DRAMHiT: A Hash Table Architected for the Speed of DRAM

Vikram Narayanan
University of Utah

David Detweiler
University of California, Irvine

Tianjiao Huang
University of California, Irvine

Anton Burtsev
University of Utah

Abstract

Despite decades of innovation, existing hash tables fail to achieve peak performance on modern hardware. Built around a relatively simple computation, i.e., a hash function, which in most cases takes only a handful of CPU cycles, hash tables should only be limited by the throughput of the memory subsystem. Unfortunately, due to the inherently random memory access pattern and the contention across multiple threads, existing hash tables spend most of their time waiting for the memory subsystem to serve cache misses and coherence requests.

DRAMHiT is a new hash table designed to work at the speed of DRAM. Architecting for performance, we embrace the fact that modern machines are distributed systems—while the latency of communication between the cores is much lower than in a traditional network, it is still dominant for the hash table workload. We design DRAMHiT to apply a range of optimizations typical for a distributed system: asynchronous interface, fully-prefetched access, batching with out-of-order completion, and partitioned design with a low-overhead, scalable delegation scheme. DRAMHiT never touches unprefetched memory and minimizes the penalty of coherence requests and atomic instructions. These optimizations allow DRAMHiT to operate close to the speed of DRAM. On uniform key distributions, DRAMHiT achieves 973Mops for reads and 792Mops for writes on 64-thread Intel servers and 1192Mops and 1052Mops on 128-thread AMD machines; hence, outperforming existing lock-free designs by nearly a factor of two.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; • **Theory of computation → Bloom filters and hashing**.

ACM Reference Format:
Vikram Narayanan, David Detweiler, Tianjiao Huang, and Anton Burtsev. 2023. **DRAMHiT**: A Hash Table Architected for the Speed of DRAM. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587457

1 Introduction

Today, hash tables are among the most critical building blocks for modern data-intensive applications. Examples include key-value stores [33], databases [2, 3, 7, 25, 56], genomic and meta-genomic analysis [36, 46], dynamic programming [59], model checking [60, 63], graph processing [35, 38], matrix multiplication [1], request load balancing [8], and many more. A common characteristic of such applications is that the hash table accesses dominate the execution time of the program.

Internally, the hash table performs a simple computation: taking a string of bits (a key) as an input, it computes a hash which is then used as an address to access the value associated with the key. The hash function is designed to uniformly distribute the storage locations for key and value tuples across the entire memory occupied by the hash table. The computation of a hash function takes only a few cycles. For example, a hardware-supported CRC32 hash function can be computed in 2-3 CPU cycles, while the more sophisticated CityHash [15] function takes 16-20 cycles on an 8-byte key. Despite minimal computation, the fastest hash tables spend 150-300 cycles per lookup and insertion operation [35].

With only a few cycles required to compute the hash function, modern hash tables spend most of the request processing time waiting on the memory subsystem while it serves memory misses and coherence requests. While the path of a cache miss through the memory subsystem is different for small and large datasets, both incur a significant performance penalty. For a large dataset (e.g., several times larger than the caching hierarchy of the CPU), the miss results in a transfer of a cache line from either local or remote memory, i.e., the memory attached to a memory controller of another socket. For a small dataset that fits in the caching hierarchy of the CPU, the miss results in a transfer of a cache line from the cache of another core or a last-level cache of the local or remote die on which it was most recently accessed. If the cache line is still on the same die, the penalty is relatively low (54-132 cycles depending on whether the cache line is evicted into the last level cache or is still present in private caches of other cores [6, 64]). Access across dies can reach 320-431 cycles due to the overhead of cross-socket links [6, 64].

Moreover, on datasets with high skew—e.g., 90% of requests accessing 10% of keys—coherence requests suffer from

EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3587457

contention, which requires expensive cache directory operations, and serializing access across multiple cores [4]. The latency of a contended coherence request, combined with the high probability of acquiring a cache line in an exclusive state, can reach thousands of cycles (Figure 2).

Today's hash tables are fundamentally limited in their peak performance due to the decades-old design choice of treating memory as a subsystem with a *synchronous* interface. Despite the fact that hash tables employ a range of optimizations to reduce the number of cache misses through the utilization of compact, cache-friendly layouts and algorithms [13, 17, 20, 24, 29, 32, 34, 40, 48, 50, 61, 68], at least one memory miss per operation remains unavoidable due to random accesses to the memory of hash table that is larger than caches of the CPU. While modern CPUs are capable of partially hiding the cost of the miss through speculative execution, the cost of the memory stall remains high. Fundamentally, the latency of the memory miss defines the upper limit on the performance of the hash table.

Our work explores the design space of hash tables aimed at achieving optimal performance on modern hardware. Architecting for performance, we embrace the fact that modern machines are distributed systems with local non-uniform memory and non-uniform caches—while the latencies of memory controllers and the coherence protocol are much lower than in the network, they are an order of magnitude larger than the rest of the hash table processing path. To hide the latencies of modern memory and coherence subsystems, we develop a range of optimizations typical for a distributed system: asynchronous interface, fully-prefetched access, batching with out-of-order completion, and partitioned design with low-overhead, scalable delegation scheme. While many of our ideas are not new, the main engineering challenge and contribution of our work is the ability to implement these optimizations with a budget in the low tens of cycles, in contrast to the thousands of cycles typical for distributed systems and prior approaches.

To avoid memory misses on the critical path, we change the interface of the hash table to support asynchronous submission of requests and out-of-order completion. This allows us to avoid wasting the CPU cycles on accesses to cache lines residing in memory or remote caches. In our design, the hash table never touches unprefetched memory. The application submits a batch of requests. The hash table computes memory addresses corresponding to the keys, and prefetches the memory locations involved in the operations, putting requests on the queue of the prefetch engine, but not touching those addresses. After enough elements are accumulated on the queue and enough time has passed for the prefetched cache-lines to reach the first-level caches of the CPU, the hash table processes the operations, potentially issuing more prefetches for keys that require additional memory accesses to resolve hash conflicts, i.e., reprobes. Pending reprobe requests are put back on the request queue.

Out-of-order completion allows us to eliminate degradation due to requests that trigger a large number of reprobes. The hash table interface takes a batch of requests and returns a batch of responses, potentially out-of-order. Requests that take very long to complete due to a large number of reprobes are returned later in subsequent invocations.

To avoid contention for workloads with a high skew, we extend our basic asynchronous design with support for delegation and partitioning. Partitions are visible to every reading thread. This allows us to process read operations locally by any of the threads accessing the hash table. Reads require no writes or atomic instructions, and hence, do not invalidate the cache-lines of other readers. To avoid expensive coherence conflicts on update operations, we delegate updates to the threads responsible for managing write access to each partition. We rely on explicit message passing to implement a scalable delegation scheme that relays update requests to a collection of update threads. Under contention, delegation allows us to outperform hardware coherence protocols.

Finally, we treat the throughput of the memory subsystem as an explicit resource. Hence we employ a range of optimizations to meet the cycle budget enforced by the memory subsystem.

Novel design decisions allow us to construct DRAMHiT, a hash table that approaches the speed of modern multichannel memory subsystems. On uniform key distributions, DRAMHiT achieves 973 Mops (reads) and 792 Mops (writes) on dual-socket 64-thread commodity Intel servers and 1192 Mops (reads) and 1052 Mops (writes) on a two-socket 128-thread AMD machine hence outperforming existing lock-free designs by nearly a factor of two. DRAMHiT explicitly trades increased latency of hash table operations for throughput. We believe that such a tradeoff is justified for a wide range of practical workloads. On a metagenomic benchmark, a partitioned version of DRAMHiT outperforms the fastest hash table equivalents by a factor of four.

2 Background

Modern hash tables have accumulated decades of algorithmic optimizations and engineering innovation geared at improving the performance of hashing functions, optimizing utilization of the caching hierarchy, minimizing overheads of synchronization, and much more. Recent advancements in CPU design—increasing core count, switching to non-uniform memory and cache architectures, and growing memory size and bandwidth—change the balance of engineering tradeoffs required to achieve peak hash table performance.

Memory bandwidth and cycle budgets Modern servers are non-uniform memory access (NUMA) machines. A typical server is deployed with several processor nodes (sockets) connected with a cross-socket link, ultra-path interconnect (UPI) on Intel machines (Figure 1). Each socket contains multiple CPU cores and several memory controllers connected to local DRAM. Each memory controller relies on a group

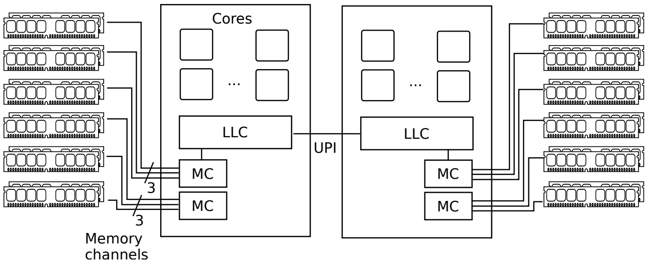


Figure 1. Memory subsystem of a two-socket, 32-core Intel server.

Table 1. Theoretical and measured bandwidth and cycle budget per one cache-line transaction if issued from 32 logical cores of one socket configured with six memory channels.

Configuration	Bandwidth (GB/s)	Cycle budget
Theoretical	127.8	41.6
Seq reads	111.0	48.3
Seq reads-writes (1:1)	95.4	55.8
Seq reads-writes (2:1)	97.5	54.5
Random reads	85.4	62
Random reads-writes (1:1)	76.3	69.7
Random reads-writes (2:1)	81.3	65.4

of memory channels to communicate with dual inline memory modules (DIMMs). Figure 1 shows the organization of a modern two-socket Intel server as an example. Each socket has two memory controllers. Each controller is connected to three memory channels (16 cores and six memory channels per socket). While we use an Intel machine as an example, conceptually, modern AMD CPUs have a similarly organized memory subsystem [64].

