DRAMHiT++: Towards faster hashtables

Joshua Tlatelpa-Agustin, Jerry Zhang, Anton Burtsev

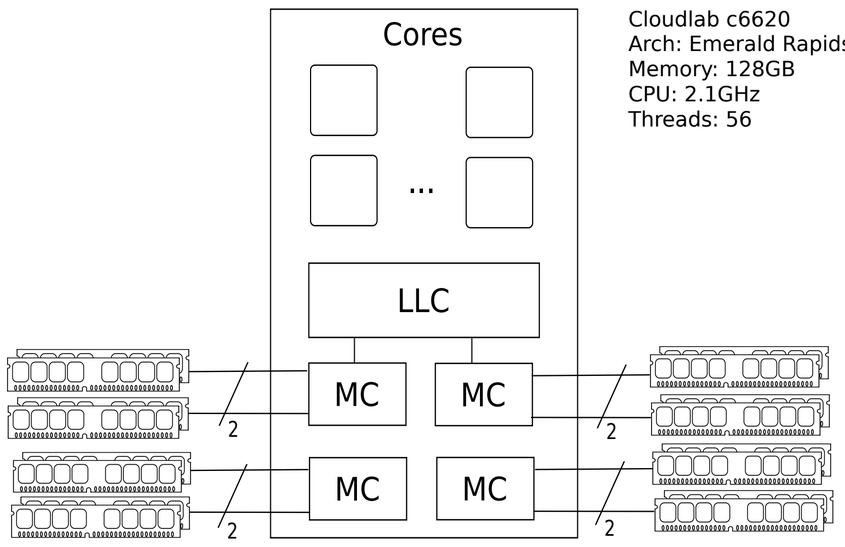
THE UNIVERSITY OF UTAH

Introduction

- Hash tables are ubiquitous and critical for modern data-intensive applications, e.g., key-value stores, databases, genomic and meta-genomic analysis, dynamic programming, model checking, graph processing, and matrix multiplication.
- Accesses to the hash table often dominate execution in these programs.
- Despite hash functions taking 2–20 cycles to compute, the fastest hash tables spend 150–300 cycles per lookup and insertion.
- The majority of time is spent waiting on memory.
- Today's hash tables are limited by the decade-old design choice of treating memory as a subsystem with a synchronous interface.

Example Machine

Configuration	Bandwidth (GB/s)	Cache lines (MOPS)	Cycle budget
Theoretical	307.2	4800	25
Seq reads	242.3	3785	31
Random reads	234.3	3660	32



Cloudlab c6620
Arch: Emerald Rapids
Memory: 128GB
CPU: 2.1GHz
Threads: 56

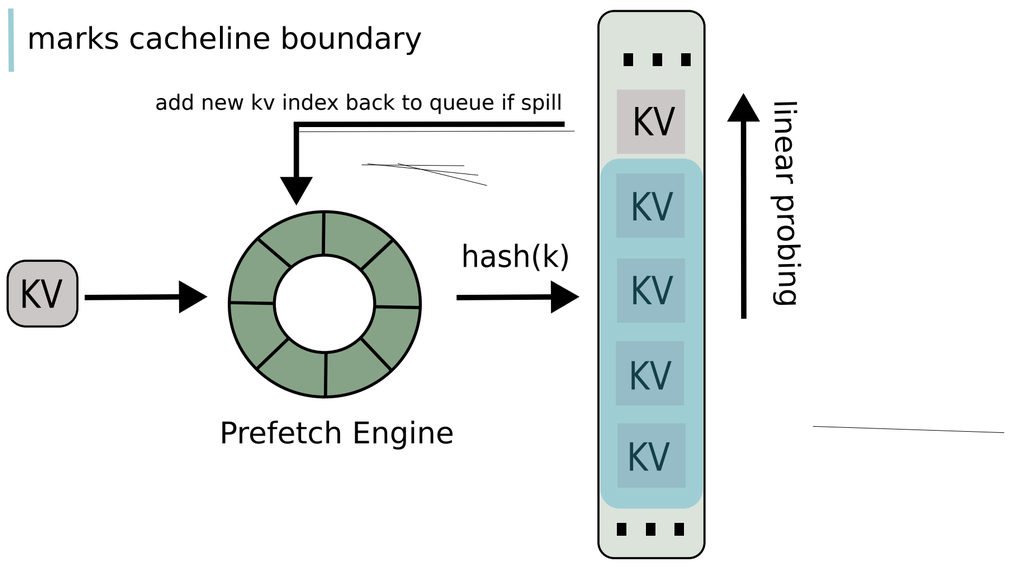
The diagram above shows 4 Memory Controllers (MC), each supporting 2 DIMM slots. The Last Level Cache (LLC) configuration is as follows: 80 KiB L1 cache per core, 2 MiB L2 cache per core, and a shared 53 MiB L3 cache for all cores.

