PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe

Li Wang*, Zining Zhang*†, Bingsheng He†, Zhenjie Zhang*
* *Singapore R&D, Yitu Pte. Ltd., Singapore*
{*li.wang, zining.zhang, zhenjie.zhang*}*@yitu-inc.com*
† *National University of Singapore, Singapore*
hebs@comp.nus.edu.sg

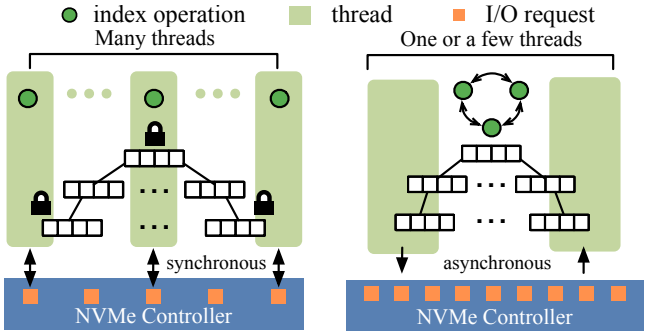
***Abstract*—With the gaining popularity of the Non-Volatile Memory (NVM) technology, NVM express (NVMe) is now becoming the de facto interface for high-end block devices. NVMe generally enables the applications to exploit the massive internal parallelism of new-generation solid-state drives (SSDs) by issuing simultaneous I/O requests. However, existing B+ Trees are unable to maximize the utilization of NVMe hardware performance because of their synchronous execution paradigm which is incompatible with the interface of NVMe. To tackle this problem, we propose *PA-Tree*, an NVMe-friendly B+ Tree with a novel, polled-mode, asynchronous execution paradigm to process multiple index operations in an interleaved and asynchronous manner. Such an execution paradigm allows PA-Tree to saturate NVMe hardware with sufficient asynchronous I/O operations, while avoiding the potential overhead of excessive multi-threading. To further unleash the power of the new paradigm, we devise a new workload-aware scheduling algorithm to optimize the access to the NVMe interface based on the unique characteristic of NVMe, which enhances throughput while minimizing processing latency as well as CPU consumption. Extensive experiments on both synthetic and real workloads demonstrate that PA-Tree achieves up to 5× improvement on throughput and 30% reduction on latency against state-of-the-art solutions.**

***Index Terms*—NVM, NVMe, B+ Tree, Index**

I. Introduction

The last decade witnesses the rapid development of novel Non-Volatile Memory (NVM), starting with flash memory [21] and phase change memory (PCM) [36], to the new-generation 3D XPoint-based memory [17]. These technologies are pushing the storage devices to a new level of performance on larger input/output operations per second (IOPS), shorter access latency, larger capacity and/or lower price. Nowadays, NVMs are commonly available to both commodity and enterprise consumers, who are looking for storage solution for their performance-critical and data-intensive applications.

While the manufacturing technology of NVM is evolving over time, the advance host controller interface (AHCI) [16], originally designed for slow magnetic disks, renders unacceptable high interface latency and CPU consumption in data access and fails to fully exploit the massive internal parallelism of new NVM [13] devices. As a result, a new interface named Non-Volatile Memory Express (NVMe) has been proposed in recent years [13]. NVMe offers higher throughput, lower latency and lower CPU consumption, which is now becoming the de facto standard for high-end NVM devices, e.g., Intel's



(a) Concurrent B+-Tree (b) PA-Tree

Fig. 1. Comparison between the synchronous execution paradigm (left) and pull-mode asynchronous execution paradigm (right) under NVMe interface.

Optane SSD. When AHCI has only one command queue with the maximal queue depth of 32, NVMe supports up to 64K command queues, each with the maximal queue depth of 64K [20]. Such design allows the high-level applications to have a large number of concurrent I/O commands submitted to the NVMs at the same time, and consequently maximizes the usage of internal parallelism of NVM and achieves extremely high IOPS. Moreover, NVMe is lock-free and efficient in command submission, due to the support of multiple lock-free queues and the avoidance of register reads when issuing a command [20]. Those features further enhance the throughput and the responsiveness of applications using NVMe as the underlying storage interface.

In this paper, we investigate how NVMe affects the design of database index and explore the potential optimization opportunities. We focus on the B+ Tree [12], which is undoubtedly the most important and common data structure for many data-related scenarios, such as relational databases [18], [35], stream processing systems [8], [34], [39], and key-value stores [6]. It is unfortunate that, despite the promising performance of NVMe, existing B+ tree index structures do not demonstrate expected performance gain with NVMe, mainly because these index structures are designed without taking account of the characteristics of NVMe. To the best of our knowledge, every B+ Tree index follows a synchronous execution paradigm such that a working thread is blocked when submitting an I/O command through NVMe and desperately waits for the completion of the submitted I/O command. As shown in Figure 1(a), to exploit the internal parallelism of the NVM, we have to spawn

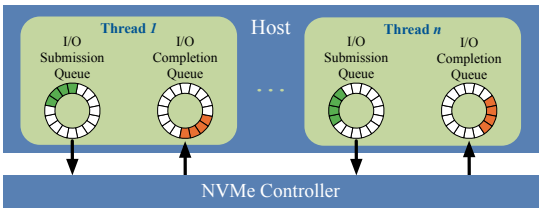


Fig. 2. NVMe interface.

a large number of concurrent working threads, much more than physical CPU cores, so that enough I/O commands can be submitted. However, concurrent execution of these threads not only introduces expensive synchronization overhead in guaranteeing the consistency of the index structure [3], but also leads to expensive context switch cost and scheduling overhead with the operating system [35]. Therefore, the internal parallelism of NVM is achieved at the expense of significant overhead and CPU consumption, consequently leading to low throughput of index operations.

To address these issues, we propose *PA-Tree*, a simple yet efficient B+ Tree variant tailored for NVMe. Instead of optimizing the internal data structures of the index based on the unique characteristics of a particular type of NVM, PA-Tree is designed for any fast storage media under the NVMe interface. The core idea in our PA-Tree is to employ a novel *polled-model*, *asynchronous* (PA) execution paradigm, in which a working thread processes multiple index operations in an interleaved manner. Such a polled-mode, asynchronous execution paradigm is able to fully exploit the internal parallelism of NVM but naturally avoids huge inter-thread synchronization and context switch cost resulted from excessively multi-threading. In addition, since the working thread is fully aware of the workload of the index operations, the index can better utilize through NVMe and CPU controls by choosing the timing to interact with the NVMe interface and to interleave the execution of the concurrent index operations.

Extensive experiments on both synthetic and real workloads demonstrate that PA-Tree achieves up to 5× improvement on throughput and 30% reduction on latency against state-of-the-art solutions.

II. Preliminaries

NVM and NVMe: In the last decade, a wide range of NVM devices, such as NAND [21], PCM [31] and the latest 3D XPoint memory [17], have emerged to offer higher performance with lower price. NVM is now playing an increasingly important role in aiding application with strong data persistency requirements as well as performance-critical constraints on throughput and latency.

AHCI [16] interface, which was originally designed for the slow magnet disks, works well with the last-generation (slow) NVMs. With the constant performance improvment of new-generation NVMs, AHCI gradually becomes the performance bottleneck in data access on high-end NVMs. AHCI supports only one command queue with the maximal queue depth of 32. This prevents AHCI from exploiting the massive internal parallelism of high-end NVMs. In addition, AHCI involves 9 accesses to un-cacheable register per queue command and requires synchronization lock to issue commands [20], which generates significant access delay and overhead of contention and synchronization.