Intel CPUs can access the first-level cache at a rate of 2.9-116.25 bytes per cycle depending on the location in the last-level cache, i.e., cache bank, and the instruction used to perform the access [64]. Furthermore, they achieve a throughput of 2.2-52.9 bytes per cycle to L2 and 1-18 bytes per cycle to L3 level caches [64]. This organization provides sufficient bandwidth for accessing all levels of the caching hierarchy.

However, the bandwidth of the memory subsystem is limited. Two factors determine the performance of the memory subsystem: the number of memory channels, and the speed of the DRAM banks. Modern DDR4 DRAM memory performs 2666 Mega-Transactions per second (each transaction allows a read or write of an 8 byte value). At 2666 MT/s, one memory channel can deliver a theoretical bandwidth of 21.3 GB/s (a six-channel system of one socket can achieve a theoretical throughput of 127.8 GB/s).

CPUs, however, communicate with DRAM in units of cache-lines (64 bytes). Therefore, to access even one byte of data, the CPU will issue a cache-line read; hence consuming 64 bytes of memory bandwidth. A single memory channel is capable of performing 333.2 million cache-line reads or writes per second (a six-channel system achieves 1.9 billion cache-line transactions). Hence, to keep the memory subsystem saturated, a single-socket system should issue a cache-line transaction every 0.5 nanoseconds. If all 16 cores are used to perform memory operations, each core has to submit a cache-line transaction every 8 ns, or 20 cycles on a 2.6GHz machine (or 40 cycles if each core runs two hardware threads of execution).

In practice, modern servers achieve only a fraction of the theoretical memory bandwidth. We use the Intel Memory Latency Checker (MLC) [5] to measure the performance of the memory subsystem of an Intel Xeon Gold 6142 16-core Skylake CPU running at 2.6 GHz, with six populated memory channels per-socket and 384 GB RAM (Table 1). We run MLC on one socket of the two-socket server, using all available 32 logical threads, and restricting memory accesses to the local NUMA node of the socket. The bandwidth changes between sequential and random accesses and with the composition of reads and writes. The system achieves 111 GB/s or 87% of the theoretical bandwidth on a sequential read-only workload (for random reads that are typical for the hash table memory access, the bandwidth drops to 85.4 GB/s). Our measurements are in line with Velten et al. [64]. Since in a typical scenario insertion into a hash table performs one or more reads due to hash conflicts and one write, we also measure achievable throughput on the workload that generates one and two reads per one write. Empirical measurements allow us to make the following observation:

> In order to saturate the memory subsystem, our example Intel system has a budget of 62 CPU cycles for the read-only lookup operation and 140 cycles for the insertion operation that requires one read and one write (69.7 cycles each).

Note that on Intel SkyLake CPUs, reads from remote NUMA nodes result in a write [23]. The reason stems from the fact that the CPU implements a "directory" feature that tracks the state of each cache line on remote NUMA nodes. The state of the cache line is maintained along with the error correction bits in memory. When the cache line is read from a remote NUMA node, it is read in an "exclusive" state (an optimization that allows the reader to write the cache line without requesting further permissions). Unfortunately, this requires a write-back of a cache line to clear the "directory" bit when the cache line is evicted. Therefore, all reads that access remote NUMA nodes will result in an additional write transaction (roughly half in our configuration). In our experiments, at 75% fill, an average lookup accesses 1.3 cache lines (and hence triggers 0.75 write transactions due to 50% reads accessing remote NUMA node).

Overheads of synchronization and coherence Concurrent access to the memory of a hash table from multiple cores results in two distinct sources of overhead: transfer of cache lines between caches and contention (i.e., linearization of concurrent accesses to the same cache line). Intel and AMD

CPUs implement a version of the MESI cache-coherence protocol [42]. To access a cache line for read or write, the core performs a coherence request that brings it into the first-level cache of the core from either memory or different levels of the caching hierarchy (i.e., local and remote L1, L2, and L3 caches). If multiple cores are trying to request access to the same cache line concurrently, requests are linearized by the cache directory implemented by the caching agent of the last-level cache [4]. The latency of acquiring a cache line in an exclusive state grows linearly with the number of cores requesting the cache line [4].

On modern machines, the transfer of a cache-line between two cores takes 115-320 cycles, depending on how far away the cache line is in the caching hierarchy from the accessing core [6, 43, 44, 64]. Atomicity of the updates is ensured by temporarily "locking" the cache line to the core, i.e., if a coherence request from another core arrives to the core that locked the cache line, the request is delayed until the line is unlocked. If the cache line is already present in the first-level cache of the core, locking is fast as it is performed locally by the cache (locking the cache line adds an overhead of 11-30 cycles [6]). Hence, without contention, the main overhead comes from transferring cache lines between individual caches or between caches and memory. However, in the case of contended accesses, the overheads of linearization of requests from multiple cores dominate those of cache-line transfers.

To illustrate the impact of synchronization on the hash table's performance, we conduct a simple experiment that mimics cache-line access patterns typical for traditional lock-based and lock-free synchronization schemes. In our experiment, multiple threads access individual cache lines either inside a critical section protected by a spinlock, or with an atomic increment instruction (Figure 2). We utilize two datasets aimed to represent two extremes of the hash table size—small and large—and vary the skew parameter of the Zipfian distribution from 0 (uniform) to 1.2 (at the skew value of 1, roughly 90% of accesses touch 10% of cache lines). The small hash table fits into the caching hierarchy of our machine, i.e., we allocate 32 MB for the hash table on a machine that has 64 MB of last-level cache spread across two sockets. The second dataset is 1 GB and is intended to emulate a large hash table that fits in memory, but not in the caches of the CPU. The experiment utilizes 64 logical threads of our example dual-socket Intel Xeon Gold 6142 16-core Skylake CPU.

In case of a large hash table, the majority of accesses fetch the cache line from memory (the probability of contention—and hence, the probability that the cache line is already present in the last-level cache or in one of the private caches—is extremely low). In the case of the small hash table, the entire dataset is cached in the private L1 and L2 caches of all CPUs and the two last-level caches of the two sockets. A typical access fetches the cache-line from either the local or the remote cache. In both cases (large and small hash tables), the overheads of synchronization are dominated by the latency of transferring the cache line to the first-level cache of the core requesting the access (from 184 cycles from a remote cache to 394 cycles from memory). A critical section requires two atomic cache-line accesses: one to acquire and one to release the lock; however, since during the second access the cache-line is already in the modified state in the local L1, the second access introduces only minor overhead (the overhead of a spinlock and atomic increment differ by only 31-95 cycles). We make the following observation:

> On distributions with a small skew, the contention is low. The overhead is dominated by coherence transfers that fetch cache-lines from either memory or the caches of other cores.

On a skewed distribution, the probability of accessing the same cache line by multiple threads grows significantly. Concurrent accesses create contention, which forces coherence requests from multiple cores to be queued by the directory. On our 32-core (64 logical threads) dual-socket system, latency reaches 16K cycles for atomic increment and 66K cycles for a spinlock (this is consistent with papers that observe scalability bottlenecks for workloads that contend for the same cache line across multiple cores [4]).

> On high skew, contention dominates overheads of cache-line transfers.

Architecting for performance The above observations shape the following design principles that are critical for achieving peak hash table performance on modern hardware:

- **Minimal number of cache misses** For both small and large hash tables, the cost of a cache miss below the L2 cache (i.e., a miss to either memory, local last-level cache, or caches of remote cores) is prohibitive. The hash table, therefore, has to avoid accessing unprefetched memory on the critical path.
- **Minimal number of memory transactions** Hash tables that do not fit in the caching hierarchy should treat memory bandwidth as a limited resource. Additional memory accesses can sharply degrade performance (e.g., one additional access per hash table operation can effectively reduce the throughput of the hash table by half if the memory bandwidth is saturated). The hash table has to be designed to minimize the number of memory transactions through the choice of conflict resolution policy, hash table organization, and data structure layout.
- **No contention** On workloads with a high skew, the overhead of contention dominates all others. To achieve peak performance, hash tables should minimize or avoid contention.

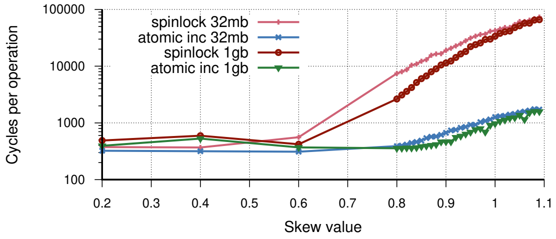


Figure 2. Synchronization overheads for 32 MB and 1 GB datasets

3 DRAMHiT Architecture

Our work, DRAMHiT, develops a new hash table that is aimed at exploiting the performance capabilities of modern memory subsystems. DRAMHiT utilizes the design principles that we articulated above. Specifically, it is designed to avoid cache misses on the critical path, treat memory bandwidth as a limited resource, and eliminate contention under high skew.

Hashtable organization DRAMHiT implements the hash table as a single contiguous array of $key, value$ tuples and relies on open addressing with linear probing for collision resolution. *Linear probing* is one of the widely used algorithms for collision resolution in a hash table. To store a $key, value$ tuple, the key is hashed using a hash function that returns an index, i.e., $index = hash(key)$. The index is used as a candidate location for storing the tuple in the array. If the candidate location is occupied, the linear probing algorithm increments the index until it finds the first vacant position in the linear sequence $index$, $index+1$, $index+2$, etc., wrapping around when the end of the array is reached.

Note that since the original hash of the key can produce an index that is larger than the array size, we use the *fastrange* function to ensure that the index is in the range $[0, size)$ in an approximately uniform manner [31]. Fastrange provides a fast alternative to the modulo reduction, allowing us to work with hash tables whose size is not limited to powers of two.

