Search Lookaside Buffer: Efficient Caching for Index Data Structures

Xingbo Wu
University of Texas at Arlington
xingbo.wu@mavs.uta.edu

Fan Ni
University of Texas at Arlington
fan.ni@mavs.uta.edu

Song Jiang
University of Texas at Arlington
song.jiang@uta.edu

ABSTRACT

With the ever increasing DRAM capacity in commodity computers, applications tend to store large amount of data in main memory for fast access. Accordingly, efficient traversal of index structures to locate requested data becomes crucial to their performance. The index data structures grow so large that only a fraction of them can be cached in the CPU cache. The CPU cache can leverage access locality to keep the most frequently used part of an index in it for fast access. However, the traversal on the index to a target data during a search for a data item can result in significant false temporal and spatial localities, which make CPU cache space substantially underutilized. In this paper we show that even for highly skewed accesses the index traversal incurs excessive cache misses leading to suboptimal data access performance. To address the issue, we introduce Search Lookaside Buffer (**SLB**) to selectively cache only the search results, instead of the index itself. SLB can be easily integrated with any index data structure to increase utilization of the limited CPU cache resource and improve throughput of search requests on a large data set. We integrate SLB with various index data structures and applications. Experiments show that SLB can improve throughput of the index data structures by up to an order of magnitude. Experiments with real-world key-value traces also show up to 73% throughput improvement on a hash table.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; **Point lookups**; • **Theory of computation** → **Caching and paging algorithms**;

KEYWORDS

Caching, Index Data Structure, Key-Value Store

ACM Reference Format:
Xingbo Wu, Fan Ni, and Song Jiang. 2017. Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *Proceedings of ACM Symposium of Cloud Computing conference, Santa Clara, CA, USA, September 24–27, 2017 (SoCC '17),* 13 pages.
https://doi.org/10.1145/3127479.3127483

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127483

1 INTRODUCTION

In-memory computing has become popular and important due to applications' demands on high performance and availability of increasingly large memory. More and more large-scale applications store their data sets in main memory to provide high-performance services, including in-memory databases (e.g., H-Store [28], MemSQL [39], and SQLite [45]), in-memory NoSQL stores and caches (e.g., Redis [43], MongoDB [41], and Memcached [38]), and large forwarding and routing tables used in software-defined and content-centric networks [4, 13, 57]. In the meantime, these applications rely on index data structures, such as hash table and B^+-tree, to organize data items according to their keys and to facilitate search of requested items. Because the index always has to be traversed to locate a requested data item in a data set, the efficiency of the index traversal is critical. Even if the data item is small and requires only one memory access, the index traversal may add a number of memory accesses leading to significantly reduced performance. For example, a recent study on modern in-memory databases shows that "*hash index (i.e., hash table) accesses are the most significant single source of runtime overhead, constituting 14–94% of total query execution time.*" [30]. A conventional wisdom to addressing the issue is to keep the index in the CPU cache to minimize index search time.

However, it is a challenge for the caching approach to be effective on reduction of index access time. The memory demand of an index (indices) can be very large. As reported, "*running TPC-C on H-Store, a state-of-the-art in-memory DBMS, the index consumes around 55% of the total memory.*" [55]. The study on Facebook's use of Memcached with their five workloads finds that Memcached's hash table, including the pointers on the linked lists for resolving hash collision and the pointers for tracking access locality for LRU replacement, accounts for about 20–40% of the memory space [2]. With a main memory of 128 GB or even larger holding a big data set, the applications' index size can be tens of gigabytes. While a CPU cache is of only tens of megabytes, search for a data item with a particular key in the index would incur a number of cache misses and DRAM accesses, unless there is strong locality in the index access and the locality can be well exploited.

Indeed, requests for data items usually exhibit strong locality. For example, as reported in the Facebook's Memcached workload study, "*All workloads exhibit the expected long-tail distributions, with a small percentage of keys appearing in most of the requests…*". For one particular workload (ETC), 50% of the keys occur in only 1% of all requests [2]. Such locality is also found in the workloads of database [30] and network forwarding table [57]. Each of the requested data items is usually associated with an entry in the index data structure. The entry corresponds to the same key as the one in the request. Example index data structures include hash table and

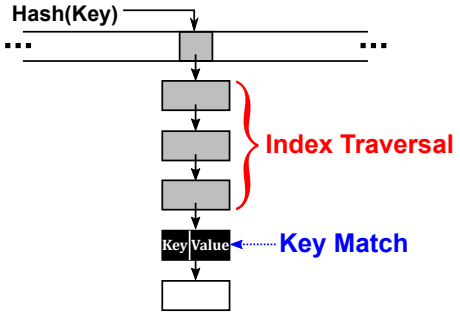


Figure 1: False temporal locality in a hash table. False temporal locality is generated on a path to a target entry in the hash table.

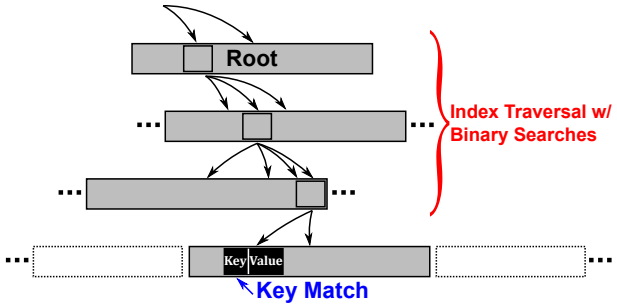


Figure 2: False temporal locality in a B$^+$-tree. False temporal locality is generated on a path to a target entry in a B$^+$-tree.

B$^+$-tree. The requested data item can be either directly included in the entry, such as a switch port number in a router's forwarding table [57], or pointed to by a pointer in the entry, such as user-account status information indexed by the hash table in Facebook's Memcached system [2]. In both cases, to access the requested data, one must search the index with a given key to reach the index entry, named *target entry*. The goal of the search is to obtain the *search result* in the target entry. The result can be the requested data item itself or a pointer pointing to the data item. Strong access locality of requested data is translated to strong locality in the access of corresponding *target entries*. However, this locality is compromised when it is exploited in the current practice of index caching for accelerating the search.

First, the temporal locality is compromised with index search. To reach a target index entry, one has to walk on the index and visit intermediate entries. For a hot (or frequently accessed) target entry, the intermediate entries on the path leading to it also become hot from the perspective of the CPU cache. This is illustrated in Figures 1 and 2 for the hash table and B$^+$-tree, respectively. However, access locality exhibited on the intermediate entries is artificial and does not represent applications' true access pattern about requested data. Accordingly, we name this locality *false temporal locality*. Such locality can increase demand on cache space by many times leading to high cache miss ratio.

Second, CPU accesses memory and manages its cache space in the unit of cache lines (usually of 64 bytes). The search result in a

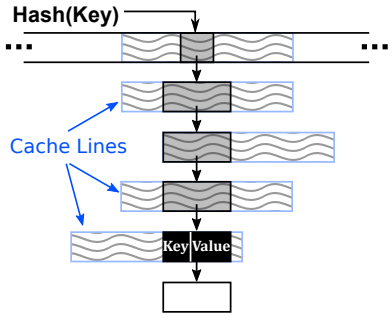


Figure 3: False spatial locality in a hash table. False spatial locality is generated in the cache lines containing intermediate entries and target entry on a path in the hash table.

target entry can be much smaller than a cache line (e.g., a 8-byte pointer vs. 64-byte cache line). In an index search spatial locality is often weak or even does not exist, especially when keys are hashed to determine their positions in the index. Because CPU cache space must be managed in the unit of cache line, (probably cold) index entries in the same cache line as those on the path to a target entry can be fetched into the cache as if there were spatial locality. We name the locality *false spatial locality*, as illustrated in Figure 3 for the hash table. This false locality unnecessarily inflates cache demand, pollutes the cache, and reduces cache hit ratio.

