Optimizing Search Layouts in Packed Memory Arrays

Brian Wheatman* Randal Burns* Aydın Buluç† Helen Xu†

Abstract

This paper introduces ***Search-optimized Packed Memory Arrays*** (SPMAs), a collection of data structures based on Packed Memory Arrays (PMAs) that address suboptimal search via cache-optimized search layouts. Traditionally, PMAs and B-trees have tradeoffs between searches/inserts and scans: B-trees were faster for searches and inserts, while PMAs were faster for scans.

Our empirical evaluation shows that SPMAs overcome this tradeoff for unsorted input distributions: on average, SPMAs are faster than B+-trees (a variant of B-trees optimized for scans) on all major operations. We generated datasets and search/insert workloads from the Yahoo! Cloud Serving Benchmark (YCSB) and found that SPMAs are about 2× faster than B+-trees regardless of the ratio of searches to inserts. On uniform random inputs, SPMAs are on average between $1.3\times - 2.3\times$ faster than B+-trees on all operations. Finally, we vary the amount of sortedness in the inputs to stress the worst-case insert distribution in the PMA. We find that the worst-case B+-tree insertion throughput is about 1.5× faster than the worst-case PMA insertion throughput. However, the worst-case input for the PMA is sorted and highly unlikely to appear naturally in practice. The SPMAs maintain higher insertion throughput than the B+-tree when the input is up to 25% sorted.

1 Introduction

The Packed Memory Array (PMA) is a cache-optimized dynamic data structure that stores all of its elements contiguously in memory. Although classical cache-friendly data structures such as B-trees [4] asymptotically match or beat PMAs on all operations, in practice, PMAs support ordered iteration (i.e., scans), a key component of range queries and maps, up to 3× faster than pointer-based structures such as trees because the PMA collocates all of its data [30]. Furthermore, previous work shows that PMAs support updates empirically much faster than the theoretical prediction suggests [36]. Due to its cache-friendliness, the PMA appears in many applications such as graph processing [15, 30, 35, 36], particle simulations [16], and computer graphics [34].

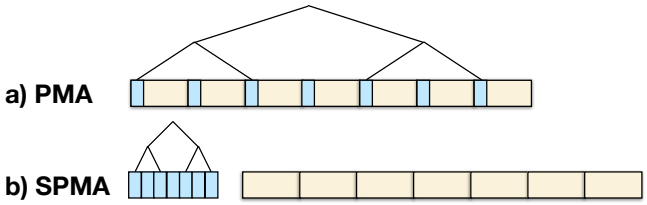

Figure 1: Search structure in PMAs and SPMAs. The yellow boxes represent PMA leaves, or chunks of size $O(\lg(n))$. The blue boxes represent the leaf heads, or the first element in each leaf. The lines represent the implicit search tree that the PMA defines. SPMAs collocate the PMA leaf heads in a separate array.

A main weakness of Packed Memory Arrays (PMAs) [6, 21] is their search cost. Given a cache-line size of B elements [2], searching a PMA with n elements takes $O(\lg(n))$ cache-line transfers. In contrast, B-trees [4] (or other hierarchical cache-friendly data structures) support searches asymptotically faster in $O(\log_B(n))$ transfers. Furthermore, PMA searches are a key component of PMA updates (inserts/deletes), so PMA updates take $\Omega(\lg(n))$ transfers as opposed to B-tree updates, which take $O(\log_B(n))$ transfers.

The PMA's cache-friendliness results in its suboptimal searches. The PMA stores elements with spaces in between them in a single contiguous array for efficient updates and moves elements around upon an insert. This array is implicitly broken up into ***PMA leaves***, or blocks of size $\Theta(\lg(n))$. Searching for an element involves a binary search on the first cells of each leaf, called ***leaf heads***, followed by a scan of the leaf that the target element might reside in. The PMA maintains the invariant that the leaf heads are nonempty. There are $O(n/\lg(n))$ leaves, so the height of the binary search is $O(\lg(n))$. Unfortunately, as shown in Figure 1(a), most of the leaf heads are far apart, so most steps of the binary search incur a cache miss.

In practice, optimizing PMA searches can substantially improve PMA updates. The cost of PMA searches empirically makes up a large fraction of the cost of PMA

*Department of Computer Science, Johns Hopkins University. {wheatman, randal}@cs.jhu.edu.

†Computational Research Division, Lawrence Berkeley National Laboratory. {abuluc, hjxu}@lbl.gov.

Data structure	*Search*	*Insert & delete*	*Range query*
PMA [6, 21]	$O(\lg(n/M))$	$O(\lg(n/M) + (\lg^2(n))/B)^\dagger$	$O(\lg(n/M) + k/B)$
SPMA-Linear	$O(\lg(n/(MB)))$	$O(\lg(n/(MB)) + (\lg^2(n))/B)^\dagger$	$O(\lg(n/(MB)) + k/B)$
SPMA-Eytzinger	$O(\lg(n/M))$	$O(\lg(n/M) + (\lg^2(n))/B)^\dagger$	$O(\lg(n/M) + k/B)$
SPMA-Btree	$O(\log_B(n/M))$	$O(\log_B(n/M) + (\lg^2(n))/B)^\dagger$	$O(\log_B(n/M) + k/B)$
B-tree [4]	$O(\log_B(n/M))$	$O(\log_B(n/M))$	$O(\log_B(n/M) + k/B)$

Table 1: PMA, SPMA, and B-tree bounds. Let B denote the cache-line size [2], let M denote the cache size [2], and let k denote the number of elements returned in a range query. Bounds with † are amortized.

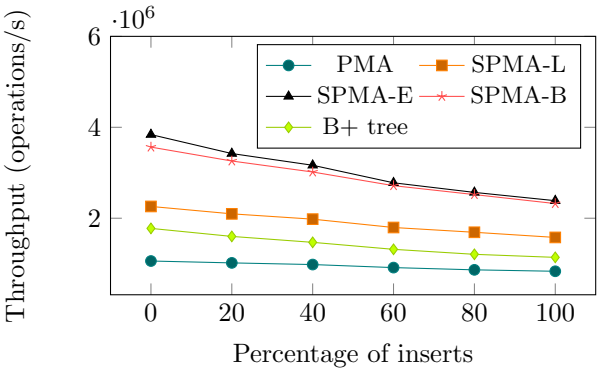

Figure 2: Throughput of serial searches/inserts on mixed workloads from YCSB.

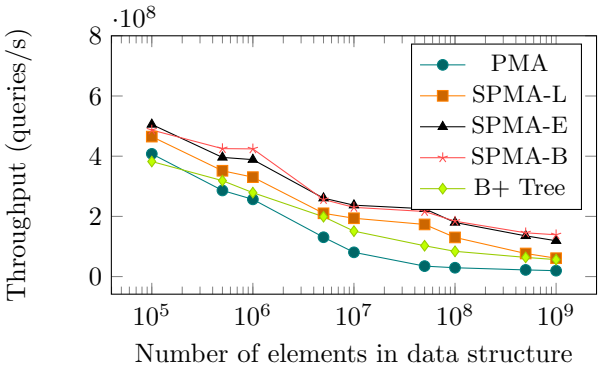

Figure 3: Throughput of simultaneous searches.

updates. For example, on uniform random inputs, PMA search time accounts for about 50% of the total time when inserting 1 million elements and over 75% of the total time when inserting 100 million elements.