Linear probing provides several attractive properties. First, a combination of open addressing and linear probing allows us to minimize the number of cache-line transactions on the memory bus. Linear probing resolves hash conflicts by accessing consecutive memory locations (in many cases, the same cache line) in order to find an unoccupied hash table slot. Our empirical observations show that on a fill factor of 75-80%, lookup and insertion operations require only 1.3 cache line accesses per request on average (i.e., reprobes that check consecutive memory locations access additional cache-lines only 30% of the time). This is critical for reducing pressure on the memory subsystem.

Second, simple conflict resolution and insertion logic allows us to implement a synchronization scheme that requires no atomic operations for reads and one atomic operation for writes. As a result, DRAMHiT benefits from caching frequently accessed elements on read-heavy workloads with a high skew (i.e., concurrent read accesses from different cores do not invalidate local copies cached in the "shared" state).

Operations DRAMHiT supports the following operations: get(), put(), delete(), and upsert(). The get() operation takes a key as an input and returns either a found value matching the key or None. The put() operation takes the key and the value and inserts the value into the hash table along with the key. If the key already exists in the hash table, put() silently overwrites it with the new value. The upsert() operation either inserts a constant passed as an argument into the hash table or updates the existing value by adding the constant. Finally, the delete() operation takes a key as input and deletes the value associated with the key if it is found in the hash table. We implement deletion by marking the element as a *tombstone*. Note that the delete operation does not free the slot of the hash table array. The space is freed only when the hash table is resized (we assume that an efficient resizing scheme can be implemented similar to Growt [35]).

Atomicity To serialize concurrent accesses, DRAMHiT implements two different protocols depending on whether the key-value tuple fits in two machine words (i.e., 8 bytes each on x86 64bit machines) or not. To implement insertion for tuples that are smaller than 16 bytes, DRAMHiT relies on a double-word compare-and-swap (cas) instruction. Concurrent updates are atomic due to the atomicity of the cas instruction. DRAMHiT implements a lookup operation as two 8-byte loads without any atomic operations. It may seem that read can observe a key-value tuple from multiple concurrent updates (i.e., a *torn read*). We ensure the linearizability of reads with respect to concurrent updates by relying on the fact that the read operation first reads the key to check if the element of the hash table is empty or not and then the value. If the read operation is interrupted by a concurrent update, i.e., an update of an already existing value, the read observes the most recent value. Since delete() operation does not free the element of the hash table array but rather marks its key as a tombstone, a concurrent read operation either observes a tombstone (i.e., key not found) or returns the old value as if read happens before deletion.

DRAMHiT uses two values from the key space to mark empty and deleted keys (empty and tombstone). To restore the key space, we use two dedicated memory locations that store values corresponding to the empty and tombstone keys.

For the key-value tuples larger than 16 bytes, we implement a simple transactional protocol that ensures the atomicity of reads [33, 62]. We maintain a 32-bit version along with each tuple. Any write operation increments the version before and after the write to the tuple, i.e., when inserting a new key or just updating the value. Reads rely on the fact that an odd version marks an in-progress update and wait for the version to become even before proceeding with the read. Finally, to ensure that the read of multiple cache lines is atomic, i.e., the key or the value are not changed with a concurrent update, the read operation compares the version before and after reading the element. If the version changes it means that the element was changed by a concurrent update and the read operation is retried.

3.1 Prefetch Engine

DRAMHiT is designed to avoid unprefetched memory accesses. To implement this design decision, we develop a lightweight prefetch engine that issues a prefetch instruction before accessing any element of the hash table. To overlap pending prefetch accesses with the processing of requests, we treat memory as an asynchronous medium.

Asynchronous interface In contrast to traditional hash tables that processes one request at a time, DRAMHiT exposes an asynchronous interface typical for modern I/O devices, e.g., network and NVMe adapters. The hash table interfaces take a batch of requests and return a batch of responses. The caller provides pre-allocated space to collect responses. The hash table can return any number of responses between zero and the size of a pre-allocated response batch.

Responses can be returned out-of-order due to different-length reprobe chains. With every request, the hash table takes a user-defined identifier which it returns along with the response. This allows the application code to match responses with requests. For example, if the application code has an array of keys and tries to look up values corresponding to each key, it can submit the position of the key as an identifier and then later use the same identifier to insert the response into the appropriate location of the array. While out-of-order completion of requests puts an additional burden on software developers using our batch API interface, we argue that the tradeoff is justified by the performance benefits. The hash table provides a flush() function that forces the hash table to complete all pending requests. Typically, flush() is called only when the application reaches the end of its dataset.

Prefetch engine To implement asynchronous request processing, DRAMHiT maintains a thread-local first-in-first-out queue of pending requests. When the request is picked from a batch of submitted requests, the hash table computes the hash of the key, determines the location in the hash table it must access, issues a prefetch for that location, and adds the request onto the queue of pending requests (Algorithm 1). We implement the queue as a bounded-size producer-consumer ring buffer. This allows us to avoid allocations on the critical pass, as the element is inserted into the queue by advancing the head pointer.

DRAMHiT accumulates PREFETCH_WINDOW requests in its queue. This allows us to adjust for the latency of access to memory or a remote cache. When PREFETCH_WINDOW requests are accumulated on the queue, we have a guarantee that the tail of the queue is already resident in the caches of the CPU. DRAMHiT then starts processing elements from the queue by dequeuing the oldest element from the tail of the queue and performing the hash table operation on it, i.e., looking for an empty location in the hash table array. If the operation triggers a reprobe that spills into a different cache line, the request is updated with the location of the next cache line and is pushed back on the queue, while the prefetch is issued for this next cache line. The operation returns when either the queue size drops below PREFETCH_WINDOW (no requests are ready) or, in the case of the lookup operation, the response batch pre-allocated by the application is full.

```
for item in batch do
    queue[head].item = item;
    h = hash(item->key);
    queue[head].index = fastrange(h, htable.size);
    prefetch(&htable[index]);
end
if queue reaches PREFETCH_WINDOW then
    for elem in queue do
        ret = insert(elem);
        if ret == reprobe then
            queue[head] = elem.index++;
            prefetch(&htable[queue[head].index]);
        end
    end
end
```

Algorithm 1: Prefetch logic for the insertion operation

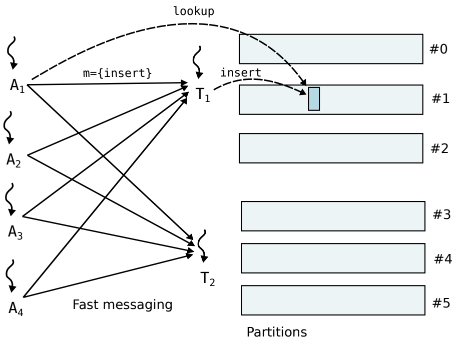


Figure 3. DRAMHiT architecture with four applications (A_i) and two delegation threads (T_i) each managing three partitions.

3.2 Partitioning for High Skew

Under high skew, the cost of linearizing conflicting cache-line accesses becomes dominant. To eliminate contention, we implement DRAMHiT-P, a partitioned version of DRAMHiT that utilizes an efficient delegation scheme to relay the processing of update requests to a dedicated set of threads. In contrast to traditional shared-memory designs in which all threads access the hash table and synchronize access via synchronization primitives, DRAMHiT-P implements the hash table as a set of partitions (Figure 3). Each partition is responsible for storing a non-overlapping part of the key space. A read operation can be executed by any of the threads, i.e., any thread can access any partition to perform a lookup. Write operations, however, are delegated to threads assigned to provide exclusive update access for each partition.

To implement delegation, we rely on explicit message passing (Section 3.3). Each delegation thread polls for client requests and executes them on the partition assigned to it.

Operations Efficient implementation of a partitioned hash table requires some changes to the hash table interface motivated by the goal of minimizing the number of messages between threads. Specifically, in DRAMHiT-P, update operations that are issued via message queues do not return a result. This allows us to avoid implementing bi-directional communication and waiting for the completion of update operations. If the key already exists in the hash table, put() silently overwrites it with the new value. The insertion may fail only if the destination partition is full. We detect that the partition is full before insertion is attempted. Since each partition is updated by a single thread, this thread maintains a counter that tracks the number of elements in the partition. Since all updates on a partition are performed by a single "writer" thread the counter can be maintained locally without incurring synchronization overheads. If the partition is full, the thread sets a flag denying further insertions. Before inserting a key, each "producer" thread checks the flag. In a stable state, while the partition is not full, the flag is cached in the *shared* state on all producer cores. Hence the check is an access to the local L1 cache.

3.3 Delegation

Several unique requirements shape the design of a delegation scheme in the context of a hash table. First, the delegation mechanism is required to scale to a large number of server threads, e.g., 32-64, to support a large number of "consumer" threads required on the write-heavy workloads (i.e., k-mer counting that we use as a case study in Section 4). Second, delegation should provide overhead of only a few tens of cycles in order to compete with the speed of the cache coherence protocol.

Most delegation schemes are designed for a small number of server threads—in a typical case, server threads execute the code of small critical sections, hence only small amount of CPU time is needed [55]. Moreover, typical delegation schemes are designed to compete with traditional synchronization primitives under high contention. For example, when applied to a hash table, a recent delegation scheme, FFWD, outperforms traditional spinlocks only on hash tables that have less than 64 buckets and under contention from 120 cores, i.e., only under extreme contention [55].

Message queue DRAMHiT-P chooses low-overhead message passing as the foundation for its delegation mechanism. Specifically, DRAMHiT-P utilizes the ideas of section queues[22, 47], but combines them with explicit queue flushing. Section queues improve the performance of the Lamport's producer-consumer queue [28] that implements lock-free communication but require checking of a shared producer-consumer pointer on every message, which results in expensive cross-core cache-line transactions. A section queue implements the queue as a ring-buffer of messages split into sections. A pair of pointers is shared between the consumer and the producer similar to the Lamport's queue, but the pointers are only updated when the producer or the consumer reaches the end of a section.