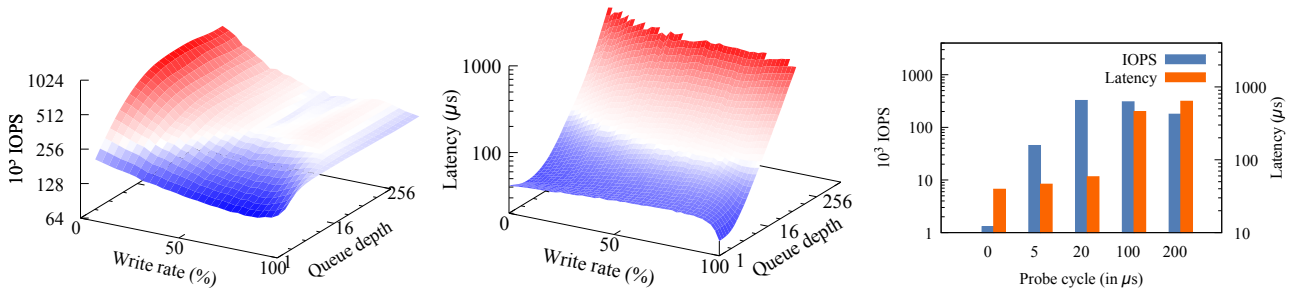
To fully exploit the performance benefits of the NVM, NVM Express (NVMe) [13] is proposed to overcome the overhead associated with AHCI. NVMe is an efficient and scalable host controller interface that is tailored for PCI Express-based, high-performing NVM devices. In contrast to AHCI, NVMe offers 64,000 command queues, 64,000 maximal queue depth per queue, and much lower access latency.

Figure 2 depicts the internal structures and working mechanism of the NVMe interface. NVMe driver, e.g., SPDK NVMe [33], provides APIs for user application to allocate queue pairs in user space. A queue pair typically consists of an I/O submission queue and an I/O completion queue, each of which is implemented as a ring buffer for efficiency concern.

To issue an I/O request, the application simply submits an I/O command to a submission queue via `io_submit()` function provided by the NVMe driver. Different from traditional disk-oriented interface where an I/O function call is blocked until the I/O is completed, `io_submit()` returns immediately after appending the I/O command to the submission queue, providing the application an option of submitting more I/O commands to the NVMe or utilizing the CPU for other work while waiting for the completion of the I/O command. The I/O commands on the submission queue are termed *outstanding commands*. When there are multiple outstanding commands available on the submission queue, the NVM processes the commands in parallel. Finally, the completion of those commands may not come back in their submission order. Once an I/O command is completed, the NVMe controller does not notify the user thread upon the completion but simply places a message, called *I/O completion*, to the corresponding I/O completion queue. It is the responsibility of the application to probe the completion queue by calling `probe()` function, which triggers the callback function associated with the completed I/O commands. For each I/O command, its callback function is specified as a parameter of the I/O command and is used to trigger any necessary logic for the user application after the I/O command is completed.

Preference to high queue depth: To quantify the importance of maintaining sufficient outstanding I/O commands on NVMe, we show the IOPS of a mainstream NVMe on Amazon EC2 i3.x2large instance with various queue depth and write rate in Figure 3(a). Clearly, the queue depth is a dominant factor to the IOPS performance of NVMe. The IOPS of NVMe with more than queue depth 32 is higher than that with only queue depth 1 by an order of magnitude. This observation leads to the most important guideline in the design of PA-Tree. That is, the user application must keep NVMe busy with sufficient outstanding I/O commands.

Unstable access latency: Figure 3(b) shows the data access latency as the write rate and the queue depth vary. Since



(a) IOPS distribution (b) Access latency distribution (c) Effects of probing cycle

Fig. 3. The effects of queue depth, write rate and probe cycle to the performance of NVM device.

the degree of internal parallelism for a given NVM device is limited and the read and write speed could be asymmetric, the access latency heavily depends on the current workload, e.g., queue depth and write rate, and may vary significantly. The unstable access latency poses a huge challenge to the objective of achieving efficient asynchronous I/O of NVMe as introduced below.

Sensitivity to probe frequency: With NVMe interface, it is up to the user threads to decide when and how often to probe the completed I/O commands. Figure 3(c) shows the IOPS and latency of the NVM as the probe cycle varies. Since each probe introduces additional overhead in NVMe, over-frequent probe lowers the IOPS. When the probe latency grows, on the other hand, the detection of a completed I/O is significantly delayed, eventually causing a high I/O latency. Such high I/O latency may prevent the application from issuing new I/O commands and consequently result in under-utilization of the NVM device. In other words, a good probe policy has the potential to enhance the throughput and latency of the data accesses, while a bad policy leads to extremely poor performance. Since the access latency of NVM heavily depends on the current workload and may vary greatly, PA-Tree needs to dynamically adjust its probing policy based on the instantaneous workload.

Out-of-order completions: The NVM processes the outstanding I/O commands in parallel for efficiency. Therefore, the completion of the I/Os may not be in the same order as they are submitted. Out-of-order processing improves the performance of NVM considerably, but the application has to take additional efforts to handle them properly.

III. Polled-mode Asynchronous Tree

PA-Tree is a generic B+-tree index for any block-oriented storage media under NVMe interface, and therefore it does not involve in optimizations based on the performance characteristics of a particular type of media. Instead, the core idea of PA-Tree is to employ a *polled-mode*, *asynchronous* programming paradigm that creates a few *working threads* to process multiple index read/write operations in an interleaved way, as shown in Figure 4. PA-Tree is asynchronous in the sense that during the process of an index operation, if the working thread submits an I/O request through NVMe, it does not wait for the completion of the I/O; instead, the operation goes into a waiting state, and the thread continues to process

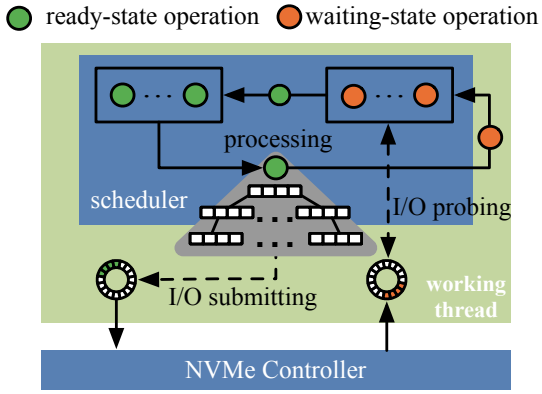


Fig. 4. An overview of PA-Tree.

other index operations and may submit more I/O commands to NVMe while waiting the completion of the first I/O command. The major advantage of PA-Tree is its capability of keeping a sufficiently large number of outstanding I/Os on the NVMe interface with only one or a few threads. This enables PA-Tree to exploit the internal parallelism of NVMe interface while avoiding the overhead of inter-thread synchronization and context switches by excessively multi-threading. Another advantage is that it is the working threads who submit I/O commands, probe their completions, and determine whether an operation transfers from waiting state to ready state. Such a polled-mode execution skips system calls made by the working threads, such as file read/write and sem_wait/sem_post, and consequently avoids the time-consuming switch between the user mode and the OS kernel mode.

PA-Tree provides exactly the same primitives as a standard B+ Tree, i.e., point / range search, insert, update and delete. This makes PA-Tree a good option for existing systems with new NVMe-compatible devices. In this paper, point search and range search are referred to as *search operations*, while the rests are called *update operations*. When an application thread calls an index primitive, the corresponding index operation is created. The application thread is then blocked until the index operation is fully processed by the working thread.

