POKER: Permutation-Based SIMD Execution of Intensive Tree Search by Path Encoding

FENG ZHANG and JINGLING XUE, UNSW Sydney, Australia

We introduce POKER, a permutation-based approach for vectorizing multiple queries over B^+ -trees. Our key insight is to combine vector loads and path-encoding-based permutations to alleviate memory latency while keeping the number of key comparisons needed for a query to a minimum. Implemented as a C++ template library, POKER represents a general-purpose solution for vectorizing the queries over indexing trees on multi-core processors equipped with SIMD units. For a set of five representative benchmarks evaluated with 24 configurations each, POKER outperforms the state of the art by 2.11x with one single thread and 2.28x with eight threads on an Intel Broadwell processor that supports 256-bit AVX2, on average. In addition, strip-mining queries will further improve POKER's performance by 1.21x (with one single thread) and 1.31x (with eight threads), on average.

CCS Concepts: • **Computer systems organization** → *Single instruction, multiple data*;

Additional Key Words and Phrases: SIMD, permute, intensive tree search

ACM Reference format:

Feng Zhang and Jingling Xue. 2018. POKER: Permutation-Based SIMD Execution of Intensive Tree Search by Path Encoding. *ACM Trans. Archit. Code Optim.* 15, 4, Article 46 (November 2018), 28 pages. <https://doi.org/10.1145/3280850>

1 INTRODUCTION

B^+ -trees [26, 30, 31], which include Binary Search Trees (BST) [15, 16, 29] as a special case, are popular indexing structures used in read-mostly Relational DataBase Management Systems (RDBMS), in, for example, decision support, data warehouse, data mining, and On-Line Analytical Processing (OLAP). In these applications, B^+ -trees are searched repeatedly by numerous independent queries but remain unchanged for an extensive period of time. For example, a decision support system that holds the franchise sale records of a retail chain may be updated once per month [27], and an OLAP system that stores the population and housing census records of a country is updated every 5 to 10 years [1].

Throughput is of great importance for RDBMS, making both CPUs and GPUs their possible computing platforms. However, as GPUs have limited memory capacity and relatively high host-device data transfer overheads [9, 32], CPUs are still a popular choice. With CPUs, a B^+ -tree can

Extension of Conference Paper [36].

This research is supported by ARC grants DP170103956 and DP180104069.

Authors' addresses: F. Zhang and J. Xue, Room 501, K-17, School of Computer Science and Engineering, UNSW Sydney, NSW 2052, Australia; emails: {fzhang, jingling}@cse.unsw.edu.au.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2018/11-ART46

<https://doi.org/10.1145/3280850>

reside entirely in main memory, eliminating the expensive disk I/O operations altogether [15, 26]. In addition, modern CPUs have multiple cores equipped with SIMD units, making it possible to exploit a large amount of DLP inherent in a large number of queries over in-memory B⁺-trees. As long as per-query response time (i.e., latency) is acceptable (for a particular RDBMS), improving the overall throughput also improves the average response time of each individual query.

However, vectorization of the queries in such an *Intensive Tree Search (ITS) problem* is often hampered due to the long latency caused by the irregular memory accesses incurred. Given a B⁺-tree, a query, which is represented by a (search) key, requests its corresponding record to be retrieved from the tree. Its core step is to compare the given search key with the sorted keys in a certain tree node by using binary search, which is optimal (in terms of key comparisons made) in the worst case [16]. However, this search process is not directly amenable to vectorization. As w queries, where w is the vector length element-wise, are handled in different vector lanes, their middle elements (accessed in the same iteration) are usually nonconsecutively stored in memory. The main challenge for harnessing the DLP in the w queries lies in how to obtain such middle elements from memory into vector registers to facilitate efficient vectorization [37, 38]. Unfortunately, the tree search process is too irregular to be expressed efficiently in a SIMD manner [12].

Two vectorization approaches exist for ITS. One conceivable but unpublished approach, denoted GATHER, is to make use of Gather instructions (available in modern SIMD units) to fetch the middle elements required in processing w queries. This simple solution results in the minimum number of key comparisons to be made for a query but at the expense of memory latency [10]. We use the number of key comparisons here as a metric because it correlates positively with the number of loads induced. On the other hand, FAST [15] and SEG-TREE [34] reduce Gather-induced memory access overhead by using vector loads but at the expense of introducing redundant key comparisons.

In this article, we introduce POKER, a permutation-based vectorization approach for tackling the ITS problem. We improve the prior work by combining vector loads and path-encoding-based permutations. This new insight enables both minimizing the number of key comparisons as in GATHER and reducing Gather-induced memory access overhead as in FAST [15] and SEG-TREE [34]. By reorganizing the sorted keys in a B⁺-tree node as a Perfect Binary Search Tree (PBST), POKER can fetch all the required middle elements for w queries in one vector load and then permute the right keys into the right lanes based on their path encodings. Compared with Gather instructions, Permute instructions are more efficient, enabling vector loads and register-to-register data reorganization based on a (dynamic) mask. In general, due to irregular memory accesses, finding the correct mask is akin to solving puzzles [10]. However, our path encoding scheme is instrumental in building the correct mask as desired.

This article makes the following contributions:

- We propose POKER, a permutation-based vectorization approach for addressing the ITS problem. This article is the first applying permutations to vectorize the queries over indexing trees.
- We make POKER available as a C++ template library, which provides a general-purpose solution for vectorizing the queries over indexing trees on modern CPUs.
- We introduce a strip mining optimization for POKER that can improve not only its overall throughput but also its per-query latency by vectorizing a given set of queries in batch mode.
- We evaluated POKER using a set of five representative benchmarks with 24 configurations, each against the state of the art. On average, POKER outperforms FAST [15] by 4.12x, SEG-TREE [34] by 3.55x, and GATHER by 2.11x in the sequential setting and GATHER (the previous best) by 2.28x with eight threads in the parallel setting. In addition, strip mining the queries

can further improve POKER's performance by 1.21x (with one single thread) and 1.31x (with eight threads), on average.

2 MOTIVATION

We first review a scalar algorithm for answering queries over B+-trees (Section 2.1). We then examine the challenges for its vectorization (Section 2.2). Finally, we introduce three existing vectorization techniques (Section 2.3) and our POKER solution (Section 2.4) by using an example.

2.1 A Scalar Algorithm for Querying B⁺-trees

A B⁺-tree of order b (known as its *branching factor*), which is a generalization of a BST, is a balanced b -way tree in which each non-leaf node has up to b children. For each non-leaf node with m keys (and thus $m + 1$ children), its i th key, where $0 \leq i < m$, is larger than the keys in its i th child's subtree and no larger than the keys in its $(i + 1)$ -th child's subtree. The root is either a leaf or has at least two children. Each internal node has between $\lceil b/2 \rceil$ and b children. The leaf nodes, which are at the same level, store between $\lceil b/2 \rceil - 1$ and $b - 1$ keys and pointers to these keys' records. The keys in each node are sorted in lexicographic order, facilitating binary search.

Algorithm 1 gives a scalar procedure for answering queries in a B⁺-tree T . A query q , represented by a search key, asks for its corresponding record to be found in T . The search starts from T 's root node and ends at a leaf where the record is found (if any). For each node n with its key count being $n.keyCnt$, a binary search is performed at the node in $\lfloor \log_2(n.keyCnt) \rfloor + 1$ iterations, without terminating early even when the search key is found, in order to facilitate its vectorization later. Therefore, max in Line 7 is needed but applies only when $i = 0$, in which case every q is smaller than all the keys at node n . Thus, q is nonexistent, captured by $q \neq n.keys[i_r]$.

BinSearch() performs two main tasks inside a node: (i) accessing a middle element (i.e., its key k (Line 13)), which can incur long memory latency due to irregular accesses and (ii) comparing query q with k (Line 14), which is computationally simple.

ALGORITHM 1: A Sequential Tree Search Algorithm

```

Procedure TreeSearch(Tree  $T$ , Array  $Q$ )
1  foreach query  $q \in Q$  do
2    return Visit( $T$ 's root node,  $q$ );

Procedure Visit(Node  $n$ , Query  $q$ )
3   $i \leftarrow \text{BinSearch}(n, q)$ ;
4  if  $n$  is an internal node then
5    return Visit( $n.children[i]$ ,  $q$ );
6  else
7     $i_r \leftarrow \max(i - 1, 0)$ ;
8    return  $q == n.keys[i_r] ? n.children[i_r] : \text{"not found"}$ ;

Procedure BinSearch(Node  $n$ , Query  $q$ )
9   $lo \leftarrow 0$ ;
10  $hi \leftarrow n.keyCnt$ ;
11  $m \leftarrow \frac{lo+hi}{2}$ ;
12 for  $i \leftarrow 1$  to  $\lfloor \log_2(n.keyCnt) \rfloor + 1$  by 1 do
13    $k \leftarrow n.keys[m]$ ;
14   if  $q < k$  then
15      $hi \leftarrow m$ ;
16   else
17      $lo \leftarrow m$ ;
18    $m \leftarrow \frac{lo+hi}{2}$ ;
19 return  $hi$ ;

```

Fig. 1. Answering queries sequentially on a B⁺-tree with a scalar algorithm given in Algorithm 1.

Figure 1 gives a running example to illustrate both the sequential and vectorized BinSearch() algorithms. For a student profile system in Figure 1(a), its index is maintained by a B⁺-tree in Figure 1(b). For the four search keys—5, 62, 32, and 50—queried for, their middle elements compared against at the root node (in all three iterations of BinSearch()) are shown in Figure 1(c). Thus, the children to be searched next for 5, 62, 32, and 50 are pointed to by P_0 , P_6 , P_3 , and P_5 , respectively.

2.2 Vectorization Challenges

Vectorization of BinSearch() for a node n is challenging even though w independent queries, $\vec{q} = [q_0, q_1, \dots, q_{w-1}]$, can be processed in parallel, where w (a power of 2) is the vector length element-wise. Ideally, $\frac{\lceil \log_2(n.keyCnt) \rceil + 1}{w}$ key comparisons are performed per query. In the first iteration, we start with the same middle element for the w queries (Line 11). However, in a subsequent iteration i , where $1 < i \leq \lceil \log_2(n.keyCnt) \rceil + 1$, the middle elements, $\vec{k} = [k_0^i, k_1^i, \dots, k_{w-1}^i]$, used (Line 13) are usually nonconsecutive in memory. The main challenge for vectorizing BinSearch() is how to obtain these keys efficiently in a SIMD manner.

We review the state of the art (Section 2.3) and describe the key idea behind POKER (Section 2.4). Figure 2 illustrates a total of four techniques (with g , f , s , and p standing for GATHER, FAST [15], SEG-TREE [34], and POKER) by showing how each vectorizes BinSearch() for the four queries, 5, 62, 32, and 59, at the root node shown in Figure 1. We assume that $w = 4$. For each technique $x \in \{g, f, s, p\}$, we depict its logical view of the root node's keys in Figure 2(x₁), its physical memory representation in Figure 2(x₂), how it vectorizes BinSearch() in Figure 2(x₃), and some statistics about its effectiveness in Figure 2(x₄). In Figure 2, the outcome of comparing a query's search key with a middle element is 0 if the former is smaller and 1 otherwise.