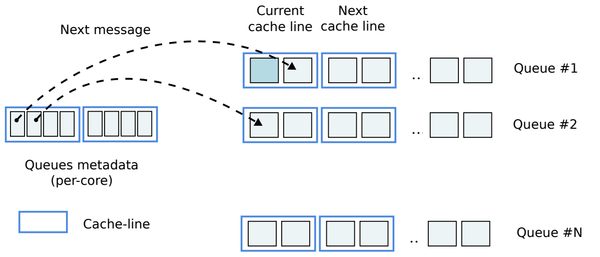


Figure 4. Producer-side queues and metadata

When the end of the section is reached, the producer or consumer checks the shared pointer to confirm if the next section is available (since the shared pointer is updated on the other side, this can result in a remote cache line access, so we prefetch the shared pointer when the queue approaches the end of a section). If the section is not available, the producer spins on the pointer.

Section queues allow us to avoid both the overheads of backtracking [67] and producer-consumer deadlocks. Compared to the queues that implement adaptive backtracking, e.g., BQueue [67], which uses a timeout before re-checking the availability of the queue element, the update to the pointer is detected immediately through a cache-coherence update, hence there is no need for an adaptive timeout. By choosing a large section, one can amortize the cost of accesses to shared pointers at the cost of increased communication latency (the messages are not reaching the other end until the entire section is full).

Scaling Scaling the number of communication queues is challenging in the face of a limited L1 cache budget. Modern Intel CPUs come with per-core 32 KB L1 caches (or 16 KB per logical thread). If a producer thread communicates with up to 64 consumers, then it leaves 256 bytes (or 4 cache lines) per queue. While in practice, a typical communication involves 32-63 queues, some fraction of the L1 cache is occupied by frequently accessed variables.

L1 cache residency is critical for the performance of the delegation mechanism, which aims for a communication overhead of a few dozen cycles (i.e., a miss from L1 into L2 adds a penalty of 10 cycles [64]). Efficient queue access requires that at least two cache lines (current and next) are resident in the L1 cache. With a budget of only 1-2 cache lines per queue, the queue has to be carefully designed to pack queue metadata into the minimal number of cache lines.

```
1  constexpr std::array<__mmask8, KV_PER_CACHE_LINE>
2  key_cmp_masks = {
3        KEY3 | KEY2 | KEY1 | KEY0, // cidx: 0; all key comparisons valid
4        KEY3 | KEY2 | KEY1, // cidx: 1; only last three comparisons valid
5        KEY3 | KEY2, // cidx: 2; only last two comparisons valid
6        KEY3, // cidx: 3; only last comparison valid
7  };
8  auto key_cmp = [&key_cmp_masks](__m512i cacheline,
9     __m512i key_mask, size_t cidx) {
10    __mmask8 cmp = _mm512_cmpeq_epu64_mask(cacheline, key_mask);
11    // zmm registers are compared as 8 uint64_t
12    // mask irrelevant results before returning
13    return cmp & key_cmp_masks[cidx];
14 };
15
16 const size_t cidx = idx & (KV_PER_CACHE_LINE−1);
17 __m512i cacheline = load_cacheline(cidx);
18 // load a vector of the key in all 4 positions
19 __m512i key_mask = load_key_mask();
20 __mmask8 eq_cmp = key_cmp(cacheline, key_mask, cidx);
21 // compute a mask for copying the key into an empty slot
22 // will be 0 if eq_cmp != 0 (key already exists in the cacheline)
23 __mmask16 copy_mask = key_copy_mask(cacheline, eq_cmp, cidx);
24 copy_key(cacheline, key_mask, static_cast<__mmask8>(copy_mask));
25 // write the cacheline back; just the KV pair that was modified
26 __mmask8 kv_mask = key_mask | val_mask;
27 store_cacheline(cacheline, kv_mask);
28 // prepare for possible reprobe
29 ...
```

Listing 1. Vectorized insertion

We carefully design DRAMHiT-P's delegation mechanism to minimize the number of cache lines per queue (Figure 4). We group the metadata of multiple queues (i.e., a pointer into the data area, and size of the queue) that are accessed by one thread together in a minimal number of cache lines. The data area is separate and is accessible through a metadata pointer. We maintain residency of two cache lines for the data area, which allows us to fit one queue in 2.5 cache lines.

L1 residency Consumer threads can control the ordering of queue accesses, e.g., round-robin. Consumer prefetches the next queue before trying to access it. Specifically, we prefetch both the queue metadata and the actual data in the queue ring. Producers access the queues in a random order depending on the distribution of keys and their mapping to hash table partitions. A cache-friendly queue organization allows us to keep critical queue data structures in the L1 cache. We prefetch only the next line of the queue data when we approach the end of the current cache-line and a shared section pointer when we approach the end of the section.

3.4 Vectorization

To explore the benefits of SIMD instructions from the AVX512 instruction set, we develop an SIMD version of the DRAMHiT-P hash table. SIMD instructions can operate on 512 bits of data in parallel hence reducing the number of iterations in the conflict resolution loop. Moreover, the support for conditional operations—specifically, conditional load, store, comparison, and arithmetic instructions provided by the AVX instruction set—allow us to implement hash table operations without conditional branches.

Intel SIMD extensions do not support atomic operations like compare-and-swap. We utilize SIMD instructions only in a partitioned version of the hash table in which each partition is updated by a single thread, hence eliminating the need for concurrent updates. We further rely on empirical evidence that aligned 512-bit read accesses remain atomic in face of concurrent writes and hence avoid torn reads [54].

AVX instructions treat the 512-bit register as a vector of eight 8-byte values and use a mask that selects which elements of the vector will be affected by the operation (Listing 1). We create a collection of masks that allow us to select which elements of the cache-line will be affected by the operation (Listing 1, lines 2–6). For example, the second entry (line 4) allows us to operate on three out of four keys.

To implement a branchless version of the insertion operation, we first compute the position of the key within the cache-line, cidx (line 16). We use the position index (0-3) to select one of the masks above, hence operating only on a subset of the cache-line. We then load the cache-line containing the key and value pointed by the hash function into a 512-bit register (line 17). To compare the same key against every element of the vector, we load it into another 512 bit register at four different positions that match the position of the keys in memory (one cache line can hold four key-value pairs, line 19). To illustrate the AVX programming techniques, we provide the code for the key_cmp() function (lines 8–14). The function first compares two 512-bit values, cacheline and key_mask with the vectorized _mm512_cmpeq_epu64_mask() instruction (in most cases we utilize compiler-provided intrinsics). After performing the parallel comparison, we select the relevant result from the 8-bit register by using the key's position in the cache line, cidx (line 13).

We then use a conditional copy operation to copy the key into the cache-line, but only if the previous comparison was true (lines 23–24). Similarly, we conditionally store the cacheline back into the hash table. Note, in the regular x86 instruction set, conditional move instructions operate only on registers, hence providing no way to conditionally store the value back to memory. The AVX instruction set, however, provides support for conditional stores. If the mask is false, no memory transaction is generated.

To implement conditional reprobe when none of the keys in the cache-line matched the requested key, we implement similar code that relies on the conditional move instruction to update the queue pointer, hence inserting the new element back into the queue or leaving it unupdated. Similarly, to implement conditional prefetch, we either increment the address prefetching the next cache line, or prefetch the same

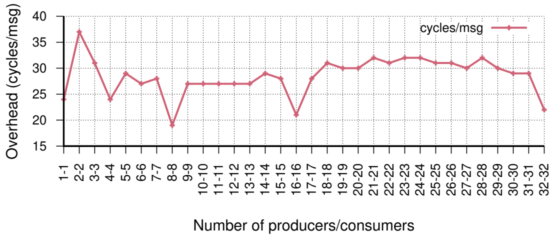


Figure 5. Latency of delegation (Intel)

cache-line again (since the cache-line is already cached, the prefetch does not generate a memory transaction).

4 Evaluation

We conduct all experiments in the CloudLab network testbed [53] and evaluate DRAMHiT on Intel and AMD architectures. Intel experiments utilize CloudLab c6420 servers configured with two Intel Xeon Gold 6142 16-core Skylake CPUs running at 2.6 GHz with 384 GB of memory. AMD experiments utilize CloudLab r6525 servers equipped with two AMD EPYC 7543 32-core Milan CPUs running at 2.8 GHz with 256 GB memory. Both systems have all memory channels populated (six channels per socket on Intel with DDR4-2666 MT/s and eight channels per socket on AMD with DDR4-3200 MT/s). All systems run 64-bit Ubuntu 20.04 with a stock kernel (turboboost, CPU idle states, and frequency scaling are disabled to reduce the variance in benchmarks).

4.1 Delegation

We first evaluate the overheads of our queue-based delegation mechanism. We execute a synthetic experiment that communicates between a group of threads (Figure 5). Each producer communicates with all consumers by repeatedly sending 16-byte messages to each consumer in a round-robin fashion. Consumers poll for messages and read the received value. Each producer sends 64 million messages. We vary the number of producers and consumers from 1 to 32. On average, it takes 22-37 cycles to send one message. The cost remains constant even when messages cross the boundary of a socket or when we scale the number of queues.

4.2 Performance on Uniformly Distributed Keys

To analyze the impact of our design ideas and optimizations on the performance of the hash table, we compare our hash tables DRAMHiT, DRAMHiT-P, and DRAMHiT-P-SIMD (an SIMD version of DRAMHiT-P) against Folklore. According to a recent study [35], Folklore is the fastest concurrent hash table that outperforms the closest competitors, i.e., Junction [51], TBB [49], cuckoo [45], Facebook folly [16], RCU [37], shunhash [58], hopscotch [20], and leahash [29]. For example, Folklore outperforms the closest competitor shunhash [58] by more than 30% for both insertions and finds on a uniform distribution of 10^8 8-byte keys and values. In many ways, our work uses the ideas of Folklore as a foundation for performance. DRAMHiT and Folklore share the same open addressing layout, linear probing as their conflict resolution mechanism, and a CAS-based synchronization scheme that avoids atomic instructions on the read path. DRAMHiT and DRAMHiT-P extend Folklore with asynchronous request completion, batching, delegation, and vectorized operations.

