SDC: A Software Defined Cache for Efficient Data Indexing

Fan Ni, Song Jiang, Hong Jiang
University of Texas at Arlington
Arlington, TX
fan.ni@mavs.uta.edu,
{song.jiang,hong.jiang}@uta.edu

Jian Huang
University of Illinois at Urbana-Champaign
Urbana, IL
jianh@illinois.edu

Xingbo Wu
University of Illinois at Chicago
Chicago, IL
wuxb@uic.edu

ABSTRACT

CPU cache has been used to bridge the processor-memory performance gap to enable high-performance computing. As the cache is of limited capacity, for its maximum efficacy it should (1) avoid caching data that are less likely to be accessed and (2) identify and cache data that would otherwise cost a program multiple memory accesses to reach. Unfortunately, existing cache architectures are inadequate on these two efforts. First, to cost-effectively exploit the spatial locality, they adopt a relatively large and fixed-size cache line as the caching unit. Thus, much of the space in a cache line can be wasted when the data locality is weak. Second, for easy use, the cache is designed to be transparent to programs, which hinders programs from fully exploiting its performance potentials.

To address these problems, we propose a high-performance Software Defined Cache (SDC) architecture providing a simple and generic key-value abstraction that allows (1) caching data at a granularity smaller than a cache line, and (2) enabling programs to explicitly insert, retrieve, and invalidate data in the cache with new instructions. By providing a program with the ability of explicitly using the cache as a lookaside key-value buffer, SDC enables a much more efficient cache without disruptively changing the existing cache organization and without substantially increasing hardware cost. We have prototyped SDC on the gem5 simulator and evaluated it with various data index structures and workloads. Experiment results show that SDC can improve the cache performance for the workloads by up to 5.3× over current cache design.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**.

KEYWORDS

software-defined cache, key value, data indexing

ACM Reference Format:
Fan Ni, Song Jiang, Hong Jiang, Jian Huang, and Xingbo Wu. 2019. SDC: A Software Defined Cache for Efficient Data Indexing. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3330345.3330353

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

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330353

1 INTRODUCTION

Computer systems have been using caches to bridge the well-known performance gap between fast CPU and slow memory. Data accesses are accelerated by keeping the most frequently accessed data in the caches to avoid memory accesses that can be an order of magnitude slower [22]. However, caches are much more expensive and smaller in capacity than memory, and must be efficiently used. To this end, the cache should make efforts to (1) cache only frequently accessed data to maximize its hit ratio, and (2) remove metadata access by enabling cache access of data directly. Existing cache architectures are inadequate in these two aspects by employing a relatively large cache access unit and transparent caching strategy, respectively.

1.1 The Issue with Large Cache Lines

The cache hit ratio can be seriously compromised with weak spatial access locality. For highly efficient memory access, low cache space overhead, and effective prefetching, the cache usually adopts a relatively large access unit, such as 64-byte cache line. However, there exist significant access patterns exhibiting weak intra-line spatial locality, leading to fetching and caching of unused data and compromised cache performance. A representative of such access patterns is pointer chasing, where a sequence of data items are accessed by following their pointers. Pointer chasing is common, especially in pointer-linked index structures, such as B+ tree, skip list, and hash tables, that are widely used in data-intensive applications [21, 26, 57].

To illustrate intra-line spatial locality exhibited in the access of B+ tree and hash table, we use each of them to build an index structure by inserting 8,000 key-value items, each with a distinct 8-byte key and an 8-byte value. The B+ tree has a fanout of 20, or at most 20 pointers in an internal node to its child nodes. The size of the hash table is 8,000. That is, each of its linked-list buckets has one item on average. Nodes in the data structures are individually allocated with glibc's *malloc()*. We issue 80,000 lookup requests to each of the data structures. To reflect normal cache use scenario, we choose a skewed key distribution (zipfian). We name every contiguous eight bytes in a cache line as a *cache slot*. We track accesses to the slots in the gem5 simulator [18] with a 32KB 8-way cache[1]. Figures 1a and 1c show access count at each slot of a cache line after serving the lookups on the B+ tree and hash table data structures, respectively. As shown, for both data structures frequency of access to different slots in a cache line is highly skewed. For example, in the 84th cache line only three slots (Slots 2, 3, and 5) have references (62852, 62852, and 125704 times, respectively) and other slots do not have any references. This uneven use of slots occurs mostly at nodes that are

[1] We use a small processor cache to match the relatively small data set size used in our experiments.

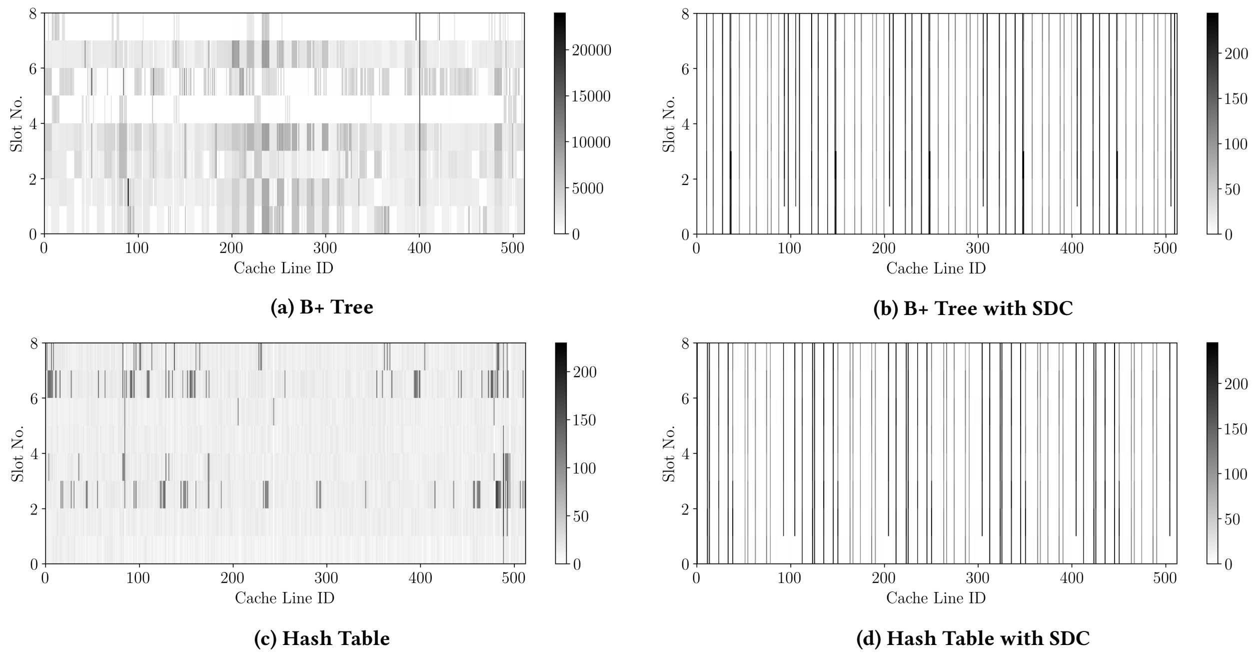


Figure 1: Number of accesses at each 8-byte slot (on the Y axis) of every 64-byte cache line (on the X axis) in a 32KB 8-way cache for serving 80,000 key lookups in each of the data structures (B+ tree and hash table) indexing 8,000 distinct keys. Keys in the lookups follow the zipfian distribution. Figures (b) and (d) show results with the proposed SDC technique.

at or close to leaves of B+ tree, which dominate cache space held by the data structure. This issue can be serious even for the hash table whose average bucket size is only one, as shown in Figure 1c.

