Hardware-Based Address-Centric Acceleration of Key-Value Store

Chencheng Ye†, Yuanchao Xu‡, Xipeng Shen‡, Xiaofei Liao†, Hai Jin†, Yan Solihin§
† National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
‡ North Carolina State University, Raleigh, North Carolina, USA
§ Computer Science, University of Central Florida, Florida, USA
{yecc,xfliao,hjin}@hust.edu.cn, {yxu47,xshen5}@ncsu.edu, Yan.Solihin@ucf.edu

***Abstract*—Efficiently retrieving data is essential for key-value store applications. A major part of the retrieving time is on data addressing, that is, finding the location of the value in memory that corresponds to a key. This paper introduces an *address-centric* approach to speed up the addressing by creating a shortcut for the translation of a key to the physical address of the value. The new technique is materialized with a novel in-memory table, STLT, a virtual-physical address buffer, and two new instructions. It creates a fast path for data addressing and meanwhile opens up opportunities for the use of simpler and faster hash tables to strike a better tradeoff between hashing conflicts and hashing overhead. Together, the new technique brings up to $1.4\times$ speedups on key-value store application Redis and up to $13\times$ speedups on some widely used indexing data structures, consistently outperforming prior solutions significantly.**

I. Introduction

Key-value store as an infrastructure is essential for cloud computing as it provides a high performance and scalable solution to materialize a simple yet flexible data model [1], [2], [3], [4], [5], [6], [7]. Cloud service providers have developed Redis [8], Memcached [9], and their variants for many uses in production environments, including queues, session store, content cache, and so on.

As the most critical performance deciding factor for key-value store, maximizing the speed of data retrieval for a given key remains the most pressing objective, essential for measures from cost reduction to response time minimization, overall throughput maximization, and so on.

Hardware caching for key-value store is a promising approach that has drawn much recent interest [10], [11]. It is prompted by the observations of non-uniform access frequencies or locality of items in key-value store [12], [13], [14], [15], [16], [17]. The essential ideas behind the existing proposals (e.g., HTA [10] and SDC [11]) are to build a key-indexed hardware cache to store frequently used or recently accessed values. We call them *value-centric* approach for their focus on caching values.

Although these proposals have demonstrated promising performance, they are subject to two major limitations. First, the record size (key and value) must not exceed a cache line [10], [11]. This limitation hinders the applications of these proposals on many practical uses where some records may exceed a single cache line. Studies [16], [14], [13], [17] have reported that the average value sizes of production workloads vary from 20 bytes to 800 bytes. Second, the main reliance on hardware cache creates the tension between the capacity and cost, and hence the limit on the ultimate benefits. As prior studies show [15], the data volume for key-value stores suggests that the cache needs to be much larger than what hardware can afford to preserve a sufficient hit rate.

This work proposes an *address-centric* approach, a new way to tackle the efficiency problem in key-value store. Two fundamental features of this new approach set it apart from the previous hardware-based proposals: (1) It caches *virtual-physical addresses* rather than *values* of records; (2) It uses a combination of off-chip table and on-chip buffer for caching. The two features address both limitations of the previous value-centric proposals. Unlike values, addresses have small fixed size; caching them rather than values makes this new approach avoid the limit on the size of a record. The combination with off-chip tables mitigates the capacity-cost tension faced by value-centric methods.

The address-centric approach emphasizes the reduction of addressing overhead—that is, the time to find the physical address of the correct record.

In a typical key-value store, an access to a record goes through some kind of indexing and data structure traversal, a process that consists of multiple address translations. As Figure 1(left) shows, before the record associated with a key is actually accessed, a hash function needs to translate the key to an integer, an indexing data structure then needs to translate the integer to the virtual address of the target record, and then the system translates the virtual address to the physical address of the record through the *Translation Lookup Buffer* (TLB) or page table walk. The latter two steps are repeated for each memory access needed to get to the target record. As Figure 1(right) shows, the translations and address finding incur excessive overhead, over 50% of the overall time of a popular key-value store application, Redis.

To give an intuition of the *address-centric* approach, Figure 2 shows a simplified design. The virtual-physical addresses of recently accessed records are stored in *system translation lookaside table* (STLT), an off-chip table indexed by keys

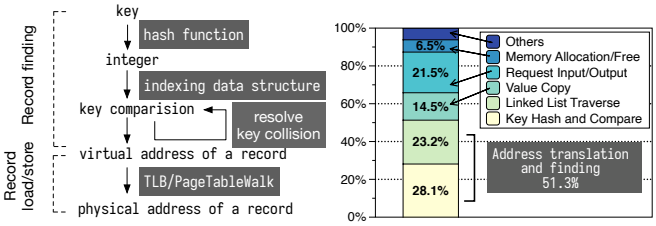


Fig. 1: Translations involved in an access to a record in key-value store (left). Breakdown of execution time of Redis (right). The results were collected using 10 million distinct keys and 100 million GET operations generated by YCSB. Redis indexes records with a hash table. Linux perf is used for measurement. To emulate the use of Redis in cloud centers that equip with fast interconnection, such as RDMA [18], [19], we use Unix domain socket as the network interface and Redis pipeline to batch requests.

whose size is not subject to area limits as an on-chip cache does. At the time of retrieving a record from a key, the runtime puts the corresponding entry in STLT into *system translation buffer* (STB), an on-chip fully associative cache with FIFO replacement. When loading the record, the runtime can then get the physical address of the record from either TLB or SLB without time-consuming page table walks. Besides saving the costly addressing overhead, this method also enables the use of much cheaper hashing functions on keys (detailed below). Both benefits could lead to large time savings.

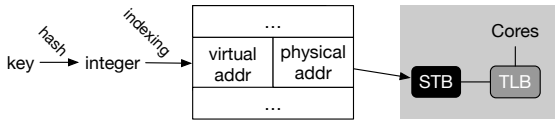


Fig. 2: Illustration of the basic idea of the *address-centric* approach

Putting this idea to work however faces several challenges: (1) how to design the STLT and STB to fit the needs and attributes of key-value store; (2) how to keep the coherence among the copies of record addresses in STLT, STB, TLB, and page tables with minimum time overhead; (3) how to keep consistency if records are moved; (4) how to deal with an application having multiple key-value store tables; (5) how to ensure security regarding both common flooding attacks[1] to key-value stores and the exposure of physical addresses of records. This paper presents the full design of the *address-centric* approach and how it addresses those challenges. The design leverages a statistical counter enhanced set-associative STLT and a software-hardware collaborative approach to resolve key queries, such that it can support keys and values of arbitrary sizes while managing physical address outside page tables in a programmer transparent way. The design employs hierarchical addressing translation and dynamic STLT switch-off to enable the use of cheaper hash functions without compromising the security. It also removes STLT updating from critical path of page swapping or migrating via lazy coherence between STLT and page table. The design handles a set of implications from moved records to multiple tables via a set of measures.

[1]Malicious keys made to share the same hashed value, causing many hash collisions [20].

It is worth mentioning that before this work, the idea of hardware-based caching addresses has been mentioned for key-value stores but has never been systematically explored. The SDC paper [11] for instance uses one sentence to mention the possibility of caching addresses rather than values; SLB [21] employs a software cache for record addresses, but without exploring architectural support, complexities to coherence, security implications, OS support, implications to hashing function designs, or treatments to multiple tables or record movements. To the best of our knowledge, this work is the first work that proposes the concept of *address-centric* approach, provides a full architectural implementation with OS and other support, and explores the opportunities on the new tradeoffs between hashing complexities and overhead.

Qualitatively, the new approach addresses the applicability limitations of the state-of-the-art hardware caching methods in record size. Quantitatively, the new approach brings up to $1.4\times$ speedups on key-value store application Redis and up to $13\times$ speedups on some widely used indexing data structures, and consistently outperforms prior software caching method SLB by 23–73%.

