respectively. In addition, we used a LSI four-port SATA/SAS 6Gbps HBA (host bus adapter) [1] for the three storage devices that we used in our experiments. These three devices are:

1. A 146GB 10K RPM SAS HDD,
2. A 400GB SAS SSD, and
3. A Smart SSD prototyped on the same SSD as above.

We implemented code for simple selection, aggregation, and selection with join queries, and uploaded that code into the Smart SSD. We also modified some components in SQL Server 2012 [2] to recognize and communicate with the Smart SSD. For each query that is used in this empirical evaluation, we have a special path in SQL Server that we created to communicate with the SSD using the API described in Section 3. Note that the results presented here are from a prototype version of SQL Server that only works on a selected class of queries.

All the results presented here are for cold experiments; i.e., there is no data cached in the buffer pool prior to running each query.

4.2 Experimental Results

To aid the analysis of the results that are presented below, the I/O characteristics of the (regular) SAS SSD and the Smart SSD are shown in Table 2.

	SAS SSD	(internal) Smart SSD
Seq.Read (MB/sec)	550	1,560

Table 2: Maximum sequential read bandwidth with 32-page (256KB) I/Os.

As can be seen in Table 2, the internal sequential read bandwidth of the Smart SSD is 2.8X faster than that of the SSD. This value can be used as the upper bound of the performance gains that this Smart SSD could potentially deliver. As described in Figure 1, over time it is likely that the gap between the SSD and the Smart SSD will grow to a much larger number than 2.8X. We also note that the improvement here (of 2.8X) is far smaller than the gap shown in Figure 1 (about 10X). The reason for this gap is that the access to the DRAM is shared by all the flash channels, and currently in this SSD device, only one channel can be active at a time, which then becomes the bottleneck. One could potentially address this bottleneck by increasing the bandwidth to the DRAM or adding more DRAM buses. As we discuss below, this and other issues must be addressed to realize the full potential of the Smart SSD vision.

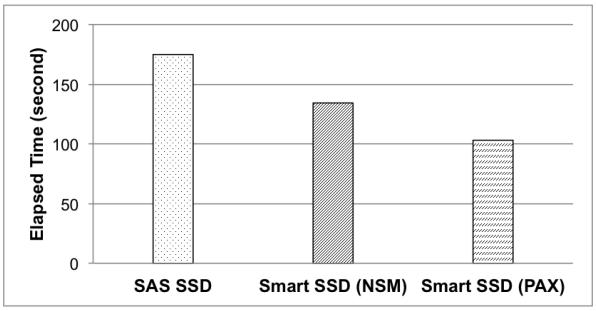


Figure 3: Elapsed time for the TPC-H query 6 on the LINEITEM table (100SF).

4.2.1 Single Table Scan Query: TPC-H Query 6

In this section, we present results for a single table scan query using TPC-H Query 6. (Additional results for single table scan queries for varying tuple sizes, scan selectivities, with and without aggregation can be found

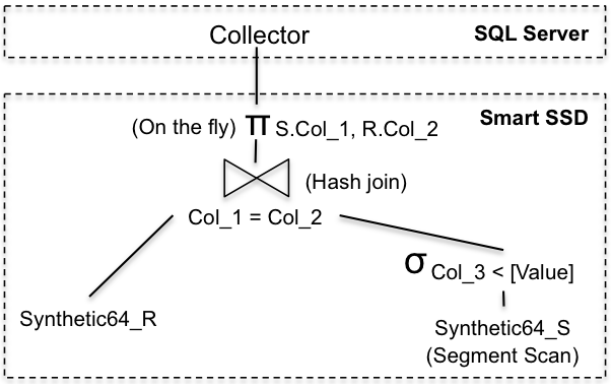


Figure 4: Query plan for the selection with join query in the Smart SSD.

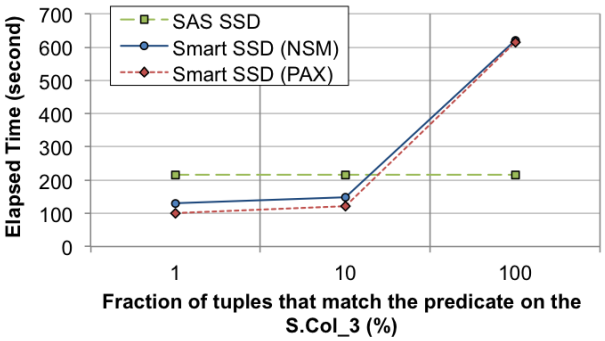


Figure 5: Elapsed time for the join query on the Synthetic64_R and the Synthetic64_S tables at various selectivity factors.

in [7].) This query is [3]:

SELECT SUM(l_extendedprice * l_discount)
FROM LINEITEM
WHERE l_shipdate >= 1994-01-01 AND l_shipdate <1995-01-01
AND l_discount >0.05 AND l_discount <0.07 AND l_quantity <24

Figure 3 shows the results with the (regular) SSD and the Smart SSD (with the NSM and the PAX layouts). As can be seen in this figure, the Smart SSD with the PAX layout improves overall the query response time by 1.7X over the SSD case. As explained in [7], the number of tuples in a data page, and the selectivity factor have a big impact on the performance improvement that is achieved using the Smart SSD. The selectivity factor (0.6%) of this query is small enough to enjoy the benefits of the Smart SSD. However, its high computation complexity (five predicates, 51 tuples per data page) saturates the CPU and the memory resources in the Smart SSD. As a result, we only achieved 1.7X speedup for this query instead of 2.8X.

4.2.2 Two-way Join Queries

In this section, we evaluate the impact of using the Smart SSD to run simple join queries.

4.2.2.1 Selection with Join Query For this experiment, we use the Synthetic64_R, the Synthetic64_S tables, and the following SQL query:

SELECT S.Col_1, R.Col_2
FROM Synthetic64_R R, Synthetic64_S S
WHERE R.Col_1 = S.Col_2 AND S.Col_3 <[VALUE]

Figure 4 shows the query plan for this query. Since the size of the Synthetic64_R table is far smaller than the Synthetic64_S table, (i.e., |S| = 400|R|), and the hash table for the Synthetic64_R table fits in memory, we used a simple hash join algorithm that builds a hash table on the Synthetic64_R table. As shown in Figure 4, the core computation of this query is carried out inside the Smart SSD and the host only collects the output from the Smart SSD. In the case of the regular SSD, we used the same query plan as the Smart SSD, but the plan was run entirely in the host.

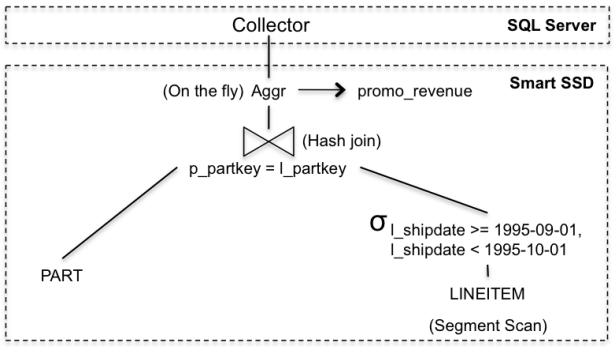


Figure 6: Query Plan for the TPC-H query 14 in the Smart SSD.

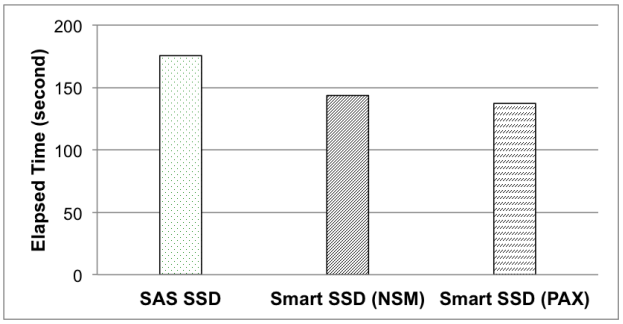


Figure 7: Elapsed time for the TPC-H query 14 on the LINEITEM and the PART table (100SF).

Similar to the previous result, the Smart SSD shows significant performance over the regular SSD case. As seen in Figure 5, the Smart SSD (with the PAX layout) improves performance for the highly selective queries by up to 2.2X over the (regular) SSD case when 1% of the tuples in the Synthetic64_S table satisfy the selection predicate. Similar to the case of single table scan queries, when the selectivity of the query is low (i.e., 100%), the performance of the Smart SSD is saturated because the query cost is dominated by the cost of the high volume of data that has to be transferred between the host and the Smart SSD.