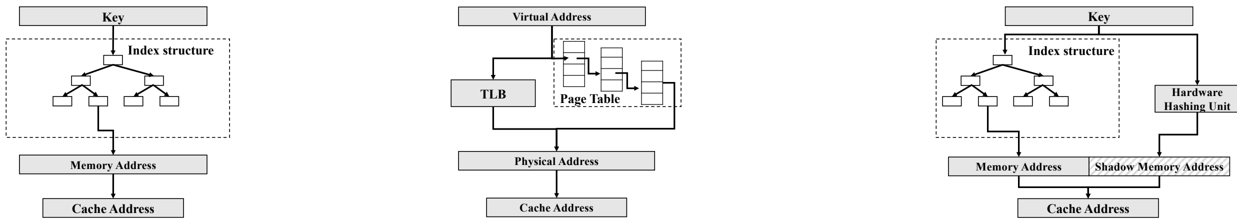
Admittedly this weak spatial locality issue is ameliorated when most lookups are successful and a large piece of data is accessed after a successful key lookup. However, referencing a small piece of data after an index lookup is common. For example, Facebook reported that in its Memcached key-value (KV) pools 90% of the KV items have values of smaller than 500 bytes [3]. In a KV pool (USR) dedicated for storing user-account statuses all values are of two bytes. In a general-purpose pool (ETC), 40% of requests to the store have values of 2, 3, or 11 bytes. In a pool for frequently accessed data, 99% of KV items are smaller than 68 bytes [35]. In Twitter's KV workloads, after compression each tweet has only 46 bytes of text [36]. In contrast, the space and time costs of pointer-linked indexes are significant. In the Facebook's Memcached servers, the hash table, including the pointers in the buckets and in the LRU lists, accounts for about 20~40% of the memory space [3]. A recent study on modern in-memory databases shows that hash index accesses are the most significant single source of run-time overhead, constituting 14~94% of total query execution time [26]. Therefore, the under-utilization of cache space due to weak spatial locality exhibited in the pointer-chasing accesses can be a serious performance issue.

1.2 The Issue with Transparent Caching

To minimize software burdens, the cache is designed to automatically cache data at any memory address referenced by the instructions, and then access them via the addresses. In a program, there are two ways in which data are addressed. One is that their addresses are directly coded in the program and they can be referenced with the addresses without any proceeding metadata accesses. Example data include variables and data elements in an array. The other is that the data have been named by the program's users, such as user-defined keys. To access the data in the memory, the program has to first use index structure(s) to translate the keys into actual memory addresses. Example data addressed in this way include values in key-value (KV) caches (such as Memcached) or in-memory KV stores, data accessed by keys in in-memory databases, and in-memory inodes accessed via file paths (as the keys) in file systems.

In existing cache designs two levels of address translation are performed to load and access a data item named by a user-defined key in the cache, as illustrated in Figure 2(a). The first level of translation (from a key to a memory address) is conducted by software with a number of memory accesses on an index structure. Note that all accessed metadata on the index will also be loaded into the cache, increasing demand on the cache space. The second level of translation (from the memory address to a cache address) is conducted by hardware. If we could access data in the cache directly via their keys, the accesses and caching of the metadata would be avoided. This is one of our design objectives.

Interestingly, the two levels of address translation also occur in a physical-memory-address-indexed cache with every memory address, as shown in Figure 2(b). The first-level translation is from a process's virtual address to the physical address, and the second-level one is from the physical address to the cache address. The first-level translation requires expensive traversal on the process's page table (a prefix tree). Fortunately, the TLB cache was invented



(a) Translate a key to a cache address. (b) From virtual addresses to cache addresses with TLB. (c) Translate a key to a cache address with SDC.

Figure 2: Various scenarios of address translation into the cache space.

to accelerate the translation by avoiding access of page table (the metadata). In this work, we'd like to make the technique generalized and make similar benefit available to user-defined index structures.

1.3 Our Solution: Software-Defined Cache

In this paper we propose a new cache design, Software-Defined Cache (SDC), as a supplement to existing cache architecture to allow the last-level cache (LLC) to serve as a look-aside cache for programmers to explicitly insert, retrieve, or invalidate data in the cache using a simple and generic key-value abstraction. As a look-aside cache, SDC is presented to programmers much like a scratch pad, whose cached data are not automatically associated to or written back, upon eviction, to the memory, except that it does not burden programmers as a scratch pad would.

First, SDC is accessed with user-defined keys, instead of memory addresses. Second, the cache space is still managed by the hardware for functions such as data replacement according to access locality, to relieve programmers from the complex and often expensive task. In addition, SDC uses a cache slot, which is much smaller than cache line, as its access unit without substantially increasing overhead cache space. These advantages of SDC are achieved by seamlessly integrating it into existing cache architecture. As illustrated in Figure 2(c), SDC maps a key to a memory address in the *shadow address space*, which is a (physical) address range that has not yet been mapped to any physical memory, using a hardware hash unit. It then leverages existing cache address mapping mechanism to map the shadow address to a cache address for caching user-supplied data. It also leverages existing cache's replacement strategy to manage its space. In the meantime, the regular transparent caching is still available, making the LLC be a hybrid cache. As part of the LLC, SDC's space size is dynamically determined by its data access locality. And cache utilization with weak spatial locality can be significantly improved, as illustrated in Figures 1b and 1d.

This paper makes three contributions. 1) We show two fundamental sources of caching inefficiency (large cache line and transparent caching) that can seriously compromise the performance of data indexing in data-intensive applications. 2) We propose SDC that uses part of the LLC as a look-aside cache accessed via user-defined keys at a fine granularity. SDC enables a KV cache at the LLC for programs to access with a simple and generic key-value interface. 3) We extensively evaluate the SDC design with a full-system simulation on gem5 with various workloads. Experiment results show that for its targeted workloads SDC significantly improves application performance. As anecdotal examples, SDC improves throughput of a real-world in-memory database by up to 3.3×.

2 THE DESIGN OF SDC

The design goal of SDC is to provide programs with explicit fine-grained LLC cache access via user-defined keys. This KV-style use of the cache provides programmers with a critical architectural support to overcome weak spatial locality and for TLB-like capability to avoid index traversal. To achieve the goal, there are a number of challenges to address to make it truly functional and efficient. First, currently data are loaded into caches as a side effect of memory accesses, and a program cannot insert data directly into the cache without issuing a memory access. We need to carefully define a new programming interface friendly to programmers and compilers and being least intrusive to existing cache architecture. Second, in the mapping from a potentially large key space to a cache space key collision is unavoidable. We must develop effective mapping and cache replacement strategy to minimize mapping conflicts. Third, allowing smaller data items in the cache means higher space cost for additional metadata. We need to carefully make trade-offs among this space overhead, the API's usability, and the cache's hit ratio to build a cost-effective software-defined cache.

2.1 Extending ISA to Enable SDC

To enable use of CPU cache as a look-aside KV cache we introduce three instructions (and more of their variants) to insert, look up, and invalidate a KV item in the cache. Their formats are shown in Table 1. A common operand of the instructions is the key, provided in a 64-bit register. Admittedly user-defined keys can be of any format and any length, such as a character string of variable length. To enforce a consistent representation, it may require programmers to first convert a key into a 64-bit number (e.g., taking the last 8 characters of the key string). Apparently, the conversion may compromise uniqueness of the original keys. A similar issue will occur in the SDC's implementation, where for reducing metadata space cost we reduce size of the tags (for matching cache slots) at a (small) risk of returning wrong values upon the lookup (a false hit). To address the issues, programmers are expected to verify the truthfulness of the value[2]. A commonly used technique for the verification is to store the original full key together with its value, and use address of the KV pair in the memory as the value in `sdc_insert` and `sdc_lookup`, or *Mem(value addr)* in Table 1.