II. Background

Finding a record in a key-value store involves many steps including hashing, traversal, and translations. First, a key is hashed to an integer value that is used to index a hash table, and then the hash table is looked up to obtain the virtual address of a record containing the key-value pair. Since multiple keys may map to the same hash table entry (i.e., collide), collision handling policy is needed. Figure 3 illustrates handling collision by chaining, where each hash table entry points to a linked list that can grow to store records that collide at the entry. In the example, records for keys user0001 and user0004 are chained in a linked list. To look up the record for user0004, three memory accesses are conducted: one to read the hash entry, another to read the first node in the list, and finally to read the user0004 record. Each of these memory accesses also involves a translation from virtual address to physical address.

Furthermore, hash table based key-value stores are vulnerable to a type of security attacks called hash-flooding denial-of-service attacks [20]. An attacker, based on analyzing the hash function, creates many colliding keys to create a long-linked list that is slow to look up. To mitigate these attacks, attack-proof hash functions, such as SipHash [20], are used. SipHash is the default hash function in Redis, Python, Rust, and many other software packages or languages. It provides better security at the expense of performance. Key-value stores use other complicated hash functions for similar purposes, for example, MongoDB uses MD5, and Aerospike

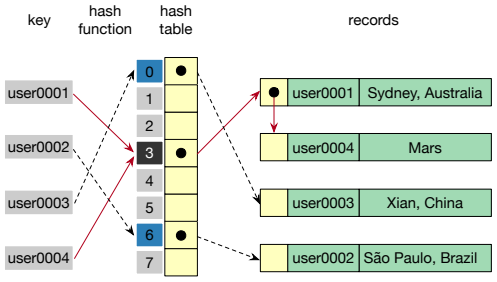


Fig. 3: An example of a chained hash table

uses RIPEMD (a family of cryptographic hash functions) for the indexing data structure.

Thus, overall, a hash table design used in key-value stores is generally optimized for ease in handling collision and attack resistance but was not designed for high performance common case.

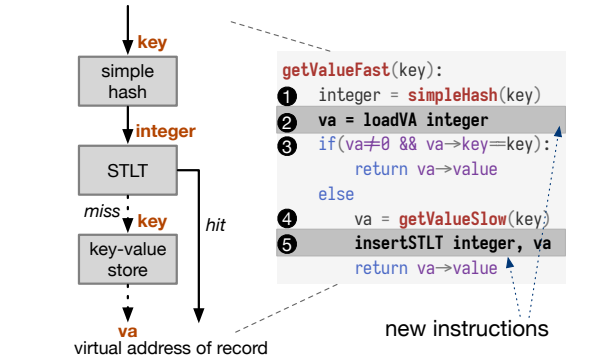
III. Address-Centric Design for Acceleration

This section presents the design of the address-centric method for accelerating key-value store operations.

A. Overview

A core data structure of the address-centric design is STLT, a table acting as a cache in kernel memory that accelerates record lookup. It is indexed by a hash table entry index. Each row contains the *virtual address* (VA) of a record and the *page table entry* (PTE) associated with the virtual address. The PTE contains the physical address (PA) of a page and its access permission. For the example in Figure 3, to look up record for user0004, STLT is indexed by a value "3", and its entry contains the VA/PA of user0004 record. Thus, given a hash entry, STLT allows us to skip hash table (and linked list) traversal and its associated address translations to directly obtain the addresses of the record. Thus, STLT acts both as an accelerator and as a cache for the key-value store. An STLT hit reduces multiple memory accesses due to hash lookup and traversal to just one. An STLT miss results in reverting back to the regular hash table lookup, hence a good hit rate is essential for performance improvement. Figure 4 the illustrates STLT use.

Every access to the key-value store ❶ hashes the key to an integer value through a simple hash function; then the integer is used ❷ to load an STLT entry via a new *loadVA* instruction. The instruction also inserts the VA-to-PA mapping into an on-chip buffer STB. In ❸, we validate the entry by checking whether the VA of a record is 0 (null pointer) and whether the record matches the input key. If the record matches, it is an STLT hit and the value of the record is found. On an STLT miss, the access looks up the key in a traditional manner involving the key-value store hash table ❹. The VA of the record is also obtained. Then, an *insertSTLT* instruction is executed that will insert the VA-to-PA translation into the STLT ❺, to increase the chance of future hits to this key.



insertSTLT reg_{in}(integer), reg_{in}(va)	find page table entry (PTE) associated with *va*, put *integer*, *va* and PTE into STLT
loadVA reg_{in}(integer), reg_{out}(va)	return *va*, load *va* and its PTE into STB (a near TLB buffer introduced later)

Fig. 4: Overview of STLT and pseudocode using it

STLT also provides the PA to the memory hierarchy, hereby adding to the *Translation Lookaside Buffer* (TLB) reach.

Because STLT is just a cache, it does not need to handle collision. A collision simply results in an STLT miss with the key passed to the key-value store for a lookup. Therefore, using an attack-resistant hash function is not critical for STLT, allowing us to rely on a simple and fast hash function. Hence, overall STLT improves performance in three ways: (1) enabling the use of a fast hash function to translate key to hash entry index, (2) allowing the retrieval of the record's VA given hash entry index, skipping hash and linked list traversal, and (3) skipping page table walk by providing PA of the record, in the event of TLB miss.

B. Hash Function

In contrast to the complex hash functions used in hash tables that must provide hash flooding attack resistance and avoid pathological cases of key collisions, a hash function in STLT can be designed to be fast for the common case. For our implementation, we use xxh3 hash function [22]. We also considered adding hardware support for calculating a fast hash function. A hardware hash gains performance at the expense of flexibility.

C. STLT Design

STLT is a set-associative cache stored in kernel memory. Since it stores not just VAs, but also PAs, it cannot be read or written directly by the application running in the user space, to avoid accidental overwrites. The user-level application only has a narrowly defined interface for interacting with the STLT, which includes *loadVA* instruction (to read only the VA of an entry), and *insertSTLT* instruction (a hint to insert VA/PA translation into STLT given the VA). Hardware handlers or the OS is responsible for reading or modifying the STLT (including looking up permission and managing PA), similar to page table walker or the OS manages the page table.

Each row of STLT has 16 bytes composed of four components depicted in Figure 5. The counter records the frequency

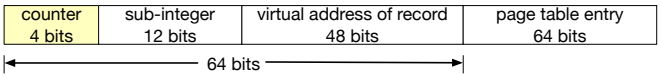


Fig. 5: A row entry of STLT.

of access to the row. The sub-integer is a few bits coming from the hash integer associated with the row. The last two fields contain the VA and PA of a key-value record.

STLT is dynamically sized (and resized as needed); its size must be a power of two. A system call is required to resize the STLT. STLT is also page aligned. STLT can be direct-mapped or set-associative (our default design). If set-associative, it maps an integer value of a hash table index to multiple adjacent candidate rows in a set. Figure 6 illustrates the mapping from a key to a set of rows. In the example, the hash function produces a 64-bit integer value. The STLT has 2^{10} rows and 4-way associativity, hence only 8 bits of the integer value are needed to index a set (*set number*), and we can choose any 8 bits from the integer. The maximum set associativity can be limited to avoid incurring two block accesses for a single *loadVA* instruction. Due to set associativity, we need to know which row in the set contains the desired entry, or whether any row in the set has the desired entry. To achieve that, we could add a full `64-8=56`-bit tag to each row, but that would create an unnecessary storage overhead. So instead, we add a 12-bit partial tag to each row which we refer to as the *sub-integer*. 12 bits are chosen for the sub-integer to avoid an STLT row to span beyond 16 bytes. If the tag matches with a 12-bit part of the integer value produced by the hash function, it indicates a potential hit[2], and the *loadVA* instruction returns the VA for the matching row. Since the tags are partial, it is possible in rare cases to find more than one matching rows in a set. In such a case, one matching row is randomly selected. In the example, we chose 12 *least significant bits* (LSBs) for the sub-integer and the 8-bit index is adjacent, but other choices are possible as long as the bits chosen for sub-integer and index do not overlap. The size of STLT is dynamic, so is the number of sets. If the STLT is enlarged, the set index bits can grow or shrink easily, while the sub-integer remains using the 12 LSBs.