In all the four techniques except GATHER, some dummy keys, with the value $\boxed{M}$, may be appended to the list of sorted keys in a B⁺-tree node so that the final list represents a perfect balanced search tree. Here, $\boxed{M}$ denotes the lexicographically maximum key. If a query uses $\boxed{M}$ as a search key, we can simply return its record (if any) directly without performing a binary search. Without loss of generality, we assume that the search keys queried are always less than $\boxed{M}$.

2.3 Existing Solutions

The prior work on vectorizing BinSearch() in a B⁺-tree node suffers from long memory latency, as in GATHER, or many redundant key comparisons, as in FAST [15] and SEG-TREE [34].

2.3.1 GATHER. GATHER exploits the DLP in w queries at a time by operating directly on the existing keys in a B⁺-tree node, without requiring any data reorganization (Figure 2(g₁) and (g₂)).

As shown in Figure 2(g₃), GATHER handles all the w queries, which are fetched with one vector load (VLD), in parallel in a SIMD manner. In the i th iteration of BinSearch(), the w middle elements required, $\vec{k} = [k_0^i, k_1^i, \dots, k_{w-1}^i]$, are obtained with one single gather instruction and compared

Fig. 2. Vectorization of BinSearch() for the four queries processed at the root shown in Figure 1 ($w = 4$).

with the w queries (CMP). Despite recent advancement in SIMD architectures, a gather instruction in Intel's AVX2 still causes what is equivalent to one (LD) in each lane [2].

For the four queries, we have VLD : 1, LD : 12 and CMP : 3 (Figure 2(g₄)). By using Gather instructions, the number of key comparisons performed at a node n per query, $\frac{\lceil \log_2(n.keyCnt) \rceil + 1}{w}$, is the smallest. However, the overall performance is hampered by such long-latency instructions.

2.3.2 FAST. Unlike GATHER, FAST [15] exploits only the DLP within a query by vectorizing the comparisons of its search key with the $w - 1$ middle elements across $\log_2(w)$ iterations of BinSearch(). Note that w is a power of 2. Logically, FAST views the list of sorted keys in a node n as a PBST (Figure 2(f₁)) by adding padding [M]'s, if necessary. Physically, FAST blocks the PBST with triangular SIMD tiles of size $w - 1$ each and stores the tiles (the keys in a tile) consecutively in breadth-first order (Figure 2(f₂)).

For a given query, as shown in Figure 2(f₃), its search key is initially broadcast (BRD) to a vector register and compared (CMP) with the $w - 1$ middle elements (in the first $\log_2(w)$ iterations of BinSearch()) loaded in a vector register (VLD) from the root tile. Then, the same key comparison process is repeated in appropriate child tiles (for later iterations of BinSearch()).

In general, a query for a node n undergoes $\lceil \frac{\log_2(n.keyCnt) + 1}{\log_2 w} \rceil$ key comparisons. Unlike GATHER, however, FAST introduces many redundant comparisons totaling $w \times \lceil \frac{\log_2(n.keyCnt) + 1}{\log_2 w} \rceil - (\lceil \log_2(n.keyCnt) \rceil + 1)$ (for GATHER) per w queries. For the four queries in our example, we have

: 4, : 8, and : 8 (Figure 2(f₄)). Compared with GATHER, FAST introduces a total of five redundant key comparisons in this running example.

2.3.3 SEG-TREE. Like FAST [15], SEG-TREE [34] also vectorizes BinSearch() for a single query. Unlike FAST, however, SEG-TREE uses a $(w + 1)$ -ary search tree (Figure 2(s₁) and (s₂)), where the w keys in one node are consecutively stored in memory to reduce the number of redundant key comparisons. Figure 2(s₃) can be understood similarly as Figure 2(f₃).

In general, a query at a node n requires $\lfloor \log_{w+1}(n.keyCnt) \rfloor + 1$ key comparisons. There are still many redundant comparisons, totaling $w \times (\lfloor \log_{w+1}(n.keyCnt) \rfloor + 1) - (\lfloor \log_2(n.keyCnt) \rfloor + 1)$ per w queries. For the four queries, we have : 4, : 8, and : 8 (Figure 2(s₄)). In this example, SEG-TREE also introduces a total of five redundant key comparisons.

2.4 Our POKER Solution

POKER improves the prior work in a novel way. Like GATHER, POKER processes w queries in parallel, thereby keeping the number of key comparisons to a minimum (without performing redundant ones). Like FAST and SEG-TREE, POKER avoids expensive Gather instructions by using vector loads. Our key novelty is to achieve the best of both worlds via path-encoding-based permutations.

For a B^+ -tree node, we vectorize BinSearch() by reorganizing its keys also into a PBST by adding pads, if needed, like FAST. Our key insight is that, in such a PBST, its level i (starting from 0) contains all the middle elements encountered in the $(i + 1)$ -th iteration of BinSearch() (starting from 1). If the w middle elements required by w queries are available in a vector register but not in a format expected, we can dynamically encode their search paths and use these path encodings to generate a mask to obtain the format as expected via register-to-register data reorganization.

In this example, Figure 2(p₁) gives our logical view of the root node shown in Figure 1 as a PBST. To ensure consecutive memory accesses, POKER blocks the PBST into SIMD tiles of size $2w - 1$ (instead of $w - 1$ in FAST) and then stores these tiles (the keys in a tile) in breadth-first order (Figure 2(p₂)). For ease of illustration, the root node in this example happens to have just one tile but can have more, as is the case of FAST (Figure 2(f₁)).

Given a SIMD tile, its path encodings are simple. For each node, its left and right edges are encoded as 0 and 1. The path from its root to a node is a concatenation of its edge encodings. For the tile in Figure 2(p₁), the path $40 \rightarrow 60 \rightarrow 50$ is encoded as the binary 10 (i.e., the decimal 2), which identifies the key 50 at the third node at level 2.

POKER, as described in Section 3, will completely process all the queries at a node (tile) before it moves to its child nodes (tiles). To explain the key idea behind POKER, it suffices to focus on how POKER handles w queries at a particular B^+ -tree node. POKER processes w queries in parallel, descending from the root tile to the leaf tiles on its tiled PBST.

In Figure 2(p₃), the PBST has just one tile. The four queries are loaded into a vector register $\vec{q} = [5, 62, 32, 59]$ ( ). At level 0, POKER starts with the path vector $\vec{c} = [-, -, -, -]$ and the permutation mask $\vec{m} = [0, 0, 0, 0]$. The singleton middle element at level 0, together with three irrelevant keys, is loaded ( ): $\vec{k}_0 = [40, \boxed{X}, \boxed{X}, \boxed{X}]$. According to $\vec{m}$, $\vec{k}_0$ is permuted ( ) to obtain $\vec{k} = [40, 40, 40, 40]$, which is then compared with $\vec{q}$ ( ). The outcome of is reflected in the path vector: $\vec{c} = [0, 1, 0, 1]$. Its decimal form $\vec{m} = [0, 1, 0, 1]$ yields the permutation mask for level 1. At level 1, the two middle elements are loaded ( ): $\vec{k}_1 = [20, 60, \boxed{X}, \boxed{X}]$, then permuted ( ) according to $\vec{m}$ to obtain $\vec{k} = [20, 60, 20, 60]$, and finally compared with $\vec{q}$ ( ). This process is repeated until level 2 is done. After this, every query will be directed to a child tile in the current node (or to a child node if the current tile is a leaf tile) according to $\vec{m}$. In this example, the node

being searched has only one SIMD tile with $\vec{m} = (0, 6, 3, 5)$. Thus, queries 5, 62, 32, and 59 will flow to the child nodes pointed to by P_0, P_6, P_3 , and P_5 , respectively.

POKER differs from GATHER in how they fetch the middle elements required by `BinSearch()`. Thus, POKER also attains the minimum number of key comparisons, $\frac{\lfloor \log_2(n.keyCnt) \rfloor + 1}{w}$, for a node n per query. However, POKER will be more efficient by avoiding gather instructions. For the four queries, we have **VLD**: 4, **PER**: 3, and **CMP**: 3 (Figure 2(p₄)).

3 APPROACH

We describe our POKER approach now. We first describe POKER (Algorithm 2) by discussing its data layout transformation (Section 3.1), its vectorized tree search (Section 3.2), and its batch processing of all the queries (Section 3.3). We then analyze its search efficiency (Section 3.4). Finally, we generalize POKER as a generic solution in terms of a C++ template (Section 3.5).

ALGORITHM 2: POKER

```

Procedure POKER (QuerySet  $Q$ , Tree  $T$ , Array  $R$ )
1   $\mathcal{T} \leftarrow \text{Transform}(T)$ ;
2   $c \leftarrow \text{CompBatchCnt}(Q, T, \mathcal{T}, R)$ ;
3  for  $c_i \leftarrow 0$  to  $c - 1$  by 1 do
4     $b_i \leftarrow c_i < |Q| \bmod c ? \lceil \frac{|Q|}{c} \rceil : \lfloor \frac{|Q|}{c} \rfloor$ ;
5     $Q \leftarrow b_i$  queries removed from  $Q$ ;
6     $\Gamma \leftarrow \Gamma \cup \text{SIMDSearch}((0, Q), \mathcal{T}, R)$ ;
7 return  $\Gamma$ ;

```

As shown in Algorithm 2, POKER takes as input a set Q of queries, a B^+ -tree T , and an array R of its actual records. Let K be the set of the keys in T . POKER produces as output a set of triplets:

$$\Gamma = \{(Q', k, r) \mid Q' \subseteq Q, k \in K, r \in R\}, \quad (1)$$

which indicates that, for each query $q \in Q'$, its record is found to be r if $q = k$. Otherwise, q is nonexistent in T . As discussed in Section 2, $\boxed{M}$ represents the lexicographically maximum key and w stands for the vector length element-wise.

DLP and TLP are orthogonal. In the case of multiple threads, the queries in Q will be evenly partitioned (sharing the same B^+ -tree), with the queries in different threads vectorized in exactly the same way. Therefore, we will only need to discuss how to vectorize Q for a single thread. To improve the overall throughput (i.e., the overall time for processing all the queries) and, consequently, per-query latency, the queries in Q are divided into c batches, with $\lceil \frac{|Q|}{c} \rceil$ queries in each of the first $(|Q| \bmod c)$ batches and $\lfloor \frac{|Q|}{c} \rfloor$ in each remaining batch. We will discuss how to select c heuristically in Section 3.3. To vectorize `SIMDSearch()`, it suffices to focus only on a single batch, as different batches are handled independently among different threads (if any).