A. Operation Transitions

To achieve the interleaved execution of index operations, PA-Tree decomposes the execution of each index operation into a finite number of transitions and interleaves the execution of multiple index operations on a transition basis. For each

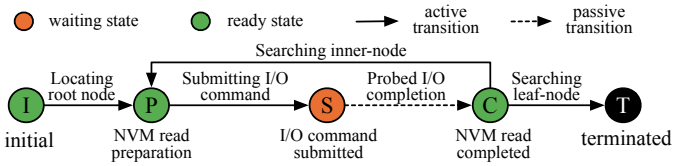


Fig. 5. The state transition graph for the point search operation.

index operation, we introduce a concept called *operation state*, which contains all the working set for the index operation. The transition of an operation is executed by the working thread, which transfers the operation from a state to another state. The states of an index operation can be categorized into ready states and waiting states. An operation is in a *ready state* if it can be processed by the working thread and transferred to the next state, or in a *waiting state* when if it must wait for certain conditions before transitioning to the next state. The transition of an operation from a ready state to another state is called *active transition*, since it is ready for transition and it is up to the working thread to decide when to perform the transition. In contrast, the transition starting from a waiting state is called *passive transition* in that the transition can only be made by the working thread when a certain condition is satisfied. The condition could be either the completion of a submitted I/O request or acquisition of latching on a particular index node.

In our implementation, the ready-state operations and the waiting-state operations are maintained in two collections, denoted as $R(\mathcal{C})$ and $W(\mathcal{C})$, respectively. When the working thread submits an I/O request to the NVMe submission queue and transfers the operation into the waiting state, it specifies a callback function in which the operation is transferred to its next state. Based on a scheduling policy discussed in details in Section 4, the working thread periodically selects a ready state operation in $R(\mathcal{C})$ to process, and probes the completion queue of NVMe to transfer the operations with completed I/Os to their respective next states.

We take the state transition graph of the point search operation shown in Figure 5 as our running example. Details of the transition graphs for other index operation types are skipped in this paper, because it is trivial to make the extensions. Generally, the working thread traverses from the root node to the very leaf node which may contain the target search key. As shown in Figure 5, to search an index node, the working thread issues an I/O command to the NVMe devices, which is an active transition and transfers the operation from P state to state S. Since S is a waiting state, it cannot be further transfered to its next state C until the I/O command is completed. Then the thread may switch to the execution of other operations and resumes the execution of this operation when the I/O request is completed and the operation is dispatched to the working thread by the scheduling policy.

Figure 6 depicts how the working thread interleaves the execution of n index operations over time. Once the ongoing executing operation is transferred to the waiting state, the working thread executes another ready-state operation and resumes the processing of the former operation shortly when the

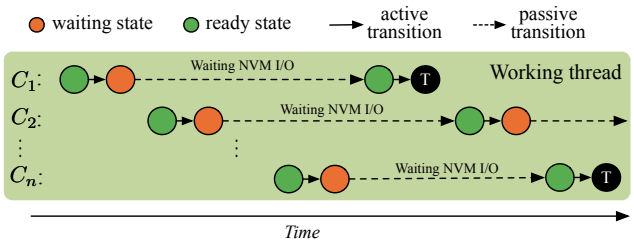


Fig. 6. The demonstration of asynchronous execution of n operations.

operation becomes ready. In other words, when waiting for the completion of the submitted I/Os, the working thread processes other ready-state operations simultaneously and may submit more I/O requests. This asynchronous workflow enables one thread to exploit the internal parallelism of the NVM, thus achieving high operation processing throughput.

B. Latching

During the execution of update operations, index nodes may be modified, split or merged, which may cause data corruption or inconsistency if not handled correctly. To guarantee data consistency against the interleaved execution of multiple index operations, we first introduce the concept of latching and then discuss how to apply existing concurrency control techniques on PA-Tree based on the operation latches.

A *latch* is a logical flag that an operation sets to a particular tree node to declare its shared-read or exclusive write ownership of the particular node. Before it releases the ownership, other operations attempting to request a conflicting ownership of this node is forced to stay in the waiting state. An operation can only set a read latch to a node when there is not any exclusive latch on the node, while it can only set a exclusive latch to a node when there are no shared latches or exclusive latches on the node. If an operation's latch request to a particular node is granted, the operation transitions to the next ready state and is called *latch-granted*. Otherwise, it stays in the waiting state and is called *latch-requesting*, until its latch request is approved.

In PA-Tree, it is the responsibility of the working thread to grant the latch to the operations and to transfer a latch-granted operation to its next state. Such a design not only enables highly efficient latch requesting and granting by avoiding the inter-thread synchronization and thread context switches, but also allows the working threads to utilize instantaneous workloads to make better scheduling decisions, as discussed in details in Section IV. To handle the latch requests, PA-Tree maintains a read latch count r, a write latch count w and a pending latch request queue for each tree node. Upon receiving an operation's latch request to an index node, the thread checks the values of r and w of the index node to determine whether the latch can be granted to the operation. Currently, all the latches are regarded equally important and thus is on a first-request-first-grant basis. Specifically, a write latch can be granted when $r = 0$ and $w = 0$, while a read latch can be granted when $w = 0$. If the latch is granted, the working thread updates r or w and continues to process the operation. Otherwise, the latch is added to the pending queue

of the tree node and the current operation goes into the waiting state. When an operation releases its latch on a particular index node, the thread updates the corresponding r or w. Then from the front to the tail, the thread processes each latch request on the pending queue based on the current value of r and w, until the thread reaches the first latch request which cannot be approved or all the latches requests are approved. If the a pending latch request can be approved, the latch request will be removed from the pending queue and its corresponding operation will be transferred to its next ready state.

With the support of the operation latches, we can simply apply the well-known latch coupling technique [3] on PA-Tree to guarantee the consistency of the index against concurrently executing index operations. We opt for this technique mainly because of its simplicity and high concurrency. Due to space limitation, we only give a high level description to point search and insertion. More details are available in articles [3] and [15]. During the traverse from the root to the leaf node in a point search, before accessing any node, the operation requests a shared latch on the node. When the request is granted, the operation releases its latch on the parent node if any. For the insertion operation, the operation requests an exclusive latch on the node. When the latch is granted, the operation will release its latch on its parent nodes if the insertion will certainly not cause splitting operations on the parent nodes, e.g., when the current nodes has at least one empty slot.

C. Buffer Management

Without buffering, every access to a tree node, either read or write, results in at least one NVM I/O. Since RAM is still at least 10 times faster than the mainstream NVM, significant performance improvement is expected by caching the frequently accessed data in RAM.

Strong Persistent Buffering: To achieve strong data persistence, the update to the tree node structure should be written directly to the NVM, so that when an update operation is completed, its modification is deemed on NVM and thus is persistent against power failures that may happen afterwards. However, it is unnecessary to always read index nodes directly from NVM for every access. The I/O requests can be answered by the data cached in RAM, provided that the data in cache is always consistent to that on NVM to avoid violating the semantics of strong data persistence.

Motivated by this, we implement a software *read-only buffer* over the NVMe interface. The read-only buffer comprises a number of in-memory data blocks and employs LRU policy [10] to evict the old blocks. The key is to make sure that data read from the read-only buffer is consistent to the data in NVM in the presence of asynchronous I/Os. To this end, when a write I/O is submitted, we do not write the data block into the read-only buffer until the write I/O is completed. That is because, if we wrote the data block into the read-only buffer before the completion of the write I/O, the content of the data block will be visible to other concurrent operations before the completion of the I/O. This potentially introduces inconsistency if failures happen before the completion of the I/O.

Weak Persistent Buffering: While strong persistence is particularly important for the scenarios relying on the indexing structure for persistence, it introduces unnecessarily high write amplification factor in the case where data persistence on a per-operation basis is not needed. For instance, with the help of write ahead log (WAL), it is unnecessary to persist every single operation, but persisting a batch of operation is sufficient.