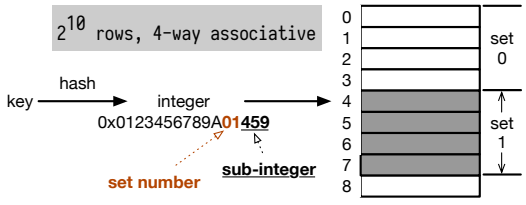


Fig. 6: Set associativity of STLT

For the rest of this section, for simplicity, we will illustrate *loadVA* and *insertSTLT* with a direct-mapped STLT (i.e., 1-way associative). We discuss set-associativity support in Section III-E.

[2]Software further validates if the returned VA is the correct one.

D. Hardware Support for STLT

The hardware implements instructions *loadVA* and *insertSTLT*. Figure 4 specifies the format of two new instructions that we introduced as the software interface to STLT, and Figure 7 depicts the modification to the processor pipeline, showing a new component *system translation unit* (STU) and a slight modification to the *memory management unit* (MMU). STU can be thought of as a specialized functional unit to execute *loadVA* and *insertSTLT* instructions.

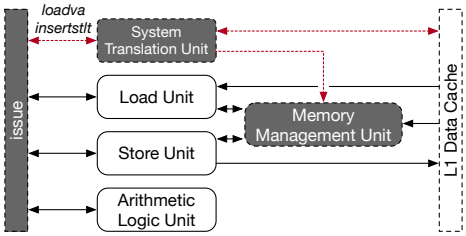


Fig. 7: Modification to core architecture

For *loadVA*, the source operand represents the integer value that is output of the hash function. It is executed in the STU. The STU calculates the effective address of the STLT set where the integer maps to, then accesses STLT by sending a memory request to L1 data cache. STLT is cacheable, hence the request may hit in the L1 data cache, or be forwarded to L2 cache, and so on. STU then buffers the row in an internal register within the unit and performs partial tag comparison to determine STLT hit or miss. If the sub-integer partial tag in the row matches the input integer, the STU writes the VA of the row into the destination register of the instruction. It also forwards the row to the MMU. The MMU treats this similar to a TLB hit: it skips the page table walk for the VA found in the STLT row and uses the PA found in the row.

For *insertSTLT*, one source operand is integer value produced by the hash function, and the other is VA of the key-value record. Both operands are passed to the STU, which calculates the row to be accessed according to the integer. The STU also obtains the PA of the record through the MMU (TLB or page table walk). Then, the appropriate STLT row is accessed, and a new entry is inserted into the table at the row.

From the viewpoint of memory consistency ordering, a *loadVA* or *insertSTLT* instruction is strictly ordered with respect to another *loadVA* or *insertSTLT* only when they have the same integer value. For example, if a preceding *insertSTLT* with integer x has not performed, a *loadVA* is stalled until the *insertSTLT* is performed. This avoids overlapping or reordering them. A *loadVA* or *insertSTLT* is ordered w.r.t. a regular load or store instruction executed by the application program only when the VA of the key-value record overlaps with the VA of a regular load or store, indicating dependence. In other words, the dependence is not checked against the effective address of the STLT row, but instead against the VA contained in the matching STLT row. Finally, *loadVA* or *insertSTLT* are executed by a single STU in hardware, hence atomic execution of each instruction is simple to achieve. With

respect to cache coherence, *loadVA* or *insertSTLT* generates a *get* or *getX* coherence request in order to get the cache line containing the STLT row, prior to reading from or writing to it, similar to a regular load or store instruction.

Now we will discuss the implementation of *loadVA* and *insertSTLT* in detail.

1) loadVA Implementation: Figure 8 illustrates how *loadVA* is implemented. We add the following on-chip hardware components: (1) a set of registers CR_S that keeps the base address and the size of STLT; (2) an *invalid page buffer* (IPB) containing recently invalidated *page table entries* (PTEs); and (3) a translation buffer (STB) containing the VA and PTE pairs.

In Figure 8, the CPU issues *loadVA* with an integer, which contains the STLT set number and the sub-integer. The STU in the CPU calculates the effective address of the set using CR_S and the index portion of the integer and scans the set to find a row that matches the sub-integer. The instruction returns 0 if no match is found or continues to check the VA against the IPB. If the VA is found in the IPB, it belongs to a recently invalidated PTE, hence a 0 is returned. Otherwise, the translation is valid, and the VA is returned to the CPU.

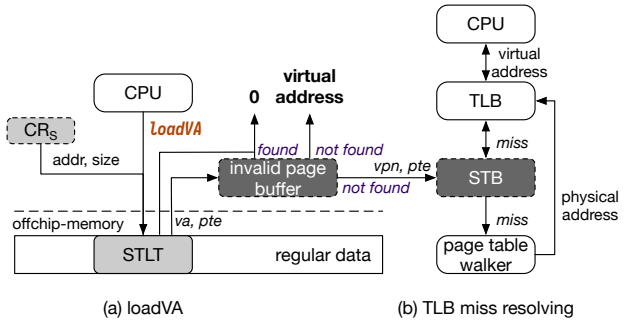


Fig. 8: Procedures of (a) load va and (b) TLB miss resolving

As illustrated by the pseudocode in Figure 4, the program accesses a record referred by the VA right after *loadVA* returns it. Figure 8(b) elaborates the address translation for the record access. If the memory access hits in the TLB, the TLB returns the PA to the processor without touching the STB. However, if the memory access misses in TLB, the hardware looks up the VA in the STB. STB is implemented as a fully associative cache with FIFO replacement, keeping a few most recently used VA/PA translation. There is no eviction for STB. If a match is found, the translation is inserted into the TLB. Otherwise, the page table walker resolves the address translation. The address translation applies to all memory accesses including both record and non-record addresses. We design the STB to be the same size as the load buffer to avoid the case that the item inserted into STB by *loadVA* is replaced before the memory access following *loadVA* access the STB. Particularly, STB is composed of 32 entries in our implementation.

As discussed earlier, one of the key performance bottlenecks is in address translation; given a VA, it can take a long time to get its PA through the page table. The TLB on modern architectures is usually complicated and is in the critical path of memory access [23]. Typically, a TLB lookup must be completed within the time to access the activate a row in the L1 cache and read the content of the row to the row buffer. This is because a typical L1 cache uses virtually indexed and physically tagged design. Hence, before L1 tag comparison, the TLB must have produced the PA for the accessed VA. Therefore, our design avoids any modifications to the TLB.

Instead, we design the STLT to accelerate key-value lookup by caching address translation and skipping page table walks. STLT keeps VA-to-PA translation information similar to a TLB. Thus, STLT provides an alternative to the TLB to obtain PA for a given VA that corresponds to a key-value record, without adding translation hardware. However, one challenge to address is the coherence between TLB, page table, and STLT. Without a coherence scheme, VA-to-PA translation may become out of date in the STLT if it is only modified in the page table and TLB. However, a coherence scheme may be expensive if it requires looking up and invalidating the STLT entry whenever VA-to-PA translation changes. Hence, we rely on *lazy* coherence scheme. Specifically, we introduce an *invalid page buffer* IPB that keeps recently invalidated *page table entries* (PTEs).

To manage IPB, we modify a set of Linux kernel TLB management functions, `flush_tlb_*`. The operating system always calls them before updating the page table. The functions wrap instruction `invlpg` on X86 architecture and invalidate one entry of TLB or the entire TLB. We insert three new instructions into the functions: (1) insert VA of the page into IPB, (2) clear the IPB, (3) check whether the IPB is full or not. Before executing `invlpg`, we check the capacity of IPB with instruction 3. If IPB is not full, the kernel function records with a kernel-space array the virtual address associated with the PTE to invalidate and inserts the virtual address into IPB with instruction 1. If IPB is full, the kernel function clears it with instruction 2 and updates STLT via searching the page table for invalidated PTEs. The array retaining invalidated virtual address is part of program context, such that on user process context-switch out, the operating system clears the IPB without updating the STLT, and on user process context-switch in, the operating system inserts all virtual addresses in the kernel-space array into IPB again with instruction 1. The IPB retains up to 32 virtual addresses. It is a fully associative cache with FIFO replacement implemented with content addressable memory. Because page invalidation is expensive yet rare, a small IPB is sufficient for most cases. Updating STLT is more expensive. However, it is even rarer than page invalidation.