3.1 Transform(): Transforming Data Layout

In Algorithm 3, we perform the data layout transformation required for a tree, as shown in Figure 3, to facilitate vectorization of `BinSearch()` in Algorithm 1. Given a B^+ -tree T , we store its nodes in breadth-first order in an array $\mathcal{T}$ with each original node represented as a *pNode* object (Line 1), with its six fields given in Figure 3. For a node $\mathcal{T}[i]$, its children are thus consecutively stored in $\mathcal{T}$. Only the index of its first child, which is a node if $\mathcal{T}[i]$ is a non-leaf and a record otherwise, is maintained explicitly (Line 3). When vectorized, `BinSearch()` does not check if a queried key is

Fig. 3. Data reorganization of a B⁺-tree T as an array $\mathcal{T}$ with each node in T represented by a six-field $pNode$ object.

ALGORITHM 3: Tree Transformation

Procedure Transform(*Tree* T)

- 1 Let $\mathcal{T}$ be an array of the nodes in T stored in breadth-first order, with each represented as a $pNode$ object (Figure 3);
- 2 **foreach** $pNode$ n at index i (starting from 0) in $\mathcal{T}$ **do**
- 3 $\mathcal{T}[i].fstcl \leftarrow$ index of the first child of $\mathcal{T}[i]$ in $\mathcal{T}$ (or R);
- 4 **if** n is a leaf node in T **then**
- 5 $\mathcal{T}[i].OKeys \leftarrow$ an array of n 's sorted keys;
- 6 Let $\mathcal{T}[i].RKeys$ be an array of n 's sorted keys;
- 7 Pad $\mathcal{T}[i].RKeys$, if needed, with $\boxed{M}$, s.t. it represents a PBST;
- 8 Relayout $\mathcal{T}[i].RKeys$ by blocking its PBST with SIMD tiles of size $2w - 1$ (starting from its root), and store these tiles (and the keys within a tile) in breadth-first order;
- 9 **return** $\mathcal{T}$;

found explicitly. For a leaf $\mathcal{T}[i]$, we use $\mathcal{T}[i].OKey$ to keep the list of its sorted keys in order to set up Γ given in Equation (1) (Lines 4 and 5) to enable a later check.

For each node $\mathcal{T}[i]$, we transform its $\mathcal{T}[i].RKeys$ into a PBST, by adding some $\boxed{M}$ pads, if necessary (Lines 6–8). The number of pads added is always the smallest such that its tree height h satisfies $h = \lfloor \log_2(\mathcal{T}[i].RKeys.size()) \rfloor + 1$ (Line 7). Now, $\mathcal{T}[i].RKeys$ represents a PBST as an array (indexed from 0). The key with index $2^{h-1} - 1$ is the root at level 0. For a non-leaf key with index i at level l , its two child keys reside at indices $i - 2^{h-l-2}$ and $i + 2^{h-l-2}$, respectively. Finally, the PBST is triangularly tiled with a size of $2w - 1$ (Line 8), as shown in Figure 2(p₁) and (p₂) for $w = 4$ (and in Figure 2(f₁) and (f₂) if $w = 2$ instead). Finally, a tiled PBST is always a perfect $2w$ -ary tree with its height given by $\lceil \frac{h}{\lceil \log_2(2w-1) \rceil} \rceil$.

All the four SIMD techniques—GATHER (for BSTs only), FAST, SEG-TREE (for B⁺-trees only), and POKER—may consume some extra spaces due to the data layout transformations required. In Section 4, we will experimentally compare and analyze their space overheads.

3.2 SIMDSearch(): Vectorizing Binary Search

To vectorize BinSearch() on $\mathcal{T}$, we aim to maximize both SIMD utilization and data reuse. To this end, the queries that visit the same tree node or SIMD tile are queued and processed together. Vectorization of BinSearch() happens only within a SIMD tile for w queries at a time.

Algorithm 4 consists of three procedures. We define a task as a pair (idx, Q_{idx}) , where Q_{idx} is a set of queries that will visit a tree node (record) or a SIMD tile whose absolute index in $\mathcal{T}$ can be derived from idx (directly or indirectly). In addition, Φ_n (Φ_t) is a set of such tasks that will visit different nodes (SIMD tiles). These three procedures are described below.

- SIMDSearch() visits the tree nodes in $\mathcal{T}$ (with its records in R) recursively, starting from the root, and processes all the queries that visit a node at the same time. To prepare for the

ALGORITHM 4: SIMD Tree Search

```

Procedure SIMDSearch(Task  $\phi$ , Array  $\mathcal{T}$ , Array  $R$ )
1   $n \leftarrow \mathcal{T}[\phi.idx]$ ;
2   $\Phi_n \leftarrow \text{VisitNode}((0, \phi.Q), n, 0)$ ;
3  foreach Task  $\phi' \in \Phi_n$  do
4    if  $n.IsLeaf$  then
5       $i_r \leftarrow \max(\phi'.idx - 1, 0)$ ; // index into actual record
6       $\Gamma \leftarrow \Gamma \cup \{(\phi'.Q, n.OKeys[i_r], R[n.fstcl + i_r])\}$ ;
7    else
8       $\phi'.idx \leftarrow \phi'.idx + n.fstcl$ ;
9      SIMDSearch( $\phi'$ ,  $\mathcal{T}$ ,  $R$ );
10 return  $\Gamma$ ;

Procedure VisitNode(Task  $\phi$ , pNode  $n$ , int  $level$ )
11  $\Phi_t \leftarrow \text{VectorizeTile}(\phi, n, level)$ ;
12 if  $level == h_{tiled\_PBST} - 1$  then
13   return  $\Phi_t$ ;
14 else
15   foreach Task  $\phi' \in \Phi_t$  do
16      $\Phi_n \leftarrow \Phi_n \cup \text{VisitNode}(\phi', n, level + 1)$ ;
17   return  $\Phi_n$ ;

Procedure VectorizeTile(Task  $\phi$ , pNode  $n$ , int  $level$ )
18  $h \leftarrow \log_2(n.RKeys.size() + 1)$ ; // height of  $n$ 's PBST
19  $h_{tiled\_PBST} \leftarrow \lceil \frac{h}{\lceil \log_2(2w-1) \rceil} \rceil$ ; // height of  $n$ 's tiled PBST
20  $h_{tile} \leftarrow level < h_{tiled\_PBST} - 1 ? n.c_{in} : n.c_{lf}$  // height of a tile
21 Let  $t_{idx}$  be the index of the first key in the current tile:
    $t_{idx} \leftarrow \frac{(2 \times w)^{level-1}}{(2 \times w) - 1} \times (2^{n.c_{in}} - 1) + \phi.idx \times (2^{h_{tile}} - 1)$ ;
22  $p \leftarrow 0$ ;
23 Let  $V[0 : h_{tile} - 1]$  be an array of vector registers;
24 for  $j \leftarrow 0$  to  $h_{tile} - 1$  by 1 do
25    $V[j] \leftarrow n.RKeys[t_{idx} + p : t_{idx} + p + w - 1]$ ;
26    $p \leftarrow p + 2^j$ ;
27 while  $\phi.Q \neq \emptyset$  do
28    $\vec{q} \leftarrow w$  queries removed from  $\phi.Q$ ;
29    $\vec{m} \leftarrow \vec{0}$ ;
30   for  $j_c \leftarrow 0$  to  $h_{tile} - 1$  by 1 do
31      $\vec{k} \leftarrow \text{Permute}(V[j_c], \vec{m})$ ;
32      $\vec{r} \leftarrow \text{VecCompare}(\vec{q}, \vec{k})$ ;
33      $\vec{m} \leftarrow \vec{m} + \vec{m} + \vec{r}$ ;
34   for  $j_m \leftarrow 0$  to  $w - 1$  by 1 do
35      $\Phi_t \leftarrow \Phi_t \cup \{(\phi.idx \times 2^{h_{tile}} + \vec{m}[j_m], \{\vec{q}[j_m]\})\}$ ;
36 return  $\Phi_t$ ;

```

- visit at a node n (Line 1), $\phi = (idx, Q_{idx})$ is the task to be processed at n , where idx is its (absolute) index in $\mathcal{T}$ and Q_{idx} is the set of queries visiting the node. Initially, $\phi = (0, Q)$, passed from POKER in Algorithm 2, where Q is the set of queries to be processed in the current batch.
- VisitNode() recursively visits the tiles in the tiled PBST at a node n , starting from its root tile, and processes all the queries at a tile at the same time. When called to visit a tile, three input arguments are passed in: (i) n is the node containing the tile, (ii) $level$ is the number of levels in the tiled PBST at n , and (iii) $\phi = (idx, Q_{idx})$, where idx is a relative index identifying the tile among all the tiles at $level$ in the tiled PBST at n , counted from left to right, and Q_{idx} is the set of queries to be processed at the tile. Initially, the task at the root tile of n is set up as $(0, \phi.Q)$ at Line 2.
 - VectorizeTile() vectorizes BinSearch() by processing all the queries, w at a time, that can possibly reach a tile. In Lines 18–21, we find the absolute index t_{idx} for the first key in the tile in $\mathcal{T}$ based on the relative index $\phi.idx$. In Lines 22–35, we vectorize all the queries in $\phi.Q$, w at a time (as illustrated already in Figure 2(p₁)–(p₃)). Two comments are in order about adding a new task $(\phi.idx \times 2^{h_{tile}} + \vec{m}[j_m], \{\vec{q}[j_m]\})$ to Φ_t to be sent to the child tiles (if any) of the current tile at Line 35. First, its relative index at its own level in the tiled PBST at node n is again used. Second, for a set Φ_t of tasks (idx, Q_{idx}) to be sent to the child tiles of the current tile, Q_{idx} is implemented as a vector, (idx, Q_{idx}) as a struct, and Φ_t as an array of such structs. It is understood that if Φ_t contains two tasks (idx_1, Q_{idx_1}) and (idx_2, Q_{idx_2}) such that $idx_1 = idx_2$, then both tasks will be conceptually merged into one $(idx_1, Q_{idx_1} \cup Q_{idx_2})$. In our actual implementation, such task merging never takes place as a query corresponding to idx is simply inserted into Q_{idx} . Finally, the time complexity for merging $|\phi.Q|$ queries at a tile is $O(|\phi.Q|)$.

3.3 CompBatchCnt(): Strip-Mining Queries

As an optimization, strip mining queries enables their vectorization in batch mode. In our POKER approach (Line 2 in Algorithm 2), $c \leftarrow \text{CompBatchCnt}(Q, T, \mathcal{T}, R)$ computes the number of batches, c , used for a set Q of queries over a transformed B^+ -tree $\mathcal{T}$ from a given B^+ -tree T (Figure 3). Now we discuss how to find c to improve POKER's overall throughput (i.e., the total time for processing the queries in Q) and, consequently, the average response time of a query.

Our approach for finding c relies on two empirical observations. First, POKER's throughput correlates negatively with both the number of Minor Page Faults (MPFs) and the number of Last-Level Cache (LLC) misses incurred in accessing the data structures used for storing Q , $\mathcal{T}$, and R . Second, improving POKER's throughput entails a tradeoff between these two compromise factors. As c increases (i.e., the number of queries per batch decreases), the number of MPFs tends to decrease, but the number of LLC misses tends to increase. Therefore, we will find c so that the overall latency that suffers from handling both MPFs and LLC misses is minimized.