To remove the aforementioned false localities and improve efficiency of limited CPU cache space, we introduce an index caching scheme, named Search Lookaside Buffer (**SLB**), to accelerate search on any user-defined in-memory index data structure. A key distinction of SLB from existing use of cache for the indices is that SLB does not cache an index according to its memory access footprint. Instead, it identifies and caches search results embedded in the target entries. By keeping itself small and its contents truly hot, SLB can effectively improve cache utilization. SLB eliminates both false temporal and spatial localities in the index searches, and enables search at the cache speed.

The main contributions of this paper are as follows:

- We identify the issue of false temporal and false spatial localities, in the use of major index data structures, responsible for degradation of index search performance for significant in-memory applications.
- We design and implement the SLB scheme, that can substantially increase cache hit ratio and improve search performance by removing the false localities.
- We conduct extensive experiments to evaluate SLB with popular index data structures, in-memory key-value applications, and a networked key-value store on a high-performance Infiniband network. We also show its performance impact using real-world key-value traces from Facebook.

2 MOTIVATION

This work was motivated by observation of false temporal and spatial localities in major index data structures and their performance implication on important in-memory applications. In this section we will describe the localities and their performance impact in two

representative data structures, B$^+$-trees and hash tables, followed with discussions on similar issues with process page table and on how the solution of SLB was inspired by an important invention in computer architecture—the TLB table.

2.1 False localities in B+-trees

B$^+$-tree [3] and many of its variants have been widely used on managing large ordered indices in databases [24, 46] and file systems [7, 44].

In B$^+$-tree each lookup needs to traverse the tree starting from the root to a leaf node with a key (see Figure 2). With a high fanout, The selection of a child node leads to multiple cache misses in a single node. For example, a 4-KB node contains 64 cache lines, and requires roughly six ($\log_2 64$) cache-line accesses in the binary search. One lookup operation on a B$^+$-tree of four levels could require 24 cache-line accesses. These cache lines are at least as frequently accessed as the target entry's cache line. For a target entry in the working set, all these cache lines will also be included in the working set. However, if one can directly reach the target entry without accessing these cache lines, the search can be completed by only one cache line access with the false temporal locality removed.

2.2 False localities in hash tables

A commonly used hash table is to use chaining for resolving collision. A hash directory consists of an array of pointers, each representing a hash bucket pointing to a linked list to store items with the same hash value. With a target search entry on one of the lists, the aforementioned false temporal locality exists. A longer list is more likely to have substantial false temporal locality.

In addition to the false temporal locality, the hash table also exhibits false spatial locality. To reach a target entry in a bucket, a search has to walk over one or more nodes on the list. Each node, containing a pointer and possibly a key, is substantially smaller than a 64 B cache line. Alongside the nodes, the cache lines also hold cold data that is less likely to be frequently accessed. However, this false spatial locality issue cannot be addressed by increasing the directory size and shortening the list lengths. A larger directory would lead to even weaker spatial locality for access of pointers in it. For every 8 B pointer in a 64 B cache line, 87.5% of the cache space is wasted.

Some hash tables, such as Cuckoo hashing [42] and Hopscotch hashing [21], use open addressing, rather than linked lists, to resolve collision for a predictable worst-case performance. However, they share the issue of false spatial locality with the chaining-based hash tables. In addition, open-addressing hashing usually still needs to make multiple probes to locate a key, which leads to false temporal locality.

2.3 Search Lookaside Buffer: inspired by TLB

The issues challenging effective use of CPU cache for fast search on indices well resemble those found in the use of page table for virtual address translation. First, as each process in the system has its own page table, total size of the tables can be substantial and it is unlikely to keep them all in the CPU cache. Second, the tables are frequently searched. For every memory-access instruction the table must be consulted to look up the physical address with a virtual address as the key. Third, the tree-structured table consists of multiple levels leading to serious false temporal locality. Fourth, though spatial locality often exists at the leaf level of the tables, such locality is less likely for intermediate entries. If the page tables were cached as regular in-memory data in the CPU cache, the demand on cache space would be significantly higher and the tables' cache hit ratio would be much lower. The consequence would be a much slower system.

Our solution is inspired by the one used for addressing the issue of caching page tables, which is Translation Lookaside Buffer (TLB), a specialized hardware cache [50]. In TLB, only page-table search results—recently accessed Page Table Entries (PTEs) at the leaf level—are cached. With a TLB as large as only a few hundreds of entries, it can achieve a high hit ratio, such as a few misses per one million instructions [34] or less than 0.5% of execution time spent on handling TLB misses [5].

It is indisputable that use of TLB, rather than treating page tables as regular data structure and caching them in the regular CPU cache, is an indispensable technique. "*Because of their tremendous performance impact, TLBs in a real sense make virtual memory possible*" [1]. Index search shares most issues that had challenged use of page tables decades ago. Unfortunately, the success of TLB design has not influenced the design on general-purpose indices. An anecdotal evidence is that to allow hash indices associated with database tables to be cache-resident, nowadays one may have to take a table partitioning phase to manually reduce index size [35].

While SLB intends to accommodate arbitrary user-defined indices and search algorithms on them, which can be of high variation and irregularity, it is not a good choice to dedicate a hardware cache separate from regular CPU cache and to apply customized management with hardware support for SLB. Instead, SLB takes an approach different from TLB. It sets up a buffer in the memory holding only hot target entries. SLB intends to keep itself sufficiently small and its contents truly hot so that its contents can be all cached in the CPU cache. It aims to keep search requests from reaching the indices, so that the indices can be much less accessed and less likely to pollute the CPU cache.

3 DESIGN OF SLB

SLB is designed for applications where index search is a performance bottleneck. While numerous studies have addressed the issues with specific index data structures and search algorithms to ameliorate this bottleneck, the SLB solution is intended to serve any data structures and algorithms for accelerating the search. This objective makes the solution have the risk of being over-complicated and entangled with designs of various data structures and algorithms. If this were the case, SLB would not have a clean interface to the users' programs. Fortunately, SLB is designed as a lookaside buffer and works independently of index data structures and their search algorithms. With limited interactions through the SLB API, the programs are only required to emit search results to SLB and delegate management of the search results to SLB.

To efficiently address the issue of false localities, the design of SLB will achieve the following goals:

```
// callback function types
typedef bool (*matchfunc)(void* entry,void* key);
typedef u64 (*hashfunc)(void* opaque);

// SLB function calls
SLB*  SLB_create(size_t size, matchfunc match,
    hashfunc keyhash, hashfunc entryhash);
void  SLB_destroy(SLB* b);

void* SLB_get(SLB* b, void* key);
void  SLB_emit(SLB* b, void* entry);
void  SLB_invalidate(SLB* b, void* key);
void  SLB_update(SLB* b, void* key, void* entry);

void  SLB_lock(SLB* b, void* key);
void  SLB_unlock(SLB* b, void* key);
```

Figure 4: SLB's Functions in API

- SLB ensures the correctness of operations on the original index data structure, especially for sophisticated concurrent data structures.
- SLB is able to identify hot target entries in the index data structure and efficiently adapt to changing workload patterns with minimal cost.
- SLB is able to be easily integrated into programs using any index data structures by exposing a clean and general interface.

3.1 API of SLB

SLB's API is shown in Figure 4. Its functions are used to support accessing of the SLB cache, maintaining consistency for its cached data, and currency control for its accesses.

3.1.1 Accessing the SLB cache. SLB is implemented as a library of a small set of functions that are called to accelerate key search in various index data structures. SLB is a cache for key-value (KV) items. While conceptually the KV items in the cache are a subset of those in the index, SLB uses its own key and value representations that are independent from those used in the index data structure defined and maintained by user code. The format of user-defined keys and values can be different in different user codes. For example, a key can either be a NULL-terminated string or a byte array whose size is specified by an integer. A value can be either stored next to its key in the target entry or linked to by a pointer next to the key.