2) insertSTLT Implementation: insertSTLT finds the PTE of a given VA and inserts it into the STLT. Its implementation relies on an insertion buffer and a *simplified page table walker* (SPTW). The 16-entry buffer stores outstanding stores. Each entry is composed of an STLT row illustrated by Figure 5 and the memory address to store the entry. The value of the counter is 0. The simplified page table walker utilizes the page table walker of the processor except that when it encounters a page fault, SPTW returns 0 as the PTE instead of triggering a hardware interrupt. This is appropriate because STLT is

designed to be fast and acts just as a cache, hence there is no need to handle a page fault in order to insert an entry. Therefore, *insertSTLT* in this case acts just like a hint that is ignored by the hardware.

Figure 9 elaborates the overview of the implementation of *insertSTLT*. The circled number depicts dataflow. The CPU issues *insertSTLT* and calculates the effective address of STLT set to be written according to the integer, taking into account the base address and size of STLT. The insertion buffer holds the effective VA and the written address. The CPU then propagates the VA to SPTW and the SPTW returns either the PTE or 0, which implies a page fault. If the SPTW returns 0, the execution is complete. Otherwise, the insertion buffer writes the virtual address and PTE into STLT.

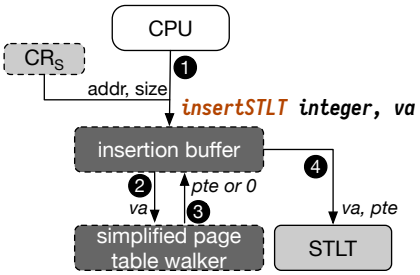


Fig. 9: Implementation of *insertSTLT*.

A problem of *insertSTLT* is concurrent writes when multiple cores simultaneously issue *insertSTLT*. Because the insertion buffer updates 16 bytes, on architectures support atomic write for 16 bytes[3]. For those architectures, the insertion buffer writes atomically the 16 bytes data into STLT. For other architectures, the programmers should provide the atomicity in software, for example, wrapping *insertSTLT* instruction with a lock or a hardware or software translation.

E. Set-Associative STLT

Direct-mapped STLT has two problems: (1) high conflict miss rate when the STLT is small, which occurs when two frequently accessed keys map to the same STLT row; and (2) low space utilization when STLT large, which is caused by random distribution of hash function, for example, mapping n items to an n-row STLT yields in $\frac{1}{e} \approx 36.8\%$ rows unoccupied while another $\frac{e-2}{e}$ rows shared by at least two keys[4]. Set-associativity solves both problems.

Consider the implementation of set-associativity of STLT for instructions *loadVA* and *insertSTLT*. For *loadVA*, the hardware scans the whole set of rows to find a row of which the sub-integer matches the input integer. The hardware then loads the row and updates the counter. *insertSTLT* finds a row and replaces it with the appropriate input VA and PTE. If it is not found, the instruction replaces the least frequently accessed row according to the counter.

[3]Intel and AMD provide atomic compare-and-switch operations for 16 bytes aligned data. However, atomicity for other operations for 16 bytes data is design specific.

[4]Suppose STLT uses a random hash function, the space utilization problem is reduced to a balls and bins problem [24].

To scan an n-way associative set, the hardware breaks *loadVA* into multiple memory operations, which depends on the data path width of the processor. For example, Intel Skylake and later architectures provide up to 512-bit back-end ports to support AVX-512 instructions [25], while AMD's early architecture K8 decodes `movaps` instruction to two micro-operations accessing the memory. The counterpart of AVX-512 for ARM is Neon, for RISC-V is a planned feature, RISC-V vector. HTA [10] implements similar scan function but without a counter to support the row replacement.

Both *loadVA* and *insertSTLT* modify the STLT. *loadVA* updates the counter on hits and *insertSTLT* always updates a row. Rather than modifying the whole set, *loadVA* modifies 4 bits while *insertSTLT* modifies 16 bytes. Thus, the data path widths between system table unit and memory vary on direction, from L1 data cache to the unit or vice-versa.

Because STLT is cache-line aligned, when $n \leq 4$, all rows of a set are in the same cache line, such that even the architecture without AVX instruction supports (RISC-V for example) has to break *loadVA* into multiple memory operations. The operations tend to incur only one L1 data cache miss. In contrast, when $n > 4$, a set spans over more than one cache line, hence it may cause more than L1 data cache miss on each STLT lookup.

The counter of a row is 4 bits as elaborated by Figure 6. To prevent the counter from overflowing fast, we use a probabilistic increasing strategy. On counter update, let the value of the counter be x, the hardware generates a random number less than 2^x. If the number is 0, the counter increases by 1, otherwise, it remains the same. The hardware generates the random number ahead of time; thus it is almost free. With the strategy, a counter overflows after 2^{17} updates on average. The consequence of the overflow is benign, such that it causes frequent accessed rows to be replaced and may harm the performance rather than the correctness of the program.

F. Other Implementation Issues

Operating System Support We allocate the STLT in the kernel space to avoid user-space load and store instructions from accidentally reading from or writing to it. The OS and hardware handlers manage STLT with the application providing input for its creation and resizing, using the following system calls:

`STLTalloc(int n)`	create an STLT of *n* rows
`STLTresize(int n)`	resize STLT to *n* rows
`STLTfree()`	deallocate STLT

Every process can have at most one STLT. `STLTalloc` allocates contiguous memory for STLT and updates register CR_S with the physical address of STLT and the size. `STLTresize` adjusts the size of STLT and clears the content of STLT as the hash function the application uses is unknown to OS. To provide flexibility on performance tuning, our design allows the key-value store user to monitor STLT miss ratio and tune the performance factors, such as space overhead, improvement in performance, or worst-case query latency.

Performance guarantee. STLT may harm performance if its hit ratio is very low, which could be caused by the STLT

being too small or poor locality presented in key access pattern. To ensure that the performance of STLT-enhanced key-value store is at least equal to the original key-value store, we may use runtime performance monitoring by periodically turning STLT on and off to determine whether or not STLT improves the performance, combined with resizing when the hit rate is too low.

Accelerating beyond hash table. STLT is applicable to indexing data structures beyond just hash tables, such as B-tree or black-red tree, as long as those structures have the same semantic as the hash table, i.e., they take a key as input and output the record matching the key. The implementation would be the same, as illustrated by the pseudocode in Figure 4, where the *getValueSlow* function uses any of those indexing data structures. Section IV demonstrates the evaluation for two implementations of hash table and two other indexing data structures.

Support for multiple indexing data structures. An application can have only one STLT. If the application has multiple indexing data structures that need to be accelerated, they can share the STLT. A challenge that needs to be overcome is that there may be key aliasing between them, i.e., using the same keys to point to different records.

To remove key aliasing, the programmer can manipulate the integer before it is used as input to *loadVA* and *insertSTLT*. First, the programmer assigns each indexing data structure a unique ID. Then, the input integer can be changed by replacing the last bit(s) of the sub-integer with the ID, which creates a globally unique integer value, as illustrated in Figure 10. This new integer can then be used for *loadVA* and *insertSTLT*.

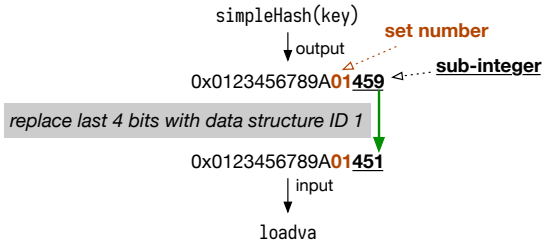


Fig. 10: Integer manipulating for shared STLT.