This is accomplished in Algorithm 5, which makes use of the following notations. For a given B^+ -tree T , we write h for its height and b for its branching factor. In its transformed B^+ -tree $\mathcal{T}$, POKER reorganizes the sorted keys in a B^+ -tree node as a PBST and then performs its vectorization on a tiled PBST (Figure 3). Thus, $h_{\text{PBST}} = \lfloor \log_2(b-1) \rfloor + 1$ gives rise to the maximum height of such a PBST and $h_{\text{tiled_PBST}} = \lceil \frac{h_{\text{PBST}}}{\lfloor \log_2(2 \times w - 1) \rfloor} \rceil$ the maximum height of its tiled PBST, where w is the vector length element-wise (as before). Let s_q be the size of a query (key) (in bytes), $s_{\mathcal{T}}$ the size of $\mathcal{T}$ (in bytes), and s_R the size of all the records R (in bytes). Let s_{page} be the size of a memory page (in bytes) and s_{line} the size of an LLC cache line (in bytes). Finally, we write l_{mpf} for the latency of a single MPF and l_{llc} the latency of a single LLC miss.

ALGORITHM 5: Computing Number of Batches Required

```

Procedure CompBatchCnt(QuerySet  $Q$ , Tree  $T$ , Array  $\mathcal{T}$ , Array  $R$ )
1   $L(x) \leftarrow \text{BuildCostFunc}(Q, T, \mathcal{T}, R)$ ;
2   $c \leftarrow \text{FindOptBatchCnt}(L(x))$ ;
3  return  $c$ ;

Procedure BuildCostFunc(QuerySet  $Q$ , Tree  $T$ , Array  $\mathcal{T}$ , Array  $R$ )
4  return  $L(x)$  defined in (2);

Procedure FindOptBatchCnt(Function  $L(x)$ )
5  Let  $D(\widehat{L})$  in (7) be defined, where  $\widehat{L} = L_{\text{ceil}}(x_{\min})$ ;
6  if  $\exists p \in D(\widehat{L})$  such that  $L(p) \leq \widehat{L}$  then
7    Let  $c$  be a minimum point found for  $L(x)$  in  $D(\widehat{L})$  by an exhaustive search (Theorem 3.3);
8  else
9    Extend  $D(\widehat{L})$  to  $D(\widehat{L}')$  iteratively (Theorem 3.4) until a minimum point  $c$  for  $L(x)$  in  $D(\widehat{L}')$  is found
    (Theorem 3.3);
10 return  $c$ ;

```

Algorithm 5 finds c in two stages, $\text{BuildCostFunc}()$, which builds a cost function related to MPFs and LLC misses, and $\text{FindOptBatchCnt}()$, which finds c by minimizing the cost function.

3.3.1 BuildCostFunc(): Building a Cost Function. To improve POKER's throughput, we will find a batch count $x = c$ that minimizes $L(x)$ (expressed as a function of the batch count x used):

$$L(x) = L_{\text{mpf}}(x) + L_{\text{llc}}(x), \quad (2)$$

where $L_{\text{mpf}}(x)$ ($L_{\text{llc}}(x)$) represents the time spent in handling the MPFs (LLC misses) incurred. As x represents the number of batches, the size of a batch is given by $\lceil \frac{|Q|}{x} \rceil$.

We derive the closed-form formulas for $L_{\text{mpf}}(x)$ and $L_{\text{llc}}(x)$ based on the data structures used:

- Φ_n and Φ_t for Storing the Queries in Q . As these data structures are shared by all the batches, it suffices to analyze their total size needed for processing one batch in Algorithm 4. For a given batch, its $\lceil \frac{|Q|}{x} \rceil$ queries are reorganized and stored at each recursive instance of Φ_n at every level of $\mathcal{T}$. Similarly, these queries are also reorganized and stored at each recursive instance of Φ_t at every level of a tiled PBST at every tree node in $\mathcal{T}$ (with Φ_t shared by all the tree nodes in $\mathcal{T}$). Thus, the space required by Φ_n and Φ_t is given by $\lceil \frac{|Q|}{x} \rceil \times s_q \times (h + h_{\text{tiled_PBST}})$ (in bytes).
- $\mathcal{T}$ and R for Storing the B^+ -Tree and Its Records. The space required is $s_{\mathcal{T}} + s_R$ (in bytes).

Given these space requirements, $L_{\text{mpf}}(x)$ can be approximated as follows:

$$L_{\text{mpf}}(x) = l_{\text{mpf}} \times \left(\left\lceil \frac{\lceil \frac{|Q|}{x} \rceil \times s_q \times (h + h_{\text{tiled_PBST}})}{s_{\text{page}}} \right\rceil + \left\lceil \frac{s_{\mathcal{T}} + s_R}{s_{\text{page}}} \right\rceil \right). \quad (3)$$

Inside $()$, the first (second) term gives the number of MPFs incurred in accessing Φ_n and Φ_t ($\mathcal{T}$ and R). Note that only in-memory B^+ -trees are considered (Section 1), and Φ_n and Φ_t are implemented as in-memory arrays. A memory page is understood to trigger only one MPF during its first access.

As a worst-case analysis, $L_{\text{llc}}(x)$ can be approximated as follows:

$$L_{\text{llc}}(x) = l_{\text{llc}} \times \left(\frac{|Q| \times s_q \times (h + h_{\text{tiled_PBST}}) + x \times (s_{\mathcal{T}} + s_R)}{s_{\text{line}}} \right). \quad (4)$$

Inside $()$, the first (second) term gives the number of LLC misses incurred in accessing Φ_n and Φ_t ($\mathcal{T}$ and R). Φ_n (Φ_t), which is shared by the x batches (all the tree nodes), enjoys no cache reuse across the batches (tree nodes). Similarly, $\mathcal{T}$ and R enjoy no cache reuse across the x batches.

3.3.2 FindOptBatchCnt(): Finding the Optimal Batch Count. We discuss how to find an optimal batch count $x = c$ such that $L(x)$ in Equation (2) is minimized. Let $L_{\text{ceil}}(x)$ ($L_{\text{floor}}(x)$) be derived from $L(x)$ such that each expression $\lceil e \rceil$ in $L(x)$ is replaced by $e + 1$ (e). For every $x \in (0, \infty)$, we have:

$$L_{\text{floor}}(x) \leq L(x) < L_{\text{ceil}}(x). \quad (5)$$

Both $L_{\text{ceil}}(x)$ and $L_{\text{floor}}(x)$ attain their minimums at the same minimum point $x_{\min}$ found from:

$$L_{\text{ceil}}(x_{\min})' = L_{\text{floor}}(x_{\min})' = 0. \quad (6)$$

The following lemma is straightforward. The same property holds for $L_{\text{ceil}}(x)$ but is not used.

LEMMA 3.1. $L_{\text{floor}}(x)$ is a continuous function in the domain $(0, \infty)$. In addition, $L_{\text{floor}}(x)$ is monotonically decreasing in $(0, x_{\min}]$ and monotonically increasing in $[x_{\min}, \infty)$.

Let $\widehat{L} \geq L_{\text{ceil}}(x_{\min})$ be fixed but arbitrary. Let $x_1^{\widehat{L}}$ and $x_2^{\widehat{L}}$ be the two solutions to the quadratic equation $L_{\text{floor}}(x) = \widehat{L}$. Based on Lemma 3.1 and the fact that $\widehat{L} \geq L_{\text{ceil}}(x_{\min}) > L_{\text{floor}}(x_{\min})$, we must have $x_1^{\widehat{L}} \neq x_2^{\widehat{L}}$. Let us assume that $x_1^{\widehat{L}} < x_2^{\widehat{L}}$. Then $[x_1^{\widehat{L}}, x_2^{\widehat{L}}]$ gives us an initial interval to search for a minimum point for $L(x)$. Before then, the following lemma shows that $x_{\min}$ lies in this interval.

LEMMA 3.2. $x_1^{\widehat{L}} < x_{\min} < x_2^{\widehat{L}}$.

PROOF. Suppose that $x_{\min} \notin (x_1^{\widehat{L}}, x_2^{\widehat{L}})$ holds. Without loss of generality, let us assume that $x_{\min} \leq x_1^{\widehat{L}} < x_2^{\widehat{L}}$. By Lemma 3.1, $L_{\text{floor}}(x)$ is monotonically increasing in $[x_{\min}, \infty)$. Then $L_{\text{floor}}(x_1^{\widehat{L}}) < L_{\text{floor}}(x_2^{\widehat{L}})$. As $x_1^{\widehat{L}}$ and $x_2^{\widehat{L}}$ are two solutions to $L_{\text{floor}}(x) = \widehat{L}$, then $L_{\text{floor}}(x_1^{\widehat{L}}) = L_{\text{floor}}(x_2^{\widehat{L}})$. A contradiction has been reached. Thus, $x_{\min} \in (x_1^{\widehat{L}}, x_2^{\widehat{L}})$, i.e., $x_1^{\widehat{L}} < x_{\min} < x_2^{\widehat{L}}$ has been established. $\square$

To find a minimum point where $L(x)$ attains its minimum, we start initially with the subdomain:

$$D(\widehat{L}) = \{x \in \mathbb{Z}^+ \mid x_1^{\widehat{L}} \leq x \leq x_2^{\widehat{L}}\}. \quad (7)$$

We then distinguish two cases. If $D(\widehat{L})$ contains some integer point p such that $L(p) \leq \widehat{L}$, then $L(x)$ must attain its minimum at one of the integer points in $D(\widehat{L})$ (Theorem 3.3). Otherwise, we extend this subdomain iteratively so that Theorem 3.3 applies in an extended subdomain (Theorem 3.4).

THEOREM 3.3. If $\exists p \in D(\widehat{L})$ such that $L(p) \leq \widehat{L}$, then $L(x)$ attains its minimum in $D(\widehat{L})$.

PROOF. Suppose that $L(x)$ attains its minimum at $p' \notin X$. Then $L(p') \leq L(p) \leq \widehat{L}$ holds. Let us consider the case when $p' < x_1^{\widehat{L}}$. The same conclusion can be obtained similarly for the case when $p' > x_2^{\widehat{L}}$ holds. By Lemma 3.1, $L_{\text{floor}}(x)$ is monotonically decreasing in $(0, x_{\min}]$. By Lemma 3.2, $x_1^{\widehat{L}} < x_{\min}$ holds. Thus, $L(p') \geq L_{\text{floor}}(p') > L_{\text{floor}}(x_1^{\widehat{L}}) = \widehat{L}$. As $L(p') \leq \widehat{L}$ and $L(p') > \widehat{L}$ cannot hold simultaneously, a contradiction has been reached. Thus, $L(x)$ must attain its minimum in $D(\widehat{L})$. $\square$

THEOREM 3.4. Suppose that $\nexists p \in D(\widehat{L})$ such that $L(p) \leq \widehat{L}$. Let $x_1^{\widehat{L}'}$ and $x_2^{\widehat{L}'}$ be the two solutions to the quadratic equation $L_{\text{floor}}(x) = \widehat{L}'$ such that $x_1^{\widehat{L}'} < x_2^{\widehat{L}'}$, where $\widehat{L}' = \max(L_{\text{ceil}}(x_1^{\widehat{L}}), L_{\text{ceil}}(x_2^{\widehat{L}}))$. Let $D(\widehat{L}') = \{x \in \mathbb{Z}^+ \mid x_1^{\widehat{L}'} \leq x \leq x_2^{\widehat{L}'}\}$. Then $\widehat{L} < \widehat{L}'$ and $D(\widehat{L}) \subset D(\widehat{L}')$.