[2]The security or privacy is not a concern here as SDC guarantees a KV item inserted by one process will not be returned to another process.

Table 1: SDC Instructions. Operands 2 and 3 specify the memory address of the value to be inserted in `sdc_insert`, and that for receiving return value in `sdc_lookup`, and the value size, respectively. For the case where the value size is 8 bytes (one cache slot), these two instructions have variants directly storing the value in a register specified by Operand 2. Operation status is a bit in the status register indicating success of the operation. To update a value in the cache, one needs to invalidate it and then inserting the new value.

Name	Operand 1	Operand 2	Operand 3	Operation status
sdc_insert	Reg(key)	Mem(value addr)	Reg(value size)	1 or 0
sdc_invalidate	Reg(key)	-	-	1 or 0
sdc_lookup	Reg(key)	Mem(value addr)	Reg(value size)	1 or 0

```
1  struct kv_entry buffer, *p;
2  uint64_t key;
3  ...
4  while (receive_req(command, &buffer) {
5      key = buffer.key;
6      switch(command) {
7       case LOOKUP:
8        if (sdc_lookup(key, &p, sizeof(p))) {
9          if(p->key == key){ /* original key comparison may be
         needed*/
10             processing_data(p);
11         }
12       } else {  /* miss, do regular search */
13         p = index_lookup(key);
14         if (p != NULL){
15           processing_data(p);
16           sdc_insert(key, &p, sizeof(p));
17         }
18       }
19       break;
20      case DELETE:
21       sdc_invalidate(key);
22       del_from_the_kv_store(key);
23       break;
24      case INSERT:
25       p = ins_into_the_kv_store(key, &buffer);
26       if (p != NULL)
27         sdc_insert(key, &p, sizeof(p));
28       break;
29     }
30 }
```

Figure 3: Pseudo code for a hypothetical KV store handling various requests with the support of SDC functions.

After receiving the address returned by a lookup, the program will compare the key at the address with the original key [2, 7, 28].

The return value (in Operands 2 and 3) is up to the programmer's interpretation. It can be either the cached value itself or a pointer to the actual value. The value size can be of 1, 2, 4, or 8 slots (or 8, 16, 32, 64 bytes, respectively). The `sdc_invalidate` instruction explicitly removes identified KV items from the cache. It is noted that, like TLB, SDC uses the cache as a look-aside buffer. When the items are evicted or invalidated, there are *no write-back operations*.

With the instructions, a library of functions can be available for programmers to use the SDC cache. Figure 3 shows an example use of the cache in a pseudo code, whose execution is facilitated by the architecture illustrated in Figure 2(c).

2.2 Two Levels of Address Mapping

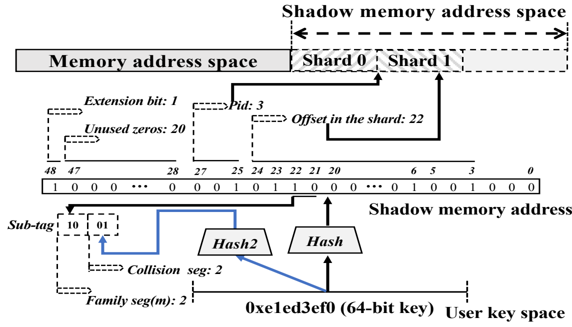
As mentioned, one of the SDC design principles is to seamlessly integrate it into the existing cache architecture. In addition to the benefit of simple design and reuse of the cache circuitry for a reduced cost, the integration allows efficient use of the cache space. To achieve high cache-space efficiency, we must keep the most frequently accessed data in the cache, regardless of their sources (either loaded via regular memory accesses or inserted by SDC instructions). If these two types of data were cached or managed separately, it would be hard to consistently compare their access locality, and it would also be difficult to determine the space allocations between them to dynamically and accurately reflect their respective space demands. To this end SDC maps its key space to a range of physical address space not mapped to any physical memory, named *shadow memory address space*, from which the existing cache scheme kicks in to further map it into cache address space.

Accordingly, there are two levels of address mapping in SDC. The first-level mapping (from the user key space to the shadow memory address space) is exclusively managed by SDC. The second-level mapping (from the memory address space, including regular and shadow memory addresses, to the cache space) is managed by the existing cache scheme at the cache-line granularity and by SDC at the slot granularity. Figure 4 illustrates these two levels of address mapping, which serves as an overview of the SDC's architecture with details explained in the following two sections.

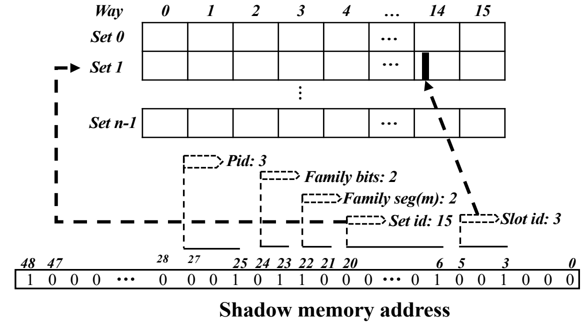
2.2.1 **Mapping from User Key to Memory**. For the first-level mapping, we need to decide where the shadow memory fits into the address space. In principle, it can be placed at any location not used in current addressing. As the memory address space is rarely fully occupied (by mapping to DRAM), we choose to double current physical memory space to make SDC compatible with any today's and future's system configurations by adding an extension bit before the MSB (Most Significant Bit) of a memory address. Accordingly, this bit will be the first bit of a cache line tag, with a "0" indicating a regular cache line and a "1" an SDC cache line. In this way, a physical address in the shadow memory will be of the format '$100:::0b_{k-1}b_{k-2}:::b_0$' with a shadow memory size of 2^k bytes. Note that for SDC cache lines the sequence of '0's before b_{k-1} does not need to be stored in their tags to reduce cache metadata.

A critical issue in the SDC design is to guarantee security and privacy of each process's data in the SDC cache. A key is unique only within a process. To isolate keys of one process from other processes and make a key unique in an SDC cache, we evenly partition the shadow memory space into *shards*. Keys of a process are mapped exclusively to a dedicated shard. Suppose an SDC supports up to eight processes simultaneously. The three bits $b_{k-1}b_{k-2}b_{k-3}$ in the aforementioned address format represent shard identifier, one for a process. As will be further explained, `sdc_lookup()` only returns values inserted by the same caller process.

An important parameter in the SDC design is the size of the shadow memory, or alternatively the size of a shard for a given



(a) An example of the first-level address mapping. A 64-bit key ("0xe1ed3ef0") in an SDC instruction issued by a process is translated into a 49-bit shadow memory address. The hash unit ("Hash") translates the key into a 22-bit offset in a shard. A 3-bit process id ("001") is placed before the offset to indicate the shard ("Shard 1") where the key is mapped. Note "sub-tag" is used in the second-level mapping.



(b) An example of the second-level address mapping. Like regular cache, the 15-bit "set id" determines into which cache set the shadow memory address is mapped. As detailed in Section 2.2.2 and illustrated in Figure 5, some other bits, such as "pid", "family bits", and "family segment", are used to determine a cache line in the set. The 3-bit "slot id" determines a slot in the cache line.