Moving records. The key-value store may move records, for example, to increase the size needed by a record. Since the PA of the record changes, the STLT row associated with the record needs to be invalidated in response to the movement. To update STLT, the programmer issues *insertSTLT* once the movement finishes. To avoid a situation where a thread calling *loadVA* while other threads moving the targeted records, programmers need to ensure that the *loadVA* and recording movement happen exclusively, for example, by using locks.

G. Cost of STLT

STLT incurs three types of cost: on-chip storage overhead, off-chip storage overhead, and software overheads.

a) Hardware Cost: Our hardware support totals less than 1KB of on-chip structures, assuming virtual address of 48 bits and the page size of 4KB, as detailed in Table I. Register CR_S keeps a 36-bit physical address that is the base address of the page-aligned STLT plus a 64-bit STLT size. The *invalid page buffer* (IPB) has 32 entries and a 6-bit counter. Each entry of the IPB contains a 36-bit virtual address of a page (i.e., virtual page number). The *system translation buffer* (STB) also contains 32 entries. Each STB entry contains a 64-bit VA and a 64-bit PTE. The insertion buffer contains 8 entries, each containing a 64-bit VA, a 64-bit PTE, and a 44-bit PA. The total number of bits is minor, less than 1KB. To put it in perspective, typical L1 data cache size per core is 64KB in size. The logic overhead is also minor, only one functional unit (STU) is added, and it is used relatively infrequently compared to a regular functional unit. The simplified page table walker also makes use of the existing page table walker, only adding the enable exception bit.

TABLE I: Hardware space overhead for STLT

Component	Cost (bits)	Detail
CR_S	64	STLT address and size
Invalid page buffer	1,158	32 entries, a 6 bits counter
STB	4,096	32 entries
Insertion buffer	1,376	8 entries
Total: 837 bytes (6,694 bits)		

b) Software Cost (Memory Consumption): The off-chip space overhead is due to STLT data structure in kernel memory (no hardware structure is involved). The size varies from zero (for applications that do not need acceleration) to MBs – hundreds of MBs, depending on the application's needs. Such cost is affordable as it is a small fraction of a typical key-value application working set, incurred only by applications needing acceleration, and it yields substantial performance improvement. The trade-offs between STLT size and performance are presented in Section IV.

c) Software Cost (Code Modification): We add 12 lines of code to apply STLT on a popular key-value store, Redis, to insert pseudocode shown in Figure 4 to the function that searches the hash table. For the other four kernel benchmarks evaluated, we modify only six lines of code, which overrides the search of the indexing data structures. An optimization may modify the insertion function as well to ensure a most recently inserted records also presents in STLT. However, we find that STLT substantially accelerates indexing data structures as reported in Section IV, thus we do not modify insertion functions.

H. Security Impact

In this section, we discuss whether STLT introduces new security vulnerabilities or makes existing security vulnerabilities worse. First, although our proposed method stores PA in the STLT, PA is never exposed to the user-level application. In addition, the OS allocates STLT in the kernel space, hence user-space load or store instructions cannot read from or write to STLT either intentionally or accidentally. Applications can only access and manage STLT through two instructions, *loadVA* and *insertSTLT*, and a set of system calls. The input

and output of the two instructions are integers and virtual addresses, and hence they do not expose the PA or allow user-level application to modify the PA.

Second, STLT does not increase vulnerabilities to hash flooding attacks, even though it allows the use of simple hash functions. The reason is that upon hash collisions, STLT would redirect the execution to the default (slow) path. In the worst case of a flooding attack, STLT could see an STLT miss on every key-value store request, which adds at most bounded constant performance overhead. However, with runtime performance monitoring, which dynamically enables and disables STLT and compares execution time, the runtime system would disable STLT in that scenario. STLT hence even the constant performance overhead could be removed.

IV. Evaluation

This section evaluates the performance of the proposed STLT and provides sensitivity studies on the design, including the size and set associativity of STLT, the choice of hash tables, and the workload distribution.

A. Methodology

Benchmarks and Dataset. We evaluate STLT on Redis, a popular key-value store application. In addition, we give a deeper evaluation of the addressing efficiency of STLT by applying it to four widely used indexing data structures that are of production-level implementation. Table II lists them all.

TABLE II: Programs used in evaluation

Program	Description
Redis-5.0.7	Production key-value store implemented in C[5]
unordered_map	Default hash table in C++ Standard Library of GCC 7.5.0
dense_hash_map	Hash table by Google in C++[6]
ordered_map	Self-balancing red-black tree in C++ Standard Library of GCC 7.5.0
btree	Btree implemented by Google in C++[7]

Data retrieving efficiency is the focus of the performance measurements on Redis. Other components of Redis (e.g., data fetching from network sockets, data validation, conversions between input/output commands and their internal representations) are excluded from the measurement. Speeding those components is not the goal of STLT, but the focus of other complementary techniques, such as RDMA [18], [26], and NIC-based techniques [27]. With them, the other operations of Redis can be significantly shortened (e.g., RDMA reduces the latency of GET in Redis from 203μs on Ethernet to 15μs).

We use YCSB [12] to generate the workloads for all benchmarks. The workloads have 10 million keys and 100 million key accesses. The keys are 24 bytes. We generate nine workloads as all the combinations of three value sizes and three key access distributions. The value sizes are 64, 128, and 256 bytes. The distributions are `zipf`, `latest`, and `uniform`. The alpha value of the `zipf` distribution is 0.99. Prior works uses these distributions on key-value stores [28], [29], [30], [31], [32]. Workloads with the `latest` distribution tend to access the latest inserted keys. Workloads with `uniform` distribution accesses every record with an even probability. The workloads are all GET operations except for workloads with `latest` distribution, of which 5% of operations are SET operations. Without noting otherwise, the reported results are on the `zipf` distribution and the value size is 64 bytes.

Hardware Simulation. We use an interval simulation-based timing accurate hardware simulator, SniperSim [33], in the evaluation. It uses Pin [34] as the frontend. We modified SniperSim to add TLB miss resolution and page table walk. The data cache caches data as well as page table entries, as modern architectures do. Table III reports the simulated architecture.

TABLE III: Simulated architecture

Component	Parameter
ISA	64-bit X86, Gainestown architecture
CPU	1 core, 2.66Ghz
L1 data TLB	4-way, 64 entries, 1 cycle
L2 shared TLB	4-way, 1536 entries, 7 cycles
L1 data cache	8-way, 64 entries, 4 cycles
L2 cache	8-way, 256KB, 12 cycles
L3 cache	8-way, 2MB, 40 cycles
Cache line	64 bytes
Memory	45 nanoseconds, 4KB page size
instructions	
loadVA	6 cycles + an STLT set load + 4 bits store
insertSTLT	4 cycles + simplified page table walker + 16 bytes store

For all benchmarks, we use 80% of the key accesses to warm up the cache and STLT, then simulate 128 thousand key accesses.

To model the latency of new instructions, we divide the latency into two parts: the latency of memory accesses and the functional operations other than memory accesses. The hardware may resolve the memory accesses within cache or memory; thus the latency varies significantly. We insert `load` and `store` instructions to simulate the latency of memory accesses. The functional operations have stable latency as the operations are fixed. We derive the associated latency by implementing the function in software, measuring the latency of the software implementation, and using the latency as the hardware latency. The latency estimate is conservative as the latencies we assume reflect fully exposed non-overlapped execution of the instructions. In modern processors, instruction fetch, decode, and execution are overlapped using *instruction-level parallelism* (ILP) techniques.

Comparison Counterparts. We use the default version of the benchmarks without STLT as the baseline. We in addition include *search lookaside buffer* (SLB) [21], a previous work in the comparison. There are some other recent proposals on speeding up data retrieving in key-value store [10], [11], but they are subject to the limitations on record size (no greater than a cache line). SLB stands for the state of the art that

[5]https://redis.io/

[6]https://github.com/sparsehash/sparsehash

[7]https://code.google.com/archive/p/cpp-btree/wikis/UsageInstructions.wiki

supports flexible record sizes as STLT is designed for. SLB uses a software cache that stores the virtual addresses of recently accessed records. It does not bypass page table walk. Therefore, by comparing to SLB, the evaluation shows the gains brought by STLT by bypassing page table walks.