Table 1. Instructions and Their Latencies for Loading Eight 32-bit Ints on a 256-bit AVX2 by GATHER and POKER

Operation	Instruction	Latency [2]	#Instructions	
			GATHER	POKER
vector load	vmoaddqu	1	0	1
add	vpaddd	1	1	1
shift left	vpslld	1	0	1
permute	vpermd	3	0	1
blend	vpblendvb	2	2	0
shift right	vpshrld	1	1	0
gather	vpgatherdd	~19	1	0
Total Latency			~25	6

PROOF. Without loss of generality, we assume that $\widehat{L}' = \max(L_{\text{ceil}}(x_1^{\widehat{L}}), L_{\text{ceil}}(x_2^{\widehat{L}})) = L_{\text{ceil}}(x_1^{\widehat{L}})$. Then we have $\widehat{L} = L_{\text{floor}}(x_1^{\widehat{L}}) < L_{\text{ceil}}(x_1^{\widehat{L}}) = \widehat{L}'$. Thus, $\widehat{L} < \widehat{L}'$ has been established.

To prove that $D(\widehat{L}) \subset D(\widehat{L}')$ holds, we prove that both $x_1^{\widehat{L}'} < x_1^{\widehat{L}}$ and $x_2^{\widehat{L}'} > x_2^{\widehat{L}}$ hold. Let us prove $x_1^{\widehat{L}'} < x_1^{\widehat{L}}$ first. As $L_{\text{floor}}(x_1^{\widehat{L}'}) = \widehat{L}'$, we have $L_{\text{floor}}(x_1^{\widehat{L}'}) < L_{\text{floor}}(x_1^{\widehat{L}})$. By Lemma 3.2, we know that $x_1^{\widehat{L}'} < x_{\min}$. As $\widehat{L} < \widehat{L}'$, we can apply Lemma 3.2 one more time to conclude that $x_1^{\widehat{L}'} < x_{\min}$. By Lemma 3.1, $L_{\text{floor}}(x)$ is monotonically decreasing in $(0, x_{\min}]$. As we know that $L_{\text{floor}}(x_1^{\widehat{L}}) < L_{\text{floor}}(x_1^{\widehat{L}'})$, we must have $x_1^{\widehat{L}'} < x_1^{\widehat{L}}$. We can prove $x_2^{\widehat{L}'} > x_2^{\widehat{L}}$ similarly. $\square$

3.4 Efficiency Analysis

We compare POKER with the state-of-the-art on vectorizing BinSearch() in terms of their algorithmic efficiency. As POKER makes the same number of key comparisons as GATHER, we compare both in terms of the latencies of their load instructions. As POKER dispenses with the redundant key comparisons introduced by FAST [15] and SEG-TREE [34], we compare all the three techniques by analyzing their relative degrees of redundancy.

— **POKER vs. GATHER.** When applying BinSearch() on w queries, the w middle elements must be loaded in each iteration. To accomplish this, GATHER uses gather instructions while POKER uses vector loads and permutations (Figure 2). Table 1 gives the instructions used by each technique and their latencies on the Intel Broadwell architecture using a 256-bit AVX2 for handling eight middle elements of type int with $w = 8$.

Consider an iteration of BinSearch() in Algorithm 1 with $w = 8$ (ints). GATHER executes two vpblendvb instructions to load lo's and hi's (Lines 15 and 17), one vpaddd and one vpsrld to compute m's (Line 18), and one vpgatherdd to gather k's (Line 13). Consider VectorizeTile(). POKER first loads the w middle elements using one vmoaddqu (Line 25), then generates a permutation mask by using one vpaddd and one vpslld (Line 33), and, finally, permutes these w middle elements by using one vpermd (Line 31). In total, GATHER takes 4x more cycles than POKER.

— **POKER vs. FAST and SEG-TREE.** Let $A = \{f, s, p\}$, with f standing for FAST, s for SEG-TREE, and p for POKER. For each vectorization approach $x \in A$, let $\mathbb{C}^x$ be the number of key comparisons performed per query in a node n , whose key count is denoted by $n.\text{keyCnt}$, and:

$$\mathbf{S}^x = \frac{\mathbb{C}^x}{\mathbb{C}^p}. \quad (8)$$

According to Knuth [16], BinSearch() is optimal in terms of key comparisons, which is given by $\lfloor \log_2(n.keyCnt) \rfloor + 1$, made at a node n in the worst case. For POKER, its in-node PBST is the smallest with its height being $\lfloor \log_2(n.keyCnt) \rfloor + 1$. POKER processes w queries in parallel. Thus we have:

$$\mathbb{C}^p = \frac{\lfloor \log_2(n.keyCnt) \rfloor + 1}{w}, \quad (9)$$

which indicates that POKER is optimal (in the worst case).

Like POKER, FAST also builds the smallest PBST for a node n , with the same height $\lfloor \log_2(n.keyCnt) \rfloor + 1$. However, FAST exploits the DLP only within a query by handling the $w - 1$ nodes in a SIMD tile that spans $\log_2(w)$ tree levels in the PBST in parallel (Figure 2). Thus, we have:

$$\mathbb{C}^f = \left\lceil \frac{\lfloor \log_2(n.keyCnt) \rfloor + 1}{\log_2(w)} \right\rceil. \quad (10)$$

This leads to:

$$\mathbf{S}^f = \frac{\mathbb{C}^f}{\mathbb{C}^p} \geq \frac{w}{\log_2 w} = \widehat{S}^f(w), \quad (11)$$

where $\widehat{S}^f(w)$ is monotonously increasing when $w \geq 2$, bounded from below by $\widehat{S}^f(2) = 2$.

Unlike FAST, SEG-TREE builds a $(w + 1)$ -ary tree of height $\lfloor \log_{w+1}(n.keyCnt) \rfloor + 1$ for a node n . Thus:

$$\mathbb{C}^s = \lfloor \log_{w+1}(n.keyCnt) \rfloor + 1. \quad (12)$$

Then we have:

$$\mathbf{S}^s = \frac{\mathbb{C}^s}{\mathbb{C}^p} = w \times \frac{\lfloor \log_{w+1}(n.keyCnt) \rfloor + 1}{\lfloor \log_2(n.keyCnt) \rfloor + 1} > w \times \frac{\log_{w+1}(n.keyCnt)}{\log_2(n.keyCnt) + 1} = \widehat{S}^s(w, n.keyCnt). \quad (13)$$

Taking the first-order partial derivatives of $\widehat{S}^s(w, n.keyCnt)$ with respect to w and $n.keyCnt$, we see that $\widehat{S}^s(w, n.keyCnt)$ is monotonically increasing over w and $n.keyCnt$. Finally, we conclude that $\mathbf{S}^s \geq 1$ always holds, since (i) $\widehat{S}^s(2, 15) > 1 \wedge \widehat{S}^s(3, 2) > 1$, and (ii) $\mathbf{S}^s \geq 1$ when $w = 2 \wedge n.keyCnt < 15$ or $w \geq 2 \wedge n.keyCnt = 1$.

As $\widehat{S}^f(w)$ and $\widehat{S}^s(w, n.keyCnt)$ are monotonically increasing over w , the performance advantages of POKER over FAST and SEG-TREE are expected to be amplified on future longer vectors [18, 28].

3.5 A C++ Template Library

To make POKER a general-purpose solution, we have designed a simple C++ template library, called *Vectorization of Intensive Tree Search* (VITS). A generic class including all the function interfaces is given in Figure 4, where Transform(), CompBatchCnt(), and SIMDSearch() are from Algorithm 2.

Figure 5 shows how the four techniques in Figure 2 are instantiated. B⁺-trees and BSTs are both supported as the underlying indexing trees. The system-specific functions hide the implementation details of a specific tree T ; they are invoked by Transform() to transform T into an abstract representation $\mathcal{T}$, which will be operated by SIMDSearch() to answer the batched queries prepared by CompBatchCnt(). Therefore, when VITS is applied to a new database system, all that is needed is to implement several simple system-specific functions without restarting from scratch.


```

1: template<class N, // node class
2:         class N_I, // inner node class
3:         class N_L, // leaf node class
4:         typename T_K, // data type of keys
5:         typename T_R> // data type of records
6: class vec_tree_approach {
7:     /***** system-specific functions *****/
8:     int get_k_cnt(N * n); // get key count in n
9:     T_K get_k_i(N_I * n, int i); // get key[i] (inner node)
10:    T_K get_k_l(N_L * n, int i); // get key[i] (leaf node)
11:    N * get_p(N_I * n, int i); // get child node i
12:    T_R get_r(N_L * n, int i); // get record i
13:    int get_h(N * root); // get the tree height
14:
15:    /***** an abstract tree T' *****/
16:    T_K * keys;
17:    bool is_leaf;
18:    int num_keys;
19:    .....
20:
21:    /***** general functions *****/
22:    void Transform(N * root);
23:    void SIMDSearch(T_K * q, T_R * a);
24:};

```

Fig. 4. A generic interface provided by VITS.

Fig. 5. The workflow of the VITS library.

4 EVALUATION

We show that POKER is superior over the state of the art, including GATHER, FAST [15], and SEG-TREE [34]. In the main algorithm of POKER (Algorithm 2), all the queries are strip mined into c batches (Line 2), with each batch processed independently. As batch processing is orthogonal to the vectorization of `BinSearch()`, we will evaluate POKER by first considering non-batch mode (i.e., with $c = 1$ assumed) and moving to batch mode (with c determined in Line 2).

We first describe our experimental settings (Section 4.1). We then separately present and discuss our results in the sequential setting (Section 4.2) and the parallel setting (Section 4.3). In either case, we first evaluate POKER operating in non-batch mode and then demonstrate that POKER can deliver even better performance in batch mode by combining vectorization and strip mining together.

Fig. 6. Permuting four double values on 256-bit AVX2.

4.1 Experimental Settings

— **Computing Platform.** Our experiments are conducted on an Intel Xeon E5-1660 v4 Broadwell 8-core processor that supports both 128-bit SSE and 256-bit AVX2 SIMD extensions. The page size is 4KB and the cache line size is 64B. All programs are compiled using the Intel ICC compiler (version 17.0.4) under `-march=core-avx2` and `-O3`.