To reduce the write amplification factor of weak-consistent PA-Tree, we employ a read-write software buffer over NVMe and provide an additional *sync()* function to the applications, which flushes all the updates to the PA-Tree on NVM. The read-write buffer handles read I/O request in exactly the same way as the read-only buffer. In contrast, to handle an write I/O, the read-write buffer updates the corresponding in-memory data block and does not immediately flush the update to NVM. Instead, the updated data blocks are only flushed on NVM when they are evicted or the user calls *sync()* function. With the help of the read-write buffer, multiple writes to the same data block can be merged into a single write to NVM, thus reducing access to NVM and improving the performance.

The read-write buffer works effectively when the workload retains a certain degree of data locality in node updates. However, when the workload lacks of data locality or a sufficiently large data buffer cannot be used due to the limited RAM capacity, the read-write buffer may not reduce the write amplification effectively. In such case, the optimizations like that used in log-structured merge (LSM) tree [27], can be employed to further improve the performance. However, applying our polled-mode, asynchronous programming model on LSM tree is out the scope of this paper.

IV. Scheduling

An advantage of employing a polled-mode, asynchronous execution paradigm in PA-Tree is that it is the responsibility of the working thread to determine when to interact with the NVMe interface and how to interleave the execution of the operations. Compared with the OS kernel which is unaware of the application-level workload, the working thread is in a position to make better decision with a better understanding of the workloads. In this section, we discuss the internal scheduling algorithm that guides the working thread and enables efficient execution of multiple operations.

The objective of the scheduling algorithm in PA-Tree is to maximize the processing throughput of index operations, while minimizing the process latency of each individual index operation as well as the CPU consumption of scheduling.

Let $\mathcal{C}$ denote the set of index operations being processed by the PA-Tree. We partition $\mathcal{C}$ into $R(\mathcal{C})$ and $W(\mathcal{C})$, which denote the set of ready-state operations and the set of waiting-state operations, respectively. Based on the waiting conditions, we further define $W(\mathcal{C}) = W_b(\mathcal{C}) \bigcup W_{IO}(\mathcal{C})$, in which $W_b(\mathcal{C})$ is the set of barrier-requesting operations waiting for the grant of operation barriers and $W_{IO}(\mathcal{C})$ is the set of I/O-blocked operations waiting for the completion of I/O requests. For an operation $c \in R(\mathcal{C})$, we use $process(c)$ to represent the maximal sequence of transitions of c until c 1) is either fully

Algorithm 1: Naive Scheduling

```
Input: operation contexts C, ready-state operation contexts R(C)
1 main loop
2     while R(C) ≠ ∅ do
3         c ← R(C);
4         process(c);
5         remove c from R(C);
6     R_IO(C) = probe();
7     add R_IO(C) to R(C);
```

processed and removed from $\mathcal{C}$, or 2) goes into a waiting-state operation and is added to $W_b(\mathcal{C})$ or $W_{IO}(\mathcal{C})$. During the process of an operation c, the barriers obtained by c may be released, which will trigger the transitions of some operations from $W_b(\mathcal{R})$ to $R(\mathcal{C})$ if possible, as discussed in Section 3.2. Therefore, moving the barrier-obtained operations from $W_b(\mathcal{C})$ to $R(\mathcal{C})$ is handled within $process()$, and the scheduling algorithm does not need explicit procedure to do that. In contrast, due to the polled-mode working mechanism of NVMe, the working thread is responsible for probing the NVMe interface for the completed I/O requests and moving the I/O-completed operation from $W_{IO}(C)$ to $R(\mathcal{C})$ so that those I/O-completed operations can be processed shortly.

Algorithm 1 shows a naive scheduling algorithm. In each main loop, the algorithm first processes all the ready-state operation contexts $R(\mathcal{C})$ (Line 2 - 5), and then calls probe() to detect the I/O-completed operation, transfers the I/O-completed operations to ready-state and adds them to $R(\mathcal{C})$.

Compared with the traditional concurrent B+-Tree, PA-Tree with the naive scheduling algorithm can effectively improve the utilization of NVM and avoid the high cost of using multiple working threads. However, there is still large optimization room for PA-Tree in terms of the performance and CPU consumption. First, the timing of calling probe() plays an interesting trade-off between the NVM utilization and the probe overhead, thus has great influence on the performance of the PA-Tree and scheduling overhead. Second, given a set of ready-state operations, the order in which the operations are processed affects the throughput and the latency of the PA-Tree considerably. Third, when the workload is low, the main loop will consume CPU cycles without doing any useful work. Therefore, it is desirable for the working thread to adaptively yield its CPU when the workload is low. The workload-aware scheduling algorithm with the three optimizations mentioned above is shown in Algorithm 2, and will be discussed in details in the rest of this section.

A. The Timing of Probe

Due to the polled-mode mechanism of NVMe, an I/O-blocked operation is not transferred to its next ready state upon the completion of its I/O requests; instead, it can only be transferred when the completion of its submitted I/O is detected by probing the NVMe interface. To reduce the processing latency of each individual operation and improve the throughput of PA-Tree, it is desirable to minimize the delay between the I/O completion and I/O detection.

We attempt to tune an optimal probing frequency that allows the working thread to detect the completed I/Os rapidly without introducing too much CPU consumption and interruption to the NVMe. Unfortunately, such global optimal setting applicable to every workload does not exist. This is because the latency to process an I/O request heavily depends on the number of outstanding I/O requests as well as the read/write ratio, and thus varies greatly as discussed in Section II. To solve this problem, we maintain a model at runtime to estimate the number of completed I/Os and only probe the NVMe interface when the model guesses an I/O command may be returned.

Given two parameters t and n, we evenly divide the recent t microseconds into n time slices. We define a vectors $\mathbf{w} = [w_1, w_2, \ldots, w_n]$, in which w_i is the number of outstanding write I/Os submitted within the i-th time slice. Similarly, we define a vector $\mathbf{r} = [r_1, r_2, \ldots, r_n]$ as the counts of the outstanding submitted read I/Os for the n time slices. Our estimation model takes $\mathbf{T} = \mathbf{w}||\mathbf{r}$, i.e., the concatenation of the two vectors, as input and outputs the estimation on (w_0, r_0), which is the number of completed write I/Os and the number of completed read I/Os, respectively.

We consider a large variety of methods to model the relationship between $\mathbf{T}$ and (w_0, r_0), but decide to use linear regression model [28] mainly because of its simplicity and effectiveness in solving our problem. In the linear regression model, the estimation on (w_0, r_0) is obtained by:

$$(w_0, r_0) = \mathbf{T}\beta,$$

where β is a $2n \times 2$-dimensional parameter matrix. We train the parameter matrix β offline using the well-known machine learning tool pandas [23]. To guarantee the accuracy and robustness of the model, we generate training data from a variety of workloads with different read/write ratio and workload intensity. In practice, we set $t = 1000$ and $n = 20$, because 99.9% I/O requests can be completed within 1000 microseconds and $n = 20$ provides enough resolutions to guarantee the accuracy of the model.

B. Prioritized Execution of Operations

Another optimization is to choose the ready-state operator to process according to their priorities based on their state and properties. Specifically, the priority of the ready-state operations is determined based on the following intuitions: a) The operation that was admitted earlier should have higher priority for processing so as to reduce the process latency for each individual operation. b) Operation holding write barriers should be processed with higher priority so that their write barriers can be released earlier to improve the concurrency of the PA-Tree for a better overall processing throughput and reduced processing latency for individual operations.