More specifically, SLB contains two tables, cache table and log table. Cache table retains the virtual addresses of most frequently accessed records. It is 7-way associative. The log table retains the access frequencies of records, four times as large as the cache table. For fair comparisons, the same hash function is used for SLB and STLT (on its fast path).

Unless noted otherwise, all evaluations of STLT use 4-way associativity; Section IV-D2 gives the sensitivity study on other associativities.

Hash function. Table IV lists all the hash functions used in the evaluation. Function `sipHash` is the one used in the original Redis, and `murmurHash` is the default hash function in the original version of the other four programs in Table II. After STLT is applied, the slow path retains the same hash function as used in the original benchmark, while its fast path, by default, uses `xxh3` as the hash function. Section IV-D3 shows the performance of STLT when the fast path uses the other hash functions Table IV lists.

TABLE IV: Hash functions used in the evaluation

Hash Function	Description
sipHash	default hash function of Redis, python, and rust.
murmurHash	default of kernel benchmarks, C++ and Java.
xxh64	64 bits xxh hash fast non-cryptographic hash
djb2	hash function specific for string
xxh3	variation of xxh64

B. Speedup on Redis

Figure 11 reports the speedup of Redis brought by SLB and STLT on nine workloads. These workloads are of three distributions and the three workloads of one distribution have record sizes set to 64B, 128B, and 256B respectively. In these experiments, the STLT table is allocated 512MB space (in 4-way associativity), and the cache and log tables in SLB use 10GB space.

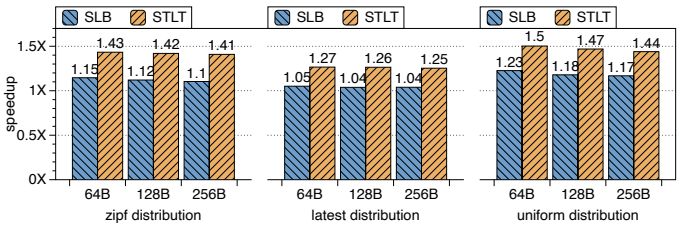


Fig. 11: Speedups brought by STLT and SLB on Redis on workloads of different record sizes and distributions

Table V lists the miss rates on STLT tables and SLB tables. Despite the 20× larger space usage, SLB has only slightly smaller miss rates. As STLT distinctively tackles address translations and TLB misses, it overall brings much larger reduction of TLB and cache misses and creates greater speedups as Figure 11 reports.

TABLE V: STLT and SLB miss rate

Distribution	SLB	STLT
zipf	1.42%	1.75%
latest	0.30%	0.85%
uniform	7.47%	3.61%

On average, STLT brings 1.38× speedups. It consistently outperforms SLB substantially. The higher speedups come from the larger reduction of TLB misses and cache misses from STLT as shown by Figure 12. The TLB miss reductions by STLT are between 27% and 31%, while the reductions are -2.6% (increases) to 10% by SLB. STLT also shows significant advantages in reducing data cache misses: 5-12% by STLT versus -3–3.7% by SLB.

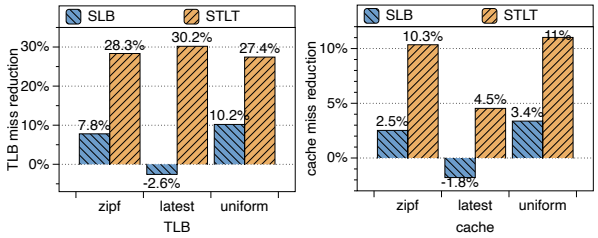


Fig. 12: TLB miss and cache miss reduction, value size is 128B.

Record size has little effect on both STLT and SLB. Experiments with 64B and 256B values show similar TLB or cache miss reduction with less than 1% difference. Workload distribution has a larger impact. In general, the benefits of STLT are more pronounced when the requests have less locality. Among the three distributions, `latest` has the best data locality, `zipf` modest, and `uniform` the worst. Workloads of `uniform` and `zipf` show much larger speedups than `latest` workloads show. It is intuitive because less locality entails higher numbers of TLB and cache misses and hence a larger potential room for improvement from using STLT.

C. Speedup on Kernel Benchmarks

On the other four kernel benchmarks, STLT also outperforms SLB substantially. Figure 13 shows speedups of two methods for six workloads (three distributions and two record sizes: 128B on top, and 256B at the bottom). The results on 64B records have the same trend, omitted in the interest of space.

For hash table based kernel benchmarks, `unordered_map` and `dense_hash_map`, SLB gains 1.70× speedup on average while STLT gains 42.3% more, reaching 2.42× speedup. For the same reason mentioned in the previous subsection, STLT gives the largest gains ($2.6 - 2.9\times$) on `zipf` and `uniform` distributions, and fewer speedups (1.7×) on the `latest` distribution. For tree-based kernel benchmarks, `map` and `btree`, SLB gains 6.46× speedup on average while STLT gains another 73% more, reaching as much as 11.2×.

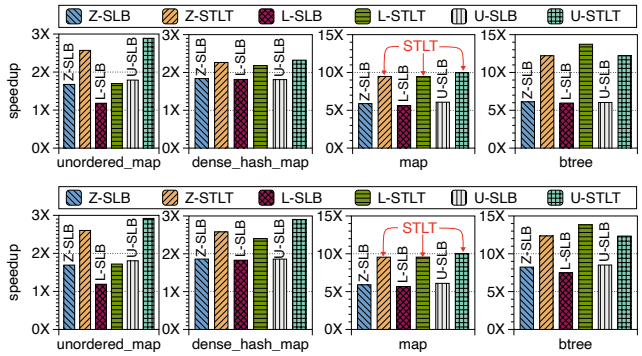


Fig. 13: Speedups by STLT and SLB on kernel benchmarks (Z for zipf, L for latest, U for uniform; top: 128B records; bottom: 256B records)

The much larger speedups than those on hash table based benchmarks are due to the more irregularity in memory accesses on trees and hence the larger potential for TLB and cache miss reductions.

D. Sensitivity Studies

In this part, we report sensitivity studies on STLT.

1) STLT Size: This section studies the relations between the speedups by STLT and its space usage. We use `zipf` workload and 64B records in the measurements.

Figure 14 reports the speedups across a range of space usage. Across all applications, the speedups increase quickly as STLT table goes from 16MB to 256MB, and then gradually flatten out as fewer and fewer TLB misses are left to reduce. STLT achieves a larger speedup for similar table space overheads compared to SLB, consistently across all applications.

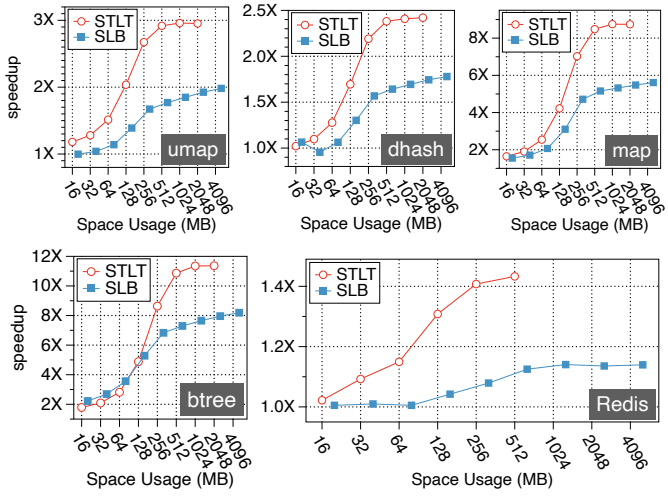


Fig. 14: Speedup sensitivity to space overhead. The space usage of SLB for the same number of table entries is 2.5× larger than STLT due to the table designs and the use of an additional log table it uses. umap is unordered_map and dhash is dense_hash_map.