Figure 4: Two-level address mapping in SDC. Bit segments and their respective lengths are shown as *segment name : segment length in bits***. A 32MB 16-way associative cache is illustrated.**

number of shards. An ill-chosen size may compromise the SDC's performance advantage. A too large shard may cause the space sparsely filled and many SDC cache lines under-utilized. However, with a too small shard there can be many keys mapped to a single address, increasing conflict misses. We set it to be the LLC cache size, ensuring the cache space can be fully utilized.

SDC uses a hash function to map a key to an address in a shard. Though a multi-choice hashing can help reduce mapping conflicts, we do not choose it to avoid extra access latency and hardware cost. Instead, SDC achieves the multi-choice capability in the second-level mapping by reusing the hardware for the set associative cache. At the first level, the hash function translate a 64-bit key into an offset in a shard, and precedes it with a process id indicating the process to which the key belongs, as illustrated in Figure 4a.

2.2.2 **Mapping from Shadow Memory to Cache**. The SDC cache is hosted only in the LLC, as it is designed for memory-intensive applications with large working sets whose caching in a small cache is unlikely to yield many hits [48]. Though LLC is relatively slow, a miss on a KV item in the SDC cache has a high penalty (multiple memory accesses for the index search). Therefore, increasing cache hit ratio outweighs achieving short hit time. Further, as LLC is shared among cores in a processor, keeping SDC data only in the LLC avoids the cost for maintaining cache coherence in a single-socket system. For multi-socket systems, consistency of the SDC data can be maintained either by hardware, using techniques such as Intel QPI [58] or AMD HyperTransport interconnection [12], or by relying on OS using techniques similar to that for TLB [1].

In SDC, in addition to the mapping conflicts from memory addresses to cache lines as that in regular caches, there is another kind of conflicts in the mapping from user keys to shadow memory addresses. To reduce the conflict misses, we introduce the concept of *cache-line family* that includes all SDC cache lines in a set whose tags differ only at the last m bits, as shown in Figure 4b. A family in a cache set has 2^m member cache lines. A shadow memory address can be mapped to any cache lines in the family of a set. Keys

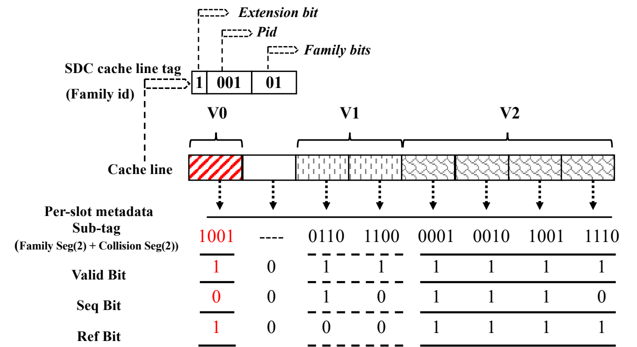


Figure 5: Status bits for an SDC cache line and its slots.

mapped to the shadow memory address can then be stored in any cache lines in the family, reducing impact of key collision. The use of cache line family can also prevent one process from using the cache space too aggressively in a multi-process context. Should the *family id* field (see Figure 5) not be introduced, a value would likely be placed in any line in a cache set.

SDC maintains some status bits for each slot in an SDC cache line as listed below and illustrated in Figure 5.

- *Sub-tag*: bits for identifying a value matching a given user key. A sub-tag of a slot consists of two bit segments. The first one is the *family segment* field as shown in Figure 4b, which is the m bits (assuming the family size is 2^m) on the left of the *set id* field. The other one is some bits generated by applying another hash function (*Hash2* in Figure 4a) on the original user key, aiming to further reduce probability of returning a value not matching a given user key (a false hit). If a value occupies multiple cache-line slots, only the first slot needs to have a sub-tag created with the above rules. Thus, the second hash function can produce more bits to fill the space originally reserved for sub-tags to further reduce probability of false hits.

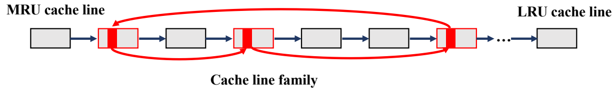


Figure 6: Two-level replacement strategy: a hypothetical LRU replacement for cache lines in a cache set and a clock replacement for SDC slots in a cache line family.

- *Valid Bit*: a bit indicating if the slot contains valid value.
- *Sequence Bit*: a bit indicating the first slot storing a user value. It is noted that a value can occupy 1, 2, 4, or 8 contiguous slots in a cache line, and is placed at a slot offset that is multiples of their respective sizes. For a value, the sequence bit of its last occupied slot is '0' and the bits for its other slots are all '1's.
- *Reference Bit*: a bit indicating if the slot has been recently referenced. It is used to facilitate the clock replacement among slots in a cache-line family. The bit is set to '1' once the slot is accessed.

For an SDC operation, there can be multiple sub-tag-matching slots in different cache lines of a family. For `sdc_lookup`, SDC randomly selects one from the slots and returns it. For `sdc_invalidate` and `sdc_insert` all the matching slots are invalidated.

When there is no free space available for inserting a new KV item, SDC employs a two-level replacement strategy. The objective is to keep hot (or frequently accessed) cache lines, either regular or SDC cache lines, from being replaced, and keep hot slots in a cache-line family from being replaced. When a KV item at a shadow memory address is mapped into a family of SDC cache lines in a cache set (all at the same cache-line offset, or slot address), the item can be placed in any of the cache lines whose corresponding slot (or multiple slots for a larger value size) is available (with '0' valid bit). If none of them in the family is available and current family size has not yet reached its maximum size, SDC will first ask the existing cache-line-level replacement scheme, such as LRU as illustrated in Figure 6, to identify a cold cache line in the set for potential replacement. If the selected cache line does not belong to the family, it is replaced and the value is stored in the corresponding slot(s) of the cache line. Other unused slots in the cache line are marked as invalid ('0' valid bits). Otherwise, instead of replacing the entire SDC cache line selected by the replacement scheme, we proceed with the slot-level CLOCK replacement within the family as described in the below.

We first assume that value is one slot long. As shown in Figure 6, a value can be stored at a slot in any cache line of the family (at the same offset). The replacement policy checks reference bits of the slots, one at a time, in a certain order. Whenever it encounters a '1', it resets it to 0 and proceeds to the next one. It stops at a slot whose reference bit is '0' and replaces its current value with the new one. Note that if the current value occupies more than one slot, which can be known by examining sequence bits of this slot and its neighboring slots, the other slots occupied by the value need to be marked as invalid. If the new value is more than one slot long, the replacement procedure is similar. The only difference is to check reference bits of multiple slots starting at the offset in each cache line and replace them together if they are all '0's. Otherwise, they are all reset to '0'. The two-level replacement policy dynamically balances cache space allocation among regular data cache and SDC.

2.3 Metadata Storage Cost of SDC

While SDC supports cache space management at a finer granularity (the slots), some metadata have to be associated with individual slots. To have an empirical estimation of the space cost, we choose a popular processor (Intel E5-2683 V4 CPUs [23]) for SDC to be implemented in its LLC (L3). Its LLC has a 20-way set associative 40MB cache with 64-byte cache lines. The size of a shard is the same as the cache size (40MB). There are eight 8-byte slots in a cache line. Let's assume a sub-tag of 4 bits (2-bit family segment from the regular tag and 2-bit collision segment from key hashing). Adding the 3 bits for the valid, sequence, and reference bits, each SDC cache line requires $7 \times 8 = 56$ bits. Assuming that the SDC supports up to 8 processes, an SDC tag has 7 bits, which is less than regular tag size (27) by 20 bits. Therefore, to accommodate SDC each cache line needs extra $56 - 20 = 36$ bits, or 36/(64*8) = 7% of the cache size. To further reduce the space cost, we can simplify the design by fixing the slot size to 16-byte, which will save the reference bit and accommodate 4 slots in a cache line. This will reduce the space cost to less than 1% ($(6 * 4\text{-}20)/(64 * 8) \approx 0.78\%$).