This work presents ***Search-optimized PMAs*** (SPMAs), a new collection of data layouts for PMAs that improve the cache-friendliness of PMA searches and, by extension, PMA updates. As shown in Figure 1(b), the main idea behind SPMAs is to collocate the PMA leaf heads in a separate array and arrange them in a more efficient order. The two-array SPMA structure is inspired by previous work [5–8, 32]. This work generalizes the design to arbitrary leaf head orders and explores both theoretically and practically efficient search layouts for the leaf head array.

Specifically, we introduce three examples of SPMAs: SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree. SPMA-Eytzinger matches the asymptotics of traditional PMAs but performs much better in practice because it is optimized to take advantage of modern CPU hardware features such as prefetching and the branch predictor. SPMA-Linear reduces the theoretical search cost of PMAs by a low-order term. Finally, SPMA-Btree decreases the theoretical search cost of PMAs by a factor of $O(\lg(B))$ to match that of B-trees. Table 1 summarizes the bounds [5,6,9,37–39] for PMAs, SPMAs, and B-trees in the Ideal-Cache model [18] detailed in Section 2.

To empirically evaluate the SPMAs, we compare them to a B+-tree [13] from the TLX library [11] and a traditional PMA on the search, insert, and scan operations. The plots in this section use SPMA-L, SPMA-E, and SPMA-B to refer to SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree respectively. All data structure implementations considered in this paper support read-only operations (i.e., queries and scans) in parallel, but are not thread-safe for writes. The search optimizations in this paper are orthogonal to prior work on concurrent PMAs, so the SPMAs can be made thread-safe for writes using techniques described in previous work [36]. In this paper, we evaluate the throughput of multiple simultaneous read-only operations and serial write operations.

The empirical evaluation shows that SPMAs overcome the traditional tradeoff between search/insert and scan performance on unsorted inputs: on average, they are faster than B+-trees for all major operations.

Figure 2 shows that the SPMAs are about 2× faster than B+-trees on an application benchmark of mixed search/insert workloads from the widely used Yahoo! Cloud Serving Benchmark (YCSB) [14]. The evaluation shows that SPMAs are faster than B+-trees on mixed workloads regardless of the ratio of searches to inserts. Section 5 also demonstrates that SPMAs are around 2.4× faster than B+-trees on mixed range query/insert workloads from YCSB. We ran all benchmarks from YCSB serially for consistency since most contain writes.

Furthermore, on uniform random inputs, SPMAs about 1.7× faster than B+-trees for searches, about 1.4× faster for inserts, about 2.3× faster for scans, and about 1.3× faster for range queries. Figure 3 shows that SPMAs improve search performance by up to 7× compared to traditional PMAs.

Section 5 also evaluates the PMA, SPMAs, and B+-tree on more structured inputs with ordered inserts and shows that the worst-case B+-tree insertion throughput is about 1.5× faster than the worst-case PMA insertion throughput. Sorted inserts are the worst case for PMAs because they maximize the number of element moves necessary to maintain the spaces spread out through the underlying array. However, the SPMAs match or exceed the B+-tree's insertion throughput when the input is up to 25% sorted.

Specifically, our contributions are as follows:

- The high-level design of SPMAs, which improve the cache-friendliness of PMA searches.
- The design and analysis of three variants of SPMAs: SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree.
- An implementation of the three SPMAs in `C++`.
- An empirical evaluation of the three SPMAs compared to PMAs and B+-trees.

Map: The rest of this paper is organized as follows. Section 2 provides background about the PMA and describes the salient features of modern computer architectures that enable the performance improvements in SPMAs. Section 3 describes the high-level SPMA design, and Section 4 introduces and analyzes SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree. Section 5 empirically evaluates the SPMAs on search, insert, scan, and range query, and shows that they improve upon PMAs and overcome traditional empirical tradeoffs between B+-trees and PMAs. Section 6 provides concluding remarks.

2 Background

We review the Ideal-Cache model [18], a variant of the classical external-memory model [2], which we will use to analyze data structure operations throughout the paper. Next, we present details about the Packed Memory Array, including a description of its operations.

We then summarize relevant features of a modern processor architecture that SPMAs take advantage of to improve overall performance. Specifically, we will discuss the latency of fetches between CPU and RAM and how processors alleviate latency issues with caches, prefetching, and pipelining. This discussion is inspired by a similar summary by Khuong and Morin [23].

Analysis method: The Ideal-Cache model contains two levels of memory: a small bounded-size cache of size M, and an unbounded-size memory. Any data must be brought to the cache first before it is processed. Data is transferred in blocks of size B between the cache and the memory, and transfers have unit cost. The cost of algorithms is measured in terms of transfers between cache and memory. The Ideal-Cache model includes the ***tall-cache assumption***: that is, it assumes the cache size $M = \Omega(B^2)$.

Packed Memory Arrays: The PMA is a dynamic dictionary data structure that stores n elements in sorted order in a $\Theta(n)$-sized array with empty cells interspersed between elements in order to support efficient updates [6, 21]. PMAs support scans of k elements in $O(k/B)$ transfers.

In this paper, we will consider the following major data structure operations:

- `search(x)`: returns a pointer to the smallest element that is at least `x` in the PMA.
- `insert(x)`/ `delete(x)`: inserts/deletes element `x` into/from the PMA.
- `range_map(start, end, f)`: applies the function `f` to all elements in the range [`start`, `end`).

In this paper, we will use the more popular term "range query" instead of "range map,", but range maps are more general and are used to implement range queries.

As mentioned in Section 1, searching a PMA takes $O(\lg(n))$ cache-line transfers because it involves a binary search on the $O(n)$ leaf heads. The top $O(\lg(M))$ levels of the binary search are stored in the cache, so the total number of transfers in a PMA search is $O(\lg(n) - \lg(M)) = O(\lg(n/M))$. Range maps can be implemented with a search for the start element and a scan from that point.

Although search is a key subroutine in PMA insert (and delete), it is asymptotically a low-order term in the theoretical insert cost. Suppose we are inserting an element x into a PMA. The insert operation first performs a search for x, which has $O(\lg(n/M))$ cost. If x is not in the PMA, the insert then places x in the target leaf discovered by the search. After the place step, the PMA counts the number of elements in the leaf that was inserted into to check if it is "too full." If the leaf is too full, the PMA may shuffle elements around in a "redistribute" to maintain enough empty

spaces in the PMA. The amortized cost of counting and redistributing is $O((\lg^2(n))/B)$, resulting in the PMA insert bound of $O(\lg(n/M) + (\lg^2(n))/B)$ [6, 21]. B-trees asymptotically beat PMAs on inserts and support updates in $O(\log_B(n/M))$ transfers.

Modern computer architecture: At a high level, a modern multicore computer is made up of a processor (CPU) connected to a random-access memory (RAM). If the CPU requires the contents of RAM, the main bottleneck is often the latency of memory transfers, which can take hundreds of cycles.

To alleviate the cost of reading data from RAM, modern architectures contain steep cache hierarchies, or multiple levels of cache, between the CPU and RAM. When a CPU reads data from RAM, the RAM moves a ***cache line***, or a contiguous block of data (often 64 bytes), to intermediate caches between the CPU and RAM. Specifically, the cache-line size parameter B in the Ideal-Cache model captures how well algorithms take advantage of cache-line transfers.

To further reduce the latency of bringing data from RAM to CPU, modern processors also contain ***hardware prefetchers***, which attempt to predict future memory accesses and pre-load the data before the CPU accesses it [27]. Successful prefetching ensures that the data is in cache when the program needs it. Taking full advantage of prefetching requires collocating as much data as possible because modern prefetchers can only guess simple access patterns [23].