Design Principles

- Minimal number of cache misses: The cost of a miss below the L2 cache is prohibitive and should be avoided on the critical path.
- Minimal number of memory transactions: Additional memory accesses can sharply degrade performance if memory bandwidth is saturated. Memory transactions should be minimized through efficient conflict resolution policy, hash table organization, and data structure layout.
- No contention: On workloads with high skew the overhead of contention dominates all others. To achieve peak performance hash tables should minimize or avoid contention.

DRAMHiT Hashtable

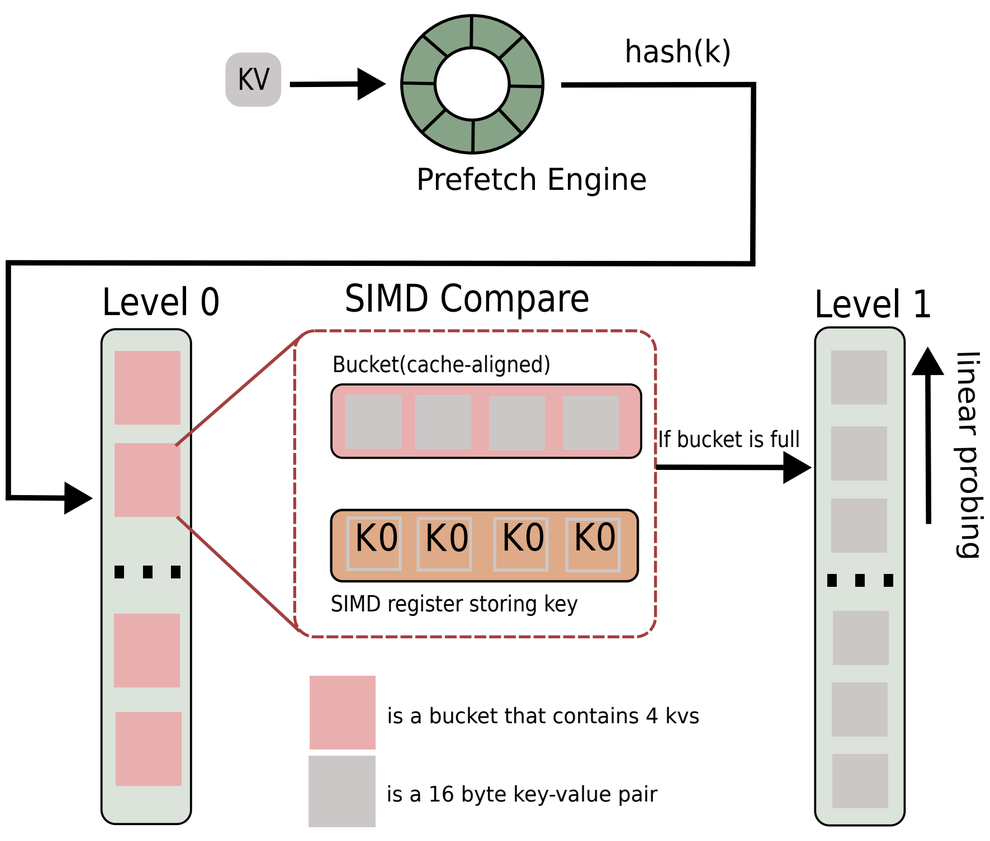
DRAMHiT uses prefetch engine to prefetch workload and do operations in a batch manner. After all items in the queue have been prefetched it uses linear probing to resolve collisions.



Under this collision scheme, a given kv pair could potentially spill over a cacheline, and in order to preserve principle of never accessing unprefetch memory, DRAMHiT will add kv back to the prefetch engine with spill over index.

Multilevel Hashtable

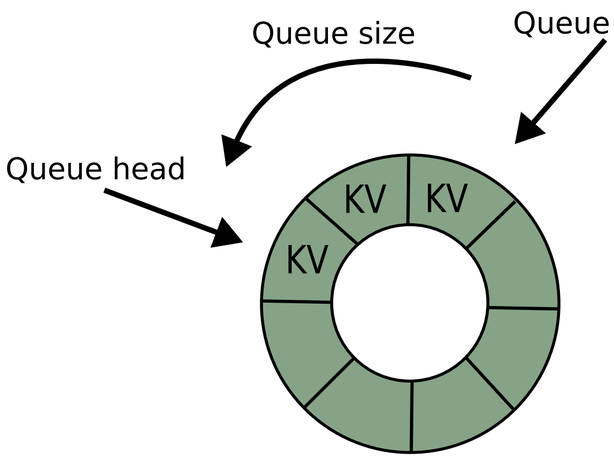
We observe that on uniform distribution, DRAMHiT experiences 1.3 reprobes per find operation on average at 70% fill factor. Thus, Multilevel hashtable intends to reduce the spill problem by adding an additional level that contains buckets. Each bucket represents a cacheline with 4 key values. When a bucket is full, we will insert into the second level where the implementation is identical as a regular DRAMHiT.



