Leveraging Caches to Accelerate Hash Tables and Memoization

Guowei Zhang
zhanggw@csail.mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Daniel Sanchez
sanchez@csail.mit.edu
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

ABSTRACT

Hash tables are widely used, but they are inefficient in current systems: they use core resources poorly and suffer from limited spatial locality in caches. To address these issues we propose HTA, a technique that accelerates hash table operations via simple ISA extensions and hardware changes. HTA adopts an efficient hash table format that leverages the characteristics of caches. HTA accelerates most operations in hardware, and leaves rare cases to software.

We present two implementations of HTA, Flat-HTA and Hierarchical-HTA. Flat-HTA adopts a simple, hierarchy-oblivious layout and reduces runtime overheads with simple changes to cores. Hierarchical-HTA is a more complex implementation that uses a hierarchy-aware layout to improve spatial locality at intermediate cache levels. It requires some changes to caches and provides modest benefits over Flat-HTA.

We evaluate HTA on hash table-intensive benchmarks and use it to accelerate *memoization*, a technique that caches and reuses the outputs of repetitive computations. Flat-HTA improves the performance of the state-of-the-art hash table-intensive applications by up to 2×, while Hierarchical-HTA outperforms Flat-HTA by up to 35%. Flat-HTA also outperforms software memoization by 2×.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; *Serial architectures*; *Parallel architectures.*

KEYWORDS

hash table, memoization, cache, microarchitecture, specialization

ACM Reference Format:
Guowei Zhang and Daniel Sanchez. 2019. Leveraging Caches to Accelerate Hash Tables and Memoization. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358272

1 INTRODUCTION

The impending end of Moore's Law is making transistors a scarce resource. Therefore, it is crucial to investigate new abstractions and mechanisms that span hardware and software to make better use of existing architectural components. In this work we focus on providing architectural support to accelerate *hash table* operations. Hash tables are widely used and consume the majority of cycles on key applications in databases [22] and genomics [31]. While hash tables have been extensively studied and optimized in software, they leave significant performance on the table in current systems due to an inexpressive hardware-software interface.

Specifically, we observe that hash tables suffer from two key inefficiencies in conventional systems (Sec. 2):

(1) **Poor core utilization:** Each hash table operation consists of a long sequence of instructions to compute hash values, memory accesses to keys and values, and comparisons. These instructions include hard-to-predict, data-dependent branches that add wasted cycles, and incur long-latency cache misses that limit instruction-level parallelism.
(2) **Poor spatial locality:** To reduce mapping conflicts, hashing spreads key-value pairs uniformly across the hash table's allocated memory. This causes poor spatial locality when key-value pairs have mixed reuse, as the same-line neighbors of a frequently accessed pair are rarely accessed. This wastes a significant portion of cache capacity.

To address these problems we propose HTA, a technique that accelerates hash table operations through a combination of expressive ISA extensions and simple hardware changes (Sec. 3). HTA adopts a hash table format that leverages the associative nature of caches. HTA introduces new instructions to perform hash table lookups and updates. These instructions are designed to leverage existing core structures and prediction mechanisms. For example, hash table lookups have branch semantics and thus leverage the core's branch predictors to avoid control-flow stalls. With a simple HTA function unit, these instructions consume far fewer pipeline resources than conventional hash table operations, allowing more instruction-level and memory-level parallelism to be exploited. HTA accelerates most hash table operations, leaving rare cases to a *software path* that allows overflowing to conventional software hash tables.

We present two implementations of HTA, Flat-HTA (Sec. 4) and Hierarchical-HTA (Sec. 5). Both implementations introduce simple changes to cores to reduce runtime overheads. Flat-HTA adopts a simple, hierarchy-oblivious layout that works well for hash tables with uniform reuse. Hierarchical-HTA adopts a multi-level, hierarchy-aware layout that lets fast caches hold more frequently accessed key-value pairs, improving spatial locality when hash tables have mixed reuse. Hierarchical-HTA requires changing cache controllers and provides modest benefits over Flat-HTA. These implementations do not reserve space in caches. Instead, they dynamically share cache capacity with non-HTA data.

We evaluate HTA on hash table-intensive benchmarks and use it to accelerate *memoization*, a technique that caches the results of repetitive computations, allowing the program to skip them (Sec. 6).

MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358272

Flat-HTA accelerates hash table-intensive applications by up to 2×, while Hierarchical-HTA outperforms Flat-HTA by up to 35%. Moreover, HTA outperforms software memoization by 2× and achieves comparable performance to conventional hardware memoization but without the need for specialized on-chip storage.

2 BACKGROUND

Hash tables are unordered associative containers. They hold *key-value pairs* and support three operations: *lookups* to retrieve the data associated with a particular key, and *insertions* and *deletions* to add or remove key-value pairs. Hash tables perform these operations with amortized constant-time complexity. They are heavily used in many domains, like databases [22], key-value stores [23, 24, 25], networking [37], genomics [31], and memoization [32, 38].

A hash table is typically implemented using an array to hold key-value pairs, which is indexed by a *hash* of the key. A *collision* happens when multiple key-value pairs map to the same array index. Collisions become more common as array utilization grows. To support high utilization, hash tables include a collision resolution strategy, such as probing additional locations, and resize the table when its utilization reaches a certain threshold.

Implementations vary in several aspects, like hash function selection, collision resolution, and resizing mechanisms. Simple hash functions such as XOR-folding and bit selection [20] are fast but are prone to hotspots, while more complex hash functions such as universal hashing [4] distribute key-value pairs more uniformly but incur more overheads. Basic collision resolution strategies include chaining and open addressing [28]. Upon a collision, chaining appends the new key-value pair to the existing ones, forming a linked list, while open addressing probes other positions in the hash table. Resizing can be performed all-at-once or incrementally.

There is a wide range of hash table implementations with different algorithmic tradeoffs, e.g., trading space efficiency for lookup efficiency [33]. For instance, Cuckoo hashing [33] improves space efficiency and worst-case lookup performance at the cost of increasing average-case lookup complexity. Its variants further focus on either reducing the memory accesses per lookup [3] or improving locality [16].

2.1 Hash table performance analysis

Despite the wide range of hash table implementations, we observe that state-of-the-art designs suffer from two issues: *poor core utilization* and *poor spatial locality*:

1. Poor core utilization adds overheads that limit the performance of many hash table-intensive applications [22]. To analyze the causes of high overheads, we evaluate three common hash table operations under different hash table implementations using detailed simulation (see Sec. 7 for methodology details). We use two state-of-the-art software baselines, `libstdc++`'s C++11 `unordered_map` and Google's `dense_hash_map`, as well as Flat-HTA.

Fig. 1 compares the execution time (lower is better) of each implementation under three cases: (a) lookups, (b) updates, and (c) insertions. In (a) and (b), each hash table is initialized with 1 million randomly generated key-value pairs, and has a footprint of about 64 MB. Then, the program performs back-to-back lookups or updates to existing, randomly chosen keys. In (c), each hash table starts empty and 1 million distinct, randomly chosen key-value pairs are inserted into it. Over time, the hash table grows to accommodate the inserted pairs.

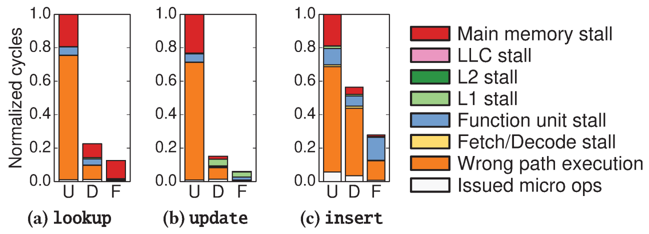


Figure 1: Execution time and cycle breakdown of three hash table microbenchmarks using three hash table implementations: libstdc++'s C++11 Unordered map, Google's Dense hash map, and Flat-HTA.

Fig. 1 breaks down execution time into the cycles cores spend on different activities, following the CPI stack methodology [15]. Specifically, each bar shows the cycles cores spend *(i)* issuing committed instructions; *(ii)* executing wrong-path instructions due to a branch misprediction; *(iii)* stalled (i.e., unable to issue) due to the frontend (fetch or decode); and *(iv)* stalled due to different backend events: functional units, L1 cache, L2 cache, LLC, or main memory.

Fig. 1 reveals two key sources of overhead in software hash tables: hard-to-predict branches and underutilized backend parallelism.