3 EVALUATION

To evaluate the performance of SDC, we prototype SDC in gem5 [6] and conduct extensive experiments with micro-benchmarks, real-world in-memory database and key-value cache traces from a production system to reveal its performance insights.

3.1 Evaluation Methodology

W modify gem5's memory system model [18] and enable SDC in the LLC. We implement the three SDC instructions (`sdc_insert`, `sdc_invalidate`, and `sdc_lookup`) as pseudo-instructions in the X86 ISA. The processor's pipeline is drained before simulated execution of a gem5's pseudo-instruction. This operation and its performance penalty do not occur in a real hardware implementation. To estimate the penalty of pseudo-instruction execution, we tentatively modify the SDC instructions to remove any cache/memory accesses. In this case these gem5's pseudo-instructions have an execution time of about 20ns (50 cycles), much longer than expected real execution time (about 1-2 cycles). To compensate the effect, we conservatively deduct 4ns from each SDC instruction's execution time. Our experiments are carried out in the full-system simulation mode and the simulation configurations are summarized in Table 2. We choose a small LLC (3MB) for the sake of moderate simulation time. The workloads' data set sizes are determined accordingly. For parameters not listed here, default values specified in the gem5 release are used.

We conduct experiments on the simulator by running an in-memory key-value store serving insertion, lookup, and deletion requests. The core index structure can be a B+ tree or a hash table. The B+ tree used in the evaluation has a fanout of 20, and its source code is taken from the open-source database LMDB [46]. The hash table has 2^{21} linked-list buckets, and the code is taken from the open-source Memcached KV cache [17]. Both indexes are initially populated with 1.5 million KV items. For the hash table the items' keys are uniformly distributed in the buckets after hashing. Between the two data structures, the B+ tree, which has five levels and stores all KV items at the leaf nodes, represents index structures with a

Table 2: Simulation Configurations

Processor Configuration	
ISA	X86 (64-bit)
CPU	1 core, 2.5Ghz
Separated L1 I/D Cache	32KB, 8-way, 3-cycle
Unified L2 Cache (LLC)	3MB, 12-way, 22-cycle
Cache line size	64-byte
Cache replacement policy	LRU
DRAM Configuration	
Memory	DDR3_1600_8x8
DRAM bus bandwidth	12.8 GB/s
SDC-related Configuration	
Key	64-bit, zipfian distribution
Shadow memory size (one shard)	3MB unless noted
Sub-tag bits	3-bit unless noted
cache line family size	4 unless noted
Hash function for generating address	xorshift [37]
Hash function for generating sub-tag	hash64shift [47]
Hardware hashing unit latency	5-cycle
SDC access latency	29-cycle
Data slot length	8-byte
Simulation Software Configuration	
OS	Linux 3.4.112, x86_64
Compiler	gcc-6
Compile option	O3

high penalty for a key lookup miss in the cache. In contrast, the hash table is configured so that each bucket has less than one KV item on average, representing a relatively low miss penalty.

For a key-value item in the stores, the key and the value are 8 bytes and 512 bytes long, respectively. For SDC operations, the 8-byte key and 8-byte memory address of the KV item are actually key and value, respectively, in *sdc_insert* for insertion. The value returned by an *sdc_lookup* instruction is interpreted as the address for locating the corresponding key-value item. Unless noted otherwise only the first 64 bytes of a value are accessed after a successful key lookup. After a KV store is populated, we continuously issue lookup requests and measure the store's throughput. The lookup keys are pre-generated using Yahoo's cloud serving benchmark tool (YCSB) [14]. The keys follow a zipfian distribution with skewness of 0.99 to simulate workload of strong temporal locality. Data set involved in the lookups in an experiment can be a subset of KV items pre-inserted in the stores. To calculate the data set's size, for values we only include the data that is actually accessed after a lookup (64 bytes by default). In each experiment we will first replay the lookup sequence three times to warm up the cache before actual measurements are conducted.

For performance comparison, we use a system with regular LLC cache configurations (see Table 2) as the baseline. Furthermore, we compare SDC with a software solution for accelerating index structure lookups, named SLB (Search Lookaside Buffer) [52]. SLB is a carefully-tuned application-managed look-aside buffer, implemented as a hash table, to cache frequently accessed data in an index structure. In the experiments the SLB buffer is set to be of the LLC cache size (3MB), at which it achieves its best performance [52]. In addition to the synthetic workloads for micro-benchmarking, we also evaluate SDC in real-systems in Sections 3.5.

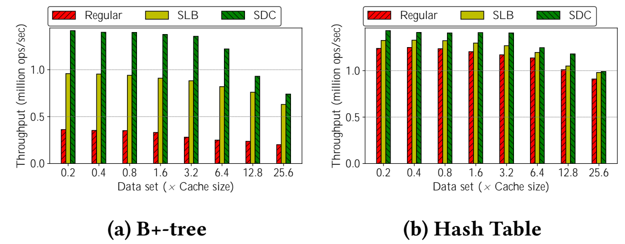


Figure 7: Throughput of lookup requests in the B+-tree and hash-table based stores with various cache set sizes.

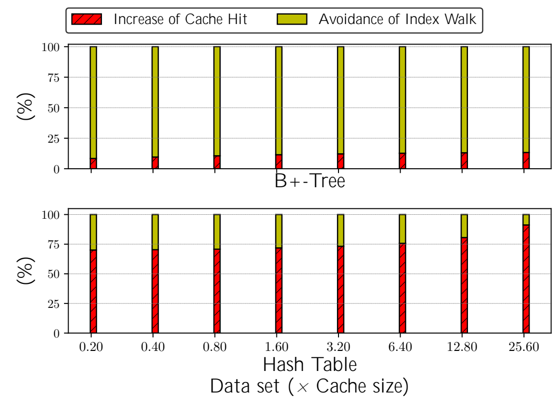


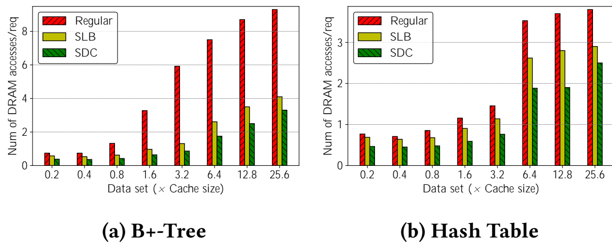
Figure 8: Contributions to memory access reduction.

3.2 Microbenchmark Performance

In the evaluation with the micro-benchmarks, we are interested in SDC's quantitative performance advantages, sources of its performance benefits, and impact of SDC's parameters.

Figure 7 shows the lookup throughput of the two KV stores on the three caches with varying data set sizes. As shown, SDC produces the highest throughput in all testing scenarios. For the B+-tree-based store, the improvements can be as high as 5.3X over those of *Regular* at smaller data sets. The improvements with the hash-table-based store are lower (at about 9.6%~23%). SDC improves the performance mainly for two reasons. First, with SDC a lookup request can be served by the SDC cache if the KV item has been inserted and stays in the cache, removing the need of index walk, which can be very expensive as it usually introduces a number of memory accesses. Second, SDC makes efficient use of cache line space by reducing idle data items in a cache line even with weak spatial access locality, which improves cache hit ratio.