Finally, CPUs improve throughput by ***pipelining*** instructions rather than performing them one at a time [19]. Each instruction is composed of several discrete stages. The CPU arranges the instructions in a pipeline so that multiple instructions at different stages are executed at the same time. [31] Since effective pipelining depends on the CPU knowing which instruction to fetch and perform next, modern processors also include ***branch predictors*** that guess which instructions come next after a conditional jump (e.g., an if statement) that changes the flow of execution. Branch prediction works best when the conditions are highly predictable, e.g. in simple loops.

3 Search-optimized PMAs

This section motivates the cache-friendliness of Search-optimized PMAs (SPMAs) and formalizes their design. At a high level, SPMAs improve the cache-friendliness of PMAs by separating the PMA into two arrays: one for the leaf heads, and one for the rest of the data. The SPMA design improves cache utilization by allowing larger SPMAs to fit their leaf heads into cache. We present a framework for creating SPMAs by defining the order of the leaf heads. Finally, we show to how analyze the search and insert operations in an SPMA based on the leaf head order. For clarity, this section focuses on inserts because deletes are symmetric.

SPMA design and benefits: More formally, SPMAs divide the traditional single array of a PMA into two arrays: a ***leaf head array*** that stores the leaf heads, and a ***data array*** that stores the rest of the PMA. Each of the arrays is stored contiguously. Figure 1(b) illustrates the high-level structure of an SPMA.

Collocating the leaf heads improves the PMA's cache-friendliness in two main ways. First, the leaf head array is much smaller than the rest of the PMA (there are $O(n/\lg(n))$ leaf heads in a PMA with n elements), which enables it to fit in cache even as n grows larger than the cache size. Furthermore, the leaf head array puts the relevant elements in the search closer together, so that each cache-line fetch is likely to contain a larger fraction of elements along the search path.

SPMA specification: To define an SPMA, one needs to specify a ***mapping*** called `leaf_to_head` between the PMA leaves and the indices in the leaf head array. The mapping is a function that defines the ***search layout***, or the order of elements in the leaf head array that is used during a search or insert. For example, the simplest mapping is the identity (i.e., the leaf index is the same as the leaf head index), which underlies SPMA-Linear. As we shall see, SPMAs may change the mapping from the identity to a more complex function in order to enable more efficient searches. For example, SPMA-Eytzinger stores the leaf heads in "Eytzinger order," and SPMA-Btree stores the leaf heads in "B-tree order."

Searching an SPMA: Just as in a PMA, searching an SPMA takes two steps: (1) finding the correct leaf that the target element resides in, and (2) a search of that leaf. Step (1) involves performing a search in the leaf head array based on the search layout. Since step (2) is the same as in a traditional PMA and a low-order term, the main factor in an SPMA search is the cost of finding the right leaf.

Inserting in an SPMA: Inserting into an SPMA has the same steps as inserting into a PMA described in Section 2:

1. A search to find the leaf to insert the target element into,
2. A place to put the element in sorted order, and
3. A "redistribute" to ensure that there are enough empty spaces throughout the PMA.

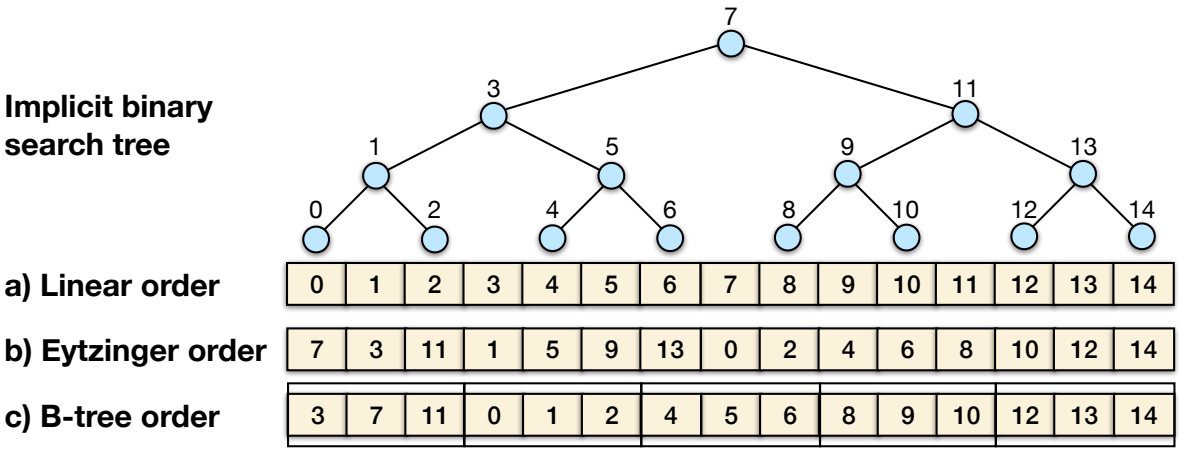

Figure 4: Mappings used in the SPMAs. The numbers in the cells represent the leaf index, and the examples illustrate the ordering of the leaf heads in the leaf head array. In this example, $B = 3$ for the B-tree order. The larger boxes around several cells in B-tree order illustrate the blocking of elements.

Placing the element is trivial once we have found the correct leaf to insert into, so the focus of the analysis is on redistributes. A ***redistribute*** takes as input a ***region*** of the PMA, or a nonempty set R of contiguous leaves, and spreads elements evenly amongst the leaves in the region. Redistributing a region of $|R|$ leaves in a PMA with n elements takes $O((|R| \lg(n))/B)$ cache-line transfers, since the leaves are stored contiguously and each leaf has $O(\lg(n))$ cells.

Showing that an SPMA maintains the PMA's insertion bound is equivalent to showing that gathering the leaf heads for the leaves in a region incurs no additional asymptotic cost over the search and redistribute steps. Given a region R, let $G(R)$ be the number of transfers required to gather the leaf heads of the leaves in R in an SPMA. Given a SPMA with n elements with search cost S_n, it suffices to show that $G(R) = O(\max\{S_n, (|R| \lg(n))/B\})$.

4 Specific search layouts in SPMAs

This section introduces three examples of SPMAs: SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree. The choice of search layouts in this section is inspired by previous studies of static array layouts for comparison-based searching [1, 23].

We demonstrate that these SPMAs maintain or improve upon traditional PMA bounds using the analysis framework presented in Section 3. The proofs in this section use the Ideal-Cache model defined in Section 2.

This section explains the empirical search performance of the different layouts based on both their asymptotic bounds as well as their usage of hardware features. All of the SPMAs considered in this paper match the PMA's asymptotic scan performance. We will omit proofs of scan performance because they directly follow from efficient gathering of leaf heads, which we will show in the proofs of insert performance.

SPMA-Linear: The most straightforward SPMA is ***SPMA-Linear***, which collocates the leaf heads in the order that they appeared in the PMA. The SPMA mapping between leaf indices to leaf heads is the identity. Searching in SPMA-Linear is simply a binary search on the leaf head array followed by a scan of the target leaf. Gathering the leaf heads during a redistribute does not incur any additional asymptotic cost, since it just involves a scan through the relevant part of the leaf head array. Figure 4(a) illustrates an example of a leaf head array in SPMA-Linear.

LEMMA 4.1. *SPMA-Linear supports searches in* $O(\lg(n/(MB)))$ *transfers and inserts in amortized* $O(\lg(n/(MB)) + (\lg^2(n))/B)$*transfers.*

Proof. The height of the binary search is $O(\lg(n))$, but when performing repeated searches, the top $O(\lg(M))$ (most accessed) levels of the search are cached and do not incur additional transfers. Furthermore, the last $O(\lg(B))$ levels of the search only occur in a single cache line, so they do not require additional transfers. Therefore, searches take $O(\lg(n) - \lg(M) - \lg(B)) = O(\lg(n/(MB)))$ transfers. The insert bound follows directly from the search bound. ∎

Although $\lg(B)$ is a lower-order term, it significantly improves the practical search performance of SPMA-Linear compared to a traditional PMA by decreasing the number of cache misses.