First, Fig. 1 shows that hard-to-predict branches in hash table probing add many cycles: up to 74% of cycles are spent on wrong-path execution. This is because, in both `unordered_map` and `dense_hash_map`, such branches direct the control flow to either the end of an operation or another hash table probing. These branches depend on data loaded from memory, so they take a long time to resolve and are challenging for branch predictors.

Second, hash table operations make poor use of backend resources to exploit instruction-level and memory-level parallelism. Each hash table operation takes a sequence of instructions including hash calculation, memory accesses, comparisons, and branches. These instructions occupy tens to hundreds of micro-op (µop) slots, comparable to the reorder buffer size. As shown in Fig. 1, this limits memory-level parallelism significantly: most backend stalls are spent waiting for main memory responses, and the reorder buffer does not have enough resources to overlap multiple misses.

Flat-HTA effectively reduces these overheads and improves performance by up to 2.5×. First, its design avoids hard-to-predict branches, reducing or even eliminating wrong-path execution. Second, each hash table operation takes far fewer µop slots, improving memory-level parallelism and reducing backend stalls by up to 5.6×.

2. Poor spatial locality is the other issue in software hash tables [3]. Hash tables spread key-value pairs uniformly across the table's allocated memory. This hinders spatial locality, as the neighboring pairs of a frequently-accessed pair are usually not frequently accessed. This wastes a significant portion of cache capacity.

To illustrate this, we design a microbenchmark similar to the previous ones. First, we size the hash tables to occupy 256 MB and pre-insert 1 million key-value pairs. By sizing the hash tables to 256 MB instead of their natural 64 MB, we keep hash table load artificially low to reduce branch mispredictions in the software versions (at this low load, the first probe almost always succeeds). This is, however, space-inefficient. Then, the benchmark performs a series of dependent lookups to a subset of 8 thousand keys.

Fig. 2 shows the execution time and cycle breakdown for the previous hash table implementations plus Hierarchical-HTA. Since there are no branch mispredictions and lookups depend on each other, the benchmark has limited parallelism. As a result, Flat-HTA outperforms the best software implementation by only 1%, as it spends 80% of cycles waiting for LLC responses.

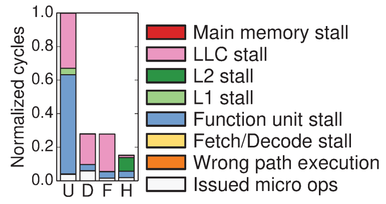


Figure 2: Execution time and cycle breakdown of the mixed-reuse microbenchmark for the previous hash tables and Hierarchical-HTA.

By contrast, by adopting a multi-level hierarchy-aware hash table layout, Hierarchical-HTA densely packs frequently accessed key-value pairs in lower-level caches, reducing misses. As shown in Fig. 2, Hierarchical-HTA serves most of the lookups from the L2 instead of the LLC, outperforming Flat-HTA by 84%.

2.2 Prior work in accelerating hash tables

Prior work has introduced hardware support to reduce hash table overheads. In databases, prior work [18, 22] leverages the inter-key parallelism of database operators such as join to exploit data-level parallelism. These techniques optimize the throughput but not the latency of hash table accesses.

Near-memory [26] and near-storage acceleration [40, 46] bypass the cache hierarchy entirely, and are a sensible choice when operating on large hash tables with no locality. They do avoid the spatial locality problems of caching hash tables. However, they incur high latency and work poorly when hash table accesses have locality.

Other hardware techniques introduce specialized hardware units for hashing and comparisons instead of using the processor pipeline, and allocate dedicated on-chip storage for hash tables. They are typically specialized for applications such as PHP processing [17] and distributed key-value stores [5, 25]. Like HTA, these techniques do reduce the latency of hash table operations. Unlike HTA, these techniques introduce large storage structures that rival or exceed the area of the L1 cache. For example, Da Costa et al.'s proposal to accelerate memoization [11] consumes 98 KB. However, not all applications can benefit from this storage. In these cases, this dedicated storage not only wastes area that could otherwise be devoted to caches, but also hurts energy consumption [7].

By contrast, HTA is general and optimizes both the throughput and the latency of hash table operations. HTA avoids specialized on-chip storage by storing hash tables in caches, so they share scarce on-chip memory capacity with other program data.

2.3 Memoization

Memoization is a technique to improve performance and energy efficiency. Memoization caches the results of repetitive computations, allowing the program to skip them. Memoized computations must be pure and depend on few, repetitive inputs. Memoization is the cornerstone of many important algorithms, such as dynamic programming, and is widely used in many languages [29, 30], especially functional ones. It was first introduced by Michie in 1968 [32]. Since then, it has been implemented using software and hardware, but both have significant drawbacks, which we address with HTA.

Software memoization relies on hash tables to memoize input-output pairs. The high runtime overheads of hash tables hamper software memoization significantly. As we later show, many memoizable functions are merely 20 to 150 instructions long, comparable to or even cheaper than hash table lookups. Memoizing them is harmful. For example, Citron and Feitelson [8] show that software memoization incurs significant overheads on short functions: when memoizing mathematical functions indiscriminately, software memoization incurs a 7% performance loss, while a hardware approach yields a 10% improvement. To avoid poor performance, software schemes must apply memoization selectively, relying on a careful cost-benefit analysis of memoizable regions, done by either compilers [13, 38], profiling tools [12], or programmers [39].

Prior work has proposed hardware support to accelerate memoization [8, 38, 42], and thus can unlock more memoization potential. Much prior work on hardware memoization focuses on automating the detection of memoizable regions at various granularities [11, 19, 36, 42], while others rely on ISA and program changes to select memoizable regions [8, 9, 38]. However, all prior hardware techniques require dedicated storage for memoization tables. Such tables require similar or even larger sizes than L1 caches. Therefore, they incur significant area and energy overheads [7], especially for programs that cannot exploit memoization.

Other prior work has proposed architectural [43] or runtime [34] support to track implicit inputs/outputs of memoized functions, enabling memoizing impure functions. This is orthogonal to the acceleration of hash tables, which is the focus of our work. HTA could be easily combined with them.

3 HTA HARDWARE/SOFTWARE INTERFACE

HTA is a Hash Table Acceleration technique. HTA handles most hash table accesses in hardware, and leaves rare cases such as overflows and table resizing to a slow software path. HTA introduces two key software-visible features to accelerate hash table operations in hardware.

First, HTA adopts a format for hash tables that exploits the characteristics of caches to make lookups and updates fast. HTA stores hash tables in cacheable memory. This avoids the large costs of specialized hardware caches used by prior hardware techniques. HTA does not statically partition cache capacity between hash table data and normal program data. Instead, both types of data are managed by the unified cache replacement policy, and share cache capacity dynamically based on access patterns.

Second, HTA introduces hash table instructions for lookups and updates that are amenable to a fast and simple implementation. Whereas software hash table lookups use multiple instructions and hard-to-predict branches, HTA hash table lookups are done through a single instruction with branch semantics. The outcome of a lookup (resolved or not) can be predicted accurately by the core's existing branch predictors, avoiding most control-flow stalls.

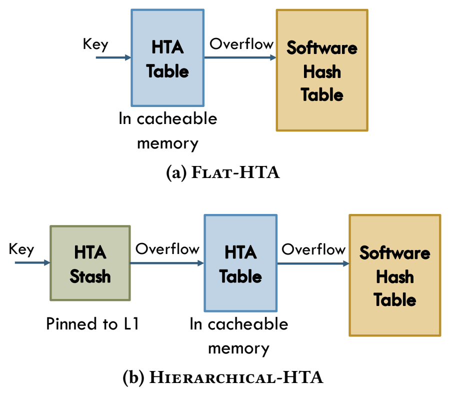


Figure 3: Overview of HTA implementations.

Fig. 3 gives an overview of how both HTA implementations, Flat-HTA and Hierarchical-HTA, use these features. Flat-HTA (Fig. 3a) stores key-value pairs across an *HTA table* and a software hash table. The HTA table is stored in cacheable memory, and may be spread across multiple caches or main memory. The HTA table is sized to hold most key-value pairs, and the software hash table is used as a victim cache, to hold pairs that overflow the HTA table.

Hierarchical-HTA (Fig. 3b) extends Flat-HTA by letting cache levels retain individual key-value pairs rather than cache lines. Specifically, they cache key-value pairs of the HTA table in small, cache-level-specific regions called *HTA stashes*. A pair that overflows an HTA stash is handled by the next level. This improves spatial locality at intermediate caches levels, as their lines fill up with frequently-accessed pairs. However, Hierarchical-HTA does not improve spatial locality at the last-level cache (doing so would complicate the interface with main memory), so its benefits over Flat-HTA are modest. Fig. 3b shows an example of Hierarchical-HTA with one HTA stash pinned to the L1.