4.2.2.2 TPC-H Query 14 For this experiment, we used the LINEITEM, the PART table, and Query 14 from the TPC-H benchmark [3]. This query is:

SELECT 100 * SUM(case when p_type LIKE 'PROMO%' then l_extendedprice *
(1 - l_discount) else 0 end) / SUM(l_extendedprice * (1 - l_discount)) AS promo_revenue
FROM LINEITEM, PART
WHERE l_partkey = p_partkey AND l_shipdate >= date '1995-09-01'
AND l_shipdate <date '1995-09-01' + interval '1' month

Figure 6 shows the query plan for this query when run in the Smart SSD. Overall, this query plan is the same as the plan that was used in Section 4.2.2.1 (except that the selection was replaced by an aggregation), since the size of the PART table is also much smaller than the LINEITEM table, (i.e., $|\text{LINEITEM}| \simeq 30|\text{PART}|$), and the hash table for the PART table fits in memory.

Figure 7 shows the results with the (regular) SSD, and the Smart SSD (with the NSM and the PAX layouts). The Smart SSD with the PAX layout improves the overall query response time by 1.3X over the (regular) SSD case. As discussed in [7], the Smart SSD achieves greater benefits when the query requires fewer computations (number of CPU cycles) per data page. However, this query requires a large number of CPU cycles to be executed per page compared to the query used in Section 4.2.2.1, which saturates the performance of the Smart SSD. That is why the performance improvement of this query is lower than the selection with join query in Section 4.2.2.1 (1.3X vs. 2.2X).

4.2.3 Energy Savings

In this section, we show the energy consumption profile with the TPC-H Query 6 [3]. Table 3 shows the results with the HDD, the (regular) SSD, and the Smart SSD (with the NSM and the PAX layouts). We note that compared to the Smart SSD case with PAX, the HDD case consumes 11.6X more energy at the whole server/system level, and about 14.3X more energy in just the I/O subsystem. In addition, using the Smart SSD (with PAX) is 1.9X and 1.4X more energy efficient than the regular SSD in the entire system and I/O subsystem

	SAS HDD	SAS SSD	Smart SSD (NSM)	Smart SSD (PAX)
Elapsed time (seconds)	1,186	175	134	103
Entire System Energy (kJ)	430	69	48	37
I/O Subsystem Energy (kJ)	200	19	18	14

Table 3: Energy Consumptions for the TPC-H Query 6

perspectives. The energy gains are likely to be much bigger with more balanced host machines than our test-bed machine. As explained, we observed 11.6X and 1.9X energy gains for the entire system, over the HDD and the SSD, respectively. However, if we only consider the energy consumption over the base idle energy (235W), then these gains become 12.4X and 2.3X over the HDD and the SSD, respectively.

4.3 Discussion

On the DBMS side, the implication of using a Smart SSD for query processing has other ripple effects. One key area is around caching in the buffer pool. If there is a copy of the data in the buffer pool that is more current than the data in the SSD, pushing the query processing to the SDD may not be feasible. Similarly, queries with any updates cannot be processed in the SSD without appropriate coordination with the DBMS transaction manager. If the database is immutable then some of these problems become easier to handle.

In addition, there are other implications for the internals of existing DBMSs, including query optimization. If all or part of the data is already cached in the buffer pool, then pushing the processing to the Smart SSD may not be beneficial (from both the performance and the energy consumption perspectives). In addition, even when processing the query the usual way is less efficient than processing all or part of the query inside the Smart SSD, we may still want to process the query in the host machine as that brings data into the buffer pool that can be used for subsequent queries.

Finally, using a Smart SSD can change the way in which we build database servers/appliances. For example, if the issues outlined above are fixed, and Smart SSDs in the near future have both significantly more processing power and are easier to program, then one could build appliances that have far fewer compute and memory resources in the host server than what typical servers/appliances have today. Thus, pushing the bulk of the processing to Smart SSDs could produce a data processing system that has higher performance and potential a lower energy consumption profile than traditional servers/appliances.

At the extreme end of this spectrum, the host machine could simply be the coordinator that stages computation across an array of Smart SSDs, making the system look like a parallel DBMS with the master node being the host server, and the worker nodes in the parallel system being the Smart SSDs. The Smart SSDs could basically run lightweight isolated SQL engines internally that are globally coordinated by the host node. Of course, the challenges associated with using the Smart SSDs (e.g. buffer pool caching and transactions as outlined above) must be addressed before we can approach this end of the design spectrum.

5 Conclusion

The results in this paper show that Smart SSDs have the potential to play an important role when building high-performance database systems/appliances. Our end-to-end results using SQL Server and a Samsung Smart SSD demonstrated significant performance benefits and a significant reduction in energy consumption for the entire server over a regular SSD. While we acknowledge that these results are preliminary (we only tested a limited class of queries and on only one server configuration), we also feel that there are potential new opportunities for crossing across the traditional hardware and software boundaries with Smart SSDs.

A significant amount of work remains. On the SSD vendor side, the existing tools for development and debugging must be improved if Smart SSDs are to have a bigger impact. We also found that the hardware inside our Smart SSD device is limited, and that the CPU quickly became a bottleneck as the Smart SSD that we used was not designed to run general purpose programs. The next step must be to add in more hardware (CPU, SRAM and DRAM) so that the DBMS code can run more effectively inside the SSD. These enhancements are absolutely crucial to achieve the 10X or more benefit that Smart SSDs have the potential of providing (see Figure 1). The hardware vendors must, however, figure out how much hardware they can add to fit both within their manufacturing budget (Smart SSDs still need to ride the commodity wave) and the associated power budget for each device. On the software side, the DBMS vendors need to carefully weigh the pros-and-cons associated with using smart SSDs. Significant software development and testing time will be needed to fully exploit the functionality offered by Smart SSDs. There are many interesting research and development issues that need to be further explored, including extending the query optimizer to push operations to the Smart SSD, designing algorithms for various operators that work inside the Smart SSD, considering the impact of concurrent queries, examining the impact of running operations inside the Smart SSD on buffer pool management, considering the impact of various storage layout, etc. To make these longer-term investments, DBMS vendors will likely need the hardware vendors to remove the existing roadblocks.

Overall, the computing hardware landscape is changing rapidly and Smart SSDs present an interesting additional new axis for thinking about how to build future high-performance database servers/appliances. Our results indicate that there is a significant potential benefit for database hardware and software vendors to come together to explore this opportunity.

6 Acknowledgements

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab, and by the National Science Foundation under grant IIS-0963993.

References

[1] Lsi, sas 9211-4i hba. http://www.lsi.com/channel/products/storagecomponents/Pages/LSISAS9211-4i.aspx.

[2] Microsoft sql server 2012. http://www.microsoft.com/sqlserver.

[3] Tpc benchmark h (tpc-h). http://www.tpc.org/tpch.

[4] A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. Technical report, Oracle Corp, 2012.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[6] P. Francisco. The netezza data appliance architecture: A platform for high performance data warehousing and analytics. In *IBM Redbook*, 2011.

[7] J. Do, Y-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: opportunities and challenges In *SIGMOD Conference*, pages 1221–1230. ACM, 2011.

Supporting Transactional Atomicity in Flash Storage Devices

Woon-Hak Kang[†] Sang-Won Lee[†] Bongki Moon[‡]

Gi-Hwan Oh[†] Changwoo Min[†]

[†]College of Info. and Comm. Engr.
Sungkyunkwan University
Suwon 440-746, Korea
{*woonagi319, swlee,wurikiji,multics69*}*@skku.edu*

[‡]School of Computer Science and Engineering
Seoul National University
Seoul, 151-744, Korea
bkmoon@snu.ac.kr

Abstract

Flash memory does not allow data to be updated in place, and the copy-on-write strategy is adopted by most flash storage devices. The copy-on-write strategy in modern FTLs provides an excellent opportunity for offloading the burden of guaranteeing the transactional atomicity from a host system to flash storage and for supporting atomic update propagation. This paper presents X-FTL *as a model case of exploiting the opportunity in flash storage to achieve the transactional atomicity in a simple and efficient way.* X-FTL *drastically improves the transactional throughput almost for free without resorting to costly journaling schemes. We have implemented* X-FTL *on an SSD development board called OpenSSD, and modified* SQLite *and* `ext4` *file system minimally to make them compatible with the extended abstractions provided by* X-FTL*. We demonstrate the effectiveness of* X-FTL *using real and synthetic* SQLite *workloads for smartphone applications.*

1 Introduction

An update action in a database system may involve multiple pages, and each of the pages in turn usually involves multiple disk sectors. A sector write is done by a slow mechanical process and can be interrupted by a power failure. If a failure occurs in the middle of a sector write, the sector might be only partially updated. A sector write is thus considered non-atomic by most contemporary database systems [3, 11].