To implement the prioritized processing of operations with minimized overhead, we use a priority queue to maintain the

Algorithm 2: Workload-aware Scheduling

Input: All operations $\mathcal{C}$, ready-state operations $R(\mathcal{C})$, parameter matrix in the predication model β, the granularity of yielding CPU t.

```
main loop
    if R(C) ≠ ∅ then
        get c with the highest priority from R(C);
        process(c);
        remove c from R(C);
    construct vector T = w||r;
    (w_0, r_0) = Tβ;
    if w_0 ≥ 1 or r_0 ≥ 1 then
        R_IO(C) = probe();
        add R_IO(C) to R(C);
    reconstruct T = w||r to estimate (w_0, r_0) after t μs;
    (w_0, r_0) = Tβ;
    if R(C) = ∅ and w_0 = 0 and r_0 = 0 then
        yield CPU for t μs;
```

ready-state operations and let the working thread process the operation with the highest priority. In particular, the priority of an operation is represented as an unsigned 8-byte integer with a smaller value implying higher priority. The most significant bit of the integer being 0 indicates the operation is holding a write barrier and should possesses highest priority. Otherwise, the most significant bit is 1. The rest of the bits are used to store the sequential ID of the operation, which is generated in an increasing order upon operation admission.

C. CPU Yielding

PA-Tree is polled-mode, asynchronous in that its working thread will not be blocked by the requests of I/Os or barriers, but continues to process other ready-state operations. This avoids the expensive switches between the user and kernel space, and thus enables a single working thread to process one million operations per second. However, due to the absence of blocking system calls, when the workload of the PA-Tree is very low, e.g., 1K operations per second, the working thread may waste a large amount of CPU cycles in the main loops without doing useful work.

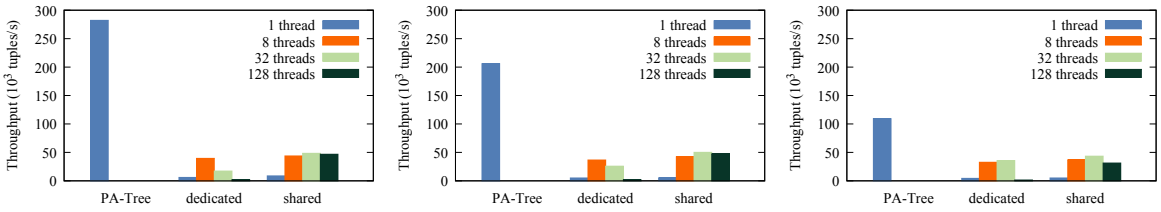
To avoid wasting CPU cycles, the working thread yields its CPU if it detects that there is unlikely to be any work to do in the following t μs, e.g., $t = 100$. In particular, at the end of the main loop, the algorithm employs the model in Section IV-A to estimate if any outstanding I/O requests can be completed after t μs. If $R(\mathcal{C}) = \emptyset$ and no outstanding I/O requests can be completed after t μs, it indicates that the working thread has no real work to do in the following t μs. In such case, the working thread will yield its CPU for t μs, so that the CPU can be used by other application threads. To minimize the negative effects on the performance of PA-Tree, we set t to be the same length as the time slice used in the operating system, so that the operating system can reschedule the working threads immediately after t μs.

V. Experiments

We implement PA-Tree in C++, open-sourced in [30]. We employ a single-threaded implementation for PA-Tree, because a single-threaded implementation is able to saturate the IOPS of the underlying hardware while avoiding the additional synchronization overhead and implementation complexity from multi-threading. We use the storage performance development kit (SPDK) NVMe drive [33] to interact with the NVMe interface. Since the minimal access granularity in the NVMe is 512 bytes, we set 512 bytes as the index node size to reduce the read and write amplification factor. The code is compiled by GCC 4.8 with -O3 flag. The experiments are run over a dedicated EC2 i3 x2large instance running Amazon Linux with kernel version 4.14.70. The instance has 64GB RAM, 8 core Intel E5@2.3GHz CPU, and 1900GB NVMe SSD. The NVMe SSD supports up to 256 queue pairs, each with 2,048 maximal queue depth.

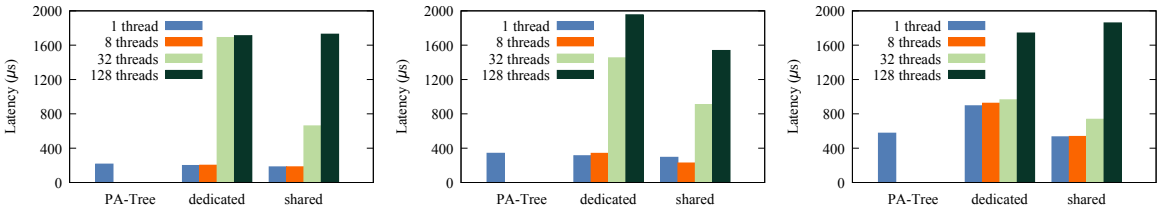
We employ a synthetic workload and two real workloads in our evaluations. The synthetic dataset is generated based on Yahoo! cloud serving benchmark (YCSB) [9]. Using a synthetic dataset enables us to easily control the distribution of keys as well as the workload characteristics for various evaluation purposes. Specifically, we target at three representative workloads, namely *read-only* workload, *default* workload and *update-heavy* workload. The default workload comprises 10% update operations and 90% read operations, and is used for most evaluations unless otherwise stated. In the update-heavy workload, 50% of operations are updates while the rest are reads. All the read and update operations are based on index keys, which are randomly generated following Zipfian distribution. Unless otherwise specified, the skewness factor, denoted by α, for the Zipfian distribution is set to 0.3 throughout our evaluation. The key and the payloads are 8 bytes in size.

The first real workload is *T-Drive* [38], which contains trajectories of over 10,000 taxis in Beijing for a week. Each record consists of a number of attributes, including taxi ID, a z-code computed by apply z-ordering [26] on latitude and longitude, and timestamp. There are over 15 million records and total driving distance of the trajectories is over 9 million kilometers. Each query in this workload aims to find all the records within a given range of z-code. This is an extremely update-heavy workload, which consists of 70% updates. The second real-world workload is *SSE* [34]. It contains anonymized order records for stock trading in Shanghai Stock Exchange (SSE), collected over three months with around 8 million records per trading hour. An order record is made by a user, who specifies his bid and asking prices for a specified volume of a particular stock. We define the stock ID, the price and the timestamp as the index attribute and store the outstanding orders in the B+ tree, so that a new order can be efficiently matched against the outstanding orders using the B+ Tree. A key-value pair is 108 bytes on average. This workload contains 28% updates.



(a) Read-only Workload (b) Default Workload (c) Update-heavy Workload

Fig. 7. Throughput comparison of PA-Tree and two baseline methods over YCSB workload.



(a) Read-only Workload (b) Default Workload (c) Update-heavy Workload

Fig. 8. Latency comparison of PA-Tree and two baseline methods under YCSB workload.

A. *Effect of the new scheme:*

We first quantify the performance gains from the adoption of our new polled-model, asynchronous execution scheme. To this end, we compare the performance of PA-Tree against two baseline approaches, namely *shared approach* and *dedicated approach*. Both baseline approaches employ exactly the same index data structure as our PA-Tree, but follow the traditional execution paradigms, in which the access to the NVMe interface is synchronous and multiple working threads are required to issue more than one I/O commands to the interface simultaneously. Unless otherwise stated, buffering is disabled in all the three approaches.

In the dedicated approach, each working thread is allocated a dedicated pair of a submission queue and a completion queue. When a working thread submits an I/O request to its submission queue, it will repeatedly probe its completion queue until the I/O is completely finished. To avoid wasting CPU cycles, the duration between any consecutive probe is set to 100 μs. Multiple working threads can be created to improve the utilization of NVMe. We employ the same latch coupling technique [3] as the PA-Tree based on semaphore wait and post primitives, to guarantee the correctness under the multi-threaded execution of the index operations.

