STORAGE DEVELOPER CONFERENCE
SDC 22
Fremont, CA
September 12-15, 2022
BY Developers FOR Developers

A SNIA Event

Challenges and Opportunities in Developing a Hash Table Optimized for Persistent Memory

New paradigms for a new storage technology

Steve Heller

Why yet another hash table/KV store?

Compare and contrast with existing technology

	Algorithm name			
Feature	TwoMisses	unordered_map	open addressing	redis
Variable-length?	Yes	Yes	No	Yes
Persistent?	Optional	No	Maybe	Optional
Performance guarantees?	1 or 2 random accesses	Asymptotic	1 random access	No
Incremental rehash?	Yes	No	Maybe	Yes
Storage efficiency	Good	Low	Best	Low
Power-fail consistency	Optional	No	No	Optional
Processing model				
Embedded/server	Embedded	Embedded	Embedded	Server
Threading	Single	Single	Single	Multiple

When would you choose the TwoMisses hash table?

You are currently using or are considering unordered_map or a similar node-based hash table because you need variable-length capabilities, and any of the following are true:

1. You want better storage efficiency and/or better speed than these other algorithms.
2. You want incremental rehashing rather than a big-bang rehash.
3. You want the option to persist your data across program termination.

When would you not use TwoMisses?

If any of the following are true, the TwoMisses hash table is not for you:

1. You want multi-user operation via threading.
2. You want a server-based system rather than embedded.
3. You want an open-source solution.

The opportunity

Approaching the theoretical minimum latency for a hash table stored on persistent memory

What is the minimum access latency of a persistent hash table?

- At least one random access to the underlying memory or storage is needed to retrieve a record from any hash table, assuming reasonably uniform access.
- For fixed-length records, there is a well-known solution that achieves this minimum: open addressing.
- For variable-length records, no such solution has existed. Instead, pointers have been required, resulting in a minimum of two random accesses.
- The new TwoMisses heterogeneous hash table achieves the theoretically optimal limit of one random access to retrieve a “short” variable-length record, i.e., a record less than a configurable size such as 40 bytes of key + data. For longer records, this hash table requires exactly two random accesses.

How the new 2Misses heterogeneous hash table works

- Instead of dividing the table into fixed-size record locations, the table is divided into “tranches” of a fixed size larger than any “direct” record that will be stored in the table itself; longer records will use indirection.
- When calculating a record location, the hashing algorithm must generate the address of the start of one of these tranches.
- Once the tranche address has been determined, the contents of that tranche are parsed. The beginning of a tranche containing two “direct” records might look like this:

Byte	Byte	Byte	Multibyte	Multibyte	Byte	Byte	Byte	Multibyte	Multibyte
Record type	Key length	Value length	Key…..	Value…..	Record type	Key length	Value length	Key…..	Value…..

Storing a record

- First the record size is compared to the “direct” record length limit. If it is within that limit, the record will be stored in the table itself. Otherwise, the record data is stored in an “overflow file” and an “indirect” record pointing to that storage is created to be stored in the table.
- Then the size of the record to be stored in the table is compared to the remaining empty space in the tranche. If there is enough space to add the record, it is added, possibly at the end of the tranche.
- Otherwise, sequential collision resolution is employed to locate a tranche containing enough free space to store the record. No further random accesses within the table will be required.

Rehashing

- When storing a record, if the algorithm determines that the file is nearly at its capacity, a rehashing round is scheduled to begin.
- Each round relocates records to different locations in the table, which grows in the process. This is generally done in an incremental way so as not to cause a prolonged service outage; the overflow file is not reorganized in this operation.

Looking up a record

- The target tranche is searched to determine whether the record is present.
- If so, a reference to the record data is returned; otherwise, an algorithm compatible with the collision resolution method used for storing a record is used to determine whether it might be in another tranche. If so, the search continues; otherwise, the search is terminated with the status “not found”.
- Since this algorithm makes exactly one random access to the table itself and one optional additional random access to access data in the “overflow file” for indirect records, the maximum number of random accesses to retrieve any record by key is two.

Some of the challenges

"We're not in Kansas anymore."

