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

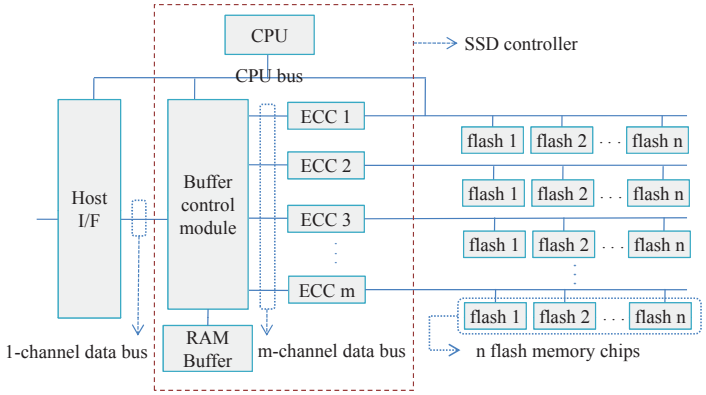


Figure 1: Internal architecture of flash SSDs

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.

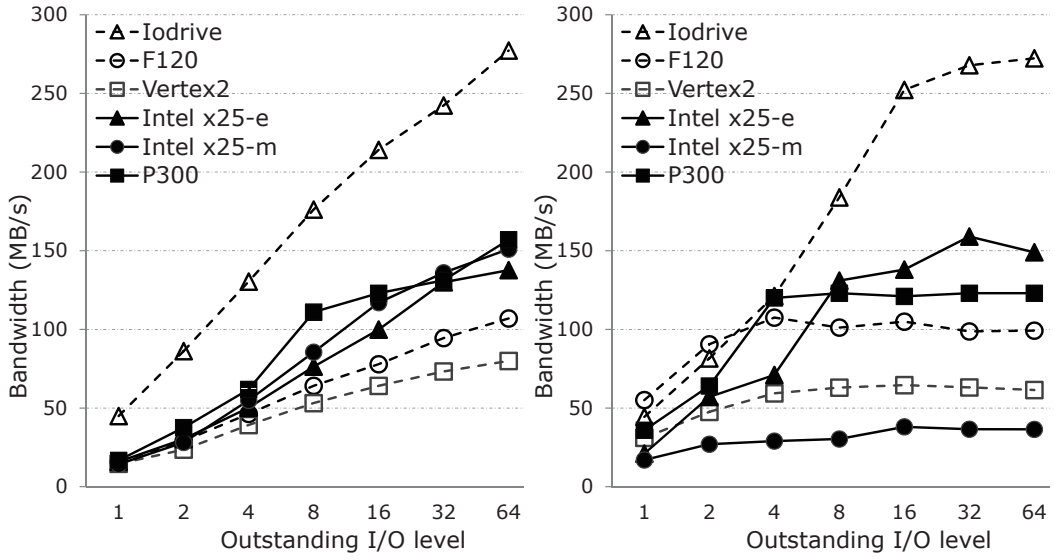
Due to the high performance gain from channel-level parallelism on a single flash SSD, we need to treat I/O parallelism as a top priority for optimizing I/O performance. In order to achieve it, a more lightweight method is needed since parallel processing (or mutlithreading) cannot be applied in every 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.

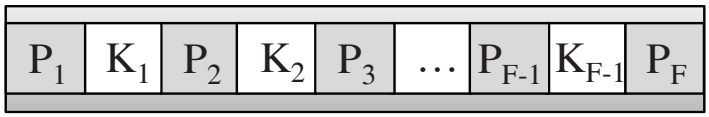


Figure 3: Internal node structure

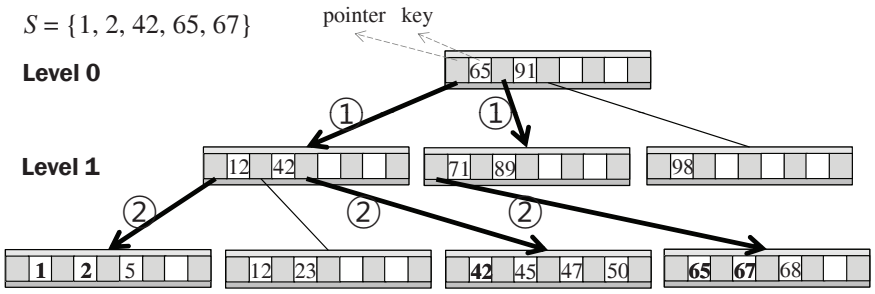


Figure 4: MPSearch with two psync I/Os

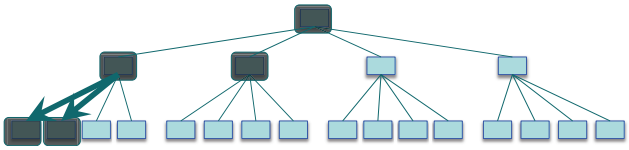


Figure 5: MPSearch with PioMax of 2

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

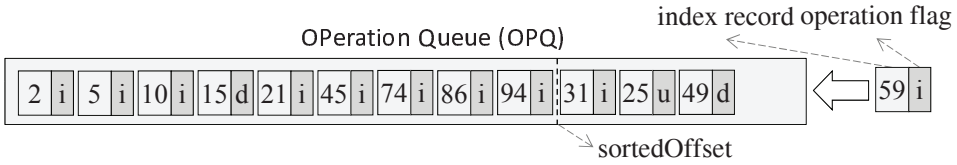


Figure 6: MPSearch with two psync I/Os

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.