We execute our experiments on the hash table with 8-byte keys and 8-byte values. In each test, we create two hash tables: small and large. The small hash table occupies 16 MB of memory (1 million elements), and the large is 16 GB (1 billion elements). The size of the small hash table is chosen to fit into the caching hierarchy of a single socket on our Intel server. We rely on a zipf generator with a skew of 0 to generate a uniform distribution of keys. We populate the hash table, so it remains 75% full (the performance of hash tables that rely on open addressing degrades sharply at higher fill factors). CRC32 is used as the hash function. We then vary the number of logical CPU threads involved in the test from 1 to 64 (maximum on the Intel machine). In our experiments, we use a batch size of 16 requests, and uniformly distribute execution threads between sockets—e.g., in a test with two threads, the threads run on different NUMA nodes. Finally, we split the memory of the hash table in half, and allocate each half on a different NUMA node to ensure that the tests utilize all available memory channels.

Insertions We first perform a basic insertion test by inserting 0.75 million (small hash table) and 805 million (large hash table) unique uniformly-distributed keys into an empty hash table such that it becomes 75% full (Figure 6a and Figure 6b). We run DRAMHiT-P and DRAMHiT-P-SIMD with a 1-to-3 proportion between producers and consumers, e.g., 16 producers and 48 consumers for a 64-core configuration (we empirically found this configuration to result in the highest throughput).

With one memory miss on the insertion path, Folklore remains limited to the maximum of 417 Mops or 50% of the theoretical bandwidth on a large hash table. Leveraging the prefetch engine, DRAMHiT comes close to saturating the memory bandwidth with 792 Mops. Our Intel system supports the maximum bandwidth of 1,192 M cache-line transactions per second per socket (2,384 M for a two-socket system) on a workload of alternating reads and writes (Table 1). The hash table insertion requires two cache line transactions—one to read the cache line and one to write it back. However, on average, a large hash table requires 1.3 cache line reads due to conflicts. Additionally, every read that accesses the memory of a remote NUMA node, i.e., half of the reads on average, results in a write-back to clear the directory bits [23]. Hence, an average insertion requires 1.3 read- and 1.65 write-cache-line transactions, limiting the theoretical throughput to 808 Mops (we confirm the number of transactions empirically with the Intel VTune performance analysis tool).

Moreover, DRAMHiT comes close to saturating memory bandwidth with only 32 cores, which allows for the possibility of doubling the number of memory channels, and hence doubling the throughput of the hash table. On a uniform distribution, neither DRAMHiT-P nor DRAMHiT-P-SIMD can benefit from partitioning (contention is low), but pay the price of delegation. DRAMHiT-P achieves a throughput of 671 Mops. DRAMHiT-P-SIMD is slightly slower at 667 Mops.

On the small hash table, Folklore is limited by the latency of cache misses to remote caches (i.e., private L2 caches of other cores that most recently accessed the hash table entry and L3). Folklore reaches 441 Mops. DRAMHiT can fully benefit from prefetching capabilities and achieves an insertion throughput of 1180 Mops. DRAMHiT-P further leverages the locality of insertions, but as it loses a fraction of the CPU cores to producer threads, it cannot reach the performance of DRAMHiT, staying at a maximum throughput of 975 Mops (DRAMHiT-P-SIMD reaches 885 Mops).

Lookups For read-only tests, we first pre-initialize a hash table with uniformly distributed keys and then perform the same number of get() operations (Figure 6a and Figure 6b). In general, reads are faster than insertions, as they require a smaller number of coherence operations (on Intel machines, a read from a remote NUMA memory triggers a write to update the directory information). Hence, on average, a read requires 1.3 read transactions (0.3 due to reprobes) and 0.65 write transactions (half of the 1.3 read transactions trigger write-backs to remote NUMA memory). If we use an empirical MLC throughput measurement for a composition of 2 random reads and 1 random write (Table 1), the maximum achievable throughput is 1.3 Mops. On a large hash table, Folklore remains bottlenecked on accesses to memory, achieving only 451 Mops. Both DRAMHiT and DRAMHiT-P benefit from prefetching and achieve 973 Mops and 951 Mops, respectively, on 64 cores. DRAMHiT-P-SIMD is slightly faster at 1008 Mops.

On a small hash table, Folklore benefits from a lean lookup path as most of the hash table is cached in the last level cache of each socket (1616 Mops). DRAMHiT pays the price of the prefetch engine overhead (1513 Mops), and DRAMHiT-P, the additional overhead of partition lookups (1224 Mops). DRAMHiT-P-SIMD is marginally faster (1270 Mops).

Impact of individual optimizations Individual optimizations, i.e., prefetching, partitioning, and SIMD (vectorization), have different impacts depending on the operation, hash table size, and key distribution. Compared to Folklore, DRAMHiT, which implements a prefetching optimization, achieves 89-230% improvement on all configurations besides reads of a small hash table on which the overhead of the prefetch engine degrades the performance by 2-7%. A combination of partitioning and prefetch is only helpful on write-dominated workloads with high skew (5-163% improvement over prefetch) but degrades performance in all other cases. Finally, SIMD optimizations improve the performance of the partitioned hash table by only a few cycles on large hash tables (1-10 cycles or 1-11% improvement over DRAMHiT-P) but generally degrade performance on small ones (1-10%).

Impact of cache pollution To measure how performance degrades if a hash table competes for a fraction of the cache with the application itself, we design an experiment in which, after every hash table operation, we pollute the cache by prefetching several random cache lines from the memory of a large array (Figure 6c). We vary the number of cache lines prefetched from 0 to 512 and run our experiment on a large hash table with a uniform distribution of keys on 64 threads. Both AMD and Intel machines have 32 KB of L1 data cache (512 cache lines) that is shared between two hyperthreads.

Performance of both DRAMHiT and DRAMHiT-P degrades gracefully until it blends with Folklore when two hyperthreads pollute the entire cache by prefetching 256 cache lines each.

Impact of batching To measure how the performance of the hash tables degrades if an application cannot accumulate a batch of requests, we vary the batch sizes and measure the insertion and find throughput (Figure 7). We vary the batch size from 1 to 16 in power-of-two increments and run our experiment on a large hash table with a uniform distribution of keys on 64 threads. Performance of both DRAMHiT and DRAMHiT-P stays almost constant across various batch sizes for the insert operation.

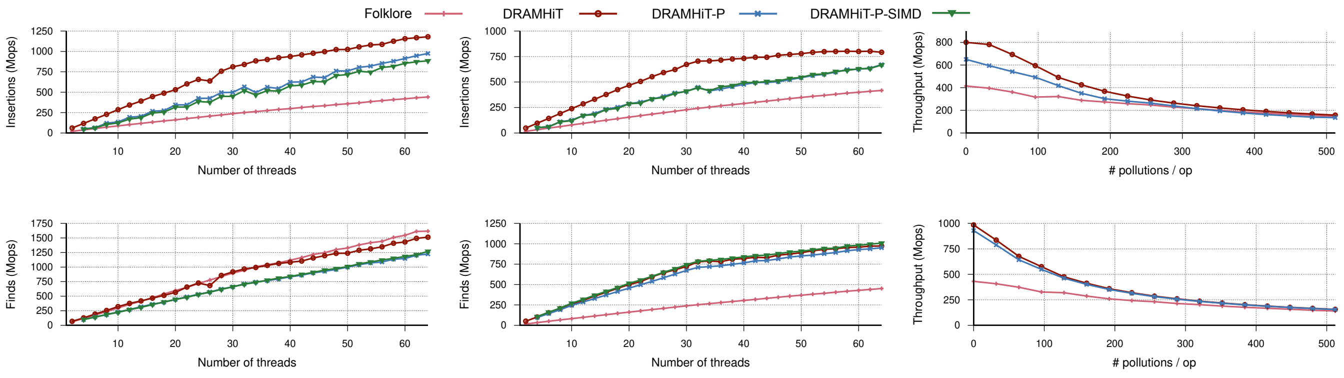
For finds, a batch size of 4 and 8 yield slightly better throughput for DRAMHiT and DRAMHiT-P, respectively. We observe only a difference of fewer than 10 cycles per operation across all batch sizes.

Mixed insertions and lookups To measure the performance of our hash tables on a mix of insertions and lookups, we perform an experiment that uses 64 threads on both uniform and zipfian (skew of 1.09) distributions (Figure 8c). We first pre-initialize the hash table with the corresponding distribution and then measure the throughput of mixed insertions and finds by varying the read probability ($p = 0$ corresponds to all writes, and $p = 1$ corresponds to all reads). The performance of all hash tables predictably goes up as the fraction of reads increases.

4.3 Performance on Skewed Distributions

To measure how DRAMHiT performs on distributions in which some fraction of the keys are accessed more frequently compared to the rest, we run a test in which we use 64 logical cores and vary the *theta* parameter of the Zipf distribution from 0 to 1.09. A *theta* of 0 results in a uniform distribution. With a *theta* of 1.09, roughly 10% of keys are accessed by 90% of requests.

Insertions On high skews (1.09), insertions suffer from the overheads of coherence protocol contention (Figure 8a and Figure 8b). On both small and large hash tables, Folklore and DRAMHiT perform poorly, achieving only 132-143 Mops.



(a) Uniform insertions and lookups (small) **(b)** Uniform insertions and lookups (large) **(c)** Impact of cache pollution (uniform, large)

Figure 6. Lookups and insertions on uniform distribution

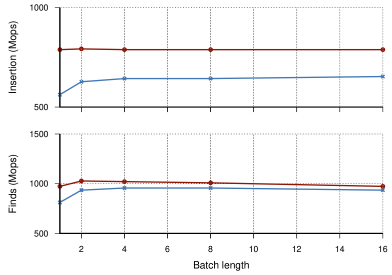


Figure 7. Impact of a batch size (uniform, large)

DRAMHiT-P, on the other hand, can benefit from delegation. On a skew of 1.09, it achieves 245 Mops on large and 351 Mops on small hash tables. SIMD optimizations provide a small improvement over DRAMHiT-P (3-23%).