SPMA-Eytzinger: The next SPMA we study, ***SPMA-Eytzinger***, arranges the leaf heads in Eytzinger order, or breadth-first order. This layout first appeared in the 1500s in the context of genealogical trees [17], but was more recently adapted as an implementation of binary heaps [40]. More formally, a sorted array in Eytzinger order stores the root of the implicit search tree at index 0 and the values of the left and right children of the node at index i at indices $2i+1$ and $2i+2$, respectively. Figure 4(b) presents an example of a leaf head array in SPMA-Eytzinger.

Searching in SPMA-Eytzinger starts at the beginning of the leaf head array and skips forward to the next level of the search at each step. To insert an element in SPMA-Eytzinger, we perform arithmetic to convert the leaf head index from Eytzinger order to linear order to find the target leaf in the data array.

SPMA-Eytzinger empirically performs much better than both the PMA and SPMA-Linear because its access pattern makes better use of the branch predictor [23]. For example, let us consider 8-byte elements and a system with 64-byte cache lines. The location of the next level of the binary search is within a range of 16 bytes because Eytzinger order places siblings next to each other. With these parameters, the machine can accurately predict the location of the next three levels of the search because each cache line contains 8 elements, which is enough to hold all possible branches at the next three levels. Therefore, the machine is very likely to fetch the next correct bytes regardless of which way the branch predictor chooses.

Analyzing SPMA-Eytzinger: We will show that SPMA-Eytzinger theoretically matches the PMA.

LEMMA 4.2. *SPMA-Eytzinger supports searches in* $O(\lg(n/M))$ *cache-line transfers.*

Proof. The height of the search is $O(\lg(n))$ and the top $O(\lg(M))$ levels of the search are held in cache for a total search cost of $O(\lg(n/M))$ transfers. The first cache line in the leaf head array holds the heads for the first $O(\lg(B))$ levels. However, the first cache line does not reduce the asymptotic cost of the search because the top levels of the search are already cached. Furthermore, the mapping of `leaf_to_head` takes no additional cache-line transfers because generating the index in Eytzinger order from linear order only requires arithmetic operations. ∎

Since gathering leaf heads for a region in SPMA-Eytzinger is more complicated than in a standard PMA, we show that storing the leaf heads in Eytzinger order does not asymptotically increase the insert cost.

LEMMA 4.3. *SPMA-Eytzinger supports inserts in amortized* $O(\lg(n/M) + (\lg^2(n))/B)$ *transfers.*

Proof. We will show that accessing the leaf heads for a scan of a region R (with $|R|$ leaves) in SPMA-Eytzinger does not affect the asymptotic cost of inserts. We proceed by case analysis.

Case 1: $B = o(\lg(n))$. In this case, scanning over the leaf heads incurs asymptotically no more transfers over the cost to perform the scan over the region. The scan over R reads $O(|R|\lg(n))$ cells in $\Theta((|R|\lg(n))/B)$ transfers. Since $B = o(\lg(n))$, $\Theta((|R|\lg(n))/B) = \Omega(|R|)$, so the cost to read the leaf heads is bounded above by the cost to read the remaining cells, even if each leaf head incurs an additional transfer to read.

Case 2: $B = \Omega(\lg(n))$. In this case, scanning over the leaf heads incurs no more than an additive $O(\lg(n))$ transfers. When the heads are stored in Eytzinger order, each level of the implicit binary tree can be traversed cache-efficiently since they are stored in order. Therefore, the only additional transfers that SPMA-Eytzinger incurs are the misses to bring in the first leaf head at each level of the tree, which is bounded above by $O(\lg(n))$. Due to the tall-cache assumption, the cache has $\Omega(B)$ lines, which in this case is also $\Omega(\lg(n))$ lines. Therefore, the cache can hold a line for each of the $O(\lg(n))$ levels of the implicit search tree for efficient traversal during the scan. ∎

SPMA-Btree: Finally, we introduce ***SPMA-Btree***, which asymptotically improves the search performance of PMAs to match B-trees. SPMA-Btree arranges the leaf heads in ***B-tree*** order, or an extension of Eytzinger order from 2-way branching to $(B+1)$-way branching [23]. Instead of storing each element as a node in the search tree, B-tree order stores elements in blocks of size B. Those blocks are stored as nodes in a search tree and arranged in Eytzinger order. That is, each block has B elements and represents a node with $B+1$ children. B-tree order was first introduced for heaps [22, 24, 29] and was later adapted for comparison-based searching [23]. Furthermore, Raman [32] proposed using a traditional B-tree layout (without Eytzinger order) for storing PMA leaf heads. Figure 4(c) illustrates an example of a leaf head array stored in B-tree with Eytzinger order.

Querying an element in SPMA-Btree involves a search of height $O(\log_B(n))$ on the blocks of leaf heads defined by the mapping. Just as in SPMA-Eytzinger, an insert includes arithmetic to convert the leaf head index from B-tree order to the original linear order to find the target leaf.

LEMMA 4.4. *SPMA-Btree supports searches in*

$O(\log_B(n/M))$ *transfers and inserts in amortized* $O(\log_B(n/M) + (\lg^2(n))/B)$ *transfers.*

Proof. B-tree order asymptotically reduces the cost of searches from $O(\lg(n/M))$ transfers to $O(\log_B(n/M))$ transfers due to the layout of the leaf head array. The B-tree mapping places B pivots on each cache line for a total search height of $O(\log_B(n))$. The top $O(\log_B(M))$ levels are stored in cache for a total search cost of $O(\log_B(n/M))$ transfers. The insert bound follows directly from the search bound and the proof of Lemma 4.3. ∎

5 Evaluation

This section shows that despite the theoretical prediction, SPMAs empirically outperform B+-trees on all operations on unsorted inputs due to the PMA's cache-friendliness.

SPMAs are about 2× faster than B+-trees on mixed search/insert workloads and about 2.4× faster than B+-trees on mixed range query/insert workloads from the Yahoo! Cloud Serving Benchmark (YCSB) [14]. As we shall see, the SPMAs are robust to the skewed Zipfian [12,28] distribution present in the YCSB workloads.

Additionally, on uniform inputs, SPMAs are on average 1.7× faster on parallel searches, 1.4× faster on inserts, 2.3× faster on parallel scans, and 1.3× faster on parallel range queries when compared to B+-trees.

On more structured inputs with some sorted insertions, we expect B+-trees to have the advantage on insertions because sorted inserts are the worst case for the PMA [10]. We find that the worst-case insertion throughput for B+-trees is about 1.5× faster than the worst-case insertion throughput for the SPMAs. However, the SPMAs match or outperform the B+-tree on insertions when the input is up to 25% sorted.

The plots in this section use SPMA-L, SPMA-E, and SPMA-B to refer to SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree, respectively.

We focus on the case of insertions for clarity, but our experiments show that deletions and insertions have similar performance.

Systems setup: We implemented the PMA, SPMA-Linear, SPMA-Eytzinger, and SPMA-Btree as a `C++` library that supports the search, update, and scan operations. As mentioned in Section 1, all data structures considered in this paper support simultaneous read-only operations (i.e., searches, range queries, and scans), but currently support a single writer at a time. Therefore, we run all operations in parallel except for inserts. We implemented range queries with a search and then a scan from that point. We set the $B = 16$ in SPMA-Btree, but tested with $B = 4$ and $B = 8$ as well and found similar results. We implemented a parallel scan using `Cilk` [20] and the Tapir [33] branch of the LLVM [25, 26] compiler (version 12).