For a hash table stored on persistent memory, which of these statements can be relied on?

- It doesn't matter which CPU socket your program runs on.
- Running in a Hyper-V Windows VM slows the program down by ~10x.
- Running in a Hyper-V Ubuntu VM slows the program down by ~10x.
- Retrieval speeds are reduced after the file is rehashed.
- Reading during rehashing is much slower.
- If you have an integer key, there will be no significant performance penalty for using a bitmap to keep track of the existence of billions of records.

None of the previous statements is reliably true.

- A round trip over the socket interconnect (UPI) costs on the order of 200 nanoseconds, lengthening overall latency by about 30%.
- Running in a Hyper-V Windows VM or an Ubuntu VM can have about a 15% performance penalty, possibly partially due to inefficient placement of vCPUs.
- Retrieval latency is not necessarily increased after rehashing.
- Reading while rehashing can be as fast as when not rehashing.
- Using a bitmap to keep track of the existence of billions of records can reduce performance by about 15%.

Some general rules to live by for optimal results

- Do not employ large auxiliary data structures (exceeding cache size).
- Do not make system calls in the hot path.
- Avoid dynamic memory allocation in the hot path; e.g., return the actual memory-mapped record data rather than making a copy.
- Avoid unnecessary object creation, even on the stack, in the hot path.
- Minimize UPI traffic to the extent possible.

Implementation implications of the general rules

- Since large auxiliary data structures cause excessive page table lookups in the virtual memory system, a directoryless structure is required for best performance.
- Since system calls are slow compared to persistent memory access, normal multithreading techniques, e.g., using std::thread, are not feasible. Just one sync call to synchronize data between threads can take longer than retrieving a record from the hash table.

Effects of the rules on the API

- The 2Misses hash table has two interfaces, a simple one that looks like a C++ unordered_map, and an advanced one for performance.
- The unordered_map type interface returns a std::string, which requires a heap memory allocation and later deallocation. In addition, modifying the value requires a new lookup in order to write the new value to the table.
- The advanced interface doesn't make a copy but returns a C++ object similar to a string_view, containing a pointer to the persistent memory address of the record, and a length. This avoids heap memory usage.
- This interface also allows update-in-place if the value length does not change, eliminating the second lookup to update the value.

Other API considerations

- The algorithm can move records around any time there is an add or a delete operation, not just during rehashing. For this reason, no existing reference to a record can be considered valid after adding a new record, deleting a record, or replacing a record with a value of different length. Updating a record in place or replacing a value with a new value of the same length does not invalidate existing record references.
- Iterators must also be considered invalid after the same operations that invalidate existing record references.

What is the payoff for following this new paradigm?

Let’s see some actual performance numbers

Test results from a C++ implementation on previous-generation hardware where the hash table is on persistent memory

- Retrieving ~1 million “short records” (averaging 8 byte keys/8 byte values) per second, 95th percentile retrieval/update latency ~1.6 microseconds.
- Retrieving >600k “medium records” (averaging 10 byte keys/100 byte values) per second, 95th percentile retrieval/update latency ~2.4 microseconds.
- Adding new records, while not rehashing, is about 50% faster than retrieving.
- Above results gathered at 1 billion record file size, flat through tens of billions of records.
- Test results from dual Xeon 2nd Gen 4215, 128GB DRAM, 2 TB Intel® Optane™ DC Persistent Memory, Windows 10. Ubuntu implementation is generally a few percent faster.

Test results from a C++ implementation on latest-generation hardware where the hash table is on persistent memory

- Retrieving ~1 million “short records” (averaging 8 byte keys/8 byte values) per second, 95th percentile retrieval/update latency ~1.6 microseconds.
- Retrieving >700k “medium records” (averaging 10 byte keys/100 byte values) per second, 95th percentile retrieval/update latency ~2.4 microseconds.
- Adding new records, while not rehashing, is about 50% faster than retrieving.
- Above results gathered at 1 billion record file size, flat through tens of billions of records.
- Test results from dual Xeon 3rd Gen Gold 6326, 256GB DRAM, 4 TB Intel® Optane™ DC Persistent Memory Series 200, Windows 10.