Lookups DRAMHiT, DRAMHiT-P, and Folklore implement lookups without atomic operations. Hence, frequently accessed keys are cached locally by the CPU and benefit from the temporal cache locality amplified by high skew (Figure 8a and Figure 8b). On the small hash table, Folklore outperforms the two other hash tables since frequently accessed keys fit in L1 and L2 caches of the core (on our Intel machine, L2 is 1 MB). By avoiding the overhead of the prefetch engine, Folklore achieves 4059 Mops, compared to DRAMHiT (2919 Mops) and DRAMHiT-P (2919 Mops). On a large hash table, DRAMHiT (2820 Mops) and DRAMHiT-P (2133 Mops) still benefit from the prefetch engine, which allows them to prefetch cache lines that do not fit into the caches of the CPU (even 10% of the dataset is 1.6 GB). Without prefetching, Folklore is limited to 1499 Mops.

4.4 Latency

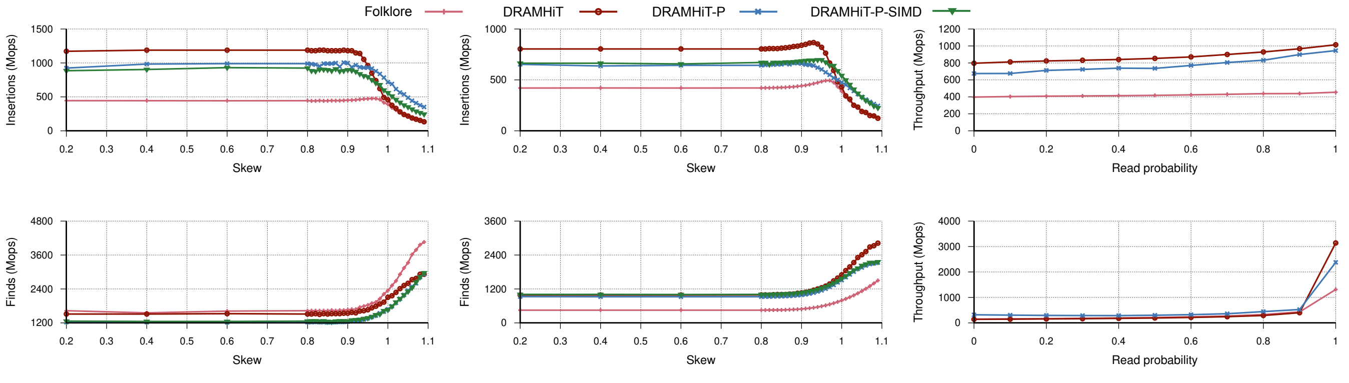
With aggressive batching, prefetching, and out-of-order completion, DRAMHiT and DRAMHiT-P trade latency for throughput. To measure how our optimizations affect the latency of hash table operations, we collect completion latency for each request and plot a latency CDF (Figure 9). On insertions, DRAMHiT-P returns immediately after submitting the request to the queue. Hence, it has the lowest insertion latency across all hash tables (90% of inserts complete within 52 cycles). Predictably, software prefetching pushes the latency of insertions and lookups in DRAMHiT to several thousands of cycles (90% of requests complete within 9090 cycles), which is much higher compared to 594 cycles for Folklore. While higher latency can hurt the performance of latency-sensitive applications, we argue that for a large class of throughput-demanding applications, the higher latency has a smaller impact, especially compared to the additional throughput provided by DRAMHiT and DRAMHiT-P.

4.5 Alternative CPU Architecture: AMD

To validate that our optimizations can be applied to different CPU architectures, we perform a collection of experiments on AMD servers. Compared to Intel, AMD EPYC CPUs support up to eight memory channels at the speed of 3200MT/s and have a larger caching hierarchy with L1 data cache of 32 KB, L2 cache of 512 KB, and a total of 256 MB of L3. A single AMD socket is comprised of 8 individual core complexes (each with 4 cores). Each core complex has a private 32 MB L3, of which 8 MB are shared across 4 cores of the complex, and 24 MB are used as 6 MB L3 slices private to each core. Despite the fact that the theoretical throughput of a single AMD socket can reach 204.8 GB/s, in practice, our system achieves 167 GB/s for random reads and 144 GB/s for a composition of one random read and one write.

We perform a collection of experiments similar to the ones above performed on the Intel system: reads and writes on a uniform and skewed distribution for both small and large hash tables (Figures 10 and 11).

An AMD system performs similar to Intel (until 32 threads) but achieves higher throughput in all the tests. One interesting anomaly is that on large datasets, AMD machines achieve peak bandwidth on 32 threads that are uniformly distributed across all core complexes. Performance drops sharply on larger core counts. While we were not able to investigate



(a) Zipfian insertions and finds (small) **(b)** Zipfian insertions and finds (large) **(c)** Mixed find/insertion tests

Figure 8. Finds and insertions on skewed distribution and mixed read/write tests

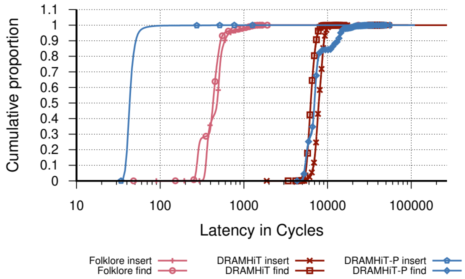


Figure 9. Cumulative latency distribution

the performance anomaly, the sharp drop in performance suggests a bottleneck in the cache-coherence subsystem (performance of a partitioned DRAMHiT-P continues to grow as it utilizes less coherence traffic).

4.6 Macrobenchmarks

K-mer counting K-mer counting, i.e., counting of substrings of length k, is a central piece in a variety of genomic algorithms ranging from genome assembly to error correction in sequenced genome reads. Specifically, k-mer counting is widely used by meta-genomic applications [11, 26, 36, 46] and can be done with a hash table in which k-mers are used as keys, and values maintain the count. Counting relies on the "upsert" operation that either inserts a new key into the hash table, or increments a value associated with the key.

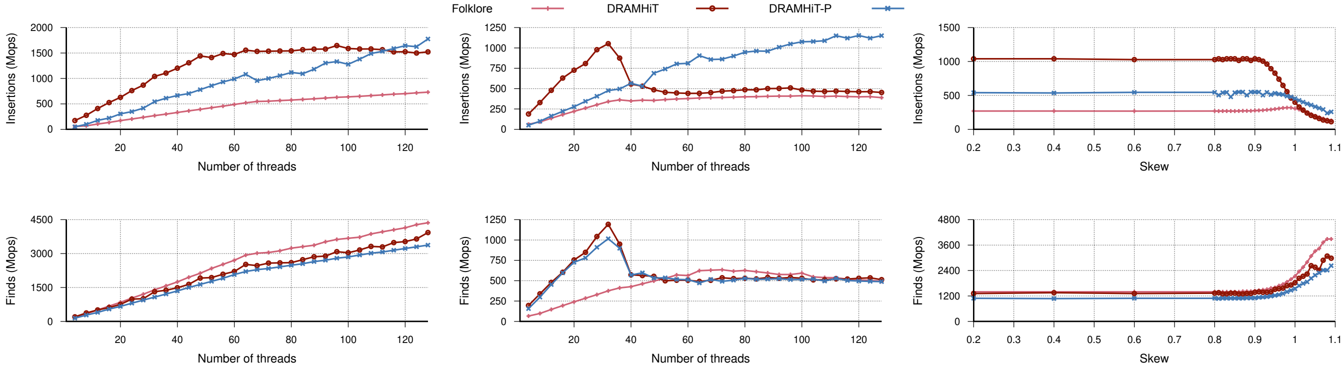
We evaluate our hash tables on the chromosomes of the *Drosophila melanogaster*, a 20GB dataset that contains 7.8 gigabases (Gbases), and *Fragaria vesca f. semperflorens*, a 15 GB dataset that contains 4.8 Gbases. We then count k-mers for different values of k, ranging it from 4 to 32 (Figure 12). We compare with CHTKC [66], a recent kmer counter that uses a lock-free hash table with chaining that showcases better performance compared to other state-of-the-art kmer counters, Jellyfish [36] and KMC3 [26]. We run the optimized version of CHTKC (chtkco) with 64 threads and 92GB of memory. To make a fair comparison, we disable the canonicalization of kmers in CHTKC as we do not perform that operation in our benchmark. CHTKC takes 40 seconds to count the kmers (K=32) for *D.melanogaster* and 51 seconds for *F.vesca*. As kmers from sequencing data often have zipfian distribution [57], DRAMHiT-P performs considerably better than other hash tables on both datasets. We empirically confirmed the distribution by measuring the frequencies of kmers from the two datasets, where the 25 most accessed kmers occupy 50-86% of the dataset.

Complexity of asynchronous interface Asynchronous submission and out-of-order completion of requests put additional engineering complexity on developers of applications that use DRAMHiT. Specifically, our macrobenchmarks submit upsertion requests in batches of 16 requests, which relies on a local array to accumulate the batch (5-10 lines of code). Arguably, the complexity of using DRAMHiT is low. DRAMHiT-P, however, requires re-structuring the application as a collection of threads, some of which are designated to perform update requests.

5 Related work

Parallel hash tables Herlihy and Shavit provide an in-depth survey of concurrent hash tables [19]. Unfortunately, many concurrent hash table designs rely on expensive synchronization, e.g., similar to early concurrent hash tables [9, 10, 21, 27], a widely-used TBB hash table relies on fine-grained locks around each bucket chain [49].