We now describe the ISA changes common to Flat-HTA and Hierarchical-HTA, then describe their implementations.

3.1 HTA hash table format

The HTA table is stored in a contiguous, fixed-size region of memory, as shown in Fig. 4.

HTA uses a storage format designed to leverage the characteristics of caches. Each cache line stores a fixed number of key-value pairs. For example, Fig. 4 shows the format of a 64-byte cache line for a hash table with 128-bit keys and 64-bit values. A given entry can map to a single cache line, but can be stored in any position within the line. A lookup thus requires hashing the input data to produce a cache line index, fetching the line at that index (as shown in Fig. 4), and comparing all keys in the line. This design requires accessing a single cache line, but retains associativity within a line to reduce collisions. To avoid the need for valid bits, HTA initializes each line's entries with *invalid* key values, which are simply keys that hash to a different line.

HTA leaves collision resolution to software. Specifically, there may be *overflowed* key-value pairs that cannot be stored in a line due to capacity constraints. These overflowed pairs are handled by the software path, which stores them in the software hash table.

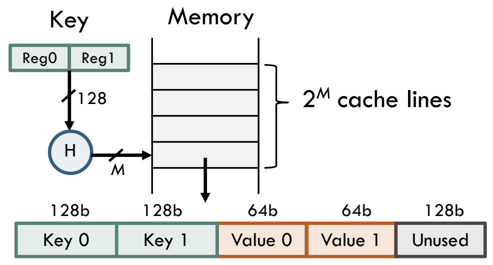


Figure 4: HTA table format.

3.2 HTA ISA extensions

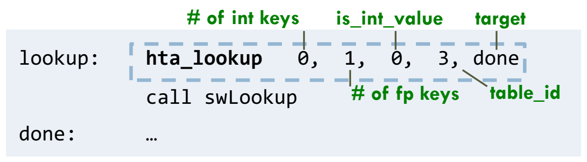
HTA stores a small number of *HTA table descriptors* in architectural registers. Each descriptor holds the table's starting address and its size. Our implementations support four HTA table descriptors. If the program uses more than four hash tables, it should manage their descriptors accordingly, loading them into registers before operating on the hash table.

HTA adds four instructions to perform hash table operations: `hta_lookup`, `hta_update`, `hta_swap`, and `hta_delete`. These instructions have branch semantics. Fig. 5 and Fig. 6 show sample code that uses them to implement single-threaded lookups and insertions. (Sec. 4.4 describes how these instructions are used to implement thread-safe hash tables for multithreaded applications.)

1. `hta_lookup` performs a lookup in the HTA hash table whose descriptor is specified by **table_id**. `hta_lookup` supports keys with up to four integer or floating-point words and a single integer or floating-point value, all stored in registers. As shown in Fig. 5, `hta_lookup` stores the number of integer and floating-point key registers, and the core decodes them to a fixed set of registers. We choose the same register mappings as the ISA's calling convention. For instance, in x86-64, **num_int_keys** = 2 means that the 64-bit values in registers `rdi` and `rsi` are used as a 128-bit key. Similarly, the **is_int_value** indicates whether the value is integer or floating-point. In x86-64, either `rax` or `xmm0` will be used accordingly.

If the lookup is resolved, i.e., the key is found or the line is not full, `hta_lookup` acts as a taken branch. It jumps to the **target** PC encoded in the instruction (in PC-relative format), sets the overflow flag to indicate whether the lookup succeeds, and also updates the result register with the corresponding value. If the lookup is not resolved, i.e., the key is not found *and* the line is full, `hta_lookup` acts as a non-taken branch, and continues to execute the next instruction.

2. `hta_update` is used to update the HTA hash table. Like `hta_lookup`, `hta_update` encodes the key and value registers, and the table id. If the key is found or the line is not full, `hta_update` updates the pair in the cache line and jumps to the **target** PC.



```
lookup:   hta_lookup  0, 1, 0, 3, done
          call swLookup
done:     …
```

Figure 5: Example showing how `hta_lookup` is used to implement a singled-threaded hash table lookup.

Otherwise, if the key is not found and the line is already full, `hta_update` does not modify anything and continues to execute the next instruction.

3. `hta_swap` attempts to insert a pair more aggressively than `hta_update`. Similar to `hta_update`, `hta_swap` encodes the key and value registers, and the table id. Upon a `hta_swap`, if the key is found or the line is not full, `hta_swap` performs the same operations as `hta_update`: it updates the pair in the cache line and jumps to the **target** PC. However, if the key is not found and the line is full, `hta_swap` selects a victim pair randomly to make space for the update. The victim's key and value are placed in registers and `hta_swap` acts as a non-taken branch, letting the software path finish the update, e.g., by inserting the victim pair to the software hash table. Such distinction between `hta_update` and `hta_swap` is useful for thread-safe hash tables (Sec. 4.4).

4. `hta_delete` removes a key-value pair with a matching key. Its format is identical to `hta_lookup`. If the key is found, it is replaced with a special *deleted* key, and the instruction acts as a taken branch. Otherwise, `hta_delete` acts as a non-taken branch to take the software path.

Deleted key values must be different from invalid key values, as `hta_lookup` should not interpret a deleted key as empty space (so that lookups do not miss pairs that overflowed to the software table), but `hta_update` and `hta_swap` should interpret a deleted key as empty space. In all, HTA uses four pre-specified key values: it chooses two small key values that do not map to line 0 as line 0's invalid and deleted key values, and two small key values that map to line 0 as the invalid and deleted key values for all other lines.

Single-threaded lookups: Fig. 5 shows the implementation of a hash table lookup with HTA instructions. The lookup begins with the `hta_lookup` instruction. A resolved `hta_lookup` jumps to `done` and continues program execution. Otherwise, the program goes through the software path to perform a lookup in the software hash table. In this way, the HTA hash table behaves as an *exclusive cache* of the conventional software hash table, allowing most of the accesses to be handled quickly. The software path is rarely executed, and therefore introduces little performance impact.

Single-threaded insertions: Similarly, Fig. 6 illustrates how to implement an insertion with HTA (if a pair with the same key already exists, an insertion updates its value). Either `hta_swap` or `hta_update` can be used. A resolved `hta_swap` instruction jumps to `done`, skipping the software path. An unresolved `hta_swap` runs through the software path, which *(i)* inserts the victim pair to the software hash table, and *(ii)* checks whether the software hash table has a pair with the same key as the newly inserted pair, and removes it if so, as this old, overflowed pair is now stale.

3.3 ISA design alternatives

We have designed the HTA ISA to integrate well in x86 processors: HTA instructions are encoded in a compact format and are decoded into multiple μops upon execution (Sec. 4.1). An alternative RISC-style implementation is also possible, e.g., by exposing the different μops as instructions. However, this design choice is not important: as shown in Fig. 1 and Fig. 2, the time spent on frontend stalls and issuing μops is negligible, so using CISC- vs. RISC-style instructions would not significantly change the results. The key benefit of HTA is to reduce wrong-path execution and backend stalls.

```
insert:    hta_swap   0,  1,  0,  3, done
           call swHandleInsert
done:      …
```

Figure 6: Example showing how `hta_swap` is used to implement a single-threaded hash table insertion.

4 FLAT-HTA IMPLEMENTATION

As shown in Fig. 3a, Flat-HTA uses a single-level HTA table stored in cacheable memory. Flat-HTA substantially reduces overheads over software hash tables, but still suffers from poor spatial locality.

4.1 Core pipeline changes

Flat-HTA requires simple changes to cores, shown in Fig. 7. We add a simple functional unit that executes lookup, update, swap, and delete instructions. This unit is fed the values of key registers, possibly over multiple cycles, as well as the table id.

For an `hta_lookup` instruction, the unit first hashes the input values and table size to find the line index. We restrict the system to use power-of-2 sizes for each HTA table. We use the x86 CRC32 instruction to compute the hash value; other ISAs could implement CRC or a different cheap hash [20]. We find that CRC produces good distributions in practice.

After hashing, the HTA functional unit loads the appropriate cache line, compares all the keys, and outputs whether the the software path can be skipped, whether there's a match, as well as the corresponding result if so. `hta_update` and `hta_swap` are similar, but the functional unit also takes the value to update, and stores the pair in the appropriate line. `hta_delete` is also similar, but does not return a value.