Rather than duplicating the real key-value data in its cache, SLB stores a pointer to the target entry for each cached key-value item. In addition, a fixed-size tag—the hash value of the original key—is stored together with the pointer for quick lookup (see Section 3.2). In this way the format of SLB cache is consistent across different indices and applications. It is up to the user code to supply untyped pointers to the target entries in the user-defined index, and to supply functions to extract or to compute hash tags from user-supplied keys (`keyhash()`) and cached target entries (`entryhash()`) for SLB to use. While the formats of keys and target entries are unknown to the SLB cache, SLB also needs a user-supplied function (`match()`) to verify whether a key matches a target entry. All the three functions are specified when an SLB cache is initialized with the `SLB_create()` function.

After an SLB's initialization, the cache can be accessed with two functions. `SLB_emit()` emits a target entry successfully found in an index search to the SLB cache. Note that SLB will decide whether an emitted item will be inserted into the cache according to knowledge it maintains about the current cache use. The user simply calls `SLB_emit()` for every successful lookup on the index.

With SLB, a search in the index should be preceded by a lookup in the SLB cache through calling `SLB_get()`. If there is a hit, the search result is returned and a search on the actual index can be bypassed.

3.1.2 Maintaining consistency. To prevent SLB from returning stale data, user code needs to help maintain consistency between the index and the SLB cache. For this purpose, user code should call `SLB_invalidate()` when a user request removes an item from the index, or call `SLB_update()` when an item is modified. `SLB_update()` should also be called if a target entry is relocated in the memory due to internal reorganization of the index, such as garbage collection.

As user code does not know whether an item is currently cached by SLB, it has to call `SLB_invalidate()` or `SLB_update()` functions for every item invalidation or updating, respectively. This is not a performance concern, as the invalidation or update operations on the index are expensive by themselves and execution of the function calls usually requires access only to one cache line. The performance impact is still relatively small even when the items are not in the SLB cache.

3.1.3 Managing concurrency. Applications usually distribute their workloads across multiple CPU cores for high performance. They often use concurrency control, such as locking, to allow a shared data structure to be concurrently accessed by multiple cores. Similarly, locking is used in SLB to manage concurrent accesses to its data structures. For this purpose, SLB provides two functions, `SLB_lock()` and `SLB_unlock()`, for user programs to inform SLB cache whether a lock on a particular key should be applied.

To prevent locking from being a performance bottleneck, SLB uses the lock striping technique to reduce lock contention [18, 20]. We divide the keys into a number of partitions and apply locking on each partition. By default there are 1024 partitions, each protected by a spinlock. SLB uses a 10-bit hash value of the key to select a partition.

A spinlock can be as small as only one byte. False sharing between locks could compromise the scalability of locking on multi-core systems. To address the issue, each spinlock is padded with unused bytes to exclusively occupy an entire cache line. Our use of stripped spinlocks can sustain a throughput of over 300 million

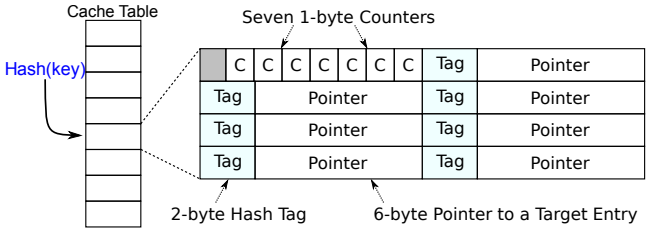


Figure 5: SLB's Cache Table

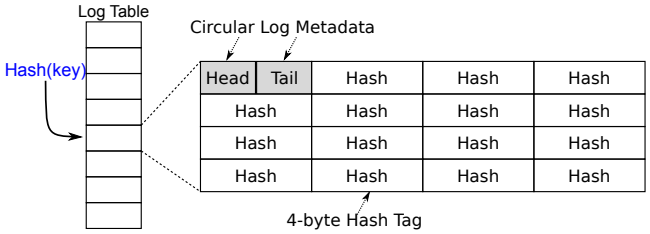


Figure 6: SLB's Log Table

lock-unlocks per second on a 16-core CPU, which is sufficient for SLB to deliver high throughput in a concurrent execution environment.

To avoid deadlocks between SLB and the index data structure, the user's code should always acquire an SLB's lock before acquiring any lock(s) for its own index. SLB's lock should be released only after all modifications to the index has been finalized and the locks on the index are released.

3.2 Data structure of the SLB cache

The SLB cache is to facilitate fast reach to requested target entries with high time and space efficiency. For this reason, the cache has to be kept small to allow its content to stay in the CPU cache as much as possible, so that target entries can be reached with (almost) zero memory access. However, the target entries can be of different sizes in different indices and can be quite large. Therefore, we cannot store target entries directly in the SLB cache. Instead, we store pointers to them.

Specifically, search results emitted into the SLB cache are stored in a hash table named *Cache Table*. To locate an item in a Cache Table, a 64-bit hash value is first obtained by calling the user-supplied functions (`keyhash()` or `entryhash()`) to select a hash bucket. As shown in Figure 5, each bucket occupies a cache line and the number of buckets is determined by the size of the SLB cache. Within each bucket there are seven pointers, each pointing to a target entry. As on most 64-bit CPU architectures no more than 48 bits are used for memory addressing, we use only 48 bits (6B) to store a pointer.

To minimize the cost for lookup of the requested target entry in a bucket, we use the higher 16 bits of the 64-bit hash value as a tag and store it with its corresponding pointer. On lookup, any target entry whose tag matches the requested key's tag will be selected and then a full comparison between the keys is performed using the user-supplied `match()` function. If there is a match the value in the target entry is returned to complete the search.

3.3 Tracking access locality for cache replacement

As the SLB cache has limited space, a decision has to be made on what items can be admitted and what items can stay in the cache based on their recent access locality, or their *temperatures*. Only comparatively hot items should be admitted or be kept in the cache. To this end, SLB needs to track temperatures for cached items and (uncached) target entries that can potentially be emitted to SLB. However, conventional approaches for tracking access locality are too expensive for SLB. For example, the list-based replacement schemes, such as LRU, require two pointers for each element, which would triple the size of Cache Table by storing three pointers for each item. Low cost replacement algorithm, such as CLOCK [11], uses only one bit per item. However it still requires global scanning to identify cold items. We develop a highly efficient locality tracking method that can effectively identify relatively hot items for caching in SLB.

3.3.1 Tracking access history of cached items. As shown in Figure 5, SLB's Cache Table has a structure similar to that of hardware-based CPU cache, which partitions cache entries into sets and identifies them with their tags. Similarly, SLB's replacement is localized within each hash bucket of a cache line size. A bucket contains seven 1-byte counters, each associated with a {tag, pointer} pair in the bucket (see Figure 5). Upon a hit on an item, its corresponding counter is incremented by one. However overflow can happen with such a small counter. To address this issue, when a counter to be incremented already reaches its maximum value (255) we randomly select another non-zero counter from the same bucket and decrement its value by one. In this way, relative temperatures of cached items in a bucket can be approximately maintained without any access outside of this bucket. To make room for a newly admitted item in a bucket, SLB selects an item of the smallest counter value for replacement.

3.3.2 Tracking access history of target entries. When a target entry is emitted to the SLB cache, SLB cannot simply admit it by evicting a currently cached item unless the new item is sufficiently hot. For this purpose, SLB also needs to keep tracking their accesses, or emissions made by the user code. However, this can be challenging. First, tracking the access history may require extra metadata attached to each item in the index. Example of such metadata include the two pointers in LRU and the extra bit in CLOCK. Unfortunately this option is undesirable for SLB as it requires intrusive modification to the user's index data structure, making it error-prone. Second, tracking temperature of cold entries can introduce expensive writes to random memory locations. For example, each LRU update requires six pointer changes, which is too expensive with accesses of many cold entries.