While both reasons can lead to reduction of memory accesses, we measure their relative contributions to the reduction in each scenarios and the results are shown in Figure 8. As shown, the two reasons contribute differently in the two stores. The B+-tree store benefits mostly from the first reason (avoidance of index walk), as an SDC lookup hit may remove multiple (likely five or more) memory accesses. In contrast, an SDC hit saves only one or two memory accesses in the hash-table index due to small bucket size. And the second reason (increase of cache hits due to higher cache



(a) B+-Tree (b) Hash Table

Figure 9: Number of memory accesses served at the DRAM per lookup in the B+-Tree and hash-table based stores.

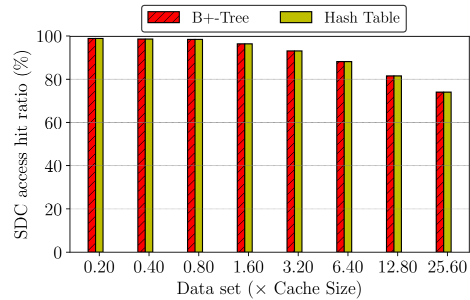
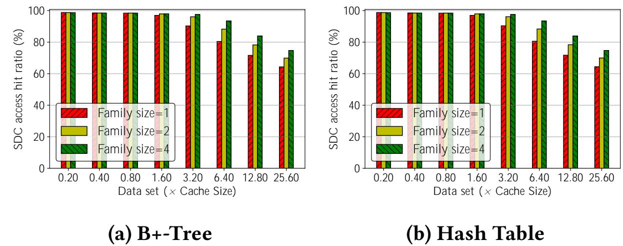


Figure 10: Hit ratio of SDC lookups for the B+-Tree and hash-table based stores with various data set sizes.



(a) B+-Tree (b) Hash Table

Figure 11: SDC access hit ratio for lookups in the B+-Tree and hash-table based stores with different family sizes.

line utilization) contributes more to hash table's throughput improvement. Figure 9 shows average number of memory accesses per lookup. SDC can significantly reduce memory accesses compared to regular cache, especially for the B+-tree store (by up to 85%).

When the data set grows beyond a certain size (about 3.2 times cache size) (see Figure 7), the working set starts to exceed the cache size and the capacity misses keep increasing. Figure 10 shows the hit ratio of the SDC cache with increase of the cache size. When the working set grows very large and the SDC cache hit ratio is reduced, SDC's performance advantage shrinks. However, as SDC is a user-defined cache, the user program may track the hit ratio and selectively insert and look up a subset of its working set in the cache and at least keep a high hit ratio for this smaller set of data.

Compared to SLB, SDC can provide up to 54% performance improvement. By maintaining an in-memory buffer, SLB requires an extra memory copy to move a data item into the buffer. Furthermore, tracking access frequency to effectively perform replacement policy requires extra space and time costs for recording, updating, and searching a set of metadata about data items in the buffer. Accordingly, SLB increases its demand on cache space. It takes more CPU cycles and memory accesses to insert data items into the cache. Therefore, its performance advantage is smaller. Note that by design SLB cannot be shared by multiple processes. If each process has its own SLB buffer, the software-managed buffers may compete with each other, leading to many cache misses and SLB's inefficacy.

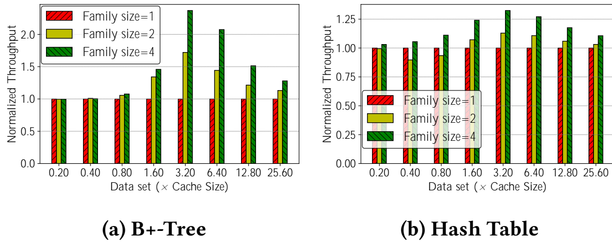
3.3 Performance Impact of Family Size

In the SDC design, the tag of an SDC cache line is actually a family id used to identify a family of SDC cache lines in a cache set. And a shadow address can be mapped to any of them. A sub-tag of a cache-line slot is to perform address matching. A larger family provides more candidate slots to serve a data insertion request and helps to reduce conflict misses and improve the cache's performance. In the experiments, we measure the stores' performance with different family sizes to study its impact. Figures 11 and 12 show SDC's hit ratio and the store's throughput with different family sizes, respectively. The throughput results are normalized to their respective counterparts for the baseline store whose family size is 1 and sub-tag is 3 bits long. When the family size doubles, one more bit is added to the sub-tag. As shown, the improvement with a larger family is substantial, especially with a larger data set and consequentially more serious collisions among multiple keys mapped to the same shadow address.

With a larger family size, the throughput can be up to 2.3× higher (for the B+-tree store). The improvement reduces if we keep increasing the data set size after the improvement peaks at around a data set of 3.2×-cache-size large. This is due to an increased ratio of false hits. With a large family size, a KV item can stay longer in the cache and likely produce more hits. However, because key space of a process can be much larger than a shard memory space, the key collision can still become increasingly serious with a large data set size. Therefore, using a larger family may also produce more false hits. Fortunately, our evaluation shows most of the increased hits are true ones (the false hit ratio is very low (< 1:5%)) and SDC almost always obtains performance improvement with a large family size. The only exceptions are with the hash-table when the data sets are small and family size is 2. This is the situation when using a large family cannot improve the hit ratio but receives more false hits. By introducing the concept of cache line family and making its size a design parameter, one can limit interference among processes competing for cache lines in a cache set, which helps reduce false hits.

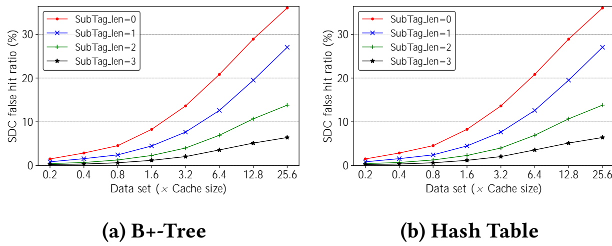
3.4 Performance Impact of Sub-tag Length

As discussed, false hit can be a threat to the cache's performance as its penalty can be higher than that of a miss. It takes an extra memory access for verification to reveal a false hit. False hits can be minimized by increasing sub-tag length. Candidate data items in slots of different cache lines in a family are more strictly screened when longer sub-tags are compared before returning a matched one to the program. For a short sub-tag, it can be easy to find one or even multiple matched data items. In the case of multiple matchings,



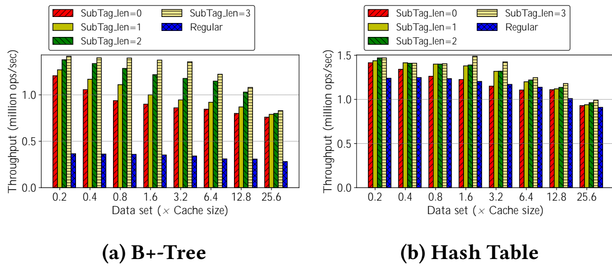
(a) B+-Tree **(b) Hash Table**

Figure 12: Throughput of lookups in the B+-Tree and hash-table based stores with different family sizes.



(a) B+-Tree **(b) Hash Table**

Figure 13: SDC false hit ratio for lookups in the B+-tree and hash-table based stores with different sub-tag lengths.



(a) B+-Tree **(b) Hash Table**

Figure 14: Throughput of lookups in the B+-tree and hash-table based stores with different sub-tag lengths.