We used the single-threaded B+-tree [13] from the TLX library [11] as a baseline. The B+-tree is a variant of B-trees with all elements at the leaves, which enables faster scans than traditional B-trees [13].

We ran all data structures as key stores (i.e. without values) with 64-bit keys.

All experiments were run on a `c5.metal` 48-core 2-way hyper-threaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [3]. Across all of the cores, the `c5.metal` contains 1.5MiB of L1 cache, 48MiB of L2 cache, and 71.5MiB of L3 cache. For each experiment, we took the median of 5 trials.

YCSB experimental setup: We evaluate all data structures on mixed query/insert workloads from YCSB [14], a popular application benchmark for data stores. Tests in YCSB have two phases: a load and run phase. The load phase generates a number of elements from some distribution to operate on in the run phase. For the load phase, we generated 10^8 elements from a Zipfian distribution using YCSB. We then tested two types of run phases: search/insert workloads, and range query/insert workloads. For the search/insert workloads, we generated 10^8 total operations made up of searches and inserts. Specifically, we modified workload A from the examples [41] to generate insertions instead of updates because updates are simply a search in key-only mode. We then varied the ratio of searches to inserts. For the range query/insert workloads, we used workload E from the examples [41] to generate workloads of 10^8 operations with 95% range queries and 5% inserts. We varied the size of the range queries by changing the `max_scan` parameter in YCSB, which determines the maximum number of elements in each range query. Each range query first searches for some element `x` and then requests a uniform random number $k \leq$ `max_scan` of subsequent elements (in sorted order). We run all YCSB experiments serially because most include writes.

YCSB searches and inserts: Figure 2 and Table 2 demonstrate that SPMAs are about 2× faster than B+-trees and 3× faster than traditional PMAs on mixed search/insert workloads from YCSB. As detailed in Section 4, SPMA-Eytzinger performs much better than the theory predicts because it takes advantage of prefetching and branch prediction, which are not accounted for in the Ideal-Cache model. Although SPMA-Eytzinger is worse theoretically than both SPMA-Linear and SPMA-Btree, SPMA-Eytzinger is always faster than SPMA-

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
% inserts	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
0	1.1E+6	2.3E+6	2.1	3.8E+6	3.6	3.6E+6	3.4	1.8E+6	1.7
20	1.0E+6	2.1E+6	2.1	3.4E+6	3.4	3.3E+6	3.2	1.6E+6	1.6
40	9.8E+5	2.0E+6	2.0	3.2E+6	3.2	3.0E+6	3.1	1.5E+6	1.5
60	9.1E+5	1.8E+6	2.0	2.8E+6	3.0	2.7E+6	3.0	1.3E+6	1.4
80	8.6E+5	1.7E+6	2.0	2.6E+6	3.0	2.5E+6	2.9	1.2E+6	1.4
100	8.3E+5	1.6E+6	1.9	2.4E+6	2.9	2.3E+6	2.8	1.1E+6	1.4

Table 2: Throughput (in operations/second) of mixed search/insert workloads from YCSB. SU denotes the speedup over the traditional PMA.

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
Max scan length	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
10	9.74E+05	1.93E+06	2.0	3.24E+06	3.3	3.14E+06	3.2	1.56E+06	1.6
100	8.88E+05	1.48E+06	1.7	2.11E+06	2.4	1.98E+06	2.2	1.06E+06	1.2
1000	4.84E+05	6.19E+05	1.3	6.94E+05	1.4	6.45E+05	1.3	2.67E+05	0.6
10000	8.52E+04	9.52E+04	1.1	1.02E+05	1.2	9.00E+04	1.1	3.10E+04	0.4

Table 3: Throughput (in operations/second) of mixed range query/insert workloads from YCSB. SU denotes the speedup over the traditional PMA.

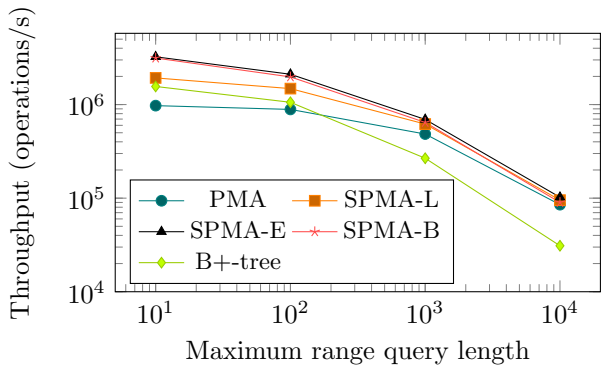

Figure 5: Throughput of serial range queries/inserts from YCSB.

Linear and similar to SPMA-Btree. The SPMAs are faster than the B+-tree on both searches and inserts, so they are faster on the mixed workloads. Although the workloads from YCSB follow a skewed (Zipfian) distribution, the YCSB generator outputs elements randomly within the distribution. The worst-case insertion workload for the PMA depends only on the sortedness of the input and not the closeness of the elements, so SPMAs outperform B+-trees on YCSB workloads regardless of the fraction of searches to inserts.

YCSB range queries and inserts: Figure 5 and Table 3 show that SPMAs are about 2.4× faster than B+-trees and about 1.5× faster than PMAs for range queries. SPMAs overcome the traditional tradeoff between searches and scans for B-trees and PMAs for improved range query performance regardless of the length of the range query. A range query in YCSB is defined by an element `x` and a number of elements k to return starting from `x` (in sorted order). Performing a range query takes two steps: 1) a point query to find the smallest element that is at least `x`, and 2) a scan from `x` over the (sorted) data structure over the next k elements. B+-trees are faster to search than PMAs, whereas PMAs are faster to scan than B+-trees. When k is small (≤ 100), B+-trees are faster than PMAs for range queries because the cost is dominated by the search. In contrast, when k is large (> 100), the cost of the scan dominates the cost of the range query, and the PMA is faster for range queries. PMAs and SPMAs converge when k is large because the search cost is small compared to the scan cost.

Experimental design for uniform random and sorted inputs: We also compare all data structures on their update, search, scan, and range query performance on uniform random inputs. For each experiment, let n be the number of elements in the data structure. To measure insert performance, we measure the time to insert n uniform random 40-bit numbers serially. To measure the space usage, we report the size of each data structure after the n insertions. To measure search performance, we then in parallel search for a group of 10^7 uniform random 40-bit numbers. Each individual search proceeds serially, but multiple searches happen simultaneously. To measure scan performance, we sum up the elements in parallel.

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
Elements	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1E5	4.1E8	4.7E8	1.1	5.1E8	1.2	4.9E8	1.2	3.8E8	0.9
1E6	2.6E8	3.3E8	1.3	3.9E8	1.5	4.2E8	1.7	2.8E8	1.1
1E7	8.1E7	1.9E8	2.4	2.4E8	2.9	2.3E8	2.9	1.5E8	1.9
1E8	2.9E7	1.3E8	4.4	1.8E8	6.1	1.8E8	6.3	8.4E7	2.9
1E9	2.0E7	6.1E7	3.1	1.2E8	6.1	1.4E8	7.1	5.6E7	2.9

Table 4: Throughput (in elements/second) of searching for elements in parallel on the uniform random input. SU denotes the speedup over the traditional PMA.