Similarly, the shared approach employs multiple working threads and the latch coupling technique, but the NVMe accesses of the working threads are delegated to a dedicated thread to reduce the interruption to the processing of the NVMe. Specifically, a global I/O request queue is maintained to accommodate all the I/O requests from the working thread, and there is a daemon thread responsible for handling the I/O requests in the queue. When a working thread submits an I/O request to the queue, it is blocked until the daemon thread notifies it upon the completion of the I/O. The synchronization between the working threads and the daemon thread is via semaphore wait and post primitives, which shows significant performance improvement over spin-lock in our scenario.

Throughput: Figure 7 shows the throughput of PA-Tree and the baseline approaches with various number of working threads under the read-only default and update-heavy workloads, respectively. Despite using the same data structure and latching techniques, PA-Tree with only one working thread achieves at least 5 times higher throughput than the baseline approaches with multiple working threads. By comparing the results in the sub-figures of Figure 7, the throughput of all three approaches drops, as the update rate increases. Higher percentage of update in the workload also generates more NVMe writes, incurring higher overhead than the NVMe reads do. We also observe that for either of the baseline approaches, the throughput is extremely low when there is only one working threads. A sole working thread under the synchronous paradigm only issues one I/O request at any time. Such paradigm fails to utilize the internal parallelism of the NVMe and consequently leads to poor index throughput. When more working threads are deployed in the baseline approaches, the better utilization of the internal parallelism of the NVMe leads to higher throughput. However, when the number of working threads exceeds 32, the performance gain starts to drop, due to the huge context switch and thread contention overhead when the number of running working threads exceeds the number of physical CPU cores.

Latency: Figure 8 illustrates the latency of PA-Tree and the baseline approaches with different number of working threads under three representative workloads. The latency of PA-Tree is highly competitive when compared against the smallest latency achieved by the baseline methods. By deploying more threads, the latency in both baseline approaches grows significantly. The latency with 128 threads, for example, is over $10,000$ μs, which is unacceptable for most applications. The root problem with the baseline approaches with over 32 threads are in twofold. First, when there are only 8 CPU cores available, the employment of more than 8 threads results in frequent context switches. Second, with more working

TABLE I
RUNTIME STATISTICS OF THE APPROACHES IN THE EXPERIMENTS.

methods	outstanding I/Os	IOPS (10^3) (max=415.7)	CPU consumption	context switches
shared	29.88	68.1	4.16	12,579,138
dedicated	18.10	58.5	6.50	6,291,027
PA-Tree	48.56	387.3	0.78	121

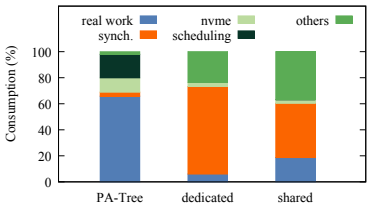


Fig. 9. Breakdown of CPU consumption for PA-Tree and baseline methods

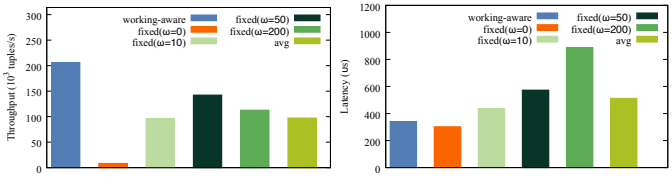
threads, it is more likely that a working thread may be blocked by other threads during the process of an individual index operation, causing poor latency. The performance of the baseline approaches in Figure 7 and Figure 8 implies that, with various number of threads, both baseline approaches are unable to achieve reasonably high throughput and low latency at the same time. In contrast, the PA-Tree is the only solution maintaining high throughput and low latency. As a short summary, the new polled-mode, asynchronous execution paradigm is much more NVMe-friendly than the traditional synchronous execution paradigm, which is the major reason driving the favored performance advantage of PA-Tree.

Performance analytics: To better understand the performance gap between PA-Tree and the baselines, we measure the number of outstanding I/Os on the NVMe, the IOPS, the CPU consumptions, and the number of context switches for the three approaches and show the numbers in Table I. The CPU consumption is measured by the average number of CPU cores used by the working threads. As there are 8 physical CPU cores in our evaluation hardware, CPU consumption varies from 0.0 to 8.0. The number of context switches is measured by using *perf* tool [1]. Since both baseline methods achieve the best performance with 32 threads, we only show their measurements under 32 working threads in the table. The results imply, despite using only working thread, PA-Tree is able to keep a larger number of outstanding I/Os in the NVMe than the baseline methods. This enables the PA-Tree to fully exploit the internal parallelism of NVMe and nearly saturate the NVMe SSD, thus achieving much better performance. Interestingly, the actual IOPS of the shared and the dedicated approaches is way lower than the IOPS they are expected to achieve given their numbers of outstanding I/Os. As discussed in Section II, this is mainly because they probe the NVMe too often, which introduces expensive overhead in the NVMe driver. Due to the single-threaded execution paradigm, the context switch cost of PA-Tree is 1,000× less than the baseline approaches. This further deepens the performance gap between the PA-Tree and the baseline approaches and also entitles PA-Tree with 5× CPU efficiency improvement over baselines.

CPU efficiency: Table II lists the CPU consumption per

TABLE II
AVERAGE CPU CONSUMPTION PER OPERATION FOR VARIOUS METHODS.

method	PA-Tree	dedicated	shared
CPU cycles (10^3)	3.23	175.3	156.2



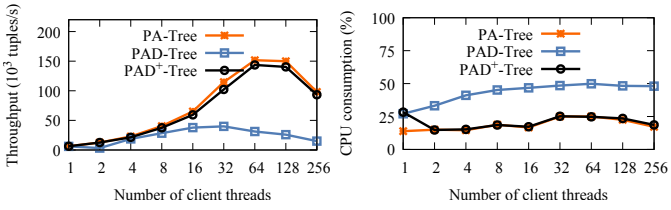
(a) Throughput (b) Latency

Fig. 10. Performance comparison of PA-Tree with and without workload-aware probing.

operation by PA-Tree and its competitors. It is clear that CPU cycles consumed by the baseline methods is larger than that of PA-Tree by two orders of magnitude. This explains why a single-threaded implementation of PA-Tree outperforms the baseline methods with 16 threads. To gain more insights into the superiority of PA-Tree in terms of CPU savings, we make a breakdown analysis over the CPU consumption and demonstrate the outcomes in Figure 9. The CPU consumption is divided into five categories: *real work*, *synchronization*, *NVMe*, *scheduling* and *others*. The real work part consists of the CPU cycles for index structure access, node split and merge. For PA-Tree, the "synchronization" refers to CPU cycles used to handle the context barriers, while in the baseline approaches it refers to the CPU cycles spent on calling semaphore primitives, i.e., wait() and post(). "NVMe" covers the CPU cycles spent on calling the NVMe drive. "Scheduling" refers to the CPU cycles for scheduling the operations and is only applicable in PA-Tree. We are unable to directly measure the OS scheduling overhead and the context switches overhead without modifying the OS kernel. Their CPU consumption is therefore in the "others" category. We observe that the synchronization and scheduling in PA-Tree take only a small fraction of CPU cycles, while more than half CPU cycles are spent on the real work. In contrast, for the baseline methods, the real work part is no more than 20% CPU consumption, while the most CPU cycles are spent on synchronization and context switches. The results validate that the CPU efficiency of PA-Tree is mainly because of its avoidance of inter-thread synchronization and context switches enabled by its single-threaded execution paradigm.