To know whether a newly emitted item is hot, we use an approximate logging scheme to track its access history in a hash table, named *Log Table* and illustrated in Figure 6. In this hash table, each bucket is also of 64 byte, the size of a cache line. In each bucket there can be up to 15 log entries, forming a circular log. When an item is emitted to SLB, SLB computes a 4-byte hash tag from the

key and appends it to the circular log in the corresponding bucket, where the item at the log head is discarded if the log has been full. The newly admitted item is considered to be sufficiently hot and eligible for caching in the Cache Table if the number of a key's hash tag in the log exceeds a threshold (three). In this history tracking scheme, different target entries may produce the same hash tag recorded in a log, which inflates the tag's occurrence. However, with 4-byte tag and a large number of buckets this inflation is less likely to take place. Even if it does happen, the impact is negligible.

3.3.3 Reducing cost of accessing the Log Table. For a more accurate history tracking in the Log Table, we usually use a large table (by default four times the size of the Cache Table) and do not expect many of its buckets stay in the CPU cache. With expected heavy cache misses for the logging operations in the table, we need to significantly reduce the operations on it. To this end, SLB randomly samples emitted items and logs only a fraction of them (5% by default) into the Log Table. This throttled history tracking is efficient and its impact on tracking accuracy is small. If the SLB cache has a consistently high or low hit ratios, the replacement would have less potential to further improve or reduce the performance, respectively. As a result, history tracking is not performance-critical and can be throttled. When the workload changes its access pattern, the changes will still be reflected in the logs even with the use of throttling (though it will take a longer time). With a workload mostly running at its steady phases, this does not pose a problem. As throttling may cause new items to enter the Cache Table at a lower rate, SLB disables throttling when the table is not full yet to allow the SLB cache to be quickly warmed up.

4 EVALUATION

We have implemented SLB as a C library and integrated it with a number of representative index data structures and memory-intensive applications. We conducted extensive experiments to evaluate it. In the evaluation, we attempt to answer a few questions:

- How does SLB improve search performance on various data structures?
- Does SLB have good scalability on a multi-core system?
- How much can SLB improve performance of network-based applications?
- How does SLB perform with real-world workloads?

4.1 Experimental setup

In the evaluation we use two servers. Hardware parameters of the servers are listed in Table 1. Hyper-threading feature in CPU is turned off in BIOS to obtain more consistent performance measurements. To minimize the interference of caching and locking between the CPU sockets, we use a single CPU socket (16 cores) to run the experiments unless otherwise noted.

The servers run a 64-bit Linux 4.8.13. To reduce the interference of TLB misses, we use Huge Pages [22] (2 MB or 1 GB pages) for large memory allocations. xxHash hash algorithm [53] is used in SLB.

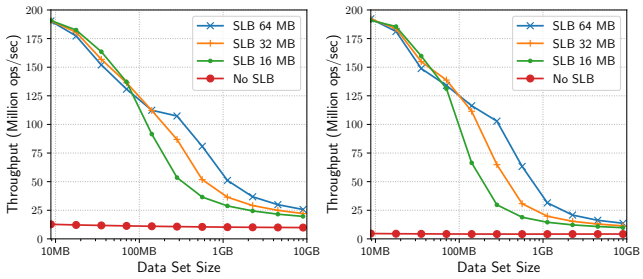
We evaluate SLB with four commonly used index data structures (Skip List, B$^+$-tree, chaining hash table, and Cuckoo hash table), and two high-performance key-value applications (LMDB [33] and MICA [32]). As it is very slow to fill up a large DRAM with small KV items, we use a data set of about 9 GB (including metadata and data) for all the experiments unless otherwise noted. We also evaluate SLB by replaying real-world key-value traces from Facebook [2], and by running SLB-enabled MICA on high-performance Infiniband network.

Table 1: Hardware parameters

Machine Model	Dell PowerEdge R730
CPU Version	Intel Xeon E5-2683 v4
Number of sockets	2
Cores per socket	16
L1 Cache (per core)	64 KB
L2 Cache (per core)	256 KB
L3 Cache (per socket)	40 MB
DRAM Capacity	256 GB (16×16GB)
DRAM Model	DDR4-2133 ECC Registered
Infiniband Network	Mellanox ConnectX-4 (100 Gb/s)

Table 2: SLB parameters

Cache Table size	16 MB	32 MB	64 MB
# target entries	1835008	3670016	7340032
Log Table size	64 MB	128 MB	256 MB
# hash tags	15728640	31457280	62914560
Total Size	80 MB	160 MB	320 MB

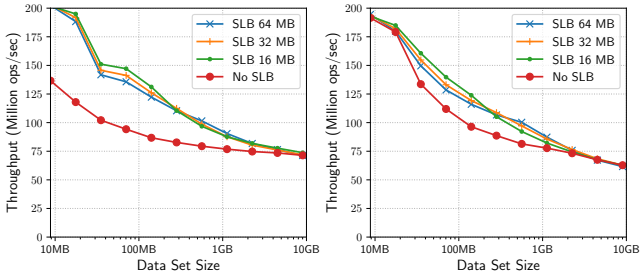


(a) B$^+$-tree **(b) Skip List**

Figure 7: Throughput with B$^+$-tree and Skip List.

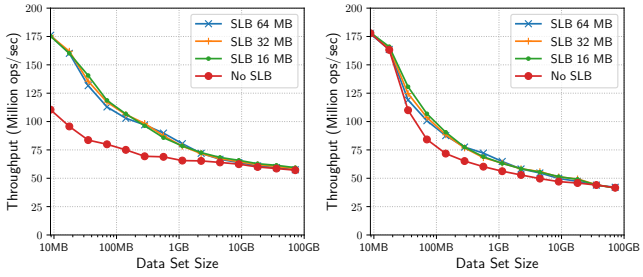
4.2 Performance on index data structures

In this experiment we first fill one of the data structures (Skip List, B$^+$-tree, chaining hash table, and Cuckoo hash table) with 100 million key-value items, each with a 8 B key and a 64 B value. Then we issue GET requests to the index using 16 worker threads, each exclusively bound to a CPU core. The workload is pre-generated in memory following the Zipfian distribution with a skewness of 0.99. For each data structure, we vary size of SLB's Cache Table from 16 MB, 32 MB, to 64 MB. We configure size of the Log Table to be 4× of the Cache Table's size. SLB's configurations are listed in Table 2. We vary the data set, or the key range used in the Zipfian generator, from 0.1 million to 100 million keys.



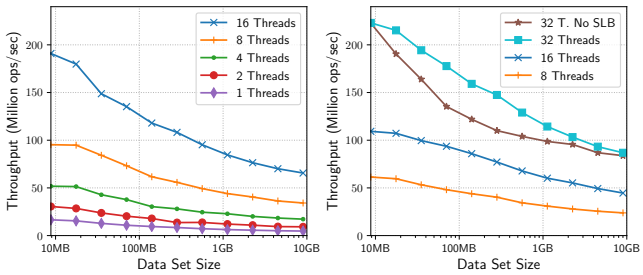
(a) Cuckoo hash table (b) Chaining hash table

Figure 8: Throughput with two hash tables.



(a) Cuckoo hash table (b) Chaining hash table

Figure 9: Throughput with 1 billion items (~90 GB).



(a) Using one CPU socket (b) Using two CPU sockets

Figure 10: Scalability of chaining hash table with 32 MB SLB.