Atomic propagation of one or more pages updated by a transaction can be implemented by shadow paging or physical logging. However, the cost will be considerable during normal processing. Although some commercial database systems adopt a solution based on physiological logging for I/O efficiency and and flexible locking granularity, others still rely on costlier but less sophisticated mechanisms based on redundant writing to limit the scale of its code base. For example, InnoDB uses a double-write-buffer strategy, and SQLite runs with rollback or write-ahead journaling for transactional atomic updates.

Most contemporary mobile devices, if not all, use flash memory as storage media to store data persistently. Since flash memory does not allow any page to be overwritten in place, a page update is commonly carried out

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

by leaving the existing page intact and writing the new content into a clean page at another location [5, 9]. This strategy is called *copy-on-write (CoW)*. Whether it is intended or not, the net effect of copy-on-write operations by a flash memory drive remarkably resembles what the shadow paging mechanism achieves [10]. This provides an excellent opportunity for supporting atomic update propagation almost for free.

This paper presents *X-FTL* [6] as a model case of exploiting the opportunity in flash storage to achieve the transactional atomicity in a simple and efficient way. We describe the X-FTL approach and its extended abstractions, and sketch how *X-FTL* can be implemented in an SSD using a development board called OpenSSD. As use cases of *X-FTL*, we explain how SQLite and `ext4` file system can be minimally modified so as to make them compatible with *X-FTL*, and demonstrate the effectiveness of *X-FTL* using real and synthetic SQLite workloads for smartphone applications.

2 Transactional Support in SQLite

In the era of smartphones and mobile computing, many popular applications such as Facebook, twitter, Gmail, and even Angry birds game manage their data using SQLite. This is mainly due to the development productivity and solid transactional support provided by the SQL interface.

In order to support transactional atomicity, SQLite processes a page update by copying the original content to a separate *rollback* file or appending the new content to a separate *write-ahead log*. This is often cited as the main cause of tardy responses in smartphone applications [7, 8]. According to a recent survey [8], approximately 70% of all write requests are for SQLite databases and related files. Considering the increasing popularity of smart mobile platforms, improving the I/O efficiency of SQLite is a practical and critical problem that should be addressed immediately.

SQLite adopts *force* and *steal* policies for buffer management. When a transaction commits, all the pages updated by the transaction are force-written to a stable storage using the `fsync` command. When the buffer runs out of free pages, even uncommitted updates can be written to a stable storage.

In order to support the atomicity of transaction execution without the atomicity of a sector write, SQLite operates usually in either *rollback* mode [3] or *write-ahead log* mode [4]. If a transaction updates a page in *rollback* mode, the original content of the page is copied to the rollback journal before updating it in the database, so that the change can always be undone if the transaction aborts. The opposite is done in *write-ahead log* mode. If a transaction updates a page in *write-ahead log* mode, the original content is preserved in the database and the modified page is appended to a separate log, so that any committed change can always be redone by copying it from the log. The change is then later propagated to the database by periodical checkpointing. The delayed propagation in *write-ahead log* mode allows a transaction to maintain its own *snapshot* of the database and enable readers to run fast without being blocked by a writer.

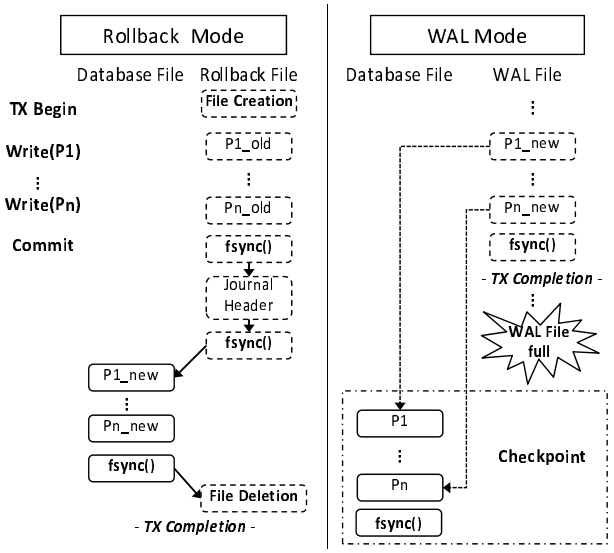


Figure 1: SQLite Journal Modes

The I/O behaviors of SQLite, as depicted in Figure 1, depend on which mode it runs in. If SQLite runs in

rollback mode, a journal file is created and deleted whenever a new transaction begins and ends. This increases I/O activities significantly for updating metadata. If SQLite runs in *write-ahead log* mode, a log file is reused and shared by many transactions until the log file is cleared by a checkpointing operation. Thus, the overhead of updating metadata is much lower when SQLite run in *write-ahead log* mode. Another aspect of I/O behaviors is how frequently files are synced. SQLite invokes `fsync` system calls more often when it runs in *rollback* mode than in *write-ahead log* mode. Since the header page of a journal file requires being synced separately from data pages, SQLite needs to invoke at least one more `fsync` call for each committing transaction.

SQLite relies heavily on the use of *rollback* journal files and *write-ahead log* as well as frequent file sync operations for transactional atomicity and durability. The I/O inefficiency of this strategy is the main cause of tardy responses of applications running on SQLite. Our goal is to achieve I/O efficient, database-aware transactional support at the flash based storage level so that SQLite and similar applications can simplify their logics for transaction support and hence run faster. We have developed a new flash translation layer (FTL) called *X-FTL*. With *X-FTL*, SQLite and other upper layer applications such as a file system can achieve transactional atomicity and durability as well as metadata journaling with minimum overhead and redundancy.

3 X-FTL for Transactional Atomicity

This section provides a brief overview of the design principles and the abstractions of *X-FTL*. *X-FTL* supports atomic page updates at the flash based storage level, so that upper layers such as SQLite and a file system can be freed from the burden of heavy redundancy of duplicate page writes. Also, we present the performance evaluation carried out to analyze the impact of *X-FTL* on SQLite.

3.1 Design Principles

The design objectives of *X-FTL* are threefold. First, *X-FTL* takes advantage of the copy-on-write mechanism adopted by most flash-based storage devices [5], so that it can achieve transactional atomicity and durability at low cost with no more redundant writes than required by the copy-on-write operations themselves. This is especially important for SQLite that adopts the *force* policy for buffer management at commit time of a transaction.

Second, *X-FTL* aims at providing atomic propagation of page updates for individual pages separately or as a group without being limited to SQLite or any specific domain of applications. So the abstractions of *X-FTL* must introduce minimal changes to the standards such as SATA, and the changes must not disrupt existing applications.

Third, SQLite and other upper layer applications should be able to use *X-FTL* services without considerable changes in their code. In particular, required changes, if any, must be limited to the use of extended abstractions provided by *X-FTL*.

This approach is novel in that it attempts to turn the weakness of flash memory (*i.e.*, being unable to update in place) into a strong point (*i.e.*, inherently atomic propagation of changes). Unlike the existing FTLs with support for atomic write [12, 13, 14], *X-FTL* supports atomicity of transactions without contradicting the steal policy of database buffer management at no redundant writes. This enables low-cost transactional support as well as minimal write amplification, which extends the life span of a flash storage device.

3.2 X-FTL Architecture and Abstractions

In the core of *X-FTL* is the *transactional logical-to-physical page mapping table* (or *X-L2P* in short) as shown in Figure 2. The *X-L2P* table is used in combination with a traditional page mapping table (or *L2P* in short) maintained by most FTLs. The *L2P* and *X-L2P* tables appear in the left and right sides of Figure 2, respectively.