On find operation, we uses SIMD hardware support on x86 to look up multiple KVs. Because buckets are cache aligned, SIMD technique is also more efficient than regular DRAMHiT.

Prefetch Engine

The Prefetch engine is implemented as a ring buffer, where the queue head advances when an item enters the queue, and the queue tail advances when the item leaves the queue.



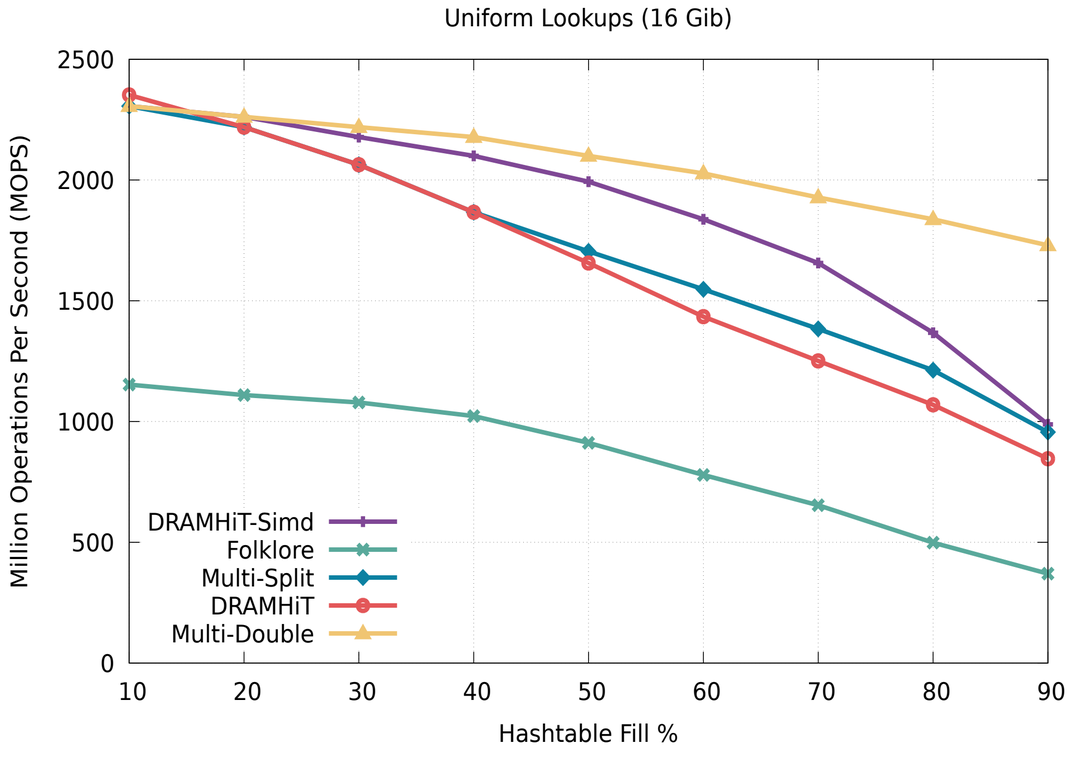

```
void add_to_prefetch_queue(kv):
    idx = hash(key)
    prefetch(idx)
    prefech_queue[queue_head] = kv
    queue_head++
    if (queue_head >= queue_sz)
        queue_head = 0
```

Prefetch engine computes hash of the KV pair upon entering the queue, and prefetches the corresponding cacheline in the hashtable.

Results

Hashtable	Prefetcher	Probing Strategy	SIMD for lookup	memory cost
Folklore	hardware	linear	No	1X
DRAMHiT	software	linear	No	1X
DRAMHiT SIMD	software	linear	Yes	1X
Multi-Split	software	linear	Yes	1X
Multi-Double	software	linear	Yes	2X

All hash tables use atomic instructions as synchronization primitives. For Multi-Split, the first and second level evenly split key space based on requested capacity, so in the worst case it can only hold 50% percent of the keys. This is ongoing work.



Graph comparing the performance of different variations of hashtable with respect to fill factor. All hashtables are run with 56 threads, constant frequency 2.1GHz, O3 gcc optimization flag on cloudlab C6620 machine.

Acknowledgements: This research is supported by NSF 2220410.