4.2.1 B+-tree and Skip List. Figure 7 shows GET throughput of the two ordered data structures: B$^+$-tree and Skip List. As shown, SLB dramatically improves the throughput of searching on the two indices by as much as 22 times. Due to existence of significant false localities in the index search, even for a small data set of less than 10 MB, the actual working set observed by the CPU cache can be much larger than the CPU's 40 MB cache, leading to intensive misses. In addition, search on the two indices requires frequent pointer dereferences and key comparisons, consuming many CPU cycles even for items that are already in the CPU cache. Consequently the two data structures exhibit consistently low throughput when SLB is not used.

When the data set grows larger, throughput with SLB reduces but remains at least more than 2× of the throughput with SLB disabled. A larger SLB cache helps to remove false localities for more target entries. This explains the fact that the throughput of the 64 MB SLB is higher than that with a smaller SLB cache on a relatively large data set (≥ 100 MB). However, the performance trend reverses for a smaller data set, where a smaller SLB cache produces higher throughput. With a small data set on the index search and a relatively large SLB cache, the cache may store many cold items that fill the SLB's cache space but produce a smaller number of hits. The relatively cold items in the SLB cache can still cause false spatial locality for a larger SLB cache. Though SLB's performance advantage is not sensitive to the SLB cache size, it is ideal to match the cache size to the actual working set size to receive optimal performance.

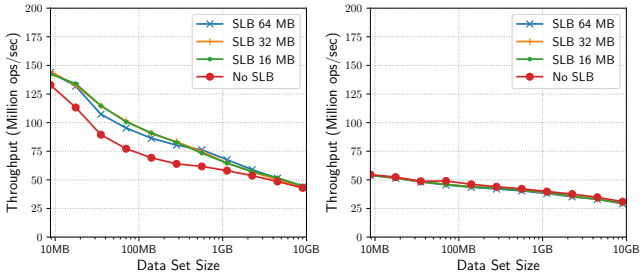
4.2.2 Hash tables. Figure 8 shows the throughput improvement of SLB with two hash tables. Without using SLB, Cuckoo hash table has lower throughput than the chaining hash table with smaller data sets. On the Cuckoo hash table each lookup accesses about 1.5 buckets on average. In contrast, we configure the chaining hash table to aggressively expand its directory so that the chain on each hash bucket has only one entry on average. For this reason the Cuckoo hash table has more significant false localities that can be removed by the SLB cache.

For the chaining hash table, the improvement mostly comes from its elimination of false spatial locality. Figure 8b shows that the chaining hash table has very high throughput with small data sets that can be all held in the CPU cache. Once the data set grows larger, the throughput drops quickly because of false spatial locality. This is the time when SLB kicks in and improves its throughput by up to 28% for medium-size data sets of 20 MB to 1 GB. When the data set becomes very large, the improvement diminishes. This is because in the Zipfian workloads with large data sets, the access locality becomes weak, and hot entries in the tables are less distinct from cold ones. SLB becomes less effective as it relies on the locality to improve CPU cache utilization with a small Cache Table.

SLB only makes moderate improvements for chaining hash table because we choose the most favorable configuration for chaining hash table. Aggressively expanding the hash directory can maximize its performance but also consume excessive amount of memory. With a conservative configuration SLB can help to maintain a high throughput by removing more false localities.

To further evaluate SLB with even larger data sets, we increase the total number of KV items in the table to 1 billion, which consumes about 90 GB of memory. We rerun the experiments on the large tables. As shown in Figure 9, with a larger table the overall throughput of all test cases reduces. This is mainly because the random access over a larger index leads to increased TLB misses. Even so, the relative improvement made by the SLB cache mostly remains.

4.2.3 Scalability. To evaluate the scalability of SLB, we change the number of worker threads from 1 to 16 and rerun the experiments using the chaining hash table. As shown in Figure 10a, SLB exhibits strong scalability. Doubling the number of working threads leads to almost doubled throughput. With the increase of data set size the throughput ratio between 16 threads and 1 thread increases from 11.5 to 13.8, because a larger data set has more balanced accesses across the hash table, which reduces contention.



(a) 95% GET, 5% SET **(b) 50% GET, 50% SET**

Figure 11: Throughput of chaining hash table with mixed GET/SET workload.

To evaluate SLB in a multi-socket system, we run the experiment by using equal number of cores from each of the two sockets. The two curves on the top of Figure 10b show the throughput of using 32 cores with SLB enabled/disabled. With both sockets being fully loaded, SLB can still improve the throughput by up to 34%.

We observe that the throughput with 32 cores is only 17% to 36% higher than that with 16 cores on one socket. When using 16 cores, the throughput with 8 cores on each of the two sockets is 30% lower than that with all 16 cores on a single socket. The impact of using two or more sockets in an index data structure is twofold. On one hand the increased cache size allows more metadata and data to be cached. On the other hand, maintaining cache coherence between different sockets is more expensive. Excessive locking and data sharing in a concurrent hash table can offset the benefit of increased cache size. As a result, localizing the accesses to a single socket is more cost-effective for a high-performance concurrent data structure.

4.2.4 Performance with mixed GET/SET. While SLB delivers impressive performance benefit with workloads of GET requests, SET requests can pose a challenge. Serving SET requests requires invalidation or update operations to maintain consistency between the SLB cache and the index. To reveal how SLB performs with mixed GET/SET operations, we change the workload to include a mix of GET/SET requests. On the hash table a SET operation is much more expensive than a GET. As shown in Figure 11, when SLB is not used, with a small percentage of SET requests (5%) the throughput is 31% lower than that of GET-only workload (see Figure 8b). It further decreases to less than 55 MOPS (million operations per second) with 50% SET in the workload, or another 41% decrease. When SLB is used, with 5% SET performance advantage of SLB remains (compare Figures 8b and 11a). However, with 50% SET, the benefit of SLB diminishes as expected.

4.3 Performance of KV applications

To understand SLB's performance characteristics in real-world applications, we run the experiments with two high-performance key-value stores, LMDB [33] and MICA [32].

4.3.1 LMDB. LMDB is a copy-on-write transactional persistent key-value store based on B$^+$-tree. LMDB uses `mmap()` system call to map data files onto the main memory for direct access. In a warmed-up LMDB all requests can be served from memory without any I/O operations. In total 124 lines of code are added to LMDB to enable SLB. We use the same workload consisting of GET requests described in Section 4.2.

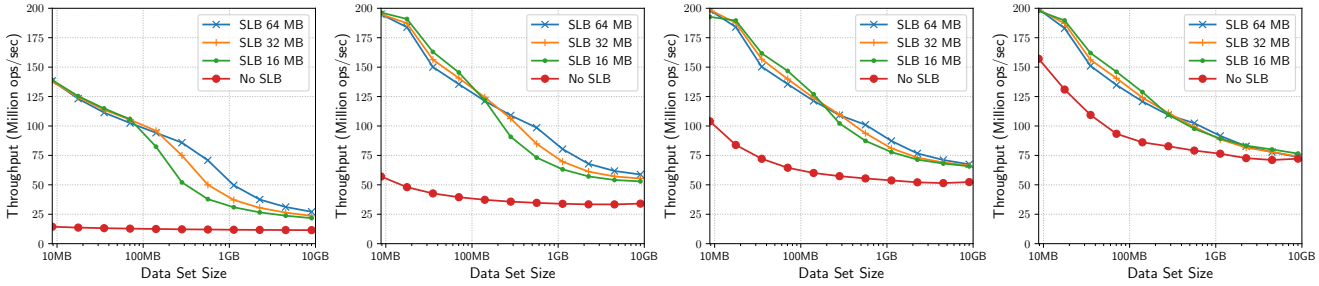
Figure 12a shows the throughput of LMDB. With larger data sets LMDB's throughput is similar to that of B$^+$-tree (See Figure 7a), because it uses B$^+$-tree as its core index structure. However, for small data sets, throughput with SLB-enabled LMDB is lower than that with B$^+$-tree. In addition to index search, LMDB has more overhead on version control and transaction support. For a small data set whose working set can almost entirely be held in the CPU cache by using SLB, LMDB spent a substantial amount of CPU cycles on the extra operations. Its peak throughput is capped at 139 MOPS, about 27% reduction over the 190 MOPS peak throughput received for B$^+$-tree with SLB.