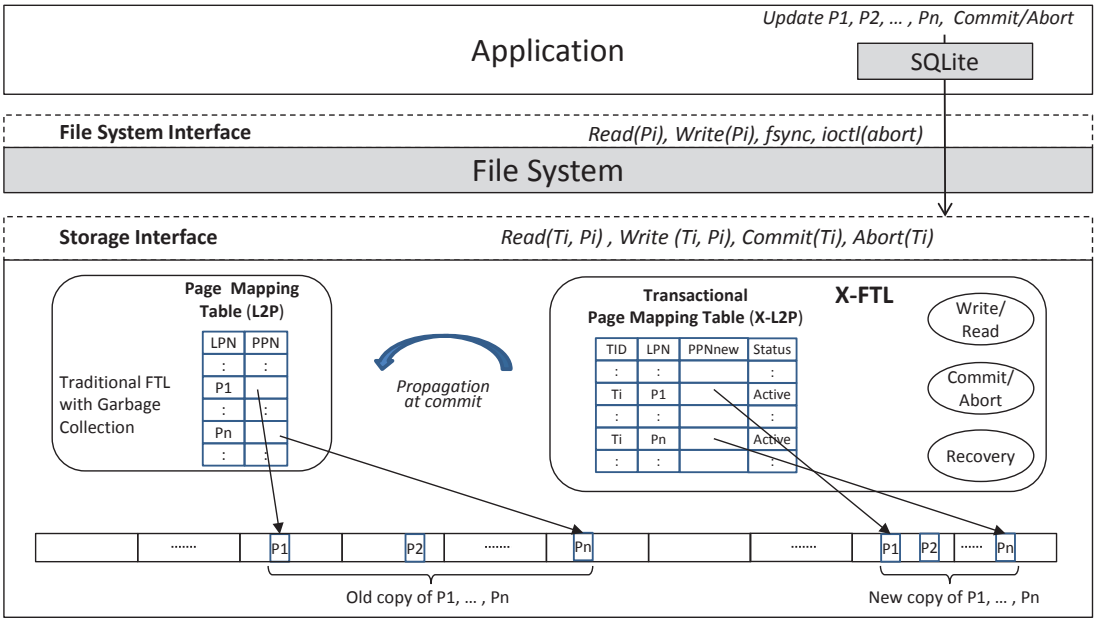


Figure 2: *X-FTL* Architecture: an FTL for Transactional Atomicity

In order to provide transactional support at the storage level, we add to it more information such as the transaction id of an updater, the physical address of a page copied into a new location, and the status of the updater transaction. This will allow us to have full control over which pages can be reclaimed for garbage collection. Specifically, an old page invalidated by a transaction will not be garbage-collected as long as the updater transaction remain active, because the old page may have to be used to rollback the changes in case the transaction gets aborted. If the updater transaction commits successfully, the information on the old page can be released from the *X-L2P* table so that it can be reclaimed for garbage collection.

Obviously the additional information on transactions must be passed from transactions themselves to the flash-based storage, but it cannot be done through the standard storage interface such as SATA. We have extended the SATA interface so that the transaction id can be passed to the storage device by `read` and `write` commands. Besides, two new commands, `commit` and `abort`, have been added to the SATA interface so that the change in the status of a transaction can also be passed. The extensions we have made to the SATA interface are summarized below.

write(tid t, page p) The write command of SATA is augmented with an id of a transaction t that writes a logical page p. This command writes the content of p into a clean page in flash memory and add a new entry $(t, p, paddr, active)$ into the *X-L2P* table, where $paddr$ is the physical address of the page the content of p is written into.

read(tid t, page p) The read command of SATA is also augmented with an id of a transaction t that reads a logical page p. This command reads the copy of p from the database snapshot of transaction t. Depending on whether t is the transaction that updated p most recently or not, a different version of p may be returned.

commit(tid t) This is a new command added to the SATA interface. When a commit command is given by a transaction t, the physical addresses of new content written by t become permanent and the old addresses are released so that the old page can be reclaimed for garbage collection.

abort(tid t) This is a new command added to the SATA interface. When an abort command is given by a

transaction t, the physical addresses of new content written by t are abandoned and those pages can be reclaimed for garbage collection.

Note that the SATA command set is not always available for SQLite and other applications that access files through a file system. Instead of invoking the SATA commands directly from SQLite, we have extended the `ioctl` and `fsync` system calls so that the additional information about transactions can be passed to the storage device through the file system.

3.3 X-FTL Advantages for SQLite

With *X-FTL* that supports atomic page updates at the storage device level, the runtime overhead of SQLite can be reduced dramatically. First, SQLite does not have to write a page (physically) more than once for each logical page write. Second, a single invocation of `fsync` call will be enough for each committing transaction because all the updates are made directly to the database file. Consequently, the I/O efficiency and the transaction throughput of SQLite can improve significantly for any workload with non-trivial update activities.

The current version (3.7.10) of SQLite supports the atomicity of a transaction that updates multiple database files but it is awkward or incomplete [3, 4]. When a transaction updates two or more database files in *rollback* mode, a master journal file, in addition to regular journal files, should be created to guarantee the atomic propagation of the entire set of updates made against the database files [3]. With *X-FTL*, in contrast, SQLite keeps trace of the multi-file updates in the *X-L2P* table under the same transaction id and supports the atomicity of the transaction without additional effort.

3.4 Performance Evaluation

In order to understand the impact of *X-FTL* on SQLite, we have implemented *X-FTL* on an SSD development board called OpenSSD, and modified SQLite and `ext4` file system minimally to make them compatible with the extended abstractions provided by *X-FTL*.

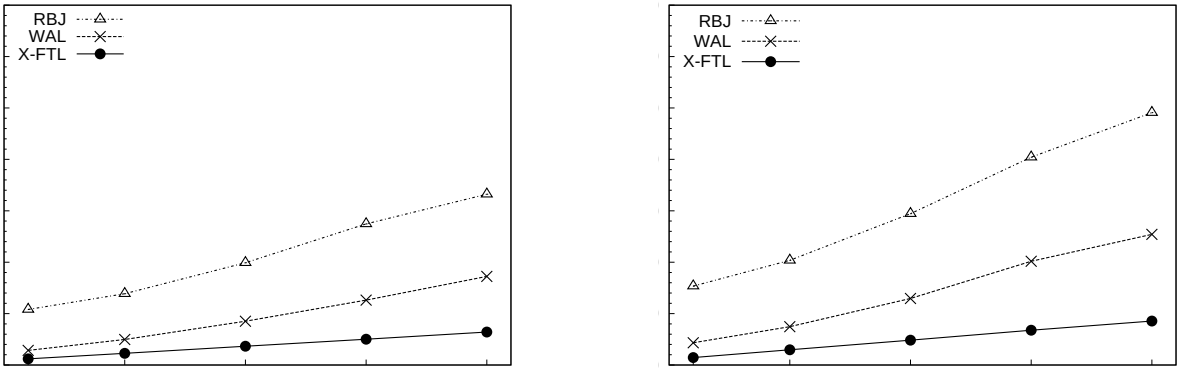
Using two different database workloads, we ran SQLite in *rollback* and *write-ahead log* modes on top of the (unchanged) `ext4` file system with the OpenSSD board running the original FTL. We also ran the modified SQLite on top of the `ext4` file system with the changed system calls with the OpenSSD board running *X-FTL*. The workloads are a synthetic workload, a set of traces from four popular Android benchmarks.

3.4.1 Experimental Setup

The OpenSSD development platform [2] is equipped with Samsung K9LCG08U1M flash memory chips. These flash memory chips are of MLC NAND type with 8KB pages and 128 pages per block. The host machine is a Linux system with 3.5.2 kernel running on Intel core i7-860 2.8GHz processor and 2GB DRAM. We used the `ext4` file system in *ordered* mode for metadata journaling when SQLite ran in *rollback* or *write-ahead log* mode. When SQLite ran on *X-FTL*, the file system journaling was turned off but the changes we added to the file system were enabled. The version of SQLite used in this paper was 3.7.10, which supports both *rollback* and *write-ahead log* modes. The page size was set to 8KB to match the page size of the flash memory chips installed on the OpenSSD board.

3.4.2 Run-Time Performance

This section demonstrates the effectiveness of *X-FTL* by comparing the performance of SQLite with and without *X-FTL*. We use the `RBJ`, `WAL` and `X-FTL` symbols to denote the execution of SQLite in *rollback* mode, *write-ahead log* mode and with *X-FTL* enabled, respectively. Each performance measurement presented in this section was an average of five runs or more.



(a) GC validity 30% (b) GC validity 50%

Figure 3: SQLite Performance (x1,000 Synthetic Transactions)

Mode	Host-side					FTL-side			
	SQLite		File System	Total Counts	fsync calls	Write	Read	GC	Erase
	DB	Journal							
RBJ	6,230	7,222	15,987	29,439	2,999	243,639	9,792	756	2,044
WAL	3,523	5,754	3,646	12,923	1,013	92,979	3,472	409	897
X-FTL	5,211	0	994	6,205	994	33,239	2,011	115	243

Table 4: I/O Count (# of updated pages per transaction = 5, GC validity = 50%)

3.4.2.1 Synthetic workload The synthetic workload consists of a `partsupply` table created by the `dbgen` tool of the TPC-H benchmark. This table contains 60,000 tuples of 220 bytes each. Each transaction reads a fixed number of tuples using random `partkey` values, updates the `supplycost` field of each tuple, and commits. In the synthetic workload, we varied the number of updates requested by a transaction from one to 20, and 1,000 transactions were executed for each fixed number of updates.