Since the TLX B+-tree does not come with a parallel scan, we implement a best-case parallel sum for the B+-tree. First, we equally split up the B+-tree with a scan through the elements which is not counted in the time. We then parallelize over these equally split parts and perform a (serial) sum in each of the parts. Finally, we reduce the partial sums but do not count this last step in the time. We take the minimum of the serial and parallel sum times.

To test range queries, we fix the size of the data structure and number of queries and vary ℓ, the range of the query. Specifically, we insert $n = 10^8$ uniform random 40-bit elements into each data structure and then pick $m = 10^5$ uniform random elements as start points for the range queries. Let $\{s_1, s_2, \ldots, s_m\}$ denote the m start points. In parallel, for all $i = 1, 2, \ldots, m$, we perform a range query in the range $[s_i,\ s_i + \ell)$. For this particular query, we count the number of elements in the range. For each ℓ, we limit the start points to fall in the range $[0, 2^{40} - \ell]$ to ensure the entire range queried is within the experiment's universe of elements.

We also compare insertion performance for all of the data structures on inputs with varying amounts of sortedness inspired by previous work [10] to stress the worst case in the PMA. The worst-case input for PMAs (and SPMAs) always inserts a new minimum element at the beginning of the array. We only measure insertion performance for these inputs because changing the input distribution does not affect the other operations (search, range query, and scan). For each of these tests, we insert $n = 10^8$ elements, each with probability p of being smaller than the current minimum element (i.e., an insert at the beginning) and probability $1 - p$ of being a uniformly random 40-bit number. We add 10^8 to all of the random 40-bit numbers to ensure that the sorted inserts at the beginning are always smaller than the current minimum element.

Searches: Figure 3 and Table 4 show that SPMA-Eytzinger and SPMA-Btree are between $1.3\times - 2.5\times$ faster for searches than B+-trees. Additionally, SPMAs are up to $7.1\times$ faster to search than a traditional PMA. At smaller input sizes below 10^7 elements, all tested data structures have relatively similar search performance (within $1.5\times$) because they all fit in cache and therefore have lower access latency. Although B+-trees asymptotically match SPMA-Btree and dominate SPMA-Eytzinger for searches, the SPMAs are faster to search because they make better use of prefetching and the branch predictor as discussed in Section 4. Therefore, the gap between SPMAs and B+-trees grows to $2.5\times$ on larger inputs when the memory access latency increases.

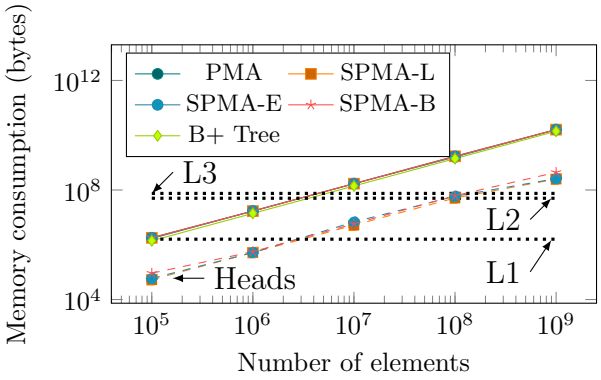

Figure 6: Memory footprints of the different data structures. The dashed lines show the size of the different head structures, and the dotted lines show the size of the various caches.

Figure 6 reports the sizes of the data structures compared to the cache sizes, which correspond to trends in search performance. Search performance in a traditional PMA matches the SPMAs for small inputs, but drops off when the PMA exceeds the cache size (before 10^7 elements). In contrast, SPMA-Linear supports fast searches until the leaf head array exceeds the size of cache (at about 10^8 elements). Furthermore, even though the head arrays of SPMA-Eytzinger/SPMA-Btree and SPMA-Linear are of similar size, SPMA-Eytzinger and SPMA-Btree maintain efficient searches with cache-optimized search layouts even when the head arrays exceed the cache size.

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
Elements	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1E5	6.0E6	8.0E6	1.3	8.5E6	1.4	6.5E6	1.1	1.0E7	1.7
1E6	4.1E6	5.9E6	1.4	6.7E6	1.6	5.7E6	1.4	5.4E6	1.3
1E7	2.1E6	3.3E6	1.6	3.9E6	1.9	3.7E6	1.8	2.6E6	1.2
1E8	1.0E6	2.0E6	2.0	2.8E6	2.8	2.6E6	2.6	1.4E6	1.4
1E9	8.0E5	1.1E6	1.4	1.8E6	2.3	1.8E6	2.3	8.9E5	1.1

Table 5: Throughput (in elements/second) of inserting all uniform random elements in serial. SU denotes speedup over the traditional PMA.

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
% Sequential	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
0	1.1E06	2.0E06	1.8	2.7E06	2.4	2.6E06	2.3	1.4E06	1.2
10	1.0E06	1.8E06	1.8	2.3E06	2.3	1.9E06	1.9	1.5E06	1.5
20	1.0E06	1.7E06	1.6	2.0E06	2.0	1.5E06	1.5	1.6E06	1.6
30	9.7E05	1.6E06	1.6	1.8E06	1.8	1.3E06	1.3	1.9E06	1.9
40	9.3E05	1.4E06	1.5	1.6E06	1.7	1.1E06	1.2	2.1E06	2.3
50	9.2E05	1.3E06	1.5	1.5E06	1.6	9.4E05	1.0	2.4E06	2.6
100	8.4E05	1.0E06	1.2	1.0E06	1.2	5.9E05	0.7	7.9E06	9.4

Table 6: Insertion throughput (in elements/second) on mixed sorted/unsorted inputs as a function of the percent of sorted inserts. We use SU to denote the speedup over the PMA.

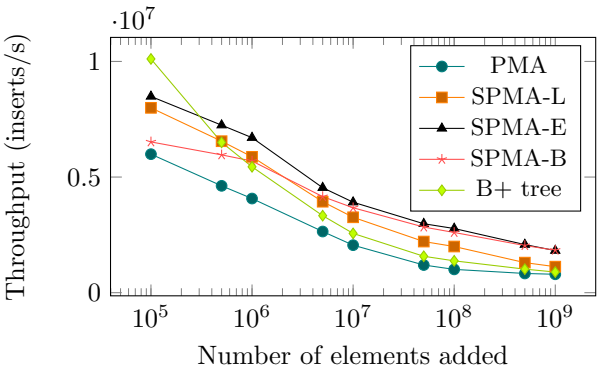

Figure 7: Insertion throughput on uniform random inputs.

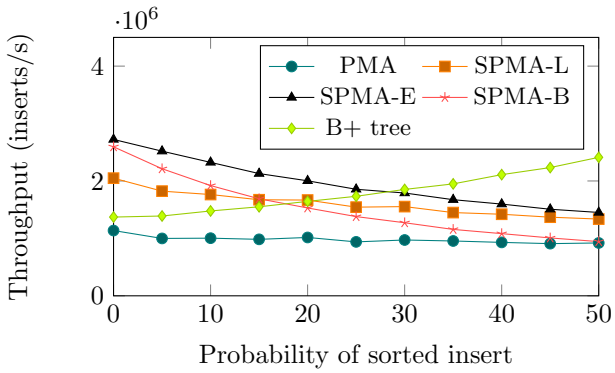

Figure 8: Insertion throughput on mixed sorted/unsorted inputs.

Uniform random inserts: Figure 7 and Table 5 show that contrary to the theoretical prediction, SPMAs perform inserts on average 1.4× faster and up to 2.1× faster than the B+-tree on unsorted inputs. Furthermore, SPMAs perform inserts up to 2.3× faster than PMAs.