HTA leverages existing core mechanisms to improve performance. We integrate these instructions into an out-of-order, superscalar x86-64 core similar to Intel's Haswell (see Sec. 7). The frontend treats HTA instructions as branches. This way, HTA instructions leverage the existing branch target buffer and branch predictor to predict whether the code following each instruction can be skipped. Thus, the core overlaps the execution of lookups and updates with either the execution of the software path (if a resolution is not predicted) or its continuation (if a resolution is predicted). We find that this effectively hides the latency of HTA instructions.

In our implementation, the backend executes `hta_lookup` using multiple RISC μops. The decoder produces one or more μops that feed each input register to the HTA functional unit, an HTA μop that instructs the functional unit to start, a branch-resolution μop,

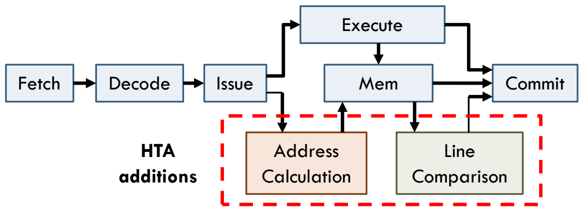


Figure 7: HTA core pipeline changes.

Circuit	Address calculation	Line comparison	Total
Area(μm^2)	6,173	9,176	15,349
Area(%core)	0.022	0.033	0.055

Table 1: Area of the HTA functional unit on a 45 nm process.

and, if the lookup is predicted to be resolved, a µop to move the lookup result into its destination register. The other instructions use a similar implementation.

4.2 Hardware costs

We implement the HTA functional unit in RTL and synthesize it using yosys [45] and the 45 nm FreePDK45 standard cell library [21]. The functional unit meets a target 3 GHz frequency. The address calculation circuit mainly consists of a 64-bit adder, 64 AND gates, and registers to store the four HTA table descriptors. The line comparison circuit includes comparators to search for a given key and an empty slot in parallel. Table 1 reports the area consumed by these components. Overall, the functional unit takes just 0.055% of the area of a Nehalem core [14], which was manufactured in a 45 nm process as well. Thus, HTA's area overheads are negligible.

4.3 Software path

The software path performs lookups and updates to a conventional software hash table. It handles the rare overflowed accesses to HTA. Besides, the software path also resizes the HTA table dynamically. The resizing algorithm HTA adopts is based on comparing the fraction of HTA accesses that take the software path with a threshold (e.g., 1%). If the fraction is above the threshold, the software path doubles the size of the HTA table and reinserts all existing elements in both the HTA table and the software hash table.

To keep track of this fraction, each HTA table is assigned a counter that is stored in memory at one word above its starting address. The counter is incremented rarely (every 100 HTA accesses in our implementation), and hence approximately monitors the number of HTA accesses of the table.

The software path also maintains a counter recording the number of software path invocations. The software path uses these counters to calculate the fraction of accesses that overflow, and decides whether to resize the HTA table.

4.4 Parallel hash table implementation

With multiple threads, the simple hash table operations shown in Fig. 5 and Fig. 6 need some refinement to be thread-safe. We leverage that HTA instructions are atomic (cores already have the machinery to ensure this for all instructions, such as line locking or verification loads). This guarantees the atomicity of operations that do not invoke the software path.

If the software path is invoked, a synchronization strategy is needed to guarantee atomicity. We use fine-grain locks, each of which protects a few lines (four in our implementation). However, HTA is orthogonal to the synchronization technique used by the software path, and can use other techniques. For example, it could be combined with transactional memory.

Thread-safe lookups: Fig. 8 shows our thread-safe implementation of lookups. The software path involves acquiring the line's lock;

```
lookup:    hta_lookup   0,  1,  0,  3, done
           call swLockLine
           hta_lookup   0,  1,  0,  3, release
           call swLookup
release:   call swUnlockLine
done:      …
```

Figure 8: Example showing how `hta_lookup` is used to implement a thread-safe hash table lookup. The software path uses fine-grain locks.

```
insert:    hta_update   0,  1,  0,  3, done
           call swLockLine
           hta_swap     0,  1,  0,  3, release
           call swHandleInsert
release:   call swUnlockLine
done:      …
```

Figure 9: Example showing how HTA instructions are used to implement a thread-safe insert. The software path uses fine-grain locks.

executing the `hta_lookup` instruction again; if needed, accessing the software hash table; and finally releasing the lock. `hta_lookup` must be invoked again after locking to avoid races with insertions.

Thread-safe insertions: Fig. 9 shows code for thread-safe updates. This code shows why `hta_update` and `hta_swap` are both needed: `hta_update` does not modify HTA table state if the software path is invoked. This is important to avoid races: by using `hta_swap` only after locking, all modifications are properly synchronized.

5 HIERARCHICAL-HTA IMPLEMENTATION

Hierarchical-HTA extends Flat-HTA to cache individual key-value pairs of the HTA table in cache-specific regions called HTA *stashes* (Fig. 3b). A stash's lines can only be stored in a specific cache level. Stashes do not reserve any capacity in their cache (i.e., they do not partition the cache). Instead, similar to Flat-HTA, each stash shares capacity with normal program data, and the actual capacity a stash consumes depends on the workload's access pattern.

This hierarchy-aware layout improves spatial locality on intermediate cache levels, improving cache utilization and reducing misses. Whereas Flat-HTA only requires changes to the core, Hierarchical-HTA also modifies cache controllers so that they can fetch and serve key-value pairs rather than cache lines. However, Hierarchical-HTA does not improve spatial locality at the LLC, as making the LLC manage key-value pairs rather than lines would complicate the interface main memory (which is optimized for wide transfers). Therefore, Hierarchical-HTA yields only modest gains over Flat-HTA.

HTA table restrictions: For simplicity, we introduce some restrictions on the backing HTA table: it must be in a contiguous region of *physical* memory, must be power-of-two sized, and must be size-aligned. (Flat-HTA tables live in pageable virtual memory so they do not have these restrictions.) These restrictions let us operate on physical addresses, avoiding the need for TLBs on caches, and simplify addressing.

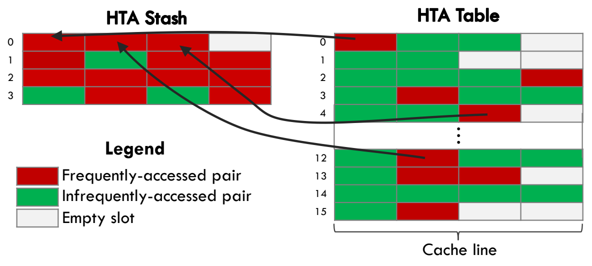


Figure 10: HTA stash format.

HTA stash format: Fig. 10 shows an example layout of an HTA stash and its corresponding HTA table. For simplicity, each HTA stash uses a contiguous range of 2^K cache lines. 2^K can be greater than the number of lines in the cache that the stash lives in. Suppose the HTA table is 2^M lines large. Then, given the HTA table address of a particular key, its HTA stash address is computed by zeroing the highest $M-K$ bits of its offset within the HTA table. Key-value pairs will map to line X in the HTA stash if they map to lines $X, X+2^K, X+2\cdot2^K, \ldots, X+(2^M-2^{M-K})$ in the HTA table.

Cache controllers store some information about each of their HTA stashes: the starting address and size of their corresponding HTA table, and the key-value pair format. This limits the number of Hierarchical-HTA hash tables that each cache may hold (to four hash tables in our implementation).

Per-pair management: Cache controllers are extended to manipulate and communicate individual key-value pairs within each level: they perform shared fetches (GETS), exclusive fetches (GETX), and dirty writebacks (PUTX) on key-value pairs, analogous to the usual requests for line fetches and evictions in conventional caches. Each HTA operation checks the hash table's HTA stashes in sequence, and the next-level HTA stash (and eventually the HTA table) is accessed only when the current stash cannot resolve the operation.

Fig. 11 illustrates Hierarchical-HTA's operation on a system with a two-level cache hierarchy and an L1-pinned HTA stash. Suppose the L1 starts empty. An `hta_lookup` triggers a `GETS` request with the key and HTA table line from the L1, as shown in Fig. 11a. The L2 accesses the right HTA table line (fetching it from memory if needed), and responds with its associated value. The L1 allocates space for the HTA stash line and installs the key-value pair there.