Furthermore, STLT plateaus at a higher speedup level than SLB. In order to find out the reason, we plot the table miss rates in Figure 15. The figure shows that as the space usage grows, the miss rates of the STLT and SLB decrease nearly identically and become close to zero when the table size increases to 512MB. From here, we can conclude that STLT's higher speedups are not due to lower miss rates, but instead due to faster address translations, which are not addressed by SLB.

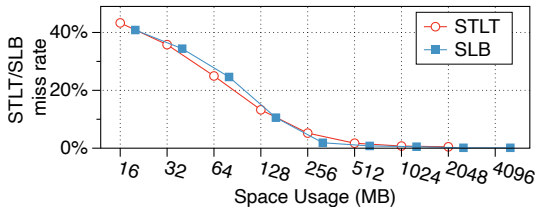


Fig. 15: Miss rates of SLB cache table and STLT

Figure 16 shows the TLB miss reduction of STLT, which contributes to the speedup. The reduction shows positive correlation with the speedups in both aspects: for single benchmark, the trend of speedup is mostly the same as the trend of TLB miss reduction; for all benchmarks, more TLB miss reduction, more speedup STLT gains on them.

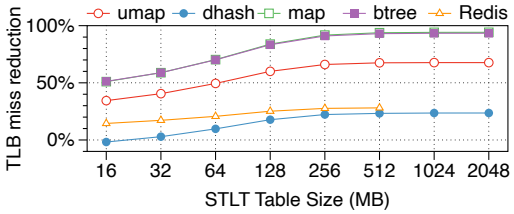


Fig. 16: Reduction in TLB misses of STLT. All benchmarks share this miss rate curve as they use the same workload. umap is unordered_map and dhash is dense_hash_map.

An exception is Redis. Because Redis incurs a large portion of computing on non-indexing data structure operations, such as parameter validating and record status maintaining, even though STLT reduces more TLB than it does on `dense_hash_map`, the speedups by STLT on Redis (1.4×) is lower than the speedups on `dense_hash_map` (2.5×).

2) Associativity: This section studies the effect of associativity of STLT table on four kernel benchmarks. We use `zipf` workload distribution and 64B records in the measurements.

Figure 17 shows the performance of STLT on different associativities. The 4-way associativity is suboptimal in some cases for various reasons. For an STLT table smaller than 64MB, 1-way associative STLT performs better or close to 4-way associative STLT as scanning a 1-way associative set is faster than doing that to a 4-way associative set. For example, in `dense_hash_map`, 1-way associative STLT outperforms 4-way associative STLT by 4.5% to 11.3% for STLT up to 64MB.

The 8-way associativity shows competitive performance for median-size STLT (from 128MB to 512MB) as its STLT miss

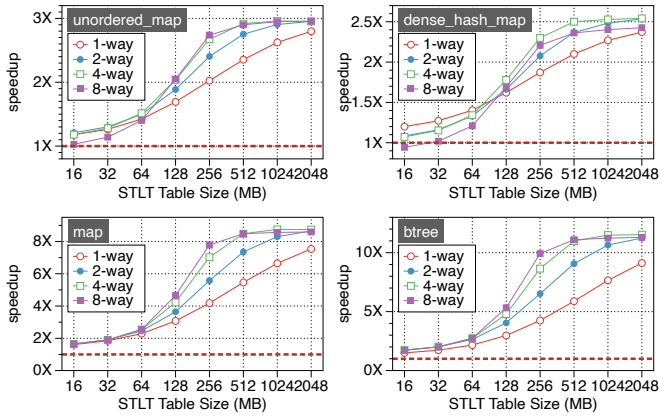


Fig. 17: Speedup of 1-, 2-, 4- and 8-way associative STLT

rates are much lower than in other designs. On `btree` and `map`, the miss rates are 43% to 99% less than the miss rates on 4-way associative STLT. However, it suffers from high overhead on scanning the set as it scans twice of the number of rows 4-way associative STLT does and is hence slow on searching on row replacement. The overhead even reduces the performance of `dense_hash_map` by 5% at 16MB STLT.

For large-size STLT, 4-way associative STLT is the optimal choice as its STLT miss ratio is as low as 8-way associative STLT and its scanning overhead is much lower. The 1- and 2-way associativities still suffer from high miss rates.

Overall, 4-way associativity gives the stablest speedup: It either gets the largest or the second-largest speedups for all benchmarks at all space usage. It is hence the choice in our current implementation.

3) Hash Function: This section studies how the performance of STLT varies when it uses different hash functions in its fast path. Figure 18(a) illustrates the speedup gained by STLT on `Redis` with 64B records and `zipf` distribution. The slow path still uses the hash function that the original Redis uses.

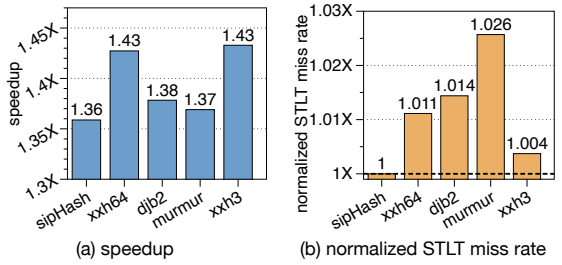


Fig. 18: Speedup and STLT miss rate vary on hash functions for Reids

The different hash functions cause up to 19.4% performance variation. Both the complexity of a hash function and its capability in avoiding conflicts affect the performance of STLT. As Figure 18(b) shows, `sipHash` generates the best randomly distributed integer and thus has the lowest STLT miss rates. It however gives the lowest speedups, due to its high time overhead. Hash function `murmursh` has the largest miss rates, but its speedups are even higher than `sipHash` gives, due to its simplicity and fast access.

E. Breakdown of Speedup

Three STLT configurations are used to study the benefits of components. Figure 19(left) shows their improvement in performance over SLB. STLT-SW is a software-only configuration retaining only virtual addresses with the in-memory table, STLT. Programs access STLT with conventional *load* and *store* instructions. STLT-VA accesses STLT with *loadva* and *insertSTLT*. It still retains only virtual addresses. STLT is the complete solution that retains page table entries of records.

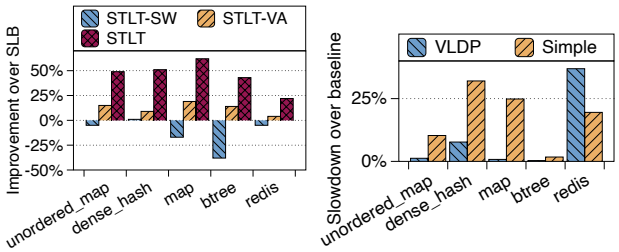


Fig. 19: Speedup for different STLT configurations over SLB (left) and slowdown caused by two LLC hardware prefetchers vs. no STLT and no prefetching (right)

The figure shows that SLB outperforms STLT-SW, especially on tree data structures. STLT-VA slightly outperforms SLB as the hardware instructions avoid frequent branch mispredictions and enable concurrent operations on STLT set scanning. STLT inherits all the benefits of STLT-VA while avoiding (at least one) VA to PA address translations, hence achieving substantial improvement over SLB.

F. Prefetchers

We evaluated the performance impact of three hardware prefetchers, including one for TLB prefetching, and two for data prefetching. This experiment does not use STLT. The baseline is the performance of benchmarks without hardware prefetching. The TLB prefetching used is distance prefetching [35]. It prefetches page table entries into the TLB. It impacts performance only marginally due to very low prefetching accuracy (up to 0.06%). The two data prefetching schemes include one scheme for complex address patterns (VLDP [36]) and another for stride-based streams ("Simple" from SniperSim). Both schemes cause performance degradation of 9.4% and 17.7% on average, respectively, as shown in Figure 19(right). All the TLB and data prefetching schemes suffer from low temporal and spatial locality due to the irregular and input dependent memory access patterns of indexing data structures. While VLDP decreases the last level cache miss rate by 7.37% on average (and by 7.15% for redis), it also incurs 1.54× more main memory accesses on average (and 3.26× for redis). The excessive memory accesses increase memory access latency by 140%, which fully negates the gain from the cache miss reduction. This is consistent with findings from prior work [37] that reports how prefetching can hurt the performance of linked data structures.