To evaluate the effect of garbage collections by FTL, we controlled aging of the OpenSSD flash memory chips such that the ratio of valid pages carried over by garbage collection was approximately 30% or 50%. Garbage collection is always done for an individual flash memory block. When a flash memory block is picked up for garbage collection, the pages marked as valid in the block are copied into a new block, while invalid ones are simply discarded. For instance, if the ratio of valid pages is 50%, 64 out of 128 pages will be copied from the victim block to a new block, and the new block will contain just 64 free pages. This implies the garbage collection will leave just half of the block available for use after all the cost of erasing a block and copying 64 pages.

Figure 3 shows the elapsed times of SQLite when it ran in rollback or write-ahead log mode and when it ran in *off* mode with *X-FTL*. In Figure 3(a), under GC validity 30%, *X-FTL* helped SQLite process transactions much faster than *write-ahead log* and *rollback* modes by 2.7 and 9.7 times, respectively. In Figure 3(b), under GC validity 50%, the improvement ratios were even higher, 3.2 and 11.7 times, respectively. The considerable gain in performance was direct reflection of reductions in the number of write operations and `fsync` system calls. Recall that, with the force policy, SQLite force-writes all the updated pages when a transaction commits.

Table 4 compares *rollback* and *write-ahead log* modes with *X-FTL* with respect to the number of writes and `fsync` calls. In the case of *rollback*, in particular, both numbers were very high. This is because SQLite had

to create and delete a journal file for each transaction and consequently had to use `fsync` call very frequently. In *write-ahead log* mode, SQLite wrote twice as many pages as running it with *X-FTL*, because it had to write pages to both log and database files. Table 4 drills down the I/O activities further for the case when the number of pages updated per transaction was five. In the 'Host-side' columns, we counted the number of page writes requested by SQLite and the number of metadata page writes requested by the file system separately as well as the total number of `fsync` calls.

In the 'FTL-side' columns, we counted the number of pages written and read (including those copied-back internally in the flash memory chips) as well as the frequencies of garbage collection (GC) and block erase operations. The write and block erase counts in Table 4 include the data pages and blocks garbage collected and the metadata blocks erased by FTL.

3.4.2.2 Android Smartphone Workload Android Smartphone workload consists of traces obtained by running four popular applications on an Android 4.1.2 Jelly Bean SDK, namely, RL Benchmark [1], Gmail, Facebook, and a web browser. RL Benchmark is a popular benchmark used for performance evaluation of SQLite on Android platforms. We modified the source code of SQLite to capture all the transactions and their SQL statements.

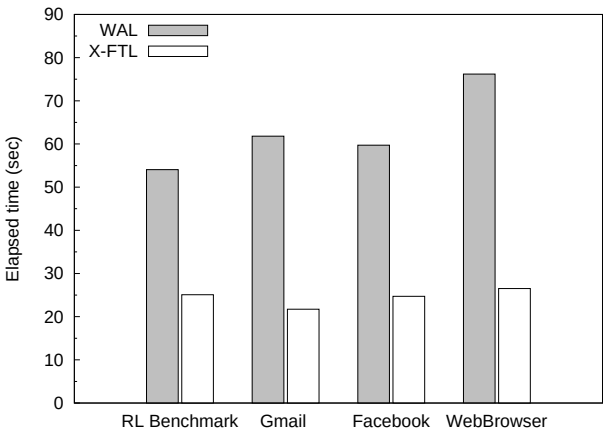


Figure 4: Smartphone Workload Performance

In Figure 4, we measured the elapsed time taken by SQLite to process each workload completely. Since the performance gap between the *rollback* and *write-ahead log* modes was similar to that observed in the synthetic workload, we did not include the elapsed time of *rollback* mode for the clarity of presentation. Across all the four traces, SQLite performed 2.4 to 3.0 times faster when it ran with *X-FTL* than when it ran in *write-ahead log* mode. These results match the elapsed times and the trend of I/O activities observed in the synthetic workloads (shown in Figure 3(b)).

4 Concluding Remarks

X-FTL is a novel transactional FTL which can efficiently support atomic propagation of one or more pages updated by transactional applications (*e.g.*, SQLite databases and NoSQL key-value stores) to a flash memory storage device. The main contributions of the *X-FTL* scheme are: (1) it realizes low-cost atomic update for individual pages, (2) it provides the awareness of database semantics in page updates to support the atomicity of transactions, and (3) it exposes an extended abstraction (and APIs) to upper layer applications such as SQLite.

There are many non-database applications that require the semantics of transactional atomicity semantics (*i.e.*, the atomic propagation of a group of data pages) without a sophisticated recovery mechanism. *X-FTL* provides an effective means for transactional support as well as a simple storage abstraction with extended functionality. *X-FTL* also demonstrates that advanced storage devices could offload essential functions from the host system to the devices and help simplify the software stack of the host system.

References

[1] RL Benchmark:SQLite. `http://redlicense.com/`.

[2] OpenSSD Project. http://www.openssd-project.org/, 2011.

[3] Atomic Commit In SQLite. http://www.sqlite.org/atomicommit.html, 2012.

[4] Write-Ahead Logging. http://www.sqlite.org/wal.html, 2012.

[5] A. Ban. Flash file system, Apr 1995. US Patent 5,404,485.

[6] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 97–108, New York, NY, USA, 2013. ACM.

[7] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of USENIX conference on File and Storage Technologies (FAST'12)*, 2012.

[8] K. Lee and Y. Won. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. In *Proceedings of ACM EMSOFT*, pages 23–32, 2012.

[9] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation. *ACM Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[10] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar 1977.

[11] C. Mohan. Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging. In *Proceedings of ICDE*, pages 324–331, 1995.

[12] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of International Conference on High-Performance Computer Architecture (HPCA '11)*, pages 301–311, 2011.

[13] S. Park, J. H. Yu, and S. Y. Ohm. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics (ISCE 2005)*, pages 155 – 160, Jun 2005.

[14] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, 2008.

Integrating SSD Caching into Database Systems

Xin Liu
University of Waterloo, Canada
x39liu@uwaterloo.ca

Kenneth Salem
University of Waterloo, Canada
kmsalem@uwaterloo.ca

Abstract

Flash-based solid state storage devices (SSDs) are now becoming commonplace in server environments. In this paper, we consider the use of SSDs as a persistent second-tier cache for database systems. We argue that it is desirable to change the behavior of the database system's buffer cache when a second-tier SSD cache is used, so that the buffer cache is aware of which pages are in the SSD cache. We propose such an SSD-aware buffer cache manager, called GD2L. An interesting side effect of SSD-aware buffer cache management is that the rate with which a page will be evicted or written from the buffer cache will change when that page is moved into or out of the second-tier SSD cache. We also propose a technique, called CAC, for managing the contents of the second-tier cache. CAC is aware that moving pages into or out of the SSD cache will change their physical read and write rates. It anticipates these changes when making decisions about which pages to cache at the second tier.

1 Introduction

Flash-based solid state storage devices (SSDs) are now becoming commonplace in server environments. When SSDs are present, there are several ways in which they could be exploited by database management systems (DBMS). On servers that have *hybrid storage systems*, consisting of both SSDs and disk drives (HDDs), one option for the DBMS is to partition the database between the SSDs and the HDDs, so that each chunk of data is stored persistently either on the SSD or on the HDD, but not both [1, 9, 13]. Alternatively, SSDs can be used as an intermediate cache between the DBMS buffer cache and the HDDs [2, 4, 5, 8, 10, 12]. In this case, the entire database resides on the HDDs, and portion of the database is also cached on the SSDs.

In this paper, we consider the latter scenario, in which the SSD is used as a persistent, intermediate cache. We also assume that both the SSD cache and the HDDs are visible to the database management system, so that it can take responsibility for managing the contents of the SSD cache. This is illustrated in Figure 1. When writing data to storage, the DBMS chooses which type of device to write it to.

The primary focus of previous work has been on how a DBMS should decide which data to cache in the SSD to maximize the system's overall I/O performance [1, 2, 5, 12, 13]. Because SSDs provide much better I/O performance than HDDs, this question is clearly important. However, if the goal is to maximize I/O performance then this work considers only part of the problem, since the DBMS manages two tiers of caching, not one. Most previous work assumes that the policies used to manage the database system's in-memory buffer cache are given and fixed. The goal is to design a caching strategy for the SSD without changing the way the DBMS buffer cache is managed.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

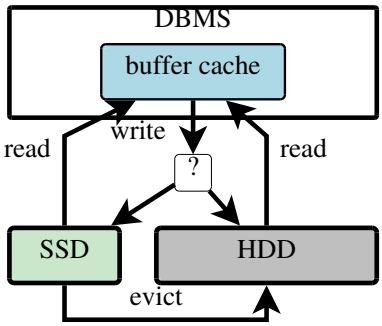


Figure 1: System Architecture

In this paper, we take the broader view, which brings both the database system's in-memory buffer cache and the SSD within scope. We consider two related problems. The first is determining which data should be retained in the DBMS buffer cache. The answer to this question is affected by the presence of an intermediate SSD cache because a database page that is evicted from the in-memory buffer cache can be brought back into the cache much more quickly if it is present in the SSD cache than if it is not. Thus, we propose a *cost-aware* DBMS buffer management technique, called *GD2L*, which takes this distinction into account.