The HTA table is inclusive of HTA stashes. Updates are similar to lookups but issue GETX (exclusive) requests. On an update, if the HTA table does not have a pair with the same key, the pair is first inserted into the HTA table.

Caches can evict HTA stash lines as shown in Fig. 11b. Individual key-value pairs are written back if the line is marked as modified, and are simply dropped if the line is clean.

Overflows: Overflows in an HTA stash are transparent to software: the cache evicts a randomly-chosen pair to the next level to make space for a new one. Overflows in the HTA table are treated the same way as in Flat-HTA, by invoking the software fallback path for updates. Note that, since the HTA table is inclusive of HTA stashes, overflows or evictions in HTA stashes never cause HTA table overflows.

Coherence: Finally, we maintain coherence conservatively. Coherence is tracked at the shared last-level cache, for each line in the HTA table. When an LLC line in the HTA table is evicted, or

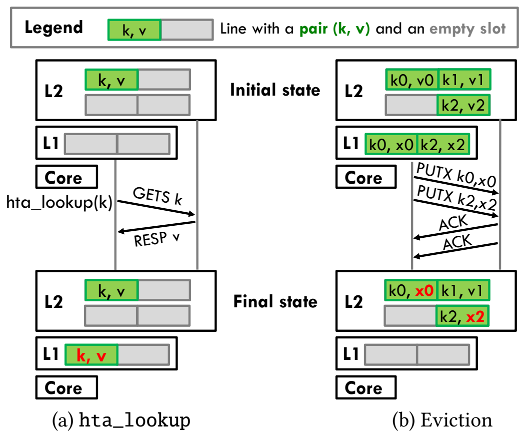


Figure 11: Pair-grain memory ops in Hierarchical-HTA.

when the line is shared and an exclusive request is received, all the sharers of the line are sent invalidations. At smaller caches that contain HTA stashes, an exclusive request (due to an update) that falls on an HTA stash line with shared permission (due to lookups) causes the pairs in the line to be dropped. These policies let us reuse line-level coherence metadata, though they are less precise than if we performed pair-by-pair coherence.

6 HTA-ACCELERATED MEMOIZATION

Memoization improves performance and saves energy by caching and reusing the outputs of repetitive computations. As discussed in Sec. 2.3, prior software and hardware memoization techniques have significant drawbacks. Software memoization suffers from high runtime overheads, and is thus limited to long computations. Prior hardware techniques achieve low overheads and can memoize short functions, but they rely on large, special-purpose memoization caches that waste significant area and energy.

We leverage HTA to accelerate memoization cheaply, matching the performance of prior hardware memoization techniques without dedicated on-chip storage.

Memoization tables are allocated for memoizable functions. Each memoization table is a hash table that stores arguments as the key and return values as the value. Since memoization tables are small, we use Flat-HTA to implement them. These Flat-HTA tables *do not have conventional software paths* that manipulate software hash tables. Instead, on an HTA table miss the software path simply calls the memoizable function. This is a good tradeoff: executing the short function is cheaper than a software hash table lookup.

Memoization operations are implemented using HTA instructions. Fig. 12 shows example code that leverages HTA instructions to memoize a single-argument, single-result function (`exp`). We

```
memo_exp: hta_lookup  0,  1,  0,  3, done
          call exp
          hta_swap    0,  1,  0,  3, done
done:     …
```

Figure 12: Example showing how HTA instructions are used to memoize the `exp()` function.

place an `hta_lookup` before the call to the memoized function. If the key is found, then the corresponding value is returned in the return value register and the function call is skipped. Otherwise, the function is executed and its result is memoized using `hta_swap`.

Since memoization tables do not grow to accommodate extra items, insertions simply replace one of the line's entries. As a result, there is no software path for `hta_swap` (i.e., target equals next PC), because the victim pair is simply dropped. This does not affect the correctness of the program. This is the right tradeoff when memoizing short functions; longer functions could use a full-blown HTA-accelerated hash table.

Exploiting memoizable regions: We have developed a pintool [27] to identify memoizable (i.e., pure) functions [48]. A function is defined as memoizable if it satisfies two conditions. First, its memory reads are either to read-only data or to its stack. Second, its memory writes are restricted to its own stack. Then, we manually added `hta_lookup` and `hta_swap` instructions to these functions' callsites. Due to its low overheads, HTA does not need to perform selective memoization based on cost-benefit analysis as in software techniques. Therefore, we memoize every function that our tool identifies as memoizable. We memoize both application and standard-library functions.

7 METHODOLOGY

We perform microarchitectural, execution-driven simulation using zsim [35]. We evaluate 1-core and 16-core chips with parameters shown in Table 2. These systems use out-of-order cores modeled after Haswell, and a 3-level cache hierarchy with a shared, inclusive LLC that has 2 MB per core.

Our HTA implementation includes registers for four HTA table descriptors. HTA instructions incur the cost of a cache-line load/store (in two 256-bit accesses), plus one cycle to perform key comparisons. We model an L1 that supports wide accesses (256 bits per cycle), which is common due to SIMD instructions (e.g., 256-bit AVX). We encode `hta_lookup`, `hta_update`, `hta_swap`, and `hta_delete` using x86-64 no-ops that are never emitted by the compiler.

We evaluate HTA using two sets of workloads: one set uses hash tables as a key part of their implementation, and the other set leverages memoization to improve performance. To achieve statistically significant results, we introduce small amounts of nondeterminism [1], and perform enough runs to achieve 95% confidence intervals ≤ 1% on all results.

Core	x86-64 ISA, 3.0 GHz, Haswell-like OOO: 16B-wide ifetch, 2-level bpred with 2 K×18-bit BHSRs + 4 K×2-bit PHT, 4+1+1+1 decoders, 6 execution ports, 4-wide commit
L1 cache	32 KB, 4-way set-associative, 3-cycle latency, split D/I
L2 cache	256 KB, 8-way set-associative, 7-cycle latency, inclusive
L3 cache	2 MB per core, 16-way set-associative, 15-cycle latency, inclusive
Main mem	DDR3-1600, 4 channels (16 cores) or 1 channel (1 core)

Table 2: Configuration of the simulated system.

	Input set	**Baseline hash table**	**# of `hta_` `lookups`**	**# of `hta_` `swaps` & `updates`**
bfcounter	ENCSR687ZCM.fastq [41]	C++11	0	26960049
lzw	the Bible	`unordered_map`	4364173	765632
hashjoin	−r-size=16777216 −s-size =268435456 −skew=1.5		268435456	23335399
ycsb-read	-z0.6 -r1.0 -w0.0	Google `dense_hash_map`	95998531	0
ycsb-write	-z0.6 -r0.0 -w1.0		0	95998531

Table 3: Hash table benchmark characteristics.

7.1 Hash table workloads

We analyze four applications that use hash tables heavily:

- `bfcounter` [31] is a memory-efficient software to count k-mers in DNA sequence data, which is essential for many methods in bioinformatics, including genome and transcriptome assembly. `bfcounter` uses a heavily-updated hash table to hold k-mers. We use a DNA sequence from ENCODE [41] as the input.
- `lzw` is a text compression benchmark based on the LZW algorithm [44], a widely-used lossless data compression technique. A hash table is used to hold the dictionary. We use the Bible as the input text file.
- `hashjoin` [2] is a single-threaded implementation of the hash join algorithm. `hashjoin` joins two synthetic tables. There are two phases in the program: in the first phase, the inner table is scanned to build a hash table; and then in the second phase, the outer table is scanned while the hash table is probed to produce output tuples.
- `ycsb` [10] is an implementation of *Yahoo! Cloud Serving Benchmark* that runs on DBx1000 [47]. Hash tables are used for hash indexes. We evaluate `ycsb` with two configurations: 100% read queries and 100% write queries.

Table 3 details these applications and their characteristics.