SDC randomly picks one and returns it. This may turn many misses (if long sub-tags are used) into (likely false) hits and artificially boost the hit ratio. Figure 13 shows the false hit ratio with different sub-tag lengths. And Figure 14 shows corresponding throughput. In the experiment the family size is fixed at 4. As expected, using a very short sub-tag (0 or 1 bit) can dramatically increase false hit ratio, and reduce the throughput. Note that the throughput reduction is moderate as most of false hits with short sub-tags are simply misses with long sub-tags. As long as SDC uses a moderately sized sub-tag (between 3 and 5), the false hit ratio can be made reasonably small.

3.5 Benefits for In-Memory Database

To evaluate the efficiency of SDC in real-systems, we run experiments with *Silo*, an open-source in-memory database [50, 51]. In Silo, a Masstree-inspired tree is used for data indexes. To enable SDC accesses, we added about 100 lines of code to Silo. We use

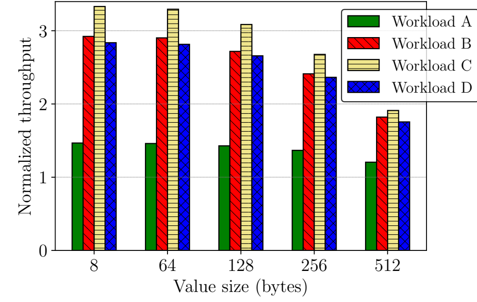


Figure 15: Throughput of the silo database with YCSB workloads at different value sizes. The throughput is normalized to that with regular system without SDC.

8-byte strings as keys and vary the value size from 8 to 512 bytes. To simulate the cost of data access, when a lookup returns a KV item, we read the first 8-byte of every 64-byte of data in the value. So accessing 512-byte data requires 8 memory accesses.

We first fill the database with 1 million KV items, and then use the YCSB benchmark to generate four workloads, each issuing 128,000 requests with zipfian distribution. The four workloads follow the same access patterns as that of YCSB's A, B, C, and D workloads [14]. Workload A has 50% read and 50% update, representing a session store. Workload B has 95% read and 5% update, representing photo tagging applications. Workload C has 100% read, and Workload D has 95% read and 5% insert with a bias towards records that are created recently. We use the first 32,000 requests to warm up the system and measure the time of serving the remaining 96,000 requests with one worker thread.

As shown in Figure 15, with a small value size SDC can significantly improve the throughput by up to 3:3× (e.g., for read-only Workload C). For workloads with a larger percentage of write requests, the improvement is less significant as SDC cannot accelerate write operations. And for each write operation an `sdc_invalidate` request needs to be issued to SDC to invalidate a possibly cached key. With a larger data size, more time is spent on accessing in-memory data and the overhead introduced by the index searching becomes less significant. When the value size is 512-byte, the throughput is improved by 1.9× for workload C. As existing studies have shown small data are common in today's data processing systems [3], the above experiment results with real-world in-memory database show that SDC is effective in improving the data indexing performance.

4 RELATED WORK

Effective use of CPU cache is critical to supporting high-performance memory-intensive applications. There have been many works on leveraging the cache to reduce or accelerate memory access.

Speeding up pointer chasing. Pointer chasing is a memory access pattern that is notoriously inefficient due to its access irregularity [16, 24, 25] and inherent serialization [24, 29, 31]. In the meantime, it is performance critical and extensively used in software such as databases, file systems, graph processing, dynamic routing tables, and garbage collections. A major technique to address the issue is to predict and prefetch the next node and pointer.

It can be hardware-based [10, 13, 42] , software-based [29, 31] , or pre-execution-based [11, 30]. This approach can be expensive by consuming too much memory bandwidth [16, 24, 41], and cannot be consistently effective on different data structures and access patterns. Concerned with limited CPU cache capacity to hold the working set accessed during pointer chasing, Hsieh et al. proposed to perform pointer chasing inside memory with the PIM (processing-in-memory) hardware support in the 3D-stacked memory [21]. In fact, one does not have to cache entire working set, or all the data in the pointer-linked nodes, in the cache. With SDC, programmers can (selectively) insert only pointers to the next node in the cache. With a much reduced data set for caching, the pointers can stay in the cache to enable efficient in-cache pointer chasing.

Improving index search performance. Traversal on index structures can be a frequent and expensive operation. There have been many efforts on optimization of index data structures, such as cuckoo hashing [43, 44], Masstree [32], Wormhole [53], CSS-Trees (Cache-Sensitive Search Trees) [38], and B^+-Trees (Cache-Sensitive B^+-Trees) [39]. These efforts are limited to specific indexes. In contrast, SDC can help to avoid many traversals on any of the indexes with expected access temporal locality. As mentioned, SLB is a software-based technique to collect pairs of key and its corresponding index search result into a memory buffer. With strong spatial and temporal localities data in the buffer will also stay in the CPU cache [52]. Compared with SDC, SLB may not be aware of the cache size and demands on cache space from other data structures and/or processes. Therefore, it does not know whether a data item is cached and does not have a control over it. Further, cost of the buffer management, including data admission and replacement, can be expensive. SDC addresses these inadequacies.

Index search can also be accelerated with hardware supports by designing new specialized hardware components [20, 26, 33] or leveraging newly available hardware features [8, 19, 54, 56]. Widx, an on-chip accelerator for database hash index lookups, is such an example. To use Widx programmers must disclose how keys are hashed into hash buckets and how walk on the nodes in the bucket is conducted. In addition to its limited applicability only on hash index, Widx increases programmers' burden and is in a sharp contrast with SDC, which does not require any knowledge on how the search is actually conducted. There are proposals on using new vector instructions for hash table search [20, 33]. However, their usage is usually limited to Vectorwise database systems.

Memoization technique. Memoization technique has been used in memory systems to cache the results of repetitive computations and allow the program to skip them. It can be implemented using software and hardware. The software memoization [15, 45] suffers from high runtime cost and is only beneficial for long memoization tasks [9, 49]. The hardware implementation [9, 49] incurs significant hardware overheads as it needs to introduce dedicate memoization caches. Since not all applications can benefit from the caches, it can waste a lot of chip area without benefits and compromise power efficiency. MCache [55] stores memoization tables in memory and allows them to share cache capacity with conventional data. Different from existing solutions, SDC reuses the regular cache space on-demand for caching frequently-accessed index entries and only needs minor changes to existing cache structures. SDC can achieve the same goal of memoization technique with low hardware cost and being easy-to-use for users.

Software-controlled use of CPU cache. There have been studies on incorporation of user knowledge on data access in the cache management. Various techniques have been proposed, from cache control instructions [40] to scratchpad memory [4]. User programs can use the instructions to disclose memory access patterns, such as data items that will only be used once. There are also instructions for prefetching a cache line, resetting data in a cache line to 0, and invalidating or evicting a cache line. Compared to SDC, these instructions add only limited capability to make the cache more effectively used. In contrast, scratchpad is a technique that removes an important cache feature – transparently mapping memory addresses to cache addresses – to allow great control over its use by providing instructions to move data to and from the main memory. A major disadvantage of scratchpad is its introduced new address space (on scratchpad) that is different from memory address space assumed in user programs and makes it very hard to adapt programs to the platform. Thus, scratchpads work poorly with irregular or input-dependent data sets [27, 34]. They are rarely used in mainstream processors and often used in embedded systems and special-purpose processors. In contrast, instead of removing the memory address space, SDC enables an address space that is closer to and more friendly to user programs' semantics (the user-defined keys) to customize cache uses. Further, SDC does not require a separate fast memory. It dynamically allocates some space from existing cache for user-defined use and enables efficient space sharing.