The gap between the SPMAs and B+-tree grows as the input size increases because the relative fraction of time spent in searches is proportional to the input size when the data is uniformly random. For small inputs of less than 10^6 elements, the B+-tree is up to 1.4× faster than the SPMAs because the entire data structure fits in cache, so the cache-friendliness of the PMA does not yet make up for its asymptotically worse searches. For example, in a traditional PMA, the search takes about 30% of the time when inserting 10^5 elements, but over 75% of the time when inserting 10^8 elements. Furthermore, even though the cost of searching is smaller in SPMAs than in the PMA, searching still makes up a substantial fraction (25% – 44%) of the insertion time. Therefore, just as in the search experiments, we find that improvements are most prominent when the data structures exceed the cache size, or are larger than 10^7 elements.

Sorted inserts: Figure 8 and Table 6 show how the different structures behave as the inputs become less

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
Elements	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1E5	2E9	2E9	1.0	1E9	0.9	2E9	1.0	6E8	0.4
1E6	4E9	3E9	1.0	3E9	0.8	3E9	0.8	9E8	0.2
1E7	6E9	6E9	1.0	6E9	0.9	6E9	0.9	3E9	0.5
1E8	7E9	7E9	1.0	7E9	0.9	7E9	0.9	4E9	0.5
1E9	7E9	8E9	1.0	7E9	1.0	7E9	1.0	4E9	0.5

Table 7: Throughput (in elements/second) of scanning the entire data structure. We use SU to denote the speedup over the PMA.

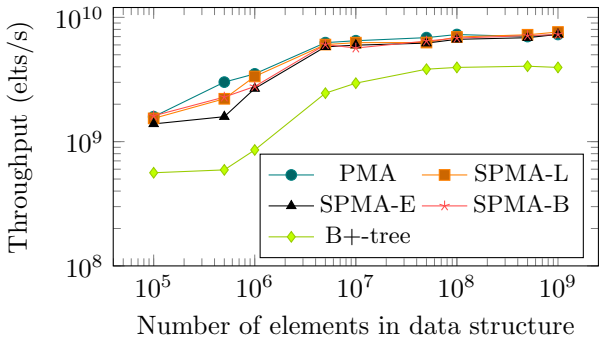

Figure 9: Throughput of parallel scan.

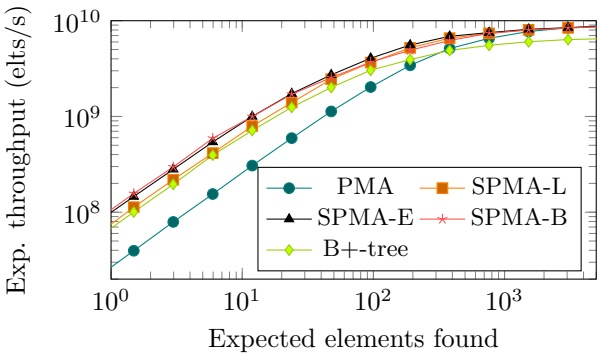

Figure 10: Expected elements processed (queries × (expected number of elements per query)) per second in simultaneous range queries.

random and more sorted. Sorted inserts of new minimum elements stress the worst case in the PMA because they maximize the number of redistributes. They are also the best case for the B+-tree because the search path and the impacted leaves will always be in cache.

We find that the original PMA is always worse for inserts than the B+-tree regardless of the sortedness of the input, but that the SPMAs are faster than the B+-tree for inserts until 25% of the input is sorted. The traditional PMA's throughput does not change much even as the fraction of sorted inserts changes because the benefit of following the same search path repeatedly is similar to the cost of more redistributes. In contrast, searching in the SPMAs is a much smaller fraction of the total cost, so the redistributes decrease the overall throughput. Finally, as the fraction of sorted inserts increases, SPMA-Btree has the slowest throughput of all the SPMAs because it has the most complicated mapping and therefore the most expensive redistributes (which must manage the leaf head array).

The worst-case insertion throughput for the B+-tree is about 1.5× faster than the worst-case insertion throughput for the SPMA. For the worst-case for the PMA (100% ordered inserts), SPMA-Eytzinger achieves about 1 million inserts per second, which is almost a 3× slowdown over random inserts. At the same time, for the worst case for the B+-tree (0% ordered inserts), the B+-tree achieves almost 1.5 million inserts a second.

In the extreme case of 100% ordered inserts, the B+-tree achieves up to 8 million inserts per second because all insertions follow the same search path in cache. However, this case is highly constructed and unlikely to appear in practice because it inserts all elements in reverse sorted order.

Scans: Figure 9 and Table 7 show that once the data structures become sufficiently large, the PMA and SPMAs are on average 2.3× faster to scan and up to 5× faster to scan than the B+-tree due to the PMA's (and SPMAs') cache-friendliness. As shown in Section 4, the SPMAs match the PMA in terms of asymptotic scan performance. Although the PMA and SPMAs asymptotically match the B+-tree for scans, in practice, the PMA/SPMAs are much faster to scan because they store their data contiguously, whereas the B+-tree performs pointer indirections to traverse its nodes.

Range queries: Figure 10 and Table 8 show how the rate of elements processed varies with the size of the range in range queries. We report the expected elements processed (queries × expected number of elements per range query) rather than just the queries because Figure 5 already addresses the range queries per second. Just as in the YCSB range queries, the SPMAs outperform B+-trees by about 1.3× regardless of the size of the range, while PMAs are slower than B+-trees for small ranges and faster for large ranges because PMAs

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*	
Exp. elts per query	*Throughput*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1.5E0	2.66E7	7.57E7	2.9	9.78E7	3.7	1.05E8	4.0	6.75E7	2.5
6.0E0	2.60E7	6.94E7	2.7	9.08E7	3.5	9.93E7	3.8	6.63E7	2.6
2.4E1	2.49E7	5.85E7	2.3	7.26E7	2.9	7.15E7	2.9	5.23E7	2.1
9.5E1	2.13E7	3.84E7	1.8	4.26E7	2.0	3.88E7	1.8	3.19E7	1.5
3.8E2	1.34E7	1.72E7	1.3	1.80E7	1.3	1.63E7	1.2	1.28E7	1.0
1.5E3	5.05E6	5.27E6	1.0	5.33E6	1.1	5.25E6	1.0	3.95E6	0.8
6.1E3	1.47E6	1.43E6	1.0	1.44E6	1.0	1.43E6	1.0	1.07E6	0.7

Table 8: Throughput (in queries/second) of range queries as a function of expected elements returned. SU denotes speedup over the traditional PMA. The elements processed per second is just the query throughput multiplied by the expected elements per query.

	PMA	*SPMA-Linear*		*SPMA-Eytzinger*		*SPMA-Btree*		*B+-tree*
Elements		*Total*	*Heads*	*Total*	*Heads*	*Total*	*Heads*	
1E5	1.7E06	1.8E06	5.3E04	1.8E06	5.9E04	1.8E06	9.2E04	1.4E06
1E6	1.7E07	1.7E07	5.2E05	1.7E07	5.2E05	1.7E07	5.6E05	1.4E07
1E7	1.7E08	1.7E08	5.2E06	1.7E08	6.8E06	1.7E08	5.8E06	1.4E08
1E8	1.6E09	1.7E09	5.1E07	1.7E09	5.9E07	1.7E09	6.2E07	1.4E09
1E9	1.6E10	1.6E10	2.5E08	1.6E10	2.6E08	1.6E10	4.4E08	1.4E10

Table 9: The memory sizes of the different data structures in bytes. The SPMA sizes are divided into the total size and the leaf head array size. For reference, L1 is 1.6E6 bytes, L2 is 5.0E7 bytes and L3 is 7.5E7 bytes.

excel on scans. The throughput in terms of queries per second decreases with the size of the range because each query processes more elements. However, the elements processed per second increases with the size of the range because larger ranges involve larger scans and scans are much faster than searches (Figures 3 and 9).