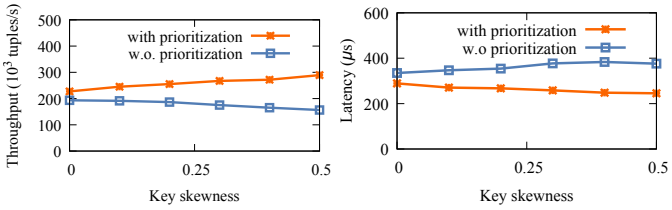
B. Scheduling Evaluation

Workload-aware probing: To evaluate our workload-aware probing strategy, we compare the performance of PA-Tree with two naive probing strategies. In the first strategy, the working thread probes the NVMe in every $avg(t)$ μs, where $avg(t)$ is the average I/O completion latency measured within the latest sliding window of 1 second length. The second strategy probes the NVMe in a pre-defined fixed rate, varying from 0 to 200 μs in different runs. The latency and throughput for the PA-Tree with different polling strategy are presented in Figure 10.



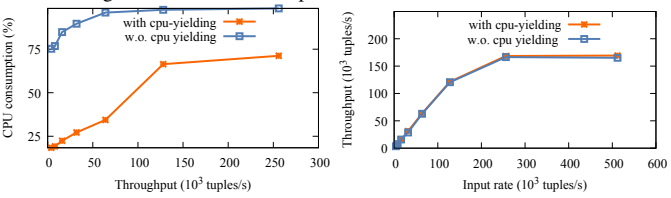
(a) Throughput (b) Latency

Fig. 11. Comparison with PA-Tree and PAD-Tree.



(a) Throughput (b) Latency

Fig. 12. The benefit of prioritized execution to PA-Tree.



(a) CPU Consumption (b) Throughput

Fig. 13. The effect of CPU yielding.

Workload-ware probing outperforms, by a substantial margin, other strategies probing the NVMe either in a fixed rate or a dynamic rate based on the average I/O completion time, in terms of both throughput and latency. When the probe cycle is short, even under short I/O completion latency of NVMe, the throughput is extremely poor. This is simply because NVMe is probed too frequently, causing high CPU consumption for the interaction with NVMe and unnecessary interruptions of the normal processing of NVMe, as discussed in Section II. With the increasing ω grows, the system triggers fewer interruptions to the NVMe, alleviating the system workload for a higher throughput. However, when ω exceeds 100 μs, the latency grows and the throughput drops significantly. Such high probe delay prevents a large number of completed I/Os from returning the results to the working thread in time, consequently lowering rate of the subsequent I/O requests from user applications. Again, appropriate utilization of the internal parallelism of NVMe is the key to fully exploit the performance of storage device through NVMe interface.

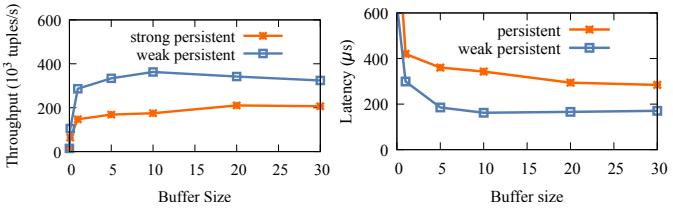
Workload-aware v.s. dedicated polling: We implement two variants of PA-Tree, denoted by *PAD-Tree*, by using a dedicated polling thread which keeps polling I/O completions in loops, and *PAD$^+$-Tree* which also uses our workload-aware pulling strategy in a dedicated polling thread. The intuition behind the two variants is not to compete CPU cycles with the working thread by using another thread handling the I/O requests. The throughput comparison is shown in Figure 11(a) with their CPU consumption results available in Figure 11(b). The first observation is that PAD-Tree performs much worse than others despite much higher CPU consumption. This is because PAD-Tree probes NVMe driver too frequently, which not only waste CPU cycles but also introduces additional overhead to the driver as discusses in Section II. Another observation is that compared with PA-Tree, PAD$^+$-Tree has similar CPU consumption but its throughput is slightly lower than PA-Tree. Since the indexing thread does not fully utilize a CPU core, doing the probing in another thread does not bring benefit but results in inter-thread synchronization, which may lead to CPU cache pollution and thus hurts the performance.

Prioritized execution: To evaluate the effects of prioritized execution to the performance of PA-Tree, we compare the throughput and latency of PA-Tree with and without prioritized execution when the key skewness varies. The results are shown in Figure 12. PA-Tree with prioritized execution achieves higher throughput and lower latency than PA-Tree without prioritized execution. The performance margin grows with higher key distribution skewness. After further investigation, we find that with higher key skewness, the contention among multiple threads is more severe. In such cases, it is more important to release the write latches on the index nodes as soon as possible to improve the concurrency.

CPU yielding: Figure 13 compares PA-Tree with and without CPU yielding when the input rate to the PA-Tree varies along the x-axis. The CPU consumption without CPU yielding is always beyond 75%, even when the input rate is extremely low, e.g., below 50,000 tuples/s. Without CPU yielding, despite the low workload, the working thread wastes a large amount of CPU cycles in the main loop without doing any real work. By enabling the CPU yielding, the working thread can adaptively yield its CPU core when it determines that it will not have any real work to do in the near future. This dramatically saves CPU consumption, especially when the workload is low, but does not decrease the overall performance as shown in Figure 13(b).

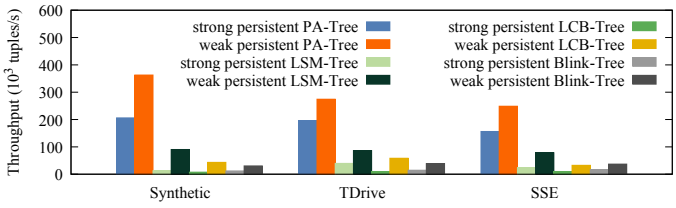
C. Data Buffering

This group of experiments examines the effects of data buffering in our PA-Tree. We evaluate the performance of PA-Tree with strong persistent buffering and with weak persistent buffering, respectively. Their performance comparison is reported in Figure 14. Data buffering effectively improves the performance of PA-Tree in terms of throughput and latency. An interesting observation is the performance boost even when a very small buffer is used. To process every index operation, the working thread has to access index nodes from the root node to the target leaf node. Better spatial and temporal locality is preserved when processing the nodes closer to the root of the tree. Therefore, buffering the root node and uppermost layers of the inner nodes effectively reduces the number of NVMe access for the index operations, finally enhancing the performance of PA-Tree. PA-Tree with weak persistence also achieves higher throughput and lower latency than strong persistent PA-Tree. To guarantee the strong consistency, every write to the NVMe must be directly flushed onto the NVMe. In contrast, in the weak persistent PA-Tree, those writes are

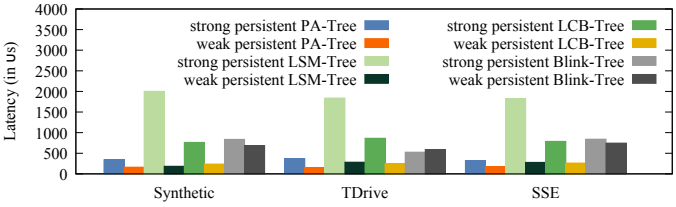


(a) Throughput (b) Latency

Fig. 14. The benefit of buffer to PA-Tree in terms of throughput and latency.



(a) Throughput



(b) Latency

Fig. 15. Performance comparison between PA-Tree and the state-of-the-art competitors.

cached in the data buffer instead and are flushed to the NVMe only when the system calls the `sync()` function.

D. End-to-End Evaluation

In this part of section, we compare the performance of PA-Tree with the state-of-the-art B+ tree implementations in LevelDB [11], a log-based consistent B+ tree (LCB-Tree) [37] and Blink-Tree [25]. LevelDB is one of the most popular fast key-value storage based on LSM-Tree [27]. Among multiple implementations of LSM-Tree, we choose LevelDB because its popularity and the support of strong consistency. LCB-Tree is a concurrent B+-Tree following the traditional synchronous execution paradigm as introduced earlier. Both LCB-Tree and Blink-Tree employ compare-and-swap (CAS) instructions to achieve locking-free. All the methods provide a `sync()` primitive. By calling `sync()` after each index update or after a group of index updates, we can guarantee strong persistent semantics or weak persistent semantics, respectively. To keep the comparison fair, the memory buffer size for each method is set to 10% to the total index size. For all the weak persistent version of each method, `sync()` is called after every 1000 index update operations. For each of the baseline method, we vary the number of threads from 1 to 128 during the evaluation and compare its best performance with PA-Tree.