4.3.2 MICA in the CREW mode. MICA is a chaining-hash-table-based key-value store that uses bulk-chaining to reduce pointer chasing during its search [32]. In the hash table each bucket is a linked list, in which each node contains seven pointers that fills an entire cache line. It also leverages load-balancing and offloading features provided by advanced NICs to achieve high throughput over high performance network [40]. In this experiment we first remove the networking component from MICA to evaluate SLB's impact on MICA's core index data structure.

MICA by default allows concurrent reads and exclusive writes (CREW) to the table. MICA uses a versioning mechanism to eliminate locking for concurrent read operations. In the meantime, writers still need to use locks to maintain consistency of the store. The implication of employing lockless concurrency model for reads is that MICA's hash table cannot be resized when it grows. With a fixed hash table size, the average length of the chains at each bucket will increase linearly with the number of stored key-value items. Consequently the long chains can lead to significant false temporal locality. To shorten the long chains one might propose to allocate a very large number of buckets when the table is created. However, this may cause the items to be highly scattered in the memory, leading to false spatial locality even for a very small data set. This drawback makes MICA's performance highly sensitive to the number of key-value items in the table. In the experiments we set up three MICA tables with different number of buckets (2^{22}, 2^{23}, or 2^{24}). Accordingly the average length of the chains in the three tables are 4, 2, and 1, respectively.

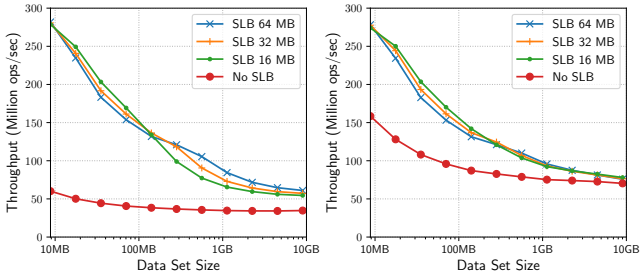
Figures 12b, 12c, and 12d show throughput of the three MICA configurations. MICA's throughput is higher with more buckets and thus shorter chains that help to reduce false temporal locality. In the meantime, SLB still improves their throughput by up to 56% even for the table whose average chain length is one (see Figure 12d). The reason is that the versioning mechanism in MICA requires two synchronous memory reads of a bucket's version number for each GET request. Synchronous reads can be much slower than regular memory reads even if the version number is already in the CPU cache.

4.3.3 MICA in the EREW mode. To further reduce the interference between CPU cores, MICA supports exclusive-read-exclusive-write (EREW) mode, in which the hash table is partitioned into a number of sub-tables, each exclusively runs on a core. As there is



(a) LMDB (b) MICA (2^{22} Buckets) (c) MICA (2^{23} Buckets) (d) MICA (2^{24} Buckets)

Figure 12: Throughput of LMDB and MICA using CREW mode. MICA is configured with three different table sizes.



(a) MICA (2^{22} Buckets) (b) MICA (2^{24} Buckets)

Figure 13: Throughput of MICA using EREW mode with 16 partitions.

no concurrent access to each sub-table, all costly protections for concurrency can be safely removed. We experiment on this mode where the SLB cache is also partitioned and its locks are also removed.

Figure 13 shows the throughput of MICA in the EREW mode with 16 partitions. The peak throughput of MICA with SLB can reach 281 MOPS, a 40% increase over its non-partitioned counterpart. For MICA of 2^{24} buckets, which has no false temporal locality, SLB can still improve its throughput by up to 95% (see Figure 13b) by removing the false spatial locality. This improvement suggests removing locking in the management of the SLB cache can further its performance advantage.

4.4 Performance of networked KV applications

While today's off-the-shelf networking devices can support very high bandwidth, SLB's performance advantage on reducing CPU cache misses becomes relevant for networked applications. For example, using three 200Gb/s Infiniband links [23] (24 GB/s × 3) can reach a throughput equal to the bandwidth of CPU's memory controller (76.8GB/s) [25]. With the ever increasing network performance, the performance of networked in-memory applications will become more sensitive to the caching efficiency. To reveal the implication of SLB on a real networked application, we port MICA of its CREW mode to Infiniband using IB_SEND/IB_RECV verbs API. We use a 100Gb/s (about 12GB/s) Infiniband link between two servers. We send GET requests in batches (2048 requests in each batch) to minimize the CPU cost on the networking operations.

Table 3: Hash table sizes after warm-up phase

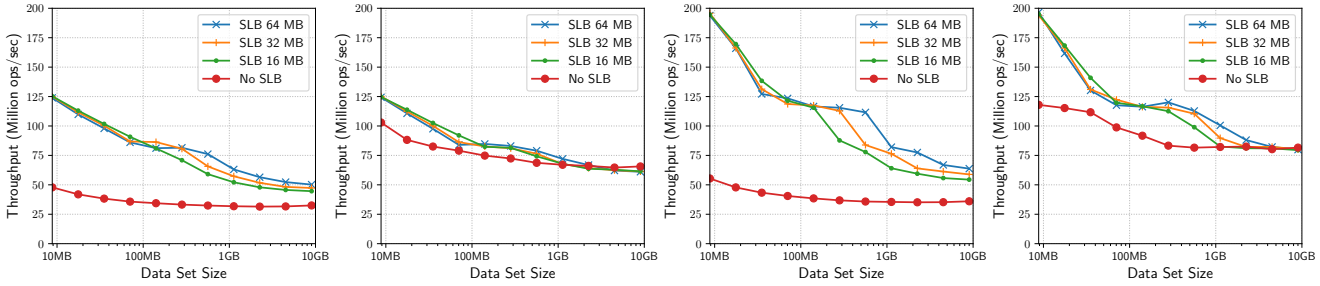
Trace Name	USR	APP	ETC	VAR	SYS
Table Size (GB)	9.6	63.8	84.3	6.2	0.08

Figures 14a and 14b show the throughput of MICA on the network. Compared to that without networking, the throughput of all configurations decreases and is capped at about 125 MOPS, as the network bandwidth becomes the bottleneck. For 64-byte values, each GET response contains 92 bytes including the value and associated metadata, and the 125 MOPS peak throughput of MICA with LSB is equivalent to 10.7 GB/s, about 90% of the network's peak throughput.

Attempting to reach the highest possible performance of the networked application, we minimize the network traffic by replacing each key-value item in the responses with a 1-byte boolean value indicating whether a value is found for the GET request. This essentially turns the GET request into a `PROBE` request. Figures 14c and 14d show the throughput for the PROBE requests on MICA with two different numbers of buckets. As the network bottleneck has been further reduced, the peak throughput recovers back to about 200 MOPS, almost the same as that of MICA without networking (see Figure 12d). In the meantime, most requests can be quickly served from cache and CPU is less involved in networking. However, the throughput drops quicker than that without networking. This is due to intensive DRAM accesses imposed by the Infiniband NIC which interfere with the DRAM accesses from the CPU.