V. Discussion

Huge pages. While using huge pages may reduce TLB misses, the impact on performance is not trivial due to high copying latency [38], [39], fragmentation [40], fairness [41], and compaction [42], especially on NUMA [43] and DRAM-NVM hybrid memory architectures [38], [44]. The official document of Redis suggests disabling huge pages as it incurs a high latency [45] (similarly with MongoDB [46] databases [47], [48]).

VI. Related Work

Besides closely related work (SLB [21], HTA [10], and SDC [11]) already covered, key-value store efficiency has been studied. DIY address translation [49] allows the programmer to define their own address translation data structure beyond a radix tree, e.g. hash table. The translation is at page granularity to accelerate embedded address translation in virtual environment. Unlike STLT, DIY neither deals with key-value systems specifically nor accelerates indexing data structure traversal.

Cooper et al. [12] studied database workload and observed that most of them follow zipfian's distribution. They proposed YCSB benchmark [12] with zipfian's distribution. Other studies confirm the case for key-value stores in production environment [13], [14], [15]. Recent studies [16], [17] report key-value store showing spatial locality. EvenDB [17] reduces write operations by leveraging spatial locality. pRedis [50] proposed locality-aware memory allocator. STLT reduces address translation overheads, hence it is complementary to prior works.

By assuming key accesses arriving in batches, Widx [51] explores parallelism between key accesses by decoupling key hashing and value loading and making them a pipeline. ASIC [52], FPGA [53], [54], [55], or processing-in-memory [56], [57] based solutions offer superb performance but less flexibility in application-specific optimization, such as hash functions for specific uses, accesses parallelism, or memory management. They in addition pose security and isolation concerns in sharing with cloud users in virtual machines [58], [59], [60], [61].

VII. Conclusion

This paper has presented a new method to accelerate data retrieving in key-value stores. It distinctively focuses on reducing data addressing overhead by proposing a technique to streamline address translations. The new solution consists of an in-memory table, STLT, that caches virtual and physical addresses of records and two new instructions for accessing STLT without exposing the physical address to user-space. It meanwhile opens the opportunity for the use of simpler faster hash tables in key-value stores. It shortens data addressing by helping skip many page table walks that are needed in the current key-value store for translating virtual addresses to physical addresses. Experiments on various request distributions and record sizes show that the new technique consistently brings significant speedups. It accelerates popular production key-value store, Redis, by up to $1.4\times$, and four widely used indexing data structure implementations by up to $13\times$.

Acknowledgement

This work is supported by the National Natural Science Foundation of China under Grant No. 61832006, 61825202, and 61702202, the National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

[1] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.

[2] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, "FPGA-accelerated compactions for LSM-based key-value store," in *18th USENIX Conference on File and Storage Technologies*, 2020, pp. 225–237.

[3] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "SLM-DB: single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies*, 2019, pp. 191–205.

[4] Y. Qiu, J. Xie, H. Lv, W. Yin, W.-S. Luk, L. Wang, B. Yu, H. Chen, X. Ge, Z. Liao, and X. Shi, "Full-kv: Flexible and ultra-low-latency in-memory key-value store system design on cpu-fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1828–1444, 2020.

[5] C. Chung, J. Koo, J. Im, and S. Lee, "Lightstore: Software-defined network-attached key-value drives," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 939–953.

[6] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[7] K. Zhang, J. Hu, B. He, and B. Hua, "DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures," in *IEEE 33rd International Conference on Data Engineering*, 2017, pp. 671–682.

[8] *Redis*, https://redis.io/.

[9] *Memcached*, https://memcached.org/.

[10] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 440–452.

[11] F. Ni, S. Jiang, H. Jiang, J. Huang, and X. Wu, "SDC: a software defined cache for efficient data indexing," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 82–93.

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

[13] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "HotRing: A hotspot-aware in-memory key-value store," in *18th USENIX Conference on File and Storage Technologies*, 2020, pp. 239–252.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "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*, 2012, pp. 53–64.

[15] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang, "zExpander: a key-value cache with both high performance and fewer misses," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–15.

[16] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies*, 2020, pp. 209–223.

[17] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovici, and R. Shahout, "EvenDB: optimizing key-value storage for spatial locality," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[18] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *USENIX Annual Technical Conference*, 2013, pp. 103–114.

[19] W. Tang, Y. Lu, N. Xiao, F. Liu, and Z. Chen, "Accelerating redis with RDMA over InfiniBand," in *Proceedings of International Conference on Data Mining and Big Data*, 2017, pp. 472–483.

[20] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input prf," in *Proceedings of International Conference on Cryptology in India*, 2012, pp. 489–508.

[21] X. Wu, F. Ni, and S. Jiang, "Search lookaside buffer: efficient caching for index data structures," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 27–39.

[22] *xxHash*, https://github.com/Cyan4973/xxHash.

[23] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall, CRC Computational Science, 2015, ISBN-13 978-1482211184.

[24] L. Johnston, M. Charikar, and G. Valiant, *Hash Tables, Universal Hash Functions, Balls and Bins*, http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Lectures/CS161Lecture08.pdf.

[25] P. Guide, "Intel® 64 and IA-32 architectures software developer's manual," *Volume 1: Basic Architecture*, vol. 1, p. 375, 2019.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 295–306.

[27] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014, pp. 429–444.

[28] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "Tucana: Design and implementation of a fast and efficient scale-up key-value store," in *USENIX Annual Technical Conference*, 2016, pp. 537–550.

[29] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *USENIX Annual Technical Conference*, 2019, pp. 739–752.

[30] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen, "Cassandra: An ssd boosted key-value store," in *IEEE 30th International Conference on Data Engineering*, 2014, pp. 1162–1167.

[31] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 477–489.

[32] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2019.

[33] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, 2014.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[35] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: an application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 195–206.

[36] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015, pp. 141–152.

[37] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 7–17.

[38] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

[39] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, 2020.

[40] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, p. 679–692.

[41] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 705–721.

[42] X. Li, L. Liu, S. Yang, L. Peng, and J. Qiu, "Thinking about a new mechanism for huge page management," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019, pp. 40–46.

[43] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on NUMA systems," in *USENIX Annual Technical Conference*, 2014, pp. 231–242.

[44] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical hybrid memory management in os for tiered memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2223–2236, 2019.

[45] *Redis*, https://redis.io/topics/latency.

[46] *MongoDB*, https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[47] *Cloudera*, https://docs.cloudera.com/cloudera-manager/7.0.2/managing-clusters/topics/cm-disabling-transparent-hugepages.html.

[48] *Couchbase*, https://docs.couchbase.com/server/current/install/thp-disable.html.

[49] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017, pp. 457–468.

[50] C. Pan, Y. Luo, X. Wang, and Z. Wang, "pRedis: Penalty and locality aware memory allocation in redis," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, p. 193–205.

[51] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 468–479.

[52] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing SoC accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 36–47.

[53] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 476–488.

[54] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-direct: High-performance in-memory key-value store with programmable NIC," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 137–152.

[55] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[56] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, 2015, pp. 1–10.

[57] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3d-stacked memory-side acceleration: Accelerator and system design," in *Workshop on Near-Data Processing (WoNDP, Held in conjunction with MICRO-47)*, 2014.

[58] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualization," in *28th International Conference on Field Programmable Logic and Applications*, 2018, pp. 131–1317.

[59] P. R. Genssler, O. Knodel, and R. G. Spallek, "Securing virtualized FPGAs for an untrusted cloud," in *Proceedings of the International Conference on Embedded Systems, Cyber-physical Systems, and Applications*, 2018, pp. 3–9.

[60] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (BiFI) – automated fault attacks against SRAM-based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, 2017.

[61] D. R. Gnad, F. Oboril, and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams," in *27th International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–7.