Memory footprint: Figure 6 and Table 9 report the sizes of the PMA, SPMAs, and B+-trees. The SPMA variants have different leaf head array sizes due to padding, but the extra allocated space is never accessed by the searches because it exceeds the search range. Finally, the PMA and SPMAs take about 1.2× more space than the B+-tree because the PMA uses a constant factor of extra spaces to maintain its bounds, whereas the B+-tree uses an asymptotically small number of elements in the internal nodes and only one extra pointer per leaf.

6 Conclusion

This paper introduces the design, analysis, and implementation of several examples of search-optimized PMAs (SPMAs). These SPMAs use cache-friendly layouts to overcome the traditional tradeoff between updates and scans in PMAs when compared with other cache-friendly data structures such as B-trees. Specifically, SPMAs are faster than B+-trees for both queries and updates on unsorted inputs. On mixed query/insert workloads from YCSB, they are between $2 - 2.4\times$ faster compared to the TLX B+-tree. On uniform random inputs, SPMAs are on average between $1.3 - 2.3\times$ faster on all operations compared to the B+-tree. On the worst-case insertion distribution for the PMA, the SPMAs' insertion throughput is about 1.5× slower than the worst-case B+-tree's insertion throughput. However, the worst case for the PMA is a highly structured reverse sorted input, and the SPMAs match or exceed the B+-tree's insertion throughput when the input is up to 25% sorted.

Acknowledgments

This research is funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] *Static B-trees*. https://en.algorithmica.org/hpc/data-structures/s-tree/, 2022.

[2] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31 (1988), pp. 1116–1127.

[3] Amazon, *Amazon web services*. https://aws.amazon.com/, 2022.

[4] R. Bayer and E. M. McCreight, *Organization and maintenance of large ordered indexes*, Acta Informatica, 1 (1972), pp. 173–189.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton, *Cache-oblivious B-trees*, in Proceedings of the 41st Annual Symposium on Foundations of Computer Science, IEEE, 2000, pp. 399–409.

[6] ———, *Cache-oblivious B-trees*, SIAM Journal on Computing, 35 (2005), pp. 341–358.

[7] M. A. Bender, Z. Duan, J. Iacono, and J. Wu, *A locality-preserving cache-oblivious dynamic dictionary*, in Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, USA, 2002, Society for Industrial and Applied Mathematics, p. 29–38.

[8] M. A. Bender, Z. Duan, J. Iacono, and J. Wu, *A locality-preserving cache-oblivious dynamic dictionary*, Journal of Algorithms, 53 (2004), pp. 115–136.

[9] M. A. Bender, J. T. Fineman, S. Gilbert, T. Kopelowitz, and P. Montes, *File maintenance: when in doubt, change the layout!*, in Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2017, pp. 1503–1522.

[10] M. A. Bender and H. Hu, *An adaptive packed-memory array*, ACM Transactions on Database Systems (TODS), 32 (2007), pp. 26–es.

[11] T. Bingmann, *TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers*, 2018. https://panthema.net/tlx, retrieved Oct. 7, 2020.

[12] A. Clauset, C. R. Shalizi, and M. E. Newman, *Power-law distributions in empirical data*, SIAM review, 51 (2009), pp. 661–703.

[13] D. Comer, *Ubiquitous B-tree*, ACM Computing Surveys (CSUR), 11 (1979), pp. 121–137.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking cloud serving systems with YCSB*, in Proceedings of the 1st ACM symposium on Cloud computing, 2010, pp. 143–154.

[15] D. De Leo and P. Boncz, *Teseo and the analysis of structural dynamic graphs*, Proceedings of the VLDB Endowment, 14 (2021), pp. 1053–1066.

[16] M. Durand, B. Raffin, and F. Faure, *A packed memory array to keep moving particles sorted*, in 9th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS), The Eurographics Association, 2012, pp. 69–77.

[17] M. Eytzinger, *Thesaurus principum hac aetate in Europa viventium*, 1590, pp. 146–147.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-oblivious algorithms*, in FOCS, 1999, pp. 285–298.

[19] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.

[20] Intel Corporation, *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.

[21] A. Itai, A. G. Konheim, and M. Rodeh, *A sparse table implementation of priority queues*, in ICALP, 1981, pp. 417–431.

[22] D. W. Jones, *An empirical comparison of priority-queue and event-set implementations*, Commun. ACM, 29 (1986), p. 300–311.

[23] P.-V. Khuong and P. Morin, *Array layouts for comparison-based searching*, ACM J. Exp. Algorithmics, 22 (2017).

[24] A. LaMarca and R. Ladner, *The influence of caches on the performance of heaps*, ACM J. Exp. Algorithmics, 1 (1996), p. 4–es.

[25] C. Lattner, *LLVM: An Infrastructure for Multi-Stage Optimization*, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec. 2002. *See* http://llvm.cs.uiuc.edu.

[26] C. Lattner and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004, p. 75.

[27] K. J. Nesbit and J. E. Smith, *Data cache prefetching using a global history buffer*, in 10th International Symposium on High Performance Computer Architecture (HPCA'04), IEEE, 2004, pp. 96–96.

[28] M. E. Newman, *Power laws, Pareto distributions and Zipf's law*, Contemporary Physics, 46 (2005), pp. 323–351.

[29] R. Niewiadomski and J. N. Amaral, *Chopping up trees to improve spatial locality in implicit k-heaps*, Tech. Rep. TR-06-06, University of Alberta Department of Computer Science, 2006.

[30] P. Pandey, B. Wheatman, H. Xu, and A. n. Buluç, *Terrace: A hierarchical graph container for skewed dynamic graphs*, in SIGMOD, 2021, p. 1372–1385.

[31] J. R. C. Patterson, *Modern microprocessors: A 90 minute guide!* https://www.lighterra.com/papers/modernmicroprocessors/, 2016.

[32] V. Raman, *Locality preserving dictionaries: Theory & application to clustering in databases*, in Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '99, New York, NY, USA, 1999, Association for Computing Machinery, p. 337–345.

[33] T. B. Schardl, W. S. Moses, and C. E. Leiserson,

Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation, ACM Transactions on Parallel Computing (TOPC), 6 (2019), pp. 1–33.

[34] J. Toss, C. A. Pahins, B. Raffin, and J. L. Comba, *Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data*, Computers & Graphics, 76 (2018), pp. 117–128.

[35] B. Wheatman and R. Burns, *Streaming sparse graphs using efficient dynamic sets*, in 2021 IEEE International Conference on Big Data (BigData), IEEE, 2021, pp. 284–294.

[36] B. Wheatman and H. Xu, *A parallel packed memory array to store dynamic graphs*, in ALENEX, 2021, pp. 31–45.

[37] D. E. Willard, *Maintaining dense sequential files in a dynamic environment*, in Proceedings of the fourteenth annual ACM Symposium on Theory of Computing, 1982, pp. 114–121.

[38] D. E. Willard, *Good worst-case algorithms for inserting and deleting records in dense sequential files*, in Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, 1986, pp. 251–260.

[39] D. E. Willard, *A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time*, Information and Computation, 97 (1992), pp. 150–204.

[40] J. W. J. Williams, *Algorithm 232: Heapsort*, Communications of the ACM, 7 (1964), p. 347–348.

[41] YCSB, *Core workloads.* `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`, 2020.