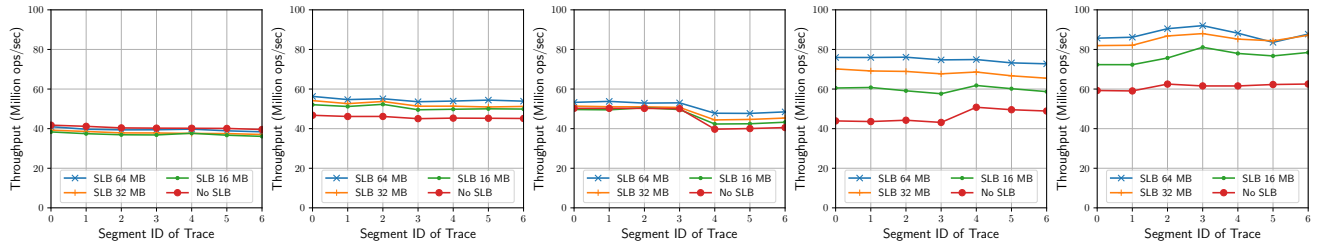
4.5 Performance with real-world traces

To study SLB's impact on real-world workloads, we replay five key-value traces that were collected on Facebook's production Memcached system [2] on an SLB-enabled chaining hash table. The five traces are USR, APP, ETC, VAR, and SYS, whose characteristics have been extensively reported and studied [2]. As the concurrency information is not available in the traces, we assign requests to each of the 16 worker threads in a round-robin fashion to concurrently serve the requests. We use first 20% of each trace to warm up the system and divide the remaining of the trace into seven segments to measure each segment's throughput. The hash table sizes after the warm-up phase are listed in Table 3.



(a) GET (2^{22} Buckets) (b) GET (2^{24} Buckets) (c) PROBE (2^{22} Buckets) (d) PROBE (2^{24} Buckets)

Figure 14: Throughput of MICA over a 100 Gb/s Infiniband.



(a) USR (b) APP (c) ETC (d) VAR (e) SYS

Figure 15: Throughput of Chaining hash table with five Facebook key-value traces.

Figure 15 shows throughput of the traces in each of their segments. The results are quite different across the traces. USR is a GET-dominant workload (GET ≥ 99%). It exhibits the least skewness compared with other traces—about 20% keys contribute to 85% of accesses. Although this is still a skewed workload, its working set can be much larger than CPU's cache size. As a result, SLB is hard to reduce its cache miss ratio. Accordingly SLB can hardly improve its throughput.

APP and ETC have much more skewed accesses than USR. In APP, 10% of the keys contribute to over 95% of the accesses. In ETC, 5% of the keys contribute to over 98% of the accesses. However, there two traces include about 10%–30% DELETE operations in their segments, which subsequently increases the miss ratio in the SLB cache. Misses in the SLB cache leads to slow index searches, which cannot be removed by SLB. For these two traces SLB increases the throughput by up to 20%.

VAR and SYS mainly comprise GET and UPDATE operations. They have high skewness and relatively small working sets that can be identified by the SLB cache and kept in the CPU cache. As a result, SLB improves their peak throughput by up to 73% and 50%, respectively.

The experiments with Facebook traces show that the effectiveness of SLB mainly depends on the skewness of the workloads and the size of the hot data set, rather than the total size of the index.

5 RELATED WORK

With intensive use of indices in in-memory computing, studies on optimizing their data structures and operations are extensive, including improvements of index performance with software and hardware approaches, and reduction of index size for higher memory efficiency.

5.1 Software approaches

There are numerous data structures developed to organize indices, such as regular hash table using buckets (or linked lists) for collision resolution, Google's sparse and dense hash maps [17], Cuckoo hashing [42], Hopscotch hashing [21], variants of B-tree [6], as well as Bitmap Index [8] and Columnar Index [31].

To speed up index search, one may reduce hops of pointer chasing in the index, such as reducing bucket size in hash tables or number of levels of trees. However, the approach usually comes with compromises. For example, Cuckoo hashing uses open addressing to guarantee that a lookup can be finished with at most two bucket accesses. However, Cuckoo hashing may significantly increase insertion cost by requiring possibly a large number of relocations or *kickouts* [49].

A tree-based index, such as B^+-tree, may reduce the depth of a tree and therefore the number of hops to reach a leaf node by employing a high fanout. However, wider nodes spanning a number of cache lines would induce additional cache misses. Masstree employs a prefix-tree to partition the key-values into multiple B^+-trees according to their key-prefixes [36]. This can reduce the cost of key comparisons with long keys. However, B^+-tree is still used in each partition to sort the key-values and the false localities in the index cannot be removed. Complementary to the techniques used by Masstree, SLB identifies the hot items in an index to further reduce the overhead on accessing them.

Specifically in the database domain efforts have been made on software optimizations for specific operations on indices, such as those on hash join algorithms to reduce cache miss rates [29, 35] and to reduce miss penalty by inserting prefetch instructions in the hash join operation [9]. These efforts demand extensive expertise on algorithms for executing corresponding database queries and their effectiveness is often limited on certain index organizations [19]. In contrast, SLB is a general-purpose solution that requires little understanding on the index structures and algorithms on them.

5.2 Hardware approaches

Other researches propose to accelerate index search with hardware-based supports. These can be either designing new specialized hardware components [19, 30, 37], or leveraging newly available hardware features [10, 16, 54, 56]. Finding that "*hash index lookups to be the largest single contributor to the overall execution time*" for data analytics workloads running contemporary in-memory databases, Kocberber et al. proposed Widx, an on-chip accelerator for database hash index lookups [30]. By building specialized units on the CPU chip, this approach incurs higher cost and longer design turn-around time than SLB. In addition, to use Widx programmers must disclose how keys are hashed into hash buckets and how to walk on the node list. This increases programmers' burden and is in a sharp contrast with SLB, which does not require any knowledge on how the search is actually conducted.

To take advantage of capability of supporting high parallelism, researchers proposed to offload index-related operations to off-CPU processing units, such as moving hash-joins to network processors [16] or to FPGAs [10], or moving index search for in-memory key-value stores to GPUs [56]. Recognizing high cache miss ratio and high miss penalty in the operations, these works exploit high execution parallelism to reduce the impact of cache misses. As an example, in the Mega-KV work, the authors found that index operations take about 50% to 75% of total processing time in the key-value workloads [56]. With two CPUs and two GPUs, Mega-KV can process more than 160 million key-value requests per second. However, to achieve such a high throughput, it has to process the requests in large batches (10,000 requests per batch). Furthermore, the latency of each request is significantly compromised because of batching. Its minimal latency is 317 microseconds in Mega-KV, much higher than that in a CPU-based store—only 6-27 microseconds over an RDMA network [51]. For workloads with high access locality, SLB can make most requests serviced within the CPU cache. In this way, SLB is expected to achieve both high throughput and low latency without requiring specialized hardware support.

5.3 Reducing index size

Large-scale data management applications are often challenged with excessively large indices that consume too much memory. Major efforts have been made on reducing index sizes for database systems and key-value stores. Finding that indices consume about 55% of the main memory in a state-of-the-art in-memory database (H-Store), researchers have proposed dual-stage architectures to achieve both high performance and high memory efficiency [55]. It sets up a front store to absorb hot writes. However, it does not help with read performance. To improve Memcached's hit ratio, zExpander maintains a faster front store and a compact and compressed backend store [52]. However, access of compressed data will use CPU cycles and may pollute the cache. In contrast, SLB reduces CPU cache miss ratio by improving caching efficiency with removed false localities.

A fundamental premise of these works is the access skew typically found in database and key-value workloads. In the workloads, there is a clear distinction of hot and cold data items and the corresponding locality is relatively stable [2, 12, 48]. This property has been extensively exploited to manage buffer for disks [14, 26, 47], to compress cold data in in-memory databases [15], and to construct and manage indices or data items in a multi-stage structures [27, 52, 55]. As use of any caches does, SLB relies on existence of temporal access locality in its workloads to be effective. Fortunately, existing studies on workload characterization and practices on leveraging the locality all suggest that such locality is widely and commonly available.

6 LIMITATIONS

Search Lookaside Buffer improves index lookup efficiency by removing the false temporal locality and false spatial locality in the process of index traversal and exploiting true access locality. For an application that uses index data structures, there are several factors that may impact the overall benefit of using the SLB cache. Here we list three possible scenarios where SLB produces only limited improvements on applications' performance.