The second problem is deciding which database pages should be retained in the SSD cache. Like some previous work, our approach bases these decisions on page read and write frequencies, attempting to fill the SSD with pages that will provide the greatest boost to I/O performance. However, the key observation is that, if the DBMS buffer cache is cost-aware, then those page read and write frequencies *depend on whether or not the page is present in the SSD*. Specifically, if a page is placed in the SSD cache, its read and write frequencies are likely to increase, since the buffer cache will view the page as a good eviction candidate. Conversely, moving a page out of the SSD cache will probably cause its read and write frequencies to drop. Thus, we propose an *anticipatory* technique, called CAC, for managing the contents of the SSDs. When deciding whether to place a page in the SSD cache, CAC predicts how such a move will affect the page's I/O frequencies and places the page into the SSD cache only if it is determined to be a good candidate under this predicted workload.

In this paper, we try to provide some intuition as to why it is important to consider both cache tiers in order to maximize I/O performance, and we present an overview of GD2L and CAC. More information, including a more detailed performance evaluation, can be found in a longer paper that appeared at VLDB'13 [11].

2 System Overview

As illustrated in Figure 1, we assume that the DBMS sees two types of storage devices, SSDs and HDDs. All database pages are stored on the HDD, where they are laid out according to the DBMS's secondary storage layout policies. In addition, copies of some pages are located in the SSD and copies of some pages are located in the DBMS buffer cache. Any given page may have a copy in the SSD, in the buffer cache, or both.

When the DBMS needs to read a page, the buffer cache is consulted first. If the page is cached in the buffer cache, the DBMS reads the cached copy. If the page is not in the buffer cache but it is in the SSD, it is read into the buffer cache from the SSD. If the page is in neither the buffer cache nor the SSD, it is read from the HDD.

If the buffer cache is full when a new page is read in, the buffer manager must evict a page according to its page replacement policy, which we present in Section 4. When the buffer manager evicts a page, the evicted page is considered for admission to the SSD if it is not already located there. SSD admission decisions are made by the SSD manager according to its *SSD admission policy*. If admitted, the evicted page is written to the SSD. If the SSD is full, the SSD manager must also choose a page to be evicted from the SSD to make room for the newly admitted page. SSD eviction decisions are made according to an *SSD replacement policy*. (The SSD

	in SSD	not in SSD
in buffer cache	$w_i W_S$	$w_i W_D$
not in buffer cache	$r_i R_S + w_i W_S$	$r_i R_D + w_i W_D$

Figure 2: Total I/O Cost for the ith Page, Depending on Cache Placement

admission and replacement policies are presented in Section 5.) If a page evicted from the SSD is more recent than the version of that page on the HDD, then the SSD manager must copy the page from the SSD to the HDD before evicting it, otherwise the most recent persistent version of the page will be lost. The SSD manager does this by reading the evicted page from the SSD into a staging buffer in memory, and then writing it to the HDD.

We assume that the DBMS buffer manager implements asynchronous page cleaning, which is widely used to hide write latencies from DBMS applications. When the buffer manager elects to clean a dirty page, that page is written to the SSD if the page is already located there. If the dirty page is not already located on the SSD, it is considered for admission to the SSD according to the SSD admission policy, in exactly the same way that a buffer cache eviction is considered. The dirty page will be flushed to the SSD if it is admitted there, otherwise it will be flushed to the HDD.

3 Page Placement Example

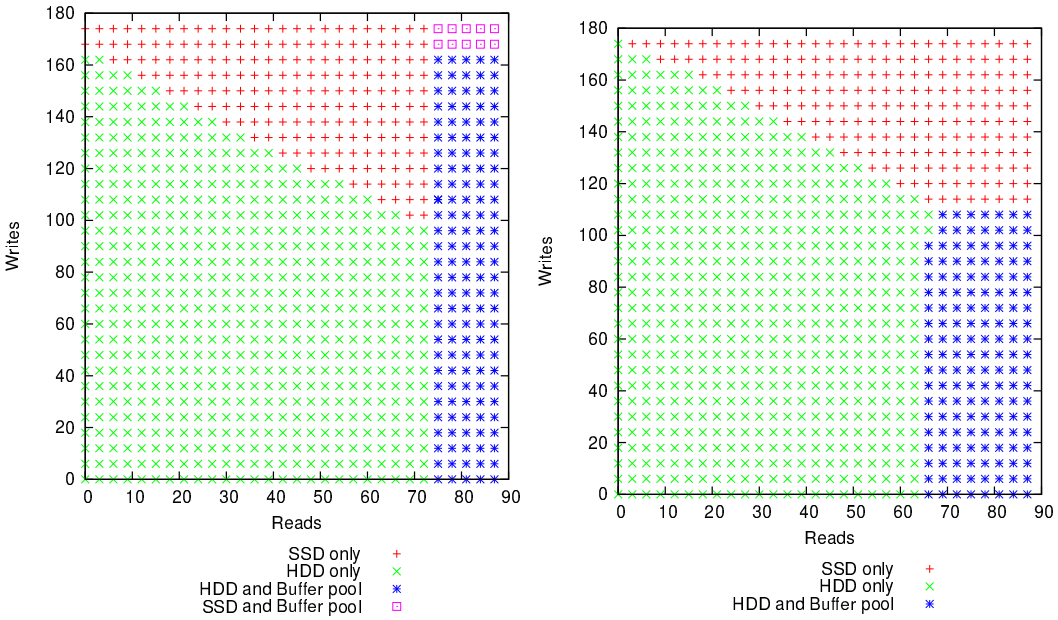
Before presenting the GD2L and CAC algorithms, we first try to develop some intuition for our two-tiered cache problem. To do this, we consider a simple static placement problem in a two-tier cache setting. Clearly, placement decisions will not be static in practice, in either cache. However, by first considering a simple static example, we hope to get some understanding of which pages belong in each cache.

Suppose we have a database consisting of N pages of data, of which at most C_M can fit in the DBMS in-memory buffer cache and C_S can fit in the SSD cache ($N > C_S > C_M$). We assume that a sequence of page read and write requests arrives from the DBMS, and that the ith page is read r_i times and written w_i times in the sequence.

Our goal is to determine which pages to place in each cache so that the total I/O cost of the request sequence is minimized. Placement is static. We'll assume R_D and W_D represent the costs of reading and writing a single page to the HDD, and R_S and W_S are the costs of reading and writing a page to the SSD. With these parameters, we can determine the total cost of the requests for each page, depending on its placement, as shown in Figure 2. The total I/O cost for the entire request sequence is simply the sum of the costs of the pages. Note that if a page is in the SSD, page writes are directed only to the SSD, not to the HDD, since the SSD cache is persistent. In this way, the SSD cache can improve the performance of writes as well as reads.

To illustrate the tradeoffs that arise in solving the two-tiered static placement problem, we consider a single problem instance in which the request sequence is chosen so that a 2D scatter plot of the pages according to their read and write counts will fill the space, as illustrated in Figure 3. This is not intended to be a realistic request sequence. Rather, it is merely intended to illustrate the read/write characteristics of the pages that get placed into each cache. We also choose $N = 900$, $C_S = 200$, and $C_M = 150$, and for I/O costs we choose $R_D = 12$, $R_S = 0.16$, $W_D = 12.5$ and $W_S = 0.4$. (These particular costs were adopted from Graefe [6].)

Figure 3(a) shows a *non-optimal* solution to this problem instance that was produced by a simple two-step process. In the figure, each point represents a page, and the points are coded to indicate where that page was placed in the solution, giving a visual overview of page placement. The first step in the two-step solution process was to determine an optimal placement of pages into memory, under the assumption that no pages are in the SSD, i.e., all pages are stored persistently on the HDD only. The second step in the process is to determine an optimal placement of pages into the SSD, given the memory placement chosen in the first step. This two-step approach is analogous to existing approaches to SSD cache management in which the management of the in-memory cache



(a) Page Placement from Two-Step Technique

(b) Optimal Page Placement

Figure 3: Two Solutions to an Instance of the Two-Tier Static Placement Problem

is taken as fixed and unchangeable.