Figure 15 investigates the throughput and latency of the all the methods under the default synthetic workload and two real workloads, whose characteristics are described at the beginning of this section. PA-Tree achieves two times higher throughput and at least 30% lower latency than the three baseline methods across all the workloads. For any of the methods, the weak persistent version outperforms the strong persistent one by a substantial margin. To achieve strong persistency, at least one NVMe write is required for each update operation, rendering higher latency and lower throughput. As for the weak persistence versions, by persisting on a group of updates, e.g., 1000 updates, rather than individual tuple level in the weak persistent versions, multiple writes to the NVMe are merged and the write overhead is amortized by multiple index operations. This effectively reduces write amplification factor and thus leads to higher throughput and lower latency. Surprisingly, the performance penalty of achieving strong consistency in LSM-Tree is extremely high. That is because LevelDB invokes `sync()` system call to guarantee the flush of the cached I/Os. `sync()` system call is known to be time-consuming and consequently leads to high latency and poor throughput.

VI. Related Work

A large body of work [14], [22], [24], [40] revisits the design of database systems and proposes a set of new techniques for flash memory to take advantage of the fast random read, overcome the limitation of expensive writes and erases by avoiding in-place update, and improve the durability by wear leveling. Oukid et. al. propose a hybrid SCM-DRAM persistent and concurrent range index, called FPTree [29]. FPTree comprises a set of optimizations to minimize the overhead of persisting data on storage-class-memory (SCM), accelerate leaf node search, and utilize the hardware transactional memory for performance enhancement. The major difference between our PA-Tree and those techniques is that PA-Tree is optimized for NVMe interface and does not involve in the optimizations on the internal index structures for the characteristics of a particular storage media. The polled-mode, asynchronous execution paradigm, as the core design in our PA-Tree, is orthogonal to those techniques. Roh et. al. propose a set of optimizations to exploit the internal parallelism of flash-based SSD [32]. Their work is similar to ours in terms of exploiting internal parallelism of storage media. However, their proposal is designed for flash memory and is based on a parallel sync I/O, which is a software wrapper of sync I/O and can only exploit a very limited degree of parallelism. Faster [5] is a highly efficient key-value store tailored for SSD, which combines concurrent hash index with hybrid log. Compared with PA-Tree, it does not support range query. LightNVM [4] provides a new physical page addressing (PPA) I/O interface to exposes SSD parallelism to the high-level applications. PA-Tree can be implemented as an application running on top of LightNVM to take advantage of its performance improvement. Asynchronous Memory Access Chaining (AMAC) [19] allows different operations to execute different and independent code stages at the same time to hide the memory access latency and maximize the throughput. This approach is similar to our PA-Tree in that the execution of multiple operators is asynchronous using a state machine, but it is not applicable in

our scenario mainly because index operations may update the same index node and thus needs coordination.

When using NVM as main memory (NVMM), cache flush instructions, e.g., CLFlush, are needed to make sure the updates to critical data are written into the NVMM rather than residing on the volatile CPU cache. Also, memory barrier instructions, e.g., MFENCE, are needed to guarantee NVMM reads and writes are in a desirable order. Since cache flush is achieved by cache invalidation, which results in cache miss, and MFENCE restricts out of order execution of CPU instructions, which leads to inefficiency in CPU execution, minimizing the usage of those instructions is the crucial in NVMRAM-based index. For instance, Yang et al. [37] propose a consistent and cache-optimized NV-tree. They only maintain leaf nodes persistent and reconstruct inner nodes upon failures to avoid the persistent overhead in inner nodes. Chen et al. propose wB+-tree [7] with a small indirect slot array/bitmap to avoid the movement of index entries while guaranteeing high search performance. Arulraj et al. propose Bztree [2], which uses a software persistent multi-word CAS operation to achieve latch-free. Those techniques target at NVMM which is bit-addressable but has similar performance as main memory.

VII. Conclusion

To address the new challenges and opportunities with the emerging NVMe interface, we present PA-Tree a new B+ tree structure designed with a new polled-mode, asynchronous execution paradigm. PA-Tree is capable of exploiting the internal parallelism of high-end NVMs without the huge overhead associated with multi-threading. Together with the new paradigm in PA-Tree, a group of optimizations are proposed to further improve the utilization of NVMe as well as to reduce operation latency and CPU consumption. Extensive experiments on both real and synthetic datasets verify the huge performance advantage of PA-Tree over existing state-of-the-art B+ tree data structures and system implementations.

VIII. Acknowledgment

Zining is partially supported by Industrial Postgraduate Programme - II under Economic Development Board of Singapore (Ref ID: S18-1259-IPP-II).

References

[1] https://perf.wiki.kernel.org/index.php/Main_Page, 2018.

[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(4), 2017.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta informatica*, 9(1):1–21, 1977.

[4] M. Bjørling, J. González, and P. Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *FAST*, 2017.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. In *SIGMOD*, pages 275–290. ACM, 2018.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 2008.

[7] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[8] K. Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154. ACM, 2010.

[10] A. Dan and D. Towsley. *An approximate analysis of the LRU and FIFO buffer replacement schemes*, volume 18. ACM, 1990.

[11] J. Dean and S. Ghemawat. https://en.wikipedia.org/wiki/LevelDB, 2018.

[12] R. Elmasri. *Fundamentals of database systems*. Pearson Education India, 2008.

[13] N. Express. http://nvmexpress.org/, 2018.

[14] H.-W. Fang, M.-Y. Yeh, P.-L. Suei, and T.-W. Kuo. An adaptive endurance-aware b+-tree for flash memory storage systems. *IEEE Transactions on Computers*, 63(11):2661–2673, 2014.

[15] G. Graefe. A survey of b-tree locking techniques. *ACM Transactions on Database Systems*, 35(3):16, 2010.

[16] Intel. https://www.intel.com/content/www/us/en/io/serial-ata/serial-ata-ahci-spec-rev1-3-1.html, 2018.

[17] Intel. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-brief.pdf, 2018.

[18] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[19] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.

[20] D. Landsman and SanDisk. https://sata-io.org/sites/default/files/documents/NVMe, 2018.

[21] J.-D. Lee, S.-H. Hur, and J.-D. Choi. Effects of floating-gate interference on nand flash memory cell operation. *IEEE Electron Device Letters*, 23(5):264–266, 2002.

[22] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 2010.

[23] P. D. A. Library. https://pandas.pydata.org/, 2018.

[24] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD*, pages 13–24. ACM, 2011.

[25] K. Malbrain. A blink tree latch method and protocol to support synchronous node deletion, 2010.

[26] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[27] P. Neil, E. Cheng, D. Gawlick, and E. Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[28] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[29] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *SIGMOD*, 2016.

[30] PA-Tree. https://github.com/wangli1426/nvm, 2020.

[31] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[32] 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. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[33] SPDK. http://www.spdk.io/doc/nvme.html, 2018.

[34] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang. Elasticutor: Rapid elasticity for realtime stateful stream processing. In *SIGMOD*, pages 573–588, 2019.

[35] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, pages 1279–1294. ACM, 2016.

[36] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[37] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *FAST*, volume 15, pages 167–181, 2015.

[38] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *SIGKDD*, pages 316–324. ACM, 2011.

[39] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[40] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacigümüş. Towards cost-effective storage provisioning for dbmss. *PVLDB*, 2011.