We modify each application to support multiple hash table implementations (using template metaprogramming to do so without runtime overheads). We compare the following implementations:

- **Baseline:** Because no single hash table design works best for all applications, we use the best of `libstdc++`'s C++11 `unordered_map` and Google's `dense_hash_map` as the baseline implementation. The best of both either matches or outperforms the application's existing hash tables.
- **Flat-HTA and Hierarchical-HTA:** To evaluate HTA, we use a hash table implementation with HTA hash tables accessed through `hta_lookup/update/swap/delete` instructions. The HTA hash table starts empty and is resized as elements are inserted. Specifically, if the fraction of software path invocations over total HTA accesses is above 1%, the size of HTA table is doubled. This involves allocating a new HTA table that is twice as large, then inserting all the pairs in both the previous HTA table and the software hash table into the new HTA table. For each application, HTA uses the same software hash table as the baseline. Since HTA rarely uses the software hash table, its performance is insensitive to the choice of software hash table.
- **HTA-SW:** To further analyze HTA and illustrate where performance differences comes from, we implement a software scheme, HTA-SW, that implements the same algorithm as HTA but without any hardware support. HTA-SW uses the same table format, the same software hash tables, and the same resizing algorithm. HTA-SW does not rely on any hardware support: all the steps in hash table operations, including

	Lang.	Benchmark suite	Input Set	Memoizable functions	Memoization table per function
bwaves	Fortran	SPECCPU2006	ref	slowpow, pow halfulp, exp1	32 KB
bscholes	C++	PARSEC	native	CDNF, exp, logf	32 KB
equake	C	SPECOMP2001	ref	phi0, phi1, phi2	4 KB
water	C	SPLASH2	1061208	exp	32 KB
semphy	C++	BioParallel	220	suffStatGlobalHom-Pos::get	4 KB
nab	C	SPECOMP2012	ref	exp, slowexp_avx	2 KB

Table 4: Memoization benchmark characteristics.

hashing, key comparison, memory accesses, and branches, are implemented purely in software. HTA-SW stores the keys within a cache line in a contiguous format, so that comparisons can be implemented with SIMD load and comparison instructions [49]. Specifically, we use Intel AVX vector load, compare, and mask instructions to exploit the parallelism in key lookups.

We fast-forward each application to skip initialization (e.g., data loading in `ycsb`) and simulate them to completion.

7.2 Memoization workloads

We analyze programs from six benchmark suites and choose one application with high memoization potential from each suite. Table 4 details these applications and their characteristics. For each application, we use the same memoization table size for all memoized functions. We report the table size that yields the best performance. Sec. 8.5 provides more insight on the effect of table size.

We fast-forward each application for 50 billion instructions. We instrument each program with *heartbeats* that report application-level progress (e.g., when each timestep or transaction finishes), and run it for as many heartbeats as the baseline system (without memoization) completes in 5 billion instructions. This lets us compare the same amount of work across schemes, since memoization changes the instructions executed.

8 EVALUATION

8.1 Flat-HTA on single-threaded applications

Fig. 13 compares the performance of the baseline, HTA-SW, and FLAT-HTA. FLAT-HTA outperforms the baseline by 24% on `bfcounter`, 70% on `lzw`, 2.0× on `hashjoin`, 23% on `ycsb-read`, and 69% on `ycsb-write`. Fig. 14 shows the breakdown of core cycles following the same format as Sec. 2.1, and shows the same trends. `bfcounter` and `lzw` benefit mainly from reduced mispredicted branches, while `hashjoin` and `ycsb` gain mostly from better backend parallelism.

Fig. 13 also shows that FLAT-HTA outperforms HTA-SW substantially, by 6.3% on `bfcounter`, 7.4% on `lzw`, 2.1× on `hashjoin`, 28% on `ycsb-read`, and 73% on `ycsb-write`. Fig. 14 shows these benefits stem from reduced wrong-path execution and backend stalls. Specifically, though FLAT-HTA incurs the same cache misses as HTA-SW, applications with abundant operation-level parallelism, like `hashjoin` and `ycsb`, benefit from HTA significantly by using the reorder buffer better: since each hash table operation uses far fewer μops, more operations are overlapped, reducing backend stalls. `ycsb-write` benefits more than `ycsb-read` because FLAT-HTA improves updates more than lookups (as Sec. 2.1 showed).

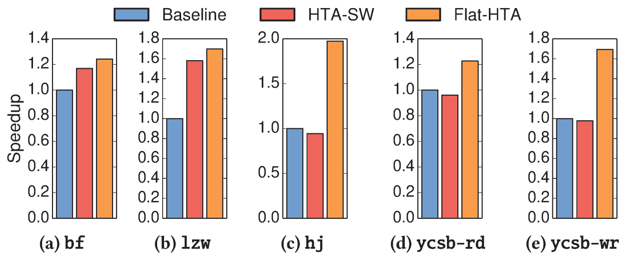


Figure 13: Speedups of FLAT-HTA and HTA-SW over the software baseline on single-threaded applications.

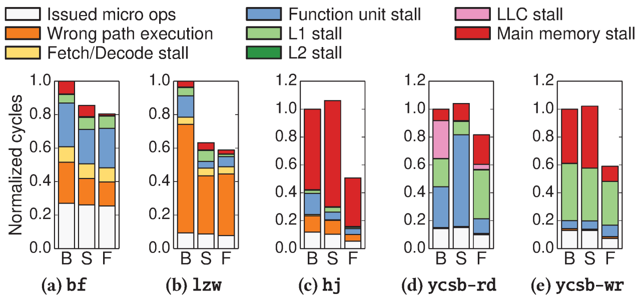


Figure 14: Cycle breakdowns for the Baseline, HTA-SW, and Flat-HTA.

Moreover, HTA-SW does not consistently improve performance. HTA-SW outperforms the baseline substantially on `bfcounter` and `lzw`, by 17% and 58% respectively, showing that the HTA design can sometimes outperform state-of-the-art hash tables even when implemented entirely in software. However, HTA-SW causes small performance degradations (up to 6%) on `hashjoin`, `ycsb-read` and `ycsb-write`. On these applications, the performance improvement of FLAT-HTA comes from hardware acceleration.

Beyond performance, space efficiency is an important consideration for hash tables. Table 5 reports the memory consumption of the different implementations. HTA-SW uses the same memory layout as FLAT-HTA, and hence has exactly the same memory consumption. Overall, results show that HTA does not cause undue storage overheads—there are differences of 2× in all cases, but note that hash tables grow exponentially over time and small differences in resizing thresholds can cause 2× size differences. On `bfcounter` and `lzw`, which use `unordered_map` as the baseline, the FLAT-HTA table is 2× larger than the baseline's. On `hashjoin` and `ycsb`, which use `dense_hash_map` as the baseline, the FLAT-HTA table is 2× smaller than the baseline's.

	Baseline		FLAT-HTA and HTA-SW	
	Type	Table	HTA table	Software path
bfcounter	`unordered`	275 MB	512 MB	2 MB
lzw	`_map`	8 MB	16 MB	117 KB
hashjoin	`dense`	512 MB	256 MB	512 B
ycsb-read	`_hash`	512 MB	256 MB	0 B
ycsb-write	`_map`	512 MB	256 MB	0 B

Table 5: Memory usage of the baseline and FLAT-HTA.

8.2 Flat-HTA on multithreaded applications

We now evaluate Flat-HTA on multithreaded applications. Since `bfcounter`, `lzw`, and `hashjoin` are single-threaded, we use the multithreaded implementations of `ycsb-read` and `ycsb-write`.

As shown by Fig. 15, at 16 cores, Flat-HTA outperforms the baseline by 33% on `ycsb-read` and by 3.5× on `ycsb-write`. These speedups are higher than in the serial version (23% and 69%) because most HTA operations are performed without acquiring locks.

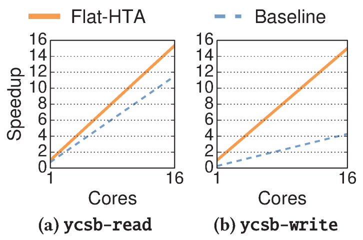


Figure 15: Speedups of Flat-HTA and the baseline on ycsb when scaling from 1 to 16 cores. Speedups are relative to the single-threaded baseline.

8.3 HTA with hierarchy-aware layout

We now compare the performance of Flat-HTA and Hierarchical-HTA. In this experiment, Hierarchical-HTA uses a 32 KB HTA stash pinned to the L1 cache, followed by a 256 KB HTA stash pinned to the L2 cache. The HTA table is cached in the LLC.

Fig. 16 compares the performance of both schemes. On `ycsb-read` and `ycsb-write`, where key-value pairs have mixed reuse, Hierarchical-HTA reduces L2 misses by 4.1× and 3.9×, respectively. This happens because Hierarchical-HTA lets the L1 and L2 hold densely-packed pairs. This miss reduction translates to a 35% performance improvement for `ycsb-read`. However, `ycsb-write` attains the same performance because Flat-HTA completely hides the latency of updates by exploiting memory-level parallelism (as we saw in Sec. 2.1). Finally, the other three applications do not exhibit mixed reuse, so Hierarchical-HTA does not significantly improve performance over Flat-HTA.