POKER makes use of permutations with dynamic masks (Figure 2(p₃)). However, permuting four 64-bit double values is not supported by 256-bit AVX2. Thus, we imitate this by using its 32-bit counterpart, as illustrated in Figure 6.

As we focus on vectorizing read-mostly B⁺-trees, the overhead of Transform() is not included in the overall search cost. The search time of a program is the average of 5 runs.

— **Benchmarks.** Table 2 lists a set of five benchmarks used, including three B⁺-trees and two BSTs. For each benchmark, we give its node size and how its child nodes are chased in Columns 3 and 4, respectively. According to Columns 6 and 7, we consider (i) three different data types, int, float, and double, for search keys; (ii) two different datasets, with 10⁴ and 10⁶ keys each; and (iii) two key insertion orders, ascending and random. For each tree (under these 12 combinations), we list its branching factor *b*, its tree height *h*, and the average key count *keyCnt* per node in Column 5. By also noting that our computing platform supports two SIMD extensions, 128-bit SSE and 256-bit AVX2, we will evaluate each benchmark under a total of 24 configurations.

To adapt the five benchmarks according to our VITS template in Figure 4, we have added only 34.6 LOC on average, with the largest for Rodinia B⁺-Tree at 51 LOC. This demonstrates the practicality of VITS in simplifying the exploitation of DLP over indexing trees on SIMD units.

— **Queries.** For each benchmark running with one thread, 10⁸ queries are generated randomly following FAST [15] and SEG-TREE [34]. For multiple threads, 10⁹ is used.

— **Vectorization Techniques.** We have implemented FAST [15] and SEG-TREE [34] faithfully as described in their papers. For FAST, page tiling is applied to the two BST benchmarks by tiling each BST with the largest triangular tiles that can fit into a page. For SEG-TREE, all the nodes in a (*w* + 1)-ary tree are stored in depth-first order [34]. Our implementations achieve performance results similar to their published ones, as shown in Figure 7(d₁)–(d₄) for FAST in terms of keys with the int and float types and Figure 7(a₁) and (a₃) for SEG-TREE in terms of keys with the int type. GATHER is actually not published but evaluated here as it becomes a promising approach due to recent advances in SIMD architectures.

Table 2. Benchmark Characteristics

Benchmark	Description	Node Size	How Child Nodes Are Chased	Tree Statistics	int/float						double					
					10 ⁴ Keys			10 ⁶ Keys			10 ⁴ Keys			10 ⁶ Keys		
					Asc	Rand		Asc	Rand		Asc	Rand		Asc	Rand	
Interactive B ⁺ -tree [4]	A straightforward B ⁺ -tree implementation	1 × page size	pointer	<i>b</i>	340						255					
				<i>h</i>	2	2		3			2	2		3		3
				$\overline{keyCnt}$	170.46	271.22		170.00			127.56	159.70		127.00		176.77
Rodinia B ⁺ -tree [6, 7]	Accelerating SELECT statements on a SQLite database	1 × page size	array index	<i>b</i>	508						339					
				<i>h</i>	2	2		3			2	2		3		3
				$\overline{keyCnt}$	250.95	303.97		253.97			167.63	271.22		169.01		240.86
STX B ⁺ -tree [5]	A set of C++ template key / value containers based on a B ⁺ -tree	4 × cache line sizes	pointer	<i>b</i>	32						16					
				<i>h</i>	4	4		6			5	4		7		6
				$\overline{keyCnt}$	14.57	21.17		14.55			7.01	10.59		7.00		10.53
Balanced BST [25]	A generic balanced BST	N/A	pointer	<i>b</i>	2						2					
				<i>h</i>	14	14		20			14	14		20		20
				$\overline{keyCnt}$	1.00	1.00		1.00			1.00	1.00		1.00		1.00
Unbalanced BST [25]	A generic unbalanced BST	N/A	pointer	<i>b</i>	2						2					
				<i>h</i>	31	31		49			31	31		49		49
				$\overline{keyCnt}$	1.00	1.00		1.00			1.00	1.00		1.00		1.00

Fig. 7. Speedups (where the five benchmarks are listed vertically in the order in which they appear in Table 2). For each benchmark under $V - O - C$, its bars are divided into three groups according to key types, int, float, and double. Each bar represents the speedup of a SIMD technique normalized to the scalar baseline, and its associated red dot represents the average number of key comparisons per query normalized to POKER.

Finally, POKER is implemented as described in Section 3, with page tiling also applied to the two BST benchmarks.

4.2 Single-Threaded Performance

4.2.1 Non-Batch Mode. Figure 7 shows the speedups of FAST, SEG-TREE, GATHER, and POKER normalized to the scalar baseline, represented by the bars (higher is better). For each bar, its associated red dot represents the average number of key comparisons per query normalized to POKER (lower is better).

Every benchmark is evaluated under 24 configurations, $V - O - C \times \{\text{int}, \text{float}, \text{double}\}$, where $V \in \{128, 256\}$ is the vector length in bits, $C \in \{10^4, 10^6\}$ gives the number of keys in the tree, $O \in \{\text{Asc}, \text{Rand}\}$ indicates the key insertion order in tree construction, and the bars are divided into three groups according to key types: int, float, and double.

For all the five benchmarks under all the configurations, POKER is always the best performer, delivering a speedup of 10.11x, on average, followed by GATHER (4.77x), FAST (2.53x), and SEG-TREE (2.69x). Note that SEG-TREE is applicable only to the three B^+ -tree benchmarks considered.

— **GATHER vs. POKER.** GATHER is the previous best (Figure 7). POKER outperforms GATHER by 2.11x, on average, although POKER makes the same number of key comparisons as GATHER. We analyze the reasons behind this in Figure 8 by incrementally transforming GATHER into POKER. Therefore, in each configuration (represented by a line), the bottom square is GATHER, the middle square is GATHER added with SIMD tiling (Figure 2(p₁)) and page tiling (Section 4.1), and the top square is POKER (with our path-encoding-based technique also added).

With both tiling techniques applied, GATHER boosts the average performance by 29.07% for the three B^+ -tree benchmarks and 99.69% for the two BST benchmarks (over the scalar

Fig. 8. Understanding the performance advantages (as speedups) of POKER over GATHER (three lines of the same color for a $C - O$ pair are for int, float, and double from left to right).

baseline) due to mainly improved cache locality. Thanks to SIMD tiling (Figure 2(p₁)–(d₂)), the middle elements at the same level in a SIMD tile are consecutive in memory, rendering their strides to fall into one cache line size. Thus, a gather instruction accesses at most two cache lines, resulting in improved cache locality. Thanks to page tiling that is further applied to the two BST benchmarks (Section 4.1), we have observed some significant improvements for the large BSTs for Balanced BST (Figure 8(d₃) and (d₄)) and Unbalanced BST (Figure 8(e₃) and (e₄)). Even when SIMD and page tiling techniques are added to GATHER, the average speedup achieved by POKER over such enhanced GATHER is still 1.41x, showing the effectiveness of our path-encoding-based technique over gather instructions in reducing memory latency. In particular, POKER outperforms the enhanced GATHER by 1.42x for the three B⁺-tree benchmarks and 1.41x for the two BST benchmarks, on average.

– **Speedups vs. Key Comparison Counts.** In almost all the configurations, their speedups are well-correlated positively with their degrees of reduction in the number of key comparisons made with respect to the scalar baseline. For Rodinia B⁺-Tree (Figure 7(b₁)–(b₈)), where the percentage reductions are the highest under all the 24 configurations among the five benchmarks, GATHER, FAST, SEG-TREE, and POKER all achieve the largest speedups overall. Note that such percentage reductions do not translate directly into performance speedups. For Rodinia B⁺-Tree, its scalar version answers in-node queries by using sequential search, leading to more key comparisons (222.46 per query), on average, but better spatial locality (compared with binary search).

For Interactive B⁺-Tree (Figure 7(a₁)–(a₄)) and STX B⁺-Tree (Figure 7(c₁)–(c₄)) under a total of eight configurations, 128 – * – * × {double}, FAST ends up using one-single-key SIMD tiles (Figure 2(f₁)) without being able to exploit any DLP. Despite an average of 1.03x key comparisons performed with respect to the baseline (due to the $\lfloor M \rfloor$ padding), FAST still achieves a speedup of 1.07x, on average, due to improved cache locality from SIMD tiling.

– **B⁺-Trees vs. BSTs.** For Balanced BST under 128 – * – * × {double} (Figure 7(d₁)–(d₄)), however, FAST also performs 1.05x more key comparisons but achieves a speedup of 1.45x, on average, against the scalar baseline. Note that POKER achieves a reduction of key comparisons by 48.7% and a speedup of 6.49x over the scalar baseline. Due to the page tiling applied to BSTs (Section 4.1), FAST enjoys an average reduction of TLB misses by 85.82% over the baseline. For the other three B⁺-tree benchmarks, the same level of reduction in TLB misses is not observed as their nodes are already designed with page/cache locality in mind (Table 2).

For Unbalanced BST under 128 – * – * × {double} (Figure 7(e₁)–(e₄)), FAST has achieved a percentage reduction of key comparisons by 20.4% and a speedup of 1.70x over the scalar baseline, on average, as FAST (like POKER) rebalances a BST into a PBST so that the original

Table 3. Average Speedups for Keys of Different Data Types

Type	Average Speedups (x)			
	FAST	SEG-TREE	GATHER	POKER
int	2.88	2.94	5.33	10.55
float	2.94	3.01	5.46	11.18
double	1.79	2.13	3.53	7.04

Table 4. Average Time Overhead of Transform() for $* - * - 10^6 \times \{\text{int}, \text{float}, \text{double}\}$

Benchmark	Transformation Overhead (secs)			
	FAST	SEG-TREE	GATHER	POKER
Interactive B ⁺ -tree	0.08	0.03	N/A	0.02
Rodinia B ⁺ -tree	0.11	0.02	N/A	0.01
STX B ⁺ -tree	0.37	0.05	N/A	0.04
Balanced BST	0.09	N/A	0.01	0.11
Unbalanced BST	0.09	N/A	0.02	0.11

tree height has been decreased. Note that POKER has improved these two numbers to 60.2% and 7.63x, respectively.