There have been works in the literature named as software-defined caching [5] or software-defined cache hierarchies [48]. They allow programs to define shares, which are collections of cache banks, and accordingly to have control over data placement and capacity allocation either at one cache level (Jigsaw [5]) or across a cache hierarchy (Jenga [48]). They function at a coarser grain for fitting working sets into the shares, while SDC allows user programs to control caching of individual data items.

5 CONCLUSION

We propose SDC, a user-programmable CPU cache architecture that enables key-value based data caching to allow programs to place their selected data items in the cache for quick access. Our design minimizes programming effort of using the user-controllable look-aside cache, and enables rich performance optimization opportunities with a customized use of the CPU cache. SDC seamlessly integrates into existing cache architecture. And its implementation requires moderate changes to current cache circuitry. With a prototype implementation in gem5, we extensively conduct experiments with various workloads. The results show that SDC improves the performance of index-based data management programs by up to 5.3× over existing cache design.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable comments and feedback. This work was supported in part by NSF grants CNS-1527076 and CCF-1815303.

REFERENCES

[1] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 27–39. http://dl.acm.org/citation.cfm?id=3154690.3154694

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1629575.1629577

[3] 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. https://doi.org/10.1145/2254756.2254766

[4] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*. IEEE, IEEE, Estes Park, CO, USA, 73–78.

[5] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable Software-defined Caches. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 213–224. http://dl.acm.org/citation.cfm?id=2523721.2523752

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] W. A. Burkhard. 1976. Hashing and Trie Algorithms for Partial Match Retrieval. *ACM Trans. Database Syst.* 1, 2 (June 1976), 175–187. https://doi.org/10.1145/320455.320469

[8] 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. https://doi.org/10.1145/2485922.2485945

[9] Daniel Citron and Dror G Feitelson. 2000. *Hardware memoization of mathematical and trigonometric functions*. Technical Report. The Hebrew University of Jerusalem.

[10] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. 2002. Pointer Cache Assisted Prefetching. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 62–73. http://dl.acm.org/citation.cfm?id=774861.774869

[11] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. ACM, New York, NY, USA, 14–25. https://doi.org/10.1145/379240.379248

[12] HyperTransport Technology Consortium et al. 2008. HyperTransport I/O link specification. *Revision* 1 (2008), 111–118.

[13] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A Stateless, Content-directed Data Prefetching Mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 279–290. https://doi.org/10.1145/605397.605427

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] Yonghua Ding and Zhiyuan Li. 2004. A Compiler Scheme for Reusing Intermediate Computation Results. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 279–. http://dl.acm.org/citation.cfm?id=977395.977679

[16] E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, Raleigh, NC, USA, 7–17. https://doi.org/10.1109/HPCA.2009.4798232

[17] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[18] gem5. 2014. Gem5-Classic Memory System. http://www.gem5.org/Classic_Memory_System.

[19] 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, 6 pages. https://doi.org/10.1145/1114252.1114260

[20] 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. https://doi.org/10.1109/MICRO.2012.24

[21] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, Phoenix, USA, 25–32. https://doi.org/10.1109/ICCD.2016.7753257

[22] INTEL. 2013. Intel Haswell processors. http://www.7-cpu.com/cpu/Haswell.html.

[23] Intel. 2016. Intel Xeon Processor E5-2683 v4. https://ark.intel.com/products/91766/Intel-Xeon-Processor-E5-2683-v4-40M-Cache-2-10-GHz-.

[24] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. ACM, New York, NY, USA, 252–263. https://doi.org/10.1145/264107.264207

[25] M. Karlsson, F. Dahlgren, and P. Stenstrom. 2000. A prefetching technique for irregular accesses to linked data structures. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. IEEE, Touluse, France, 206–217. https://doi.org/10.1109/HPCA.2000.824351

[26] 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. https://doi.org/10.1145/2540708.2540748

[27] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 707–719. https://doi.org/10.1145/2749469.2750374

[28] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[29] Mikko H Lipasti, William J Schmidt, Steven R Kunkel, and Robert R Roediger. 1995. SPAID: Software prefetching in pointer-and call-intensive environments. In *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*. IEEE, IEEE, Ann Arbor, MI, USA, 231–236.

[30] Chi-Keung Luk. 2001. Tolerating Memory Latency Through Software-controlled Pre-execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. ACM, New York, NY, USA, 40–51. https://doi.org/10.1145/379240.379250

[31] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, USA, 222–233. https://doi.org/10.1145/237090.237190

[32] 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. https://doi.org/10.1145/2168836.2168855

[33] Rich Martin. 1996. *A Vectorized Hash-Join*. Technical Report. University of California at Berkeley, California, USA.

[34] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving Dynamic Cache Management with Static Data Classification. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 113–127. https://doi.org/10.1145/2872362.2872363

[35] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[36] Brendan O'Connor. 2011. How much text versus metadata is in a tweet. http://goo.gl/EBFIFs.

[37] François Panneton and Pierre L'Ecuyer. 2005. On the Xorshift Random Number Generators. *ACM Trans. Model. Comput. Simul.* 15, 4 (Oct. 2005), 346–361. https://doi.org/10.1145/1113316.1113319

[38] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89. http://dl.acm.org/citation.cfm?id=645925.671362

[39] Jun Rao and Kenneth A. Ross. 2000. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 475–486. https://doi.org/10.1145/342009.335449

[40] Freescale Semiconductor. 2005. PowerPC e500 Core Family Reference Manual. https://goo.gl/Jjs38u

[41] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, Washington, DC, USA, 63–74. https://doi.org/10.1109/HPCA.2007.346185
[42] Nitish Kumar Srivastava and Akshay Dilip Navalakha. 2018. Pointer-Chase Prefetcher for Linked Data Structures. *CoRR* abs/1801.08088 (2018), 12. arXiv:1801.08088 http://arxiv.org/abs/1801.08088
[43] 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)*. IEEE, Santa Clara, California, USA, 1–7. https://doi.org/10.1109/MSST.2015.7208292
[44] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. 2017. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 553–565. https://www.usenix.org/conference/atc17/technical-sessions/presentation/sun
[45] Arjun Suresh, Erven Rohou, and André Seznec. 2017. Compile-time Function Memoization. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/3033019.3033024
[46] Symas. 2016. LMDB: Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/index.html.
[47] Thomas Wang. 2007. Integer Hash Function. http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm.
[48] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE/ACM, Toronto, Ontario, Canada, 652–665. https://doi.org/10.1145/3079856.3080214
[49] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. 2007. Design and Evaluation of an Auto-memoization Processor. In *Proceedings of the 25th Conference on Proceedings of the 25th IASTED International Multi-Conference: Parallel and Distributed Computing and Networks (PDCN'07)*. ACTA Press, Anaheim, CA, USA, 245–250. http://dl.acm.org/citation.cfm?id=1295581.1295621
[50] Stephen Tu. 2013. Silo source code on Github. https://github.com/stephentu/silo.
[51] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713
[52] Xingbo Wu, Fan Ni, and Song Jiang. 2017. Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 27–39. https://doi.org/10.1145/3127479.3127483
[53] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955
[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. https://doi.org/10.14778/2536206.2536210
[55] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 59–63. https://doi.org/10.1109/LCA.2017.2762308
[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. https://doi.org/10.14778/2809974.2809984
[57] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-based Multicore Cache Management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 89–102. https://doi.org/10.1145/1519065.1519076
[58] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. 2010. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. IEEE, IEEE, Mountain View, California, US, 1–6.