8.4 HTA on multiprogrammed workloads

We evaluate Flat-HTA's impact on co-running applications by running and 8-thread `ycsb` and an 8-thread SPEC OMP2012 application simultaneously on the 16-core system. Threads of both applications are pinned to cores.

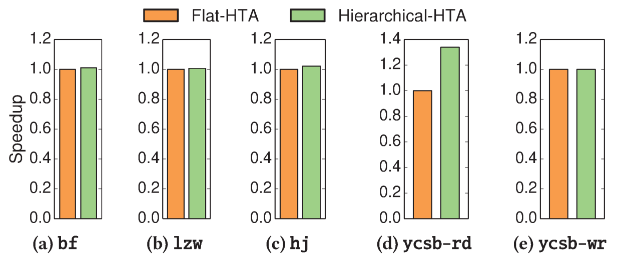


Figure 16: Speedups of Hierarchical-HTA over Flat-HTA.

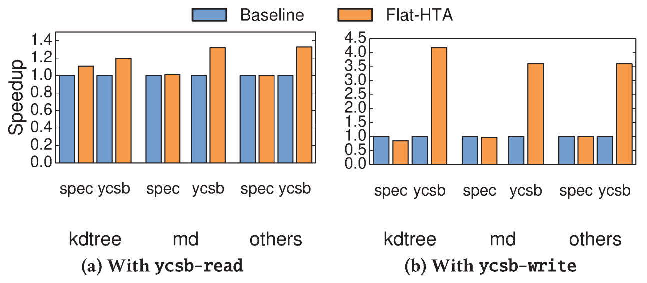


Figure 17: Speedups of HTA and the baseline when running a SPEC OMP2012 app and ycsb simultaneously.

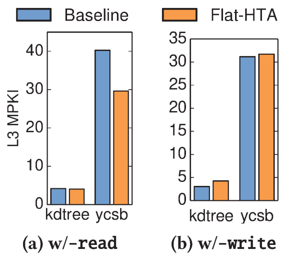


Figure 18: L3 MPKI when running kdtree and ycsb.

Fig. 17 summarizes the performance impact of Flat-HTA. The performance of all SPEC OMP2012 applications except `kdtree` and `md` is not affected by replacing the baseline hash table with Flat-HTA (as shown in the `others` bar groups). `kdtree`, the most cache-sensitive application, shows that HTA causes *less interference* than the default hash table.

First, when co-running with `ycsb-read`, using Flat-HTA causes `kdtree`'s performance to improve by 11%, even though Flat-HTA accelerates `ycsb-read` by 20%, which is thus performing hash table operations faster. Fig. 18a gives more insight into this result by reporting the changes in L3 misses per kiloinstruction (MPKI) for both `ycsb` and `kdtree` without and with Flat-HTA. Despite the higher rate of operations in `ycsb-read`, its MPKI is lower, which leaves more L3 capacity and memory bandwidth for `kdtree`.

Second, when co-running with `ycsb-write`, using Flat-HTA causes `kdtree`'s performance to drop by 15% due to increased cache capacity contention. However, note that `ycsb-write` is 4.2× faster with Flat-HTA, so it performs hash table operations much faster than the baseline. Fig. 18b shows that both `ycsb-write` versions incur a similar L3 MPKI. Overall, these results show that HTA does not introduce undue L3 and main memory memory pressure.

8.5 HTA on memoization

We leverage HTA for memoization, and compare its performance with the baseline implementation and both conventional hardware and software memoization techniques.

HTA vs. baseline: Fig. 19 compares the performance of HTA-based memoization over the baseline benchmarks, which do not perform memoization. HTA improves performance substantially, by 16× on `bwaves`, 7.5× on `bscholes`, 56% on `equake`, 27% on `water`, 17% on `semphy`, and 4% on `nab`.

Table 6 provides more details into these results by reporting per-function statistics. For example, in `bwaves`, memoizing the `pow` function provides most of the benefits. `pow` takes thousands of instructions to calculate x^y if x is close to 1 and y is around 0.75, which is common in `bwaves`. Memoizing `pow` contributes to 99.9% of the instruction reduction in `bwaves`.

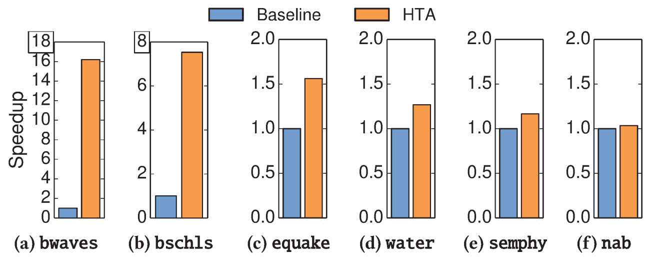


Figure 19: Speedups of HTA-based memoization over the baseline benchmarks, which do not use memoization.

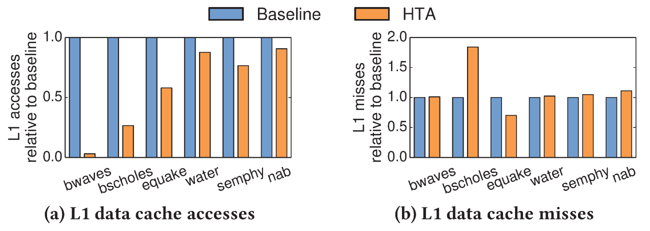


Figure 20: L1 data cache accesses and misses of HTA-based memoization relative to those of the baseline (without memoization).

Beyond reducing execution time, HTA reduces the number of L1 cache accesses significantly, as shown in Fig. 20a: L1 access reductions range from 9% on `nab` to 97% on `bwaves`. This happens because the L1 accesses saved through memoization hits exceed the additional L1 accesses incurred by memoization operations. Moreover, HTA does not incur much extra capacity contention in L1 caches. Fig. 20b shows that HTA increases L1 data cache misses by less than 5% overall. One exception is `bscholes`, which incurs 84% more L1 misses on HTA. However, this is not significant, because the baseline's L1 miss rate is only 0.2%. In fact, such misses bring in valuable memoization data that in the end improve performance by 7.5×. On `equake`, HTA even reduces L1 data misses by 30%, as the functions it memoizes have a larger data footprint than their memoization tables.

	Function	Instrs per func call	# of `hta_lookups`	Hit rate
bwaves	slowpow	485160	579	48.0%
	pow	12947	371813	96.3%
	halfulp	77	301	1.3%
	exp1	28	14443	21.7%
bscholes	CDNF	193	15547145	99.6%
	exp	115	7840164	100.0%
	logf	56	7773573	100.0%
equake	phi0	119	7953687	100.0%
	phi1	123	7953687	100.0%
	phi2	118	7953687	100.0%
water	exp	116	7806240	100.0%
semphy	get	19	67123200	94.6%
nab	exp	81	29150496	49.6%
	slowexp_avx	14756	0	N/A

Table 6: Per-function breakdown of `hta_lookups`.

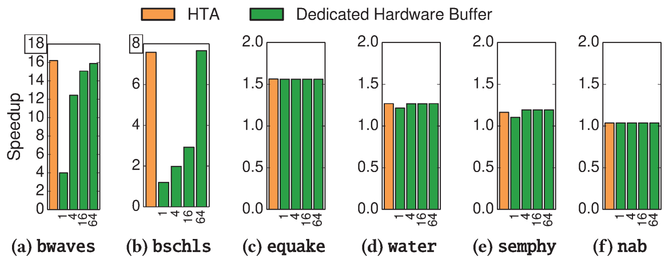


Figure 21: Per-application speedups of HTA and conventional hardware memoization with different dedicated buffer sizes (in KB).

HTA vs. conventional hardware memoization: We implement a conventional hardware memoization technique that leverages HTA's ISA and pipeline changes, but uses a dedicated storage buffer like prior work [6, 42] instead of using the memory system to store memoization tables. Beyond its large hardware cost, the key problem of conventional hardware memoization is its lack of flexibility: a too-large memoization buffer wastes area and energy, while a too-small memoization buffer sacrifices memoization potential.

Fig. 21 quantifies this problem by showing the performance of hardware memoization across a range of memoization buffer sizes: 1, 4, 16, and 64 KB. The buffer is associative, and entries are dynamically shared among all memoized functions. We optimistically model a 1-cycle buffer access latency.