- **Data Types.** All the techniques, as shown in Table 3, are less effective for double than for int/float as the amount of DLP halves for the same vector width. Note that permuting four double values with a dynamic mask is not currently supported on 256-bit AVX2. The workaround illustrated in Figure 6 also slows down the performance of POKER. Despite this, POKER still outperforms GATHER, the best of the three existing techniques, by 1.91x in the case of double.
- **POKER's Performance vs. Key Insertion Order.** The orders under which keys are inserted during the tree construction may affect the performance of a vectorization technique for the resulting tree. Let us examine POKER for Rodinia B⁺-Tree under the two configurations, $256 - \text{Asc} - 10^6 \times \{\text{int}\}$ and $256 - \text{Rand} - 10^6 \times \{\text{int}\}$ given in Figure 7(b₇) and (b₈), denoted $\text{POKER}_{\text{Asc}}$ and $\text{POKER}_{\text{Rand}}$, respectively. $\text{POKER}_{\text{Rand}}$ makes fewer key comparisons but less performance improvement than $\text{POKER}_{\text{Asc}}$. In Table 2, we have $128 \leq \text{keyCnt} < 256$ for 99.92% of the tree nodes with *Asc* and $256 \leq \text{keyCnt} < 512$ for 96.27% of the tree nodes with *Rand*. Thus, an in-node PBST has 8 levels under $\text{POKER}_{\text{Asc}}$ but 9 levels under $\text{POKER}_{\text{Rand}}$. In both cases, each SIMD tile has $2w - 1 = 15$ keys at its four levels. So $\text{POKER}_{\text{Rand}}$ ends up with one more level of SIMD tiles than $\text{POKER}_{\text{Asc}}$. As a result, $\text{POKER}_{\text{Rand}}$ spends more time on queuing the queries between SIMD tiles, thereby decreasing its computation/memory transfer ratio and thus its overall performance.
- **Transformation Overhead.** FAST, SEG-TREE (for B⁺-trees), GATHER (for BSTs), and POKER all apply Transform() to perform a tree layout transformation before the search begins. However, their transformation times are small, as shown in Table 4.

Table 5 compares the average spaces allocated for representing the transformed trees. On average, GATHER (which serves as the baseline) is the least expensive, SEG-TREE is the most expensive (1.97x) because it explicitly maintains an arrays of child pointers for each tree node [34] (Section 3.1), and both FAST and POKER sit in between (1.17x and 1.34x, respectively) due to their compressed representations of child pointers in a tree node (Figure 3).

Table 5. Average Space Overhead of Transformed Trees
for $* - * - 10^6 \times \{\text{int}, \text{float}, \text{double}\}$

Benchmark	Space Overhead Normalized to GATHER (x)			
	FAST	SEG-TREE	GATHER	POKER
Interactive B ⁺ -tree	1.00	1.99	1.00	1.01
Rodinia B ⁺ -tree	1.09	2.00	1.00	1.11
STX B ⁺ -tree	1.03	1.91	1.00	1.23
Balanced BST	1.28	N/A	1.00	1.68
Unbalanced BST	1.28	N/A	1.00	1.68

Table 6. Query Latencies of POKER for $* - * - 10^6 \times \{\text{int}, \text{float}, \text{double}\}$ with 10^8 Queries

Benchmark	Query latency (secs)		
	Minimum	Maximum	Average
Interactive B ⁺ -tree	1.60	3.79	2.39
Rodinia B ⁺ -tree	1.38	3.88	2.37
STX B ⁺ -tree	1.83	4.59	2.90
Balanced BST	1.68	3.81	2.49
Unbalanced BST	1.67	3.82	2.49

Both FAST and POKER (relative to GATHER) are more space-consuming for BSTs than for B⁺-trees, as GATHER is less space-consuming for BSTs than for B⁺-trees.

- **Search Times.** To understand the (absolute) search time incurred per query, Table 6 gives the minimum, maximum, and average search times of POKER under the same configurations as in Table 4 with 10^8 queries issued. According to Nielsen [20], such response times are acceptable for throughput-oriented systems.

4.2.2 Batch Mode. Figure 9 demonstrates that POKER can achieve improved performance in batch mode in the sequential setting. For each benchmark, we continue to consider the same 24 configurations as shown in Figure 7. Each configuration in $V - O - C \times \{\text{int}, \text{float}, \text{double}\}$ is represented as a line, decorated with six diamonds representing the speedups achieved by POKER under six different batch counts, $\lfloor \frac{1+c-1}{2} \rfloor$, $c-1$, c , $c+1$, $\lfloor \frac{c+1+c_{\text{full}}}{2} \rfloor$, and c_{full} (normalized to the performance of POKER in non-batch mode). For each B⁺-tree with N_{LeafNode} leaf nodes under a particular configuration, c is the batch count computed in Algorithm 2, and $c_{\text{full}} = \lfloor \frac{|Q|}{w \times N_{\text{LeafNodes}} \times (2w)^{h_{\text{tiled_PBST}}-1}} \rfloor$ is the batch count under which there are usually no fewer than w queries visiting a SIMD tile at a leaf tree node (if the queries in Q are evenly distributed), where $h_{\text{tiled_PBST}}$ is the maximum height of a tiled PBST (Section 3.3).

To compute c , l_{mpf} in (3) and l_{llc} in (4) must be initialized. In our platform, $l_{\text{llc}} = 34$ cycles [2]. However, l_{mpf} is not directly available. We estimate it as $l_{\text{mpf}} = \frac{n_{\text{ret_intr}} \times CPI}{n_{\text{mpf}}}$ by running POKER with 10^8 queries over Rodinia B⁺-Tree under $128 - Asc - 10^6 \times \{\text{int}\}$ in non-batch mode. From Intel Vtune, $n_{\text{ret_intr}}$ and CPI are obtained as the number of retired instructions and cycles per instruction in handling the MPFs incurred, respectively. From `/usr/bin/time -v`, n_{mpf} is read off as the number of MPFs incurred. Finally, $l_{\text{mpf}} = 904.21$ cycles is the average of 10 runs.

Strip mining queries boosts POKER’s performance further, as shown in Figure 9. For all the five benchmarks under all the 120 configurations, POKER delivers the best speedup (1.208x) with

Fig. 9. Batch processing (with the benchmarks listed vertically in the order in which they appear in Table 2). For each benchmark under $V - O - C$, its three lines represent three data types as int, float, and double, respectively. Each line represents POKER's speedups with six batch counts, $\lfloor \frac{1+c-1}{2} \rfloor$, $c-1$, c , $c+1$, $\lfloor \frac{c+1+c_{full}}{2} \rfloor$, and c_{full} , normalized to POKER's performance on the respective data type in non-batch mode. c found by Algorithm 2 is identified with a red diamond, with its value shown in red just under the line.

c batches on average. With $c-1$ and $c+1$ batches, POKER achieves roughly the same speedup (1.204x), followed by $\lfloor \frac{1+c-1}{2} \rfloor$ batches (1.179x), $\lfloor \frac{c+1+c_{full}}{2} \rfloor$ batches (1.026x), and c_{full} batches (0.903x).

- **MPFs vs. LLC Misses.** As discussed in Section 3.3, the number of MPFs tends to decrease while the number of LLC misses tends to increase as the number of batches increases. In POKER, dividing Q into c (computed in Algorithm 2) batches aims to achieve a good tradeoff between these two compromise factors. Take Interactive B⁺-Tree under $256 - Rand - 10^6 \times \{\text{double}\}$, for example, where $c = 10$. Going from non-batch mode to $c = 10$, POKER reduces the number of MPFs by 40.36% but causes the number of LLC misses to increase by 25.00%, thereby achieving a speedup of 1.20x. On the other hand, going from $c_{full} = 69$ to $c = 10$, POKER causes the number of MPFs to increase by 19.15% but reduces the number of LLC misses by 87.50%, achieving a speedup of 1.15x.
- **Batch Processing vs. Data Types.** With strip mining (i.e., in batch mode), POKER delivers generally better performance for double than for int/float. For a given configuration $V - O - C$, Φ_n (Φ_t) consumes twice as much memory for double than int/float. Without strip mining (i.e., in non-batch mode), POKER therefore suffers from more MPFs and worse cache locality for double than for int/float. With strip mining, the effects of reduced MPFs and improved cache locality on boosting performance become more pronounced for double than for int/float.

Take Balanced BST under $128 - Asc - 10^4 \times \{\text{int}\}$ (denoted as $\text{POKER}_{\text{int}}$) and $128 - Asc - 10^4 \times \{\text{double}\}$ (denoted as $\text{POKER}_{\text{double}}$) for example. Moving from non-batch mode to batch mode with $c = 72$ ($c = 91$), $\text{POKER}_{\text{int}}$ ($\text{POKER}_{\text{double}}$) achieves a relative speedup of 1.24x (1.42x). According to Vtune, the percentage of the total time spent on handling MPFs is

Fig. 10. Speedups of GATHER and POKER with multiple threads normalized to GATHER (with two threads).

33.67% (39.71%) for $\text{POKER}_{\text{int}}$ ($\text{POKER}_{\text{double}}$) in non-batch mode, but drops by 41.67% (55.12%) for $\text{POKER}_{\text{int}}$ ($\text{POKER}_{\text{double}}$) in batch mode. In addition, the fraction of pipeline slots stalled due to demand loads and stores is 8.13% (18.00%) for $\text{POKER}_{\text{int}}$ ($\text{POKER}_{\text{double}}$) in non-batch mode, but drops by 35.66% (35.93%) for $\text{POKER}_{\text{int}}$ ($\text{POKER}_{\text{double}}$) in batch mode.

- **Batch Processing vs. Tree Sizes.** POKER tends to use a larger c and achieve a better speedup under $V - O - 10^4$ than $V - O - 10^6$ for the same key type used. As discussed in Section 3.3, increasing c (starting from 1) decreases the number of MPFs but increases the number of LLC misses until its optimal balance point is reached. As a tree becomes larger, more LLC misses are incurred. Thus, such a balance point occurs later at a larger c under $V - O - 10^4$ than $V - O - 10^6$.
- **Optimal Batch Count.** In practice, c found in Algorithm 2 is not necessarily optimal. The most visible case happens in Interactive B⁺-Tree under $256 - \text{Asc} - 10^6 \times \{\text{int}\}$, where $c = 7$. POKER with $c + 1 = 8$ batches outperforms itself with 7 batches by 1.02x. However, with 7 batches, POKER still achieves a speedup of 1.14x compared with when batching processing is not used.

4.3 Multithreaded Performance

In the parallel setting, we present only our results of comparing POKER with GATHER, as (i) GATHER outperforms both FAST and SEG-TREE in all the 120 configurations evaluated in Figure 7, and (ii) all the queries are relatively independent.

4.3.1 Non-Batch Mode. Figure 10 gives our results, with the red lines for POKER and the blue lines for GATHER. For each benchmark, we consider the same 24 configurations as in Figure 7, and for every $V - O - C$, we display again its corresponding results for int, float, and double

consecutively. For each configuration represented by a line, its four squares represent the speedups achieved by 2, 4, 6, and 8 threads from the bottom to top, with the two-threaded execution of GATHER as the baseline. For every thread count, POKER outperforms GATHER for all the 120 configurations considered. On average, the speedups of POKER over GATHER are 2.21x, 2.22x, 2.25x, and 2.28x when 2, 4, 6, and 8 threads are used, respectively. Both GATHER and POKER are relatively scalable across most of the 120 configurations. In general, the performance speedups of POKER over GATHER in the single-threaded setting, as displayed in Figure 7, are carried over to Figure 10.

For all the configurations, POKER achieves good strong scaling as the number of threads increases. This holds also for GATHER except for a large-sized STX B^+ -Tree. Take $256 - Asc - 10^6 \times \{\text{double}\}$, for example. When the number of threads increases from 6 to 8, GATHER's execution time increases by 21.3% due to a low computation over memory access ratio in such tall trees.