To reduce the overhead of synchronization, Michael [40] suggested the use of a fixed-size array of lock-free lists to handle collisions. Greenwald developed a lock-free closed addressing hash table centered around double-word compare and swap (DCAS) [17]. DCAS required multiple CAS operations and hence introduced prohibitive overheads [34]. Gao et al. proposed a lock-free hash table with open addressing and support for resizing [13]. Li et al. implemented a concurrent implementation based on per-bucket locks for the hash table that relies on cuckoo-hashing [32]. While improving



(a) Uniform distribution (AMD, small) **(b)** Uniform distribution (AMD, large) **(c)** Zipfian distribution (AMD, small)

Figure 10. Lookups and insertions on AMD CPUs

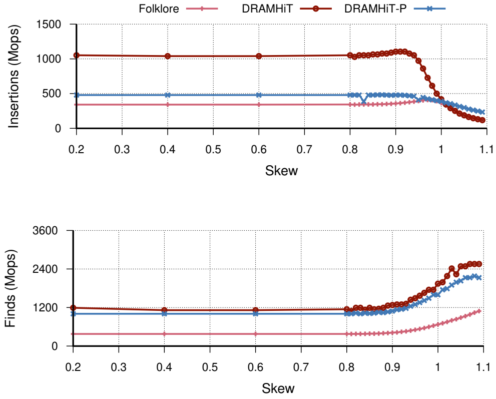
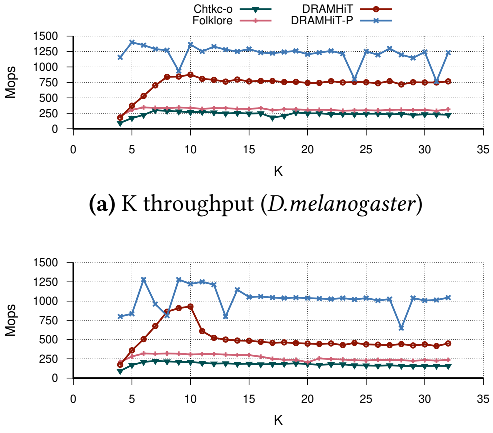


Figure 11. Lookups and insertions on zipfian distribution (AMD, large)



(a) K throughput (*D.melanogaster*)

(b) K throughput (*F.vesca f. semperflorens*)

Figure 12. Insertions throughput on K-mer datasets

over fine-grained locking, lock-free techniques only partially address the overheads of synchronization due to expensive cache line transfers and directory linearization. Our work addresses these overheads with prefetching and partitioning.

Shun and Blelloch introduce the notion of *phase concurrent hash tables* that allow the client to use either the read or write operation within a globally synchronized phase [58]. Arguably, our techniques can be used to benefit from efficient reads and writes during each phase.

Relativistic hash tables suggest the use of the read-copy update (RCU) synchronization primitive [48] that requires no synchronization for read operations [50, 61, 68]. Naturally, relativistic hash tables perform well on read-dominated workloads, but degrade quickly with an increasing fraction of updates. Leveraging a simple linearization protocol, DRAMHiT avoids writes and atomic operations for reads similar to relativistic designs.

Maier et al. implement and evaluate Folklore, an open-addressing hash table that requires only one atomic operation on the insertion path and no atomic primitives for lookup [35]. Detailed comparison demonstrates that their hash table achieves excellent performance—roughly 360 million insertions and 650 million lookups per second on a two-socket 64-thread machine with keys from a uniform distribution [35] (this is consistent with our experiments). They further extend naive Folklore implementation with support for resizing, Growt [35].

Hopscotch hashing implements a cache-friendly algorithm for open addressing [20]. While Hopscotch helps to avoid cache misses due to reprobing, our approach argues for explicit prefetching to make sure the hash table does not perform accesses to an unprefetched memory. LeaHash relies on hashing with chaining but suffers from repetitive memory misses when traversing the chain [29]. While explicit prefetching can be applied to chaining, traversing the chain introduces additional memory transactions, and therefore, will bottleneck on the throughput provided by the memory subsystem. Bolt develops a concurrent version of Robin Hood hashing [24]. Due to predictable reprobe distance, Bolt outperforms Growt but only on a relatively small 10^6 keys hash table that almost fits in the last level cache of the CPU and on a load factor of 50% (this is critical as the fast path in Bolt relies on the absence of reprobes) [24].

Delegation and combining To avoid cache coherence and synchronization overheads, delegation schemes designate one thread, a server or a combiner, that executes the code of the critical section on behalf of all other client threads. Flat combining is a dynamic delegation scheme in which any thread tries to acquire a basic spinlock to become a temporary combiner [18]. Fast-flyweight delegation (FFWD) provides an efficient NUMA-aware, static delegation scheme which is used to implement a hash table [55]. While FFWD (and delegation schemes in general) eliminates synchronization overheads and suggests a cache-coherence optimized communication protocol, the performance of one server core does not match the throughput of a multi-threaded system. FFWD outperforms traditional locking methods only when contention is extreme [55]. DRAMHiT-P relies on a hybrid technique that combines generalized, multi-server delegation with efficient cross-core communication mechanisms.

In the past, CPHash explored partitioned hash table design with the goal of avoiding cross-core synchronization overheads [39]. Unfortunately, CPHash suffers from an inefficient implementation for cross-core messaging, and a lack of lock-free read operations required to match the performance of modern lock-free hash tables.

Partitioned hash tables were further explored in the context of network-attached key-value stores [33]. Mica relies on hardware support from the network interface to delegate requests via a queue handled by a specific core [33]. Similar to DRAMHiT-P, Mica serves read requests from multiple cores, which allows scalability on read-dominated workloads. Compared to distributed systems like Mica, DRAMHiT-P has a much tighter cycle budget per request and hence needs a range of novel optimizations to meet it. Arguably, optimizations suggested in our work can be used to accelerate network-attached systems as well.

Fast inter-core communication The first concurrent lock-free queue that allowed synchronization without locks or atomic primitives was introduced by Lamport [28]. Unfortunately, despite its lock-free design, Lamport's queue suffers from constant cache thrashing of the producer and consumer pointers, i.e., transfers of cache lines between producer and consumer cores. DBLS [65], MCRingBuffer [30] and Liberty [22] optimize Lamport's design by adding a *lazy loading* optimization that reduces the number of accesses to the shared producer and consumer pointers. In addition, DBLS [65] and MCRingBuffer [30], BatchQueue [52], Liberty [22] introduce batching optimizations that keep enqueue and dequeue indices updating shared control state once per batch of enqueue operations. FastForward eliminates sharing of the control state by storing a special NULL value directly into the element of the queue after it was processed [14]. Also, to reduce cache-line bouncing between the cores, FastForward proposed an adaptive flow-control algorithm ensuring that the producer and consumer do not access the same cache line of the queue. Another batching queue, BQueue, addresses the problem of deadlock typical for batching queues by introducing an idea of backtracking, i.e., probing the batch space in the power-of-two decrements when the producer becomes idle [67]. Lynx further specializes the batch queue by removing the queue logic that is responsible for checking the boundary conditions from the critical path of the enqueue and dequeue operations [41]. To handle queue overflow, Lynx relies on CPU exceptions (delivered through signal handlers) triggered when the enqueue and dequeue code performs an access outside of the queue area. Unfortunately, signal handling does not scale well on commodity operating system kernels like Linux due to a global lock in the signal delivery path. While Lynx' demonstrates impressive performance numbers on a single-core, single-queue setup, our attempts to scale it were unsuccessful. A large fraction of Lynx's impressive performance is due to aggressive compiler inlining and optimizations possible only for a point-to-point (i.e., single queue) communication—the compiler inlines queue metadata and keeps all local state in registers. Scaling Lynx to larger number of queues introduces additional memory accesses (2-3 cycles per L1 memory access), which negatively affects Lynx' performance when more than one queue is used.

6 Conclusions

Our work explores new ways of improving hash table performance on modern hardware. We argue that modern machines should be treated as distributed systems with relatively expensive communication channels across non-uniform memory and caches. We develop a range of optimizations typical for a distributed system—asynchronous interface, fully-prefetched memory access, batching with out-of-order completion, and a scalable delegation scheme—borrowing insights from distributed systems, but applying them in the environment of a commodity server. These optimizations allow us to design a hash table that can saturate the bandwidth of a modern memory subsystem—arguably, the real architectural bottleneck on modern machines—and outperform the fastest commodity hash tables by a factor of two.

Acknowledgments

We would like to thank USENIX ATC'21 and EuroSys'23 reviewers and our shepherd, Sam H. Noh, for numerous insights helping us to improve this work. Also, we would like to thank Harishankar Vishwanathan, Daman Mohan Kumar, and Nivedha Krishnakumar who contributed to various parts of the system. This research is supported in part by the National Science Foundation under Grant Numbers CNS-1817120. Vikram Narayanan is partly supported by an IBM PhD fellowship.

References

[1] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*, 2016.

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE 2013)*, pages 362–373, April 2013.

[3] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, pages 37–48, 2011.

[4] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[5] Intel Corporporation. Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. Accessed: 2022-05-18.

[6] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 33–48, 2013.

[7] David J DeWitt and Robert Gerber. Multiprocessor Hash-Based Join Algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB '85)*, pages 151–164, 1985.

[8] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 523–535, 2016.

[9] Carla Schlatter Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '83)*, pages 106–116, 1983.

[10] Carla Schlatter Ellis. Concurrency in linear hashing. *ACM Transactions on Database Systems*, 12(2), 1987.

[11] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1):9, 2017.

[12] Flux Research Group. CloudLab Web site. http://www.cloudlab.us.

[13] Hui Gao, Jan Friso Groote, and Wim H Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, 2005.

[14] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 43–52, 2008.

[15] Google. The CityHash family of hash functions. http://code.google.com/p/cityhash/. Accessed: 2021-01-12.

[16] Google. Folly: Facebook Open-source Library. https://github.com/facebook/folly/. Accessed: 2021-01-12.