Figure 3(b) shows an optimal solution to the same problem instance. The key differences between this solution and the two-step solution occur in the top right of the figures. In the optimal solution, pages with high read and write counts are placed in the SSD and not in the in-memory buffer cache. Instead, the buffer cache is used for pages with high read counts and lower write counts. In contrast, the two-step solution places pages with high read and write counts in the buffer cache. Most are not in the SSD, but the most frequently written pages are in both the buffer cache and the SSD.

Of course, this is only a single problem instance, and the placement boundaries will shift as the parameters change. However, this does provide some intuition about how to place pages in the two caches to minimize I/O costs. The optimal solution ensures that frequently written pages are in the SSD, to keep write costs low. It also avoids *cache inclusion* [15], i.e., it avoids keeping the same page in both the buffer cache and the SSD.

4 Buffer Cache Management

Most replacement policies for DBMS buffer caches, such as 2Q [7] or variants of LRU, are cost-oblivious, i.e., they do not consider the cost of reloading a page when making eviction decisions. In this section, we describe our proposed *cost-aware* buffer cache management technique, which we refer to as GD2L. GD2L is based on the GreedyDual algorithm [16], a cost-aware technique that was originally proposed for file caching.

The original GreedyDual algorithm takes into account the access costs of cached objects when making replacement decisions. It associates a non-negative cost $H(p)$ with each cached object p. When p is brought into the cache or referenced in the cache, $H(p)$ is set to the cost of retrieving p. To make room for a new object, the object with the lowest H in the cache, H_{min}, is evicted and the H values of all remaining objects are reduced by H_{min}. By reducing the objects' H values and resetting them upon access, GreedyDual ages objects that have not been accessed for a long time. The algorithm thus integrates temporal locality and cost concerns in a seamless fashion. GreedyDual is usually implemented using a priority queue of cached objects, prioritized based on their H value. With a priority queue, handling a hit and an eviction each require $O(\log k)$ time. Cao et

al. [3] have proposed a technique to avoid the cost of subtracting H_{min} from the H values of all cached objects when a page is evicted. That technique involves maintaining a global inflation value L that increases on each eviction, and increasing the initial H value of each newly-cached page by L.

In our setting, the cached objects are database pages, and there are only two possible initial values for $H(p)$: one corresponding to the cost of retrieving p from the SSD (R_S) and the other to the cost of retrieving p from the HDD (R_D). The GD2L algorithm is designed for this special case.

We implemented GD2L in MySQL's InnoDB storage engine. GD2L uses two queues to manage pages in the buffer cache: one queue (Q_S) is for pages that are located on the SSD, the other (Q_D) is for pages that are not on the SSD. Each queue is managed using the scan-resistant variant of LRU that is used by InnoDB. When eviction is necessary, GD2L evicts the page with the lowest H value, which will be located either at the LRU end of Q_S or at the LRU end of Q_D. In part by leveraging the technique proposed by Cao et al, GD2L achieves $O(1)$ time for handling both hits and evictions.

In a DBMS, when pages are modified in the buffer cache (dirty pages), they need to be copied back to the underlying storage device. InnoDB, like many other database systems, uses dedicated by *page cleaner* threads to clean dirty pages asynchronously. Since the page cleaners attempt to clean pages that are likely eviction candidates, we changed the page cleaners to reflect the eviction policies of the new cost-aware replacement policy. Our modified page cleaners check pages from the tails of both Q_S and Q_D. If there are dirty pages in both lists, the page cleaners compare their H values and choose dirty pages with lower H values to write back to the storage devices.

The original GreedyDual algorithm also assumed that a page's retrieval cost does not change while it is cached, but this is not true in our setting. A page's retrieval cost changes when it is moved into or out of the SSD. If a buffered page is moved into the SSD (e.g, when it is cleaned), then GD2L takes that page out of Q_D and places it into Q_S. It is also possible that a buffered page that is in the SSD will be evicted from the SSD (while remaining in the buffer cache). This may occur to make room in the SSD for some other page. In this case, GD2L removes the page from Q_S and inserts it to Q_D.

5 SSD Management

Pages are considered for SSD admission when they are cleaned or evicted from the DBMS buffer cache. Pages are always admitted to the SSD if there is free space available on the device. If there is no free space on the SSD when a page is cleaned or evicted from the DBMS buffer cache, the SSD manager must decide whether to place the page on the SSD and which SSD page to evict to make room for the newcomer. The SSD manager makes these decisions by estimating the benefit, in terms of reduction in overall read and write cost, of placing a page on the SSD. It attempts to keep the SSD filled with the pages that it estimates will provide the highest benefit. Our specific approach is called Cost-Adjusted Caching (CAC). CAC is specifically designed to work together with a cost-aware DBMS buffer cache manager, like the GD2L algorithm presented in Section 4.

To decide whether to admit a page p to the SSD, CAC estimates the benefit B, in terms of reduced access cost, that will be obtained if p is placed on the SSD. The essential idea is that CAC admits p to the SSD if there is some page p' already on the SSD cache for which $B(p') < B(p)$. To make room for p, it evicts the SSD page with the lowest estimated benefit.

We refer to a read and write requests issued to the SSD or the HDD as *physical* read and write requests. Suppose that a page p is not currently in the SSD, and has experienced $r(p)$ physical read requests and $w(p)$ physical write requests over some measurement interval prior to the admission decision. If this physical I/O load on p in the past were a good predictor of the I/O load p would experience in the future, a reasonable way to estimate the benefit of admitting p to the SSD would be

$$B(p) = r(p)(R_D - R_S) + w(p)(W_D - W_S) \tag{3}$$

Symbol	Description
r_D, w_D	Measured physical read/write count while not on the SSD
r_S, w_S	Measured physical read/write count while on the SSD
$\widehat{r_D}$, $\widehat{w_D}$	Estimated physical read/write count if never on the SSD
$\widehat{r_S}$, $\widehat{w_S}$	Estimated physical read/write count if always on the SSD
α	Miss rate expansion factor

Figure 4: Summary of Notation

where R_D, R_S, W_D, and W_S represent the costs of read and write operations on the HDD and the SSD.

Unfortunately, when the DBMS buffer manager is cost-aware, like GD2L, the physical read and write counts experienced by p in the past may be particularly poor predictors of its future physical I/O workload. In particular, if p is admitted to the SSD then we expect that its post-admission physical read and write rates will be much higher than its pre-admission rates. This is because GD2L will be more likely to evict p from the database buffer cache once p has been moved into the SSD. Since p will spend less time in the buffer cache, attempts by the database system to read p will be more likely to result in physical reads, causing p's physical read rate to increase. Since the DBMS page cleaners try to keep likely eviction candidates clean, p's physical write rate will increase as well. Put another way, the *buffer pool miss rate* of p will increase if p is moved into the SSD. Conversely, if a page p that is located on the SSD is evicted from the SSD, then we expect its buffer pool miss rate, and hence its physical I/O rates, to drop. Thus, we do not expect Equation 3 to provide a good benefit estimate when the DBMS uses cost-aware buffer management.

To estimate the benefit of placing page p on the SSD, we would like to know what its physical read and write workload would be if it were on the SSD. Suppose that $\widehat{r_S}(p)$ and $\widehat{w_S}(p)$ are the physical read and write counts that p would experience if it were placed on the SSD, and $\widehat{r_D}(p)$ and $\widehat{w_D}(p)$ are the physical read and write counts p would experience if it were not. Using these hypothetical physical read and write counts, we can write our desired estimate of the benefit of placing p on the SSD as follows

$$B(p) = (\widehat{r_D}(p)R_D - \widehat{r_S}(p)R_S) + (\widehat{w_D}(p)W_D - \widehat{w_S}(p)W_S) \tag{4}$$

Thus, the problem of estimating benefit reduces to the problem of estimating values for $\widehat{r_D}(p)$, $\widehat{r_S}(p)$, $\widehat{w_D}(p)$, and $\widehat{w_S}(p)$. The notation used in these formulas is summarized in Table 4.