- For index data structures that have been highly optimized, such as some hash table implementations, there are not substantial false localities. As a result, there is limited space for SLB to improve the lookup efficiency.
- SLB's effectiveness depends on skewness of workload access pattern. For workloads with weak locality, SLB has less opportunity to improve the cache miss ratio.
- When indices are used to access large data items, only a fraction of data access time is spent on index lookup. The program's performance improvement due to the use of SLB can be limited even when the index lookup time is significantly reduced.

7 CONCLUSION

In this paper we describe Search Lookaside Buffer (SLB), a software cache that can accelerate search on user-defined in-memory index data structures by effectively improving hardware cache utilization. SLB uses a cost-effective locality tracking scheme to identify hot items on the index and caches them in a small SLB cache to remove false temporal and false spatial localities from index searches. Extensive experiments show that SLB can significantly improve search efficiency on commonly used index data structures, in-memory key-value applications, and a high performance key-value store using 100 Gb/s Infiniband. Experiments with real-world Facebook key-value traces show up to 73% throughput increase with SLB on a hash table.

ACKNOWLEDGMENTS

We are grateful to the paper's shepherd, Dr. Amar Phanishayee, and anonymous reviewers who helped to improve the paper's quality. This work was supported by US National Science Foundation under CNS 1527076.

REFERENCES

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces* (0.91 ed.). Arpaci-Dusseau Books.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. DOI: https://doi.org/10.1145/2254756.2254766

[3] B+-tree 2017. B+-tree. https://en.wikipedia.org/wiki/B%2B_tree. (2017).

[4] Masanori Bando, Yi-Li Lin, and H. Jonathan Chao. 2012. FlashTrie: Beyond 100-Gb/s IP Route Lookup Using Hash-based Prefix-compressed Trie. *IEEE/ACM Trans. Netw.* 20, 4 (Aug. 2012), 1262–1275. DOI: https://doi.org/10.1109/TNET.2012.2188643

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. DOI: https://doi.org/10.1145/2485922.2485943

[6] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '70)*. ACM, New York, NY, USA, 107–141. DOI: https://doi.org/10.1145/1734663.1734671

[7] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*.

[8] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*. ACM, New York, NY, USA, 355–366. DOI: https://doi.org/10.1145/276304.276336

[9] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst.* 32, 3, Article 17 (Aug. 2007). DOI: https://doi.org/10.1145/1272743.1272747

[10] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 261–272. DOI: https://doi.org/10.1145/2485922.2485945

[11] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Technical Report. DTIC Document.

[12] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1942–1953. DOI: https://doi.org/10.14778/2556549.2556575

[13] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 15–28. DOI: https://doi.org/10.1145/1629575.1629578

[14] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-optimized Database. *Proc. VLDB Endow.* 7, 11 (July 2014), 931–942. DOI: https://doi.org/10.14778/2732967.2732968

[15] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *Proc. VLDB Endow.* 5, 11 (July 2012), 1424–1435. DOI: https://doi.org/10.14778/2350229.2350258

[16] Brian Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. 2005. Accelerating Database Operators Using a Network Processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN '05)*. ACM, New York, NY, USA, Article 1. DOI: https://doi.org/10.1145/1114252.1114260

[17] Google Sparse Hash 2017. Google Sparse Hash. http://github.com/sparsehash/sparsehash/. (2017).

[18] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 417–428. DOI: https://doi.org/10.1145/2254064.2254114

[19] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. 2012. Vector Extensions for Decision Support DBMS Acceleration. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 166–176. DOI: https://doi.org/10.1109/MICRO.2012.24

[20] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming.* Morgan Kaufmann.

[21] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *International Symposium on Distributed Computing*. Springer, 350–364.

[22] hugepages 2017. HugeTlbPage. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt. (2017).

[23] Infiniband HDR 2017. 200Gb/s HDR InfiniBand Solutions. https://goo.gl/z6Z1xc. (2017).

[24] InnoDB B-tree 2017. Physical Structure of an InnoDB Index. https://goo.gl/mHnpFb. (2017).

[25] Intel Xeon E5 2017. Intel(R) Xeon(R) Processor E5-2683 v4. https://goo.gl/4Ls9xr. (2017).

[26] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. 2005. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4 (FAST'05)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1251028.1251036

[27] Song Jiang and Xiaodong Zhang. 2004. ULC: A File Block Placement and Replacement Protocol to Effectively Exploit Hierarchical Locality in Multi-Level Buffer Caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04) (ICDCS '04)*. IEEE Computer Society, Washington, DC, USA, 168–177. http://dl.acm.org/citation.cfm?id=977400.977997

[28] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. DOI: https://doi.org/10.14778/1454159.1454211

[29] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1378–1389. DOI: https://doi.org/10.14778/1687553.1687564

[30] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. DOI: https://doi.org/10.1145/2540708.2540748

[31] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 1177–1184. DOI: https://doi.org/10.1145/1989323.1989448

[32] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 429–444. http://dl.acm.org/citation.cfm?id=2616448.2616488

[33] lmdb 2017. Symas Lightning Memory-mapped Database. http://www.lmdb.tech/doc/. (2017).

[34] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. DOI: https://doi.org/10.1145/2445572.2445574

[35] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowl. and Data Eng.* 14, 4 (July 2002), 709–730. DOI: https://doi.org/10.1109/TKDE.2002.1019210

[36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196. DOI: https://doi.org/10.1145/2168836.2168855

[37] Rich Martin. 1996. A Vectorized Hash-Join. In *iRAM technical report*. University of California at Berkeley.

[38] Memcached 2017. Memcached - a distributed memory object caching system. https://memcached.org/. (2017).

[39] MemSQL 2017. MemSQL. http://www.memsql.com/. (2017).

[40] MICA source code 2017. MICA. https://github.com/efficient/mica/. (2017).

[41] MongoDB 2017. MongoDB for GIANT Ideas. https://mongodb.com/. (2017).

[42] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[43] Redis 2017. Redis. http://redis.io/. (2017).

[44] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. DOI: https://doi.org/10.1145/2501620.2501623

[45] SQLite 2017. In-Memory Databases - SQLite. https://sqlite.org/inmemorydb.html. (2017).

[46] SQLite B-tree 2017. Architecture of SQLite. https://goo.gl/5RaSol. (2017).

[47] Radu Stoica and Anastasia Ailamaki. 2013. Enabling Efficient OS Paging for Main-memory OLTP Databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13)*. ACM, New York, NY, USA, Article 7, 7 pages. DOI: https://doi.org/10.1145/2485278.2485285

[48] Radu Stoica, Justin J. Levandoski, and Per-Åke Larson. 2013. Identifying Hot and Cold Data in Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 26–37. DOI: https://doi.org/10.1109/ICDE.2013.6544811

[49] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, and S. Cao. 2015. MinCounter: An efficient cuckoo hashing scheme for cloud storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–7. DOI: https://doi.org/10.1109/MSST.2015.7208292

[50] Translation Lookaside buffer 2017. Translation lookaside buffer. https://goo.gl/yDd2i8. (2017).

[51] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 22, 11 pages. DOI: https://doi.org/10.1145/2807591.2807614

[52] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. 2016. zExpander: A Key-value Cache with Both High Performance and Fewer Misses. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 14, 15 pages. DOI: https://doi.org/10.1145/2901318.2901332

[53] xxHash 2017. xxHash. http://github.com/Cyan4973/xxHash/. (2017).

[54] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. DOI: https://doi.org/10.14778/2536206.2536210

[55] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1567–1581. DOI: https://doi.org/10.1145/2882903.2915222

[56] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow.* 8, 11 (July 2015), 1226–1237. DOI: https://doi.org/10.14778/2809974.2809984

[57] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. ACM, New York, NY, USA, 97–108. DOI: https://doi.org/10.1145/2535372.2535379