[17] Michael Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC '02)*, pages 260–269. Association for Computing Machinery, 2002.

[18] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*, pages 355–364, 2010.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*, chapter 13. Morgan Kaufmann Publishers, 2008.

[20] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing (DISC 2008)*, pages 350–364, 2008.

[21] Meichun Hsu and Wei-Pang Yang. Concurrent Operations in Extendible Hashing. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*, pages 241–247, aug 1986.

[22] Thomas B Jablin, Yun Zhang, James A Jablin, Jialu Huang, Hanjun Kim, and David I August. Liberty queues for epic architectures. In *Proceedings of the Eigth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology (EPIC)*, 2010.

[23] John McCalpin. Topology and Cache Coherence in Knights Landing and Skylake Xeon Processors. https://www.ixpug.org/documents/1524216121knl_skx_topology_coherence_2018-03-23.pptx. Accessed 2022-10-10.

[24] Endrias Kahssay. A fast concurrent and resizable Robin Hood hash table, 2021.

[25] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. In *Proceedings of the VLDB Endowment*, volume 2, pages 1378–1389, Aug 2009.

[26] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

[27] Vijay Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6):681–694, jun 1990.

[28] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[29] Doug Lea. util.concurrent.ConcurrentHashMap, revision 1.3. JSR-166, the Proposed Java ConcurrencyPackage. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent, 2003.

[30] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A Lock-Free, Cache-Efficient Shared Ring Buffer for Multi-Core Architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '09)*, pages 78–79, 2009.

[31] Daniel Lemire. fastrange: A fast alternative to the modulo reduction. https://github.com/lemire/fastrange/. Accessed: 2023-02-20.

[32] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, pages 1–14, 2014.

[33] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 429–444, 2014.

[34] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking K-Compare-Single-Swap. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '03)*, pages 314–323, 2003.

[35] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and General(?)! *ACM Transactions on Parallel Computing*, 5(4), February 2019.

[36] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[37] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[38] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer Science & Business Media, 2008.

[39] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*, pages 319–320, 2012.

[40] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, pages 73–82, 2002.

[41] Konstantina Mitropoulou, Vasileios Porpodas, Xiaochun Zhang, and Timothy M. Jones. Lynx: Using os and hardware support for fast fine-grained inter-core communication. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*, 2016.

[42] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–10, 2014.

[43] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. Association for Computing Machinery, 2014.

[44] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*, pages 261–270, 2009.

[45] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[46] Tony C. Pan, Sanchit Misra, and Srinivas Aluru. Optimizing High Performance Distributed Memory Parallel Hash Tables for DNA k-mer Counting. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*, pages 135–147, Nov 2018.

[47] Tian Bu Patrick P. C. Lee and Girish Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[48] Paul E. McKenney. RCU vs. locking performance on different CPUs. In *Linux.Conf.Au*, 2004.

[49] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[50] Nick Piggin. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing (resend). https://lwn.net/Articles/302132/. Accessed: 2022-10-10.

[51] Jeff Preshing. Junction. https://github.com/preshing/junction/. Accessed: 2021-01-12.

[52] Thomas Preud'homme, Julien Sopena, Gael Thomas, and Bertil Folliot. Batchqueue: Fast and memory-thrifty core to core communication. In *2010 22nd International Symposium on Computer Architecture and High Performance Computing*, pages 215–222, 2010.

[53] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[54] Erik Rigtorp. Aligned AVX loads and stores are atomic. https://rigtorp.se/isatomic/. Accessed: 2022-05-18.

[55] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. FFWD: Delegation is (Much) Faster than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 342–358, 2017.

[56] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, 1986.

[57] Moustafa Shokrof, C Titus Brown, and Tamer A Mansour. Mqf and buffered mqf: Quotient filters for efficient storage of k-mers with their counts and metadata. *BMC bioinformatics*, 22(1):1–14, 2021.

[58] Julian Shun and Guy E. Blelloch. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*, pages 96–107, 2014.

[59] Alex Stivala, Peter J Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *Journal of Parallel and Distributed Computing*, 70(8):839–848, 2010.

[60] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *33rd Design Automation Conference Proceedings, 1996*, pages 641–644, 1996.

[61] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[62] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 18–32, 2013.

[63] Freark I. van der Berg and Jaco van de Pol. Concurrent Chaining Hash Maps for Software Model Checking. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 46–54, 2019.

[64] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering (ICPE '22)*, pages 165–175, 2022.

[65] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 244–258, 2007.

[66] Jianan Wang, Su Chen, Lili Dong, and Guohua Wang. CHTKC: a robust and efficient k-mer counting algorithm based on a lock-free chaining hash table. *Briefings in Bioinformatics*, 22(3), 05 2020.

[67] Junchang Wang, Kai Zhang, Xinan Tang, and Bei Hua. B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication. *International Journal of Parallel Programming*, 41(1):137–159, 2013.

[68] Herbert Xu. bridge: Add core igmp snooping support. http://git.kernel.org/linus/eb1d16414339a6e113d89e2cca2556005d7ce919. Accessed: 2022-10-10.

A Artifact Appendix

A.1 Abstract

We release the source code of all software used in this paper along with detailed build instructions and automated scripts used for running the benchmarks as a collection of publicly-hosted Git repositories.

A.2 Description & Requirements

A.2.1 How to access The artifacts are hosted at the git repository https://github.com/mars-research/dramhit-artifacts/tree/esys23-ae-v1. The evaluated version of the artifact is available at https://doi.org/10.5281/zenodo.7719328.

A.2.2 Hardware dependencies We have tested DRAMHiT on the following hardware (available on CloudLab):

- Dell PowerEdge C6420 machine configured with two Intel Xeon Gold 6142 CPUs
- Dell Poweredge R6525 machine configured with two AMD Epyc 7543 CPUs

Though we have not tested it on other hardware, the experiments should be reproducible on a range of machines as long as all the memory channels are populated.

A.2.3 Software dependencies

The DRAMHiT build infrastructure was tested on an x86-64 Ubuntu 22.04 LTS system.

A.2.4 Benchmarks

None

A.3 Set-up

We conduct all experiments in the openly-available CloudLab cloud infrastructure testbed [12] and make our experimentation environment available via an open CloudLab [53] profile that automatically creates the software environment required to run DRAMHiT: https://github.com/mars-research/cloudlab-profiles/tree/kvstore-ae.

A.4 Evaluation Workflow

A.4.1 Major Claims

- **(C1)**: DRAMHiT achieves 973 Mops for reads and 792 Mops for writes on 64 threads outperforming existing lock-free designs by nearly a factor of two. This is proven by the experiment (E2) described in Figure 6b whose results are discussed in Section 4.2.

A.4.2 Experiments

The following experiments (E1-E3) were evaluated by the artifact evaluation committee as our peer-reviewed paper only had these experiments.

- **Experiment (E1)**: Synchronization overheads [5 human-minutes + 1 compute-hour]: measures the overheads of various synchronization primitives such as spinlocks and atomic increments on two different datasets (32 MB and 1 GB).
  The script (under the fig2 directory) from the artifact repository contains the necessary scripts to configure, run and plot Figure 2.

- **Experiment (E2)**: Hash table experiments [5 human-minutes + 12 compute-hour]: measures the throughput of insertions and lookups on two different distributions (uniform and zipfian with different skews) for two different datasets (small and large).
  The script (under the ht-bench directory) from the artifact repository contains the necessary scripts to configure, run and plot Figure 6.

- **Experiment (E3)**: Latency [5 human-minutes + 5 compute-hours]: measures the latency of insertion and lookups on a 64-thread configuration.
  The setup script under the latency directory builds and runs the program to measure the latency of insertion and lookup on folklore, DRAMHiT, and DRAMHiT-P, and plots a cumulative distribution function (CDF) (Figure 9).

- **Experiment (E4)**: Macro benchmark (kmer histogram) [5 human-minutes + 5 compute-hours]: measures the insertion throughput of k-mers for various values of k.
  The setup script under the kmer-bench directory builds and runs the program to measure the insertion throughput on folklore, DRAMHiT, and DRAMHiT-P, and compares with one of the existing state-of-the-art kmer counters that uses lock-free hash tables Figure 12.

A.4.3 Additional experiments

We performed the following additional experiments for the camera-ready version of the paper. The artifacts for these experiments are hosted at the git repository https://github.com/mars-research/dramhit-artifacts.

Set-up We use an updated profile that automatically creates the software environment required to run DRAMHiT: https://github.com/mars-research/cloudlab-profiles/tree/dramhit-ae.

- **Experiment (E5)** Hash table experiments on AMD architecture, where we perform the same set of experiments on an AMD node to understand how the optimizations behave on a different architecture.
  The script (under the ht-bench directory) from the artifact repository contains the necessary scripts to configure, run and plot Figures 10 and 11.

- **Experiment (E6)**: Cache pollution [5 human-minutes + 6 compute-hours]: measures the impact of hash table performance when an application competes for the cache space.
  The setup script under the pollutions directory builds and runs folklore, DRAMHiT, and DRAMHiT-P by polluting the cache after every operation to measure the throughput for insertions and lookups and plots Figure 6c.

- **Experiment (E7)**: Mixed workloads [5 human-minutes + 8 compute-hours]: measures the hash table performance with a mix of insertions and lookups. We vary the read probability from 0.1 to 1.0, which controls the proportion of insert and lookup operations.
  The setup script under the mixed-workloads directory builds and runs folklore, DRAMHiT, and DRAMHiT-P to plot the combined throughput for insertions and lookups in Figure 8c.

- **Experiment (E8)**: Impact of batching [5 human-minutes + 8 compute-hours]: measures how the hash table performance varies when the batch size is varied. We vary the batch length in power-of-two increments from 1 to 16.
  The setup script under the batching directory builds and runs DRAMHiT, and DRAMHiT-P by varying the batch length and plots the Figure 7.