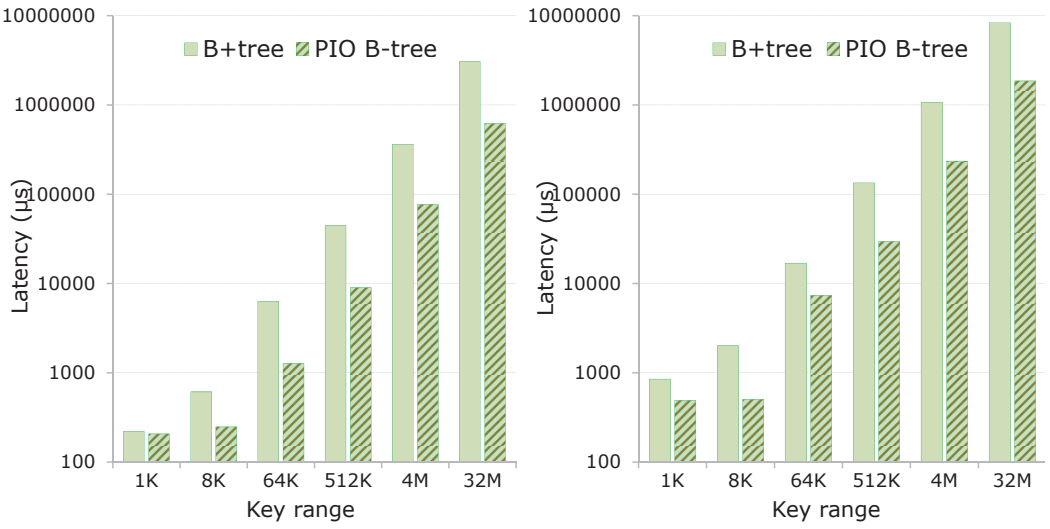
Likewise, multiple point-search operations can be queued in the expense of delaying response and simultaneously processed based on MPSearch algorithm.

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

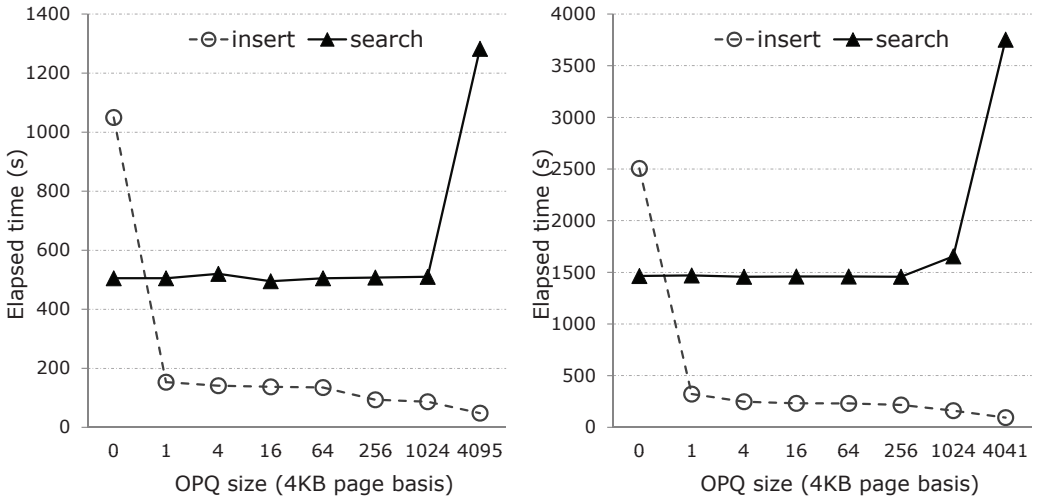
We evaluated the range search performance by requesting queries having various key ranges. The buffer pool size was fixed at 16MB. The gap between the start key and end key in the range was increased from 1024 (1K) to 33554432 (32M) by a factor of 8 at a time.

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

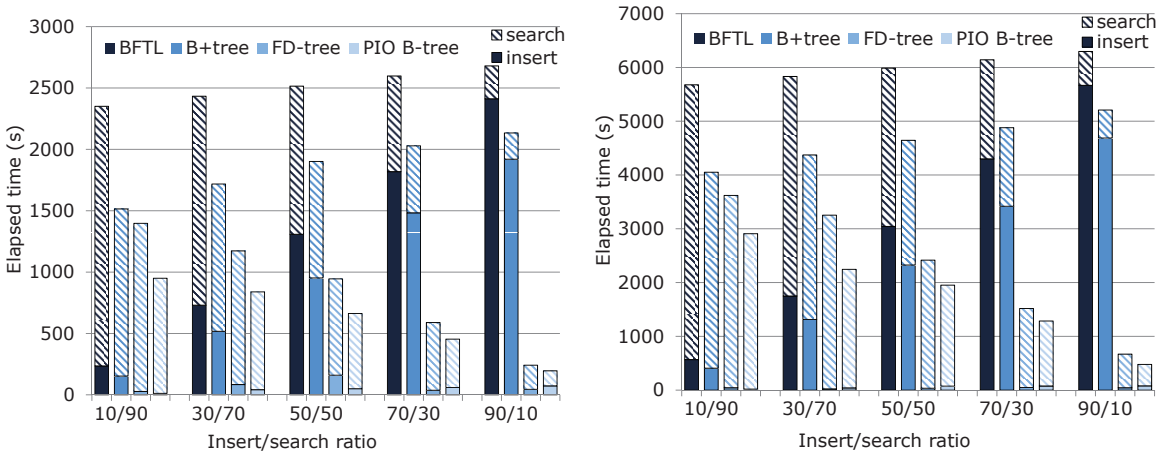
We evaluated the performance of update operations by using update-only workloads. Since index-insert, index-delete, index-update operations demonstrated almost the same performance, we only report the insert workload result. After allocating the main memory to OPQ, the rest of main memory space was allocated 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.