Fig. 21 shows that applications are quite sensitive to memoization buffer size: 1 KB is sufficient for `equake` and `nab`, while `water` and `semphy` prefer at least 4 KB, and `bwaves` and `bscholes` prefer at least 64 KB. Smaller buffers than needed by the application result in increased memoization misses and sacrifice much of the speedup of memoization.

Finally, Fig. 21 shows that HTA matches hardware memoization with a dedicated storage size of 64 KB on all applications. This is achieved even though HTA does not require any dedicated storage, saving significant area and energy. The tradeoff is that storing memoization tables in memory causes longer lookup latencies than using a dedicated buffer. However, these lookup latencies are small, as they mostly hit on the L1 or L2, and branch prediction effectively hides this latency most of the time.

HTA vs. software memoization: We implement software memoization using function wrappers similar to Suresh et al. [39]. Per-function memoization tables are implemented as fixed-size, direct-

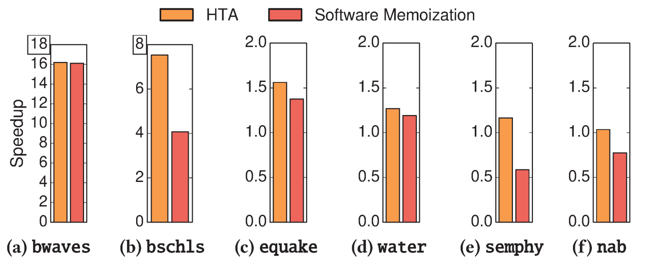


Figure 22: Per-application speedups of HTA and software memoization with per-function direct-mapped hash tables.

mapped hash tables, accessed before calling the function and updated after a memoization miss.

Fig. 22 compares the performance of HTA and software memoization. HTA outperforms software memoization by 85% on `bscholes`, 14% on `equake`, 7% on `water`, 2× on `semphy`, and 34% on `nab`. HTA outperforms software memoization due to its low overheads. For example, `semphy`'s memoizable function runs for 19 instructions on average, too short for software memoization. As a result, software memoization is 41% slower than the baseline. This explains why software memoization needs a careful cost-benefit analysis to avoid performance degradation. By contrast, HTA improves performance by 17% on `semphy`, outperforming software memoization by 2×. Similarly, software memoization makes `nab` 23% slower, while HTA improves performance by 4%.

9 CONCLUSION

We have introduced HTA, a technique that leverages caches to accelerate hash tables. HTA introduces simple ISA extensions and hardware changes to address the high runtime overheads and the poor spatial locality of conventional hash table implementations. HTA adopts a hash table format that exploits the characteristics of caches. HTA uses new instructions that leverage existing core structures to accelerate hash table lookups and updates.

We have presented two implementations of HTA: Flat-HTA and Hierarchical-HTA. Flat-HTA adopts a simple, hierarchy-oblivious memory layout and reduces runtime overheads through simple changes to cores. Hierarchical-HTA uses a multi-level hierarchy-aware layout and requires modifications in caches to further improve spatial locality.

As a result, Flat-HTA outperforms state-of-the-art implementations of hash-table-intensive applications by up to 2×, while Hierarchical-HTA outperforms Flat-HTA by up to 35%. Finally, we have shown that HTA can be leveraged to accelerate memoization. HTA bridges the gap between hardware and software memoization: Flat-HTA outperforms software memoization by up to 2×, and matches the performance of conventional hardware techniques, but avoids the overheads of large dedicated buffers.

ACKNOWLEDGMENTS

We sincerely thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Anurag Mukkara, Quan Nguyen, Hyun Ryong Lee, Suvinay Subramanian, Po-An Tsai, Victor Ying, Xiangyao Yu, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CAREER-1452994 and by DARPA SDH under contract HR0011-18-3-0007. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Alaa R Alameldeen and David A Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (2006).

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE-29)*.

[3] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing.. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.

[4] J Lawrence Carter and Mark N Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979).

[5] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA memcached appliance. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-21)*.

[6] Peng Chen, Krishna Kavi, and Robert Akl. 2006. Performance enhancement by eliminating redundant function execution. In *ANSS-39*.

[7] Byung-Soo Choi and Jun-Dong Cho. 2008. Partial resolution for redundant operation table. *Microprocessors and Microsystems* 32, 2 (2008).

[8] Daniel Citron and Dror G. Feitelson. 2000. *Hardware Memoization of Mathematical and Trigonometric Functions.* Technical Report. The Hebrew University of Jerusalem.

[9] Daniel Connors and Wen-Mei Hwu. 1999. Compiler-directed dynamic computation reuse: rationale and initial results. In *Proceedings of the 32nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-32)*.

[10] 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-1)*.

[11] Amarildo T Da Costa, Felipe M. G. França, and E. M. C. Filho. 2000. The dynamic trace memoization reuse technique. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT-9)*.

[12] Luca Della Toffola, Michael Pradel, and Thomas R Gross. 2015. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[13] Yonghua Ding and Zhiyuan Li. 2004. A compiler scheme for reusing intermediate computation results. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*.

[15] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*.

[16] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[17] Dibakar Gope, David J Schlais, and Mikko H Lipasti. 2017. Architectural Support for Server-Side PHP Processing. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*.

[18] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. 2012. Vector extensions for decision support dbms acceleration. In *Proceedings of the 45th annual IEEE/ACM international symposium on Microarchitecture (MICRO-45)*.

[19] Jian Huang and David J Lilja. 1999. Exploiting basic block value locality with block reuse. In *Proceedings of the 5th IEEE international symposium on High Performance Computer Architecture (HPCA-5)*.

[20] Raj Jain. 1992. A comparison of hashing schemes for address lookup in computer networks. *IEEE Transactions on Communications* 40, 10 (1992).

[21] Jesper Knudsen. 2008. Nangate 45nm open cell library. *CDNLive, EMEA* (2008).

[22] 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)*.

[23] Maysam Lavasani, Hari Angepat, and Derek Chiou. 2014. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters* 13, 2 (2014).

[24] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. 2015. 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 (ISCA-42)*.

[25] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. 2013. Thin servers with smart pipes: designing SoC accelerators for memcached. In *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*.

[26] Scott Lloyd and Maya Gokhale. 2017. *Near Memory Key/Value Lookup Acceleration*. Technical Report. Lawrence Livermore National Lab (LLNL).

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[28] Ward Douglas Maurer and Theodore Gyle Lewis. 1975. Hash table methods. *ACM Computing Surveys (CSUR)* 7, 1 (1975).

[29] James Mayfield, Tim Finin, and Marty Hall. 1995. Using automatic memoization as a software engineering tool in real-world AI systems. In *Proceedings the 11th Conference on Artificial Intelligence for Applications (CAIA'95)*.

[30] Paul McNamee and Marty Hall. 1998. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Notices* 33, 8 (1998).

[31] Pall Melsted and Jonathan K Pritchard. 2011. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC bioinformatics* 12, 1 (2011).

[32] Donald Michie. 1968. Memo functions and machine learning. *Nature* 218, 5136 (1968).

[33] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004).

[34] Hugo Rito and João Cachopo. 2010. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th international conference on the Principles and Practice of Programming in Java (PPPJ-8)*.

[35] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*.

[36] Avinash Sodani and Gurindar S Sohi. 1997. Dynamic instruction reuse. In *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA-24)*.

[37] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.

[38] Arjun Suresh, Erven Rohou, and André Seznec. 2017. Compile-Time Function Memoization. In *Proceedings of the 26th international conference on Compiler Construction (CC-26)*.

[39] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. 2015. Intercepting functions for memoization: a case study using transcendental functions. *ACM TACO* 12, 2 (2015).

[40] Shingo Tanaka and Christos Kozyrakis. 2014. High performance hardware-accelerated flash key-value store. In *The 2014 Non-volatile Memories Workshop (NVMW)*.

[41] The ENCODE Project Consortium. 2012. An integrated encyclopedia of DNA elements in the human genome. *Nature* 489, 7414 (2012).

[42] 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 Parallel and Distributed Computing and Networks (PDCN)*.

[43] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. 2008. SoftSig: software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*.

[44] Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 6, 17 (1984).

[45] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[46] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. BlueCache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment* 10, 4 (2016).

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment* 8, 3 (2014).

[48] Guowei Zhang and Daniel Sanchez. 2018. Leveraging Hardware Caches for Memoization. *Computer Architecture Letters (CAL)* 17, 1 (2018).

[49] Jingren Zhou and Kenneth A Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on management of data (SIGMOD)*.