To estimate $\widehat{r_S}(p)$, CAC uses *two* measured read counts: $r_S(p)$ and $r_D(p)$. In general, p may spend some time on the SSD and some time not on the SSD. $r_S(p)$ is the count of the number of physical reads experienced by p while p is on the SSD. $r_D(p)$ is the number of physical reads experienced by p while it is not on the SSD. The total number of physical reads experienced by p during the measurement interval is $r_S(p) + r_D(p)$. To estimate what p's total physical read count would be if it had been on the SSD full time during the measurement interval ($\widehat{r_S}$), CAC uses

$$\widehat{r_S}(p) = r_S(p) + \alpha r_D(p) \tag{5}$$

In this expression, the number of physical reads experienced by p while it was not on the SSD ($r_D(p)$) is multiplied by a scaling factor α to account for the fact that it would have experienced more physical reads during that period if it had been on the SSD. We refer to the scaling factor α as the *miss rate expansion factor*. A simple way to estimate α is to compare the overall miss rates of pages on the SSD to that of pages that are not on the SSD, although our implementation of CAC uses a more fine-gained estimate. CAC estimates the values of $\widehat{r_D}(p)$, $\widehat{w_D}(p)$, and $\widehat{w_S}(p)$ in a similar fashion:

$$\widehat{r_D}(p) = r_D(p) + \frac{r_S(p)}{\alpha} \quad (6) \quad \widehat{w_S}(p) = w_S(p) + \alpha w_D(p) \quad (7) \quad \widehat{w_D}(p) = w_D(p) + \frac{w_S(p)}{\alpha} \quad (8)$$

6 Performance Evaluation

We performed a variety of experiments to evaluate the performance of CAC and GD2L. All of our experiments were performed using TPC-C workloads [14]. The longer paper [11] includes a more thorough performance evaluation, including tests with different systems configurations and comparisons of GD2L and CAC to other proposed techniques for managing SSDs in database systems. Here, we present two of the experiments from that evaluation.

Our first experiment compares the TPC-C performance obtained by our MySQL test system when it uses different combinations of algorithms for managing its buffer cache and the SSD. Here, we consider three alternatives:

LRU+CC: This uses InnoDB's original, unmodified cost-oblivious algorithm (a scan-resistant variant of LRU) to manage the database buffer cache, and uses CC to manage the SSD. CC is identical to our proposed CAC technique, except that it is non-anticipatory, i.e., it does not attempt to predict the changes in page read and write rates that will occur as the page is moved into and out of the SSD. Specifically, CC uses Equation 3, rather than Equation 4, to estimate the benefit of each page.

GD2L+CC: This uses GD2L to manage the InnoDB buffer cache, and CC to manage the SSD.

GD2L+CAC: This uses GD2L to manage the buffer cache and CAC to manage the SSD.

We used a TPC-C scale factor of 300 warehouses, corresponding to an initial database sizes of approximately 30GB, fixed the SSD size at 10GB, and tested database buffer cache sizes of of 10%, 20%, and 40% of the SSD size (1GB, 2GB, and 4GB, respectively). Thus, in this setting, the database is substantially larger than the SSD.

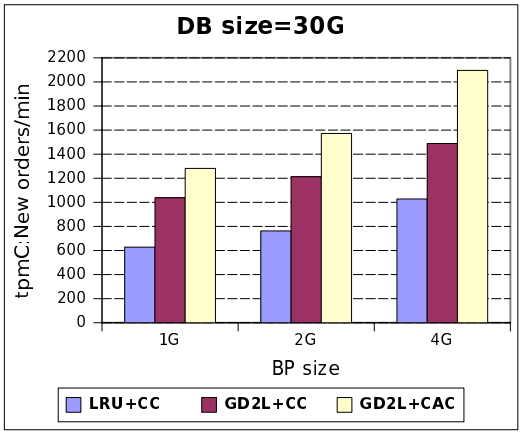
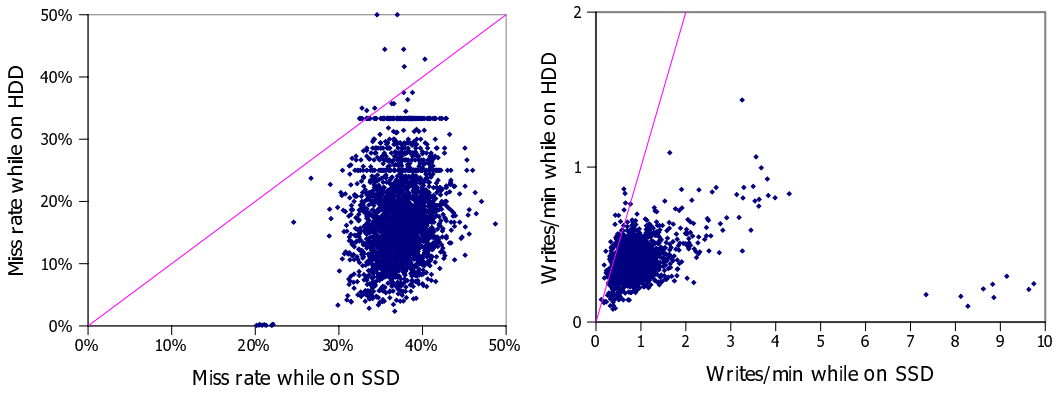


Figure 5: TPC-C Throughput Under Different Algorithm Combinations for Various DBMS Buffer Cache Sizes.

Figure 5 shows the TPC-C throughput of each of these algorithm combinations for each InnoDB buffer cache size. By comparing LRU+CC with GD2L+CC, we can see the benefit that is obtained by switching from InnoDB's original cost-oblivious buffer manager to GD2L. In our setting, this resulted in TPC-C throughput increases of about 40%-75%, depending on the size of the buffer cache. By comparing GD2L+CC with GD2L+CAC, we see the additional benefit that is obtained by switching from a non-anticipatory cost-based SSD manager (CC) to an anticipatory one (CAC). Figure 5 shows that GD2L+CAC provides additional performance gains above and beyond those achieved by GD2L+CC. Thus, it is important to use *both* cost-aware buffer management and compatible SSD manager, like CAC, to obtain the full benefit of the SSD cache. Together, GD2L and CAC provide a TPC-C performance improvement of about a factor of two relative to the LRU+CC baseline in these tests.



(a) Miss rate

(b) Write rate

Figure 6: Miss rate/write rate while on the HDD (only) vs. miss rate/write rate while on the SSD. Each point represents one page

Our second experiment is intended to show how cost-aware buffer managers, like GD2L, cause the read and write rates of database pages to change as those pages are moved into or out of the SSD. For this experiment, we drove our MySQL test system, with GD2L used for buffer management, using a TPC-C workload, and generated a log of all page I/O as well as movements of pages into and out of the SSD. By analyzing this log, we can calculate read and write rates for individual pages. We identified approximately 2000 pages in this trace that spent significant amounts of time both in the SSD cache and not in the SSD cache. For each such page, we calculated the read miss rate (in the DBMS buffer cache) of the page while it was in the SSD cache, and the read miss rate while it was not in the SSD cache. In addition, we calculated the pages physical write rate while in the SSD cache, and its physical write rate while not in the SSD cache.

Figure 6(a) shows scatter plots of the read miss rates of pages while in the SSD cache and while not in the SSD cache. Each point represents a single page. From these graphs, we can see that most pages have higher read miss rates when they are located in the SSD. This is the effect of making the DBMS buffer manager cost-aware: it is more likely to evict pages that are located in the SSD. Figure 6(b) is similar to Figure 6(a), but it shows page write rates rather than miss rates. Again, most pages have higher write rates while in the SSD cache, because GD2L's page cleaners try to clean pages that are likely to be evicted.

7 Conclusion

In this paper we present two new algorithms, GD2L and CAC, for managing the buffer cache and the SSD in a database management system. Both algorithms are cost-based and the goal is to minimize the overall I/O cost of the workload. We implemented the two algorithms in the InnoDB storage engine and evaluated them using a TPC-C workload. Our results show that we can achieve the best I/O performance using both cost-aware management of the DBMS buffer cache (GD2L) and a compatible SSD manager, like CAC. The SSD manager needs to anticipate I/O workload changes that will result from its SSD placement decisions.

References

[1] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. An object placement advisor for DB2 using solid state storage. *Proc. VLDB Endow.*, 2:1318–1329, August 2009.

[2] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3:1435–1446, September 2010.

[3] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 18–18, 1997.

[4] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010.

[5] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *Proc. SIGMOD Int'l Conf. on Management of Data*, pages 1113–1124, 2011.

[6] G. Graefe. The five-minute rule 20 years later: and how flash memory changes the rules. *Queue*, 6:40–52, July 2008.

[7] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 1994.

[8] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012.

[9] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1:514–525, August 2008.

[10] A. Leventhal. Flash storage memory. *Commun. ACM*, 51:47–51, July 2008.

[11] X. Liu and K. Salem. Hybrid storage management for database systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.

[12] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[13] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 939–950, New York, NY, USA, 2010. ACM.

[14] The TPC-C Benchmark. [online] http://www.tpc.org/tpcc/.

[15] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[16] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398