As shown in Table 2, STX B^+ -Tree features nodes where $\overline{keyCnt}$ is small. For the configuration considered, $4 \leq \overline{keyCnt} < 8$ holds for more than 99.99% of the tree nodes. For a query processed at a node, every three key comparisons will be followed by two memory accesses, one to access an array containing its children and one to access the right child for the query. According to Intel Vtune, the CPI for GATHER with 8 threads is 0.90 with 36.2% of the execution pipeline slots stalled due to demand memory accesses. As for POKER, the CPI is 0.59 with the number of stalled slots reduced by 43.89%. This happens since POKER has replaced one of the two memory accesses with a computation due to the use of a single child pointer in a node (Figure 3). As a result, POKER's execution time has dropped by 0.23% when the number of threads increases from 6 to 8.

POKER's scalability is less impressive on double with 6 and 8 threads, as POKER becomes more memory-bound on 64-bit keys. Take Rodinia B^+ -Tree under $128 - Rand - 10^4 \times \{\text{int}\}$ (denoted as $\text{POKER}_{\text{int}}^{\text{multi}}$) and $128 - Rand - 10^4 \times \{\text{double}\}$ (denoted as $\text{POKER}_{\text{double}}^{\text{multi}}$) for example. Going from 4 to 6 threads, the relative speedup achieved is 1.42x (1.28x) for $\text{POKER}_{\text{int}}^{\text{multi}}$ ($\text{POKER}_{\text{double}}^{\text{multi}}$). We have again analyzed this based on the fraction of pipeline slots stalled due to demand loads and stores. For $\text{POKER}_{\text{int}}^{\text{multi}}$, this metric is 5.50% with 4 threads but increases to 7.00% with 6 threads. For $\text{POKER}_{\text{double}}^{\text{multi}}$, however, this metric is 23.80% with 4 threads but increases to 30.10% with 6 threads. Given the same number of queries issued in both cases, larger memory access overhead is incurred for $\text{POKER}_{\text{double}}^{\text{multi}}$ (than for $\text{POKER}_{\text{int}}^{\text{multi}}$), causing execution starvation and thus hindering its scalability.

4.3.2 Batch Mode. POKER can also achieve improved performance in batch mode in the multi-threaded setting. As shown in Figure 11, each benchmark is evaluated under a total of 96 configurations: $V - O - C \times \{\text{int}, \text{float}, \text{double}\} \times \{2 \text{ Threads}, 4 \text{ Threads}, 6 \text{ Threads}, 8 \text{ Threads}\}$. For each configuration, the decimal number represents the speedup of POKER operating in batch mode over itself in non-batch mode, and the integer number below represents the number of batches found by Algorithm 2. For all the benchmarks under all the configurations, batch processing is always beneficial, allowing POKER to achieve an average speedup of 1.24x (with 1.17x for 2 threads, 1.21x for 4 threads, 1.27 for 6 threads, and 1.31x for 8 threads).

The same two observations that are made for the single-threaded setting (Section 4.2.2) also apply here. First, the effect of batch processing on performance improvements is generally more pronounced for double than for int/float. Second, POKER tends to use more batches and achieve better speedups on smaller trees with 10^4 keys (than trees with 10^6 keys).

With batch processing, POKER's scalability for 6 and 8 threads, as discussed in Section 4.3.1, has improved, especially for trees with 10^4 double keys, due to the improved performance in memory accesses. Let us revisit $\text{POKER}_{\text{double}}^{\text{multi}}$ under configuration $128 - Rand - 10^4$ on Rodinia B^+ -Tree, as discussed in Section 4.3.1, where $\text{POKER}_{\text{double}}^{\text{multi}}$ achieves a relative speedup of 1.28x going from 4 to 6 threads in non-batch mode. In batch mode, $\text{POKER}_{\text{double}}^{\text{multi}}$ achieves a performance improvement

Fig. 11. Batch processing with multithreading. In each rectangle that represents a specific configuration, the decimal number represents the speedup of POKER operating in batch mode over itself in non-batch mode, and the integer number in the parentheses represents the batch count c found by Algorithm 2.

of 24.87% with 4 threads and 32.62% with 6 threads, giving rise to a relative speedup of 1.36x. According to Vtune, the fraction of pipeline slots stalled due to demand loads and stores decreases by 12.74% with 4 threads and 17.93% with 6 threads.

5 RELATED WORK

In addition to GATHER, FAST [15] and SEG-TREE [34], we review some other related work here.

- **Vectorization of ITS.** Kim et al. [15] propose a GPU version of FAST for vectorizing BSTs by tiling a BST according to the warp size instead of the SIMD width. On a CPU-GPU system, Shahvarani et al. [26] propose to scatter the internal and leaf nodes of a B⁺-tree to GPU and CPU memories, respectively, and process in a pipelined manner. For large indexing trees, the GPU memory may be a bottleneck. To overcome this limitation, Daga et al. [9] introduce a new B⁺-tree search on a fused processor, where CPU and GPU cores are integrated on a single die and can both access the system memory, but the computing power of the fused processor can be limited [26].
- **Vectorization of Tree-Based Algorithms.** Unlike ITS, repeated tree traversals allow a query to visit all the nodes at a tree level unless truncated otherwise. Jo et al. [14] dynamically reorder queries to improve SIMD utilization on CPUs. On GPUs, AUTOROPES [11] transforms repeated traversals to improve performance. Later, SCHED [17] and REGTT [35] can better schedule repeated traversals based on some truncation-induced regularities. Abstracted as a computation tree, a recursive and task-parallel program is vectorized by combining code transformations and scheduling strategies [22]. On a random forest containing a collection of binary decision trees, Ren et al. [23] develop an intermediate language and a runtime scheduler for efficient SIMD execution.
- **SIMDization of Irregular Applications.** In addition to tree algorithms, many irregular applications are no longer SIMD-prohibited, including sparse matrix computation [8], finite state machine transition [31], and hash-based aggregation [13]. These approaches combine gather instructions and loop tiling [33] to reduce the strides between memory accesses. On GPUs, Rhu et al. [24] apply *SIMD lane permutation* to improve the compaction of divergent branches. Nuzman et al. [21] vectorize nonconsecutive memory accesses with power-of-two strides on CPUs by exploiting spatial locality. Anderson et al. [3] later generalize this earlier work by handling arbitrary but compile-time-known strides using permute and blend operations.

6 CONCLUSION

We introduce POKER, a new approach for vectorizing a large number of queries over indexing trees by leveraging permutation instructions, which are known as an efficient means for data recognition but hard to utilize in practice. POKER, as a general-purpose solution, outperforms the prior art significantly. We hope that our POKER approach can provide some insights on vectorizing other irregular applications with data reorganization.

ACKNOWLEDGMENTS

The authors wish to thank the reviewers for their comments.

REFERENCES

- [1] 1997. *Principles and Recommendations for Population and Housing Censuses*. United Nations.
- [2] 2016. Intel® 64 and IA-32 architectures optimization reference manual. Intel Corporation.
- [3] Andrew Anderson, Avinash Malik, and David Gregg. 2015. Automatic vectorization of interleaved data revisited. *ACM Trans. Archit. Code Optim.* 12, 4 (2015), 50:1–50:25.
- [4] Amittai F. Aviram. 2008. Interactive B⁺-tree. <http://www.amittai.com/prose/bplustree.html>. (2008).
- [5] Timo Bingmann. 2013. Stx B⁺-tree C++ template classes. <http://panthema.net/2007/stx-btree>. (2013).
- [6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE, 44–54.
- [7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. 2010. A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE, 1–11.

- [8] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting recent SIMD architectural advances for irregular applications. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 47–58.
- [9] Mayank Daga and Mark Nutter. 2012. Exploiting coarse-grained parallelism in B+ tree searches on an APU. In *Proceedings of the SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 240–247.
- [10] Franz Franchetti and Markus Püschel. 2008. Generating SIMD vectorized permutations. In *Proceedings of the International Conference on Compiler Construction*. Springer, 116–131.
- [11] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 10:1–10:12.
- [12] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. 2016. Future vector microprocessor extensions for data aggregations. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. IEEE, 418–430.
- [13] Peng Jiang and Gagan Agrawal. 2017. Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation. In *Proceedings of the International Conference on Supercomputing*. ACM, 24:1–24:11.
- [14] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 363–374.
- [15] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and gpus. In *Proceedings of the International Conference on Management of Data*. ACM, 339–350.
- [16] D. E. Knuth. 1971. Optimum binary search trees. *Acta Inform* 1, 1 (1971), 14–25.
- [17] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the International Conference on Supercomputing*. ACM, 2:1–2:12.
- [18] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD vectorization of fast fourier transforms for the larrabee and AVX instruction sets. In *Proceedings of the International Conference on Supercomputing*.
- [19] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 317–328.
- [20] Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann.
- [21] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 132–143.
- [22] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient execution of recursive programs on commodity vector hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 509–520.
- [23] Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 1–10.
- [24] Minsoo Rhu and Mattan Erez. 2013. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 356–367.
- [25] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms, 4th Edition*. Addison-Wesley Professional.
- [26] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *Proceedings of the International Conference on Management of Data*. ACM, 1523–1538.
- [27] Karthik Sonti. 2016. Building event-driven batch analytics on AWS. <https://aws.amazon.com/cn/blogs/bigdata/building-event-driven-batch-analytics-on-aws/>.
- [28] Nigel Stephens. 2016. ARMv8-A next-generation vector architecture for HPC. (2016).
- [29] H. R. Strong, G. Markowsky, and A. K. Chandra. 1979. Search within a page. *Journal of the ACM* 26, 3 (1979), 457–482.
- [30] Tomas Karnagel, Roman Dementiev, Ravi Rajwar Konrad Lai Thomas Legler Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel transactional synchronization extensions. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*. IEEE, 476–487.
- [31] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling concurrent search trees under contention using HTM. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 385–399.
- [32] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB* 2, 1 (2009), 385–394.
- [33] Jingling Xue. 2000. *Loop Tiling for Parallelism*. Kluwer Academic Publishers.
- [34] Steffen Zeuch, Johann-Christoph Freytag, and Frank Huber. 2014. Adapting tree structures for processing with SIMD instructions. In *Proceedings of the 17th International Conference on Extending Database Technology*. OpenProceedings.org, 97–108.

- [35] Feng Zhang, Peng Di, Hao Zhou, Xiangke Liao, and Jingling Xue. 2016. RegTT: Accelerating tree traversals on GPUs by exploiting regularities. In *Proceedings of the 45th International Conference on Parallel Processing*. IEEE, 562–571.
- [36] Feng Zhang and Jingling Xue. 2018. Poker: Permutation-based SIMD execution of intensive tree search by path encoding. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 87–99.
- [37] Hao Zhou and Jingling Xue. 2016. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization* (2016), 11:1–11:26.
- [38] Hao Zhou and Jingling Xue. 2016. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 59–69.

Received May 2018; revised August 2018; accepted September 2018