Test results from a C++ implementation on previous-generation hardware where the hash table is cached in memory

- Retrieving ~1.6 million “short records” (averaging 8 byte keys/8 byte values) per second, 95th percentile retrieval/update latency ~0.8 microseconds.
- Retrieving ~1 million “medium records” (averaging 10 byte keys/100 byte values) per second, 95th percentile retrieval/update latency ~1.5 microseconds.
- Adding new records, while not rehashing, is slightly faster than retrieving.
- Short record speeds tested at 4 billion record count, medium records at 500 million record count, both limited by DRAM.
- Test results from dual Xeon 2nd Gen 4215, 128GB DRAM, Windows 10.

Test results from a C++ implementation on latest-generation hardware where the hash table is cached in memory

- Retrieving ~1.8 million “short records” (averaging 8 byte keys/8 byte values) per second, 95th percentile retrieval/update latency ~0.75 microseconds.
- Retrieving ~1 million “medium records” (averaging 10 byte keys/100 byte values) per second, 95th percentile retrieval/update latency ~1.5 microseconds.
- Adding new records, while not rehashing, is slightly faster than retrieving.
- Short record speeds tested at 9 billion record count, medium records at 1 billion record count, both limited by DRAM.
- Test results from dual Xeon 3rd Gen Gold 6326, 256GB DRAM, Windows 10.

How close is this to the theoretical limit?

Theory vs. practice for variable-length records, 50% read/50% update on persistent memory with previous-generation server

- With short records, averaging 8-byte keys and 8-byte values, the Visual Studio profiler indicates that about 50% of the time is spent reading and writing. As a sanity check, the average total latency is about 1000 nanoseconds and one persistent memory random access takes ~350 nanoseconds, so we are at least within a factor of 3 of the theoretical limit.
- With medium records, averaging 10-byte keys and 100-byte values, the Visual Studio profiler indicates that about 70% of the time is spent reading and writing. As a sanity check, the average total latency is about 1500 nanoseconds and even a (hypothetical) algorithm that requires only one persistent memory random access would take a minimum of ~350 nanoseconds, so we are within a factor of 4 of the theoretical limit.

Is this algorithm applicable to DRAM or SSD storage?

Applicability to DRAM

When the hash table is stored entirely in DRAM, the speeds will be comparable to those shown above for a DRAM-cached memory-mapped hash table, although startup and shutdown delays are minimized for the DRAM-only version. Of course, without some persistence mechanism, e.g., battery or supercapacitor, persistence features will be unavailable.

Applicability to SSD storage

One would expect that SSD performance will not be of the same order of magnitude as with DRAM or persistent memory, as even the fastest SSD has random access times of 5-10 microseconds, at least 10x that of persistent memory.

However, it is possible to split the hash table into multiple shards that do not have to coordinate with one another very closely, thus allowing a type of multiprocessing that does not limit performance as a standard multithreading approach would. With this in mind, how much throughput can we achieve with an SSD?

Test results from a multiprocessing C++ implementation on SSD

	Variable-length key/value averaging 8 bytes each, 100% Read, 32K tranches		
	15B records, total storage 300 GB	**30B records, total storage 580 GB**	
Operation	Optane SSD, old server, 16 shards	Samsung SSD, new server, 32 shards	Dapustor SSD, new server, 32 shards
Read a billion records in original order after 4-minute warmup	1703K	6100K	6027K
Overall throughput in same order at 1800 seconds	1228K	3353K	3110k
Overall throughput in different order at 1800 seconds	129K	101K	139K

	Variable-length key/value averaging 8 bytes each, 50/50 Read/update, 32K tranches		
	15B records, total storage 300 GB	**30B records, total storage 580 GB**	
Operation	Optane SSD, old server, 16 shards	Samsung SSD, new server, 32 shards	Dapustor SSD, new server, 32 shards
Read a billion records in original order after 4-minute warmup	1309K	5695K	4193K
Overall throughput in same order at 1800 seconds	1004K	2775K	2383K
Overall throughput in different order at 1800 seconds	~108K (anomalous behavior filtered out)	95K	129K

Comments on SSD performance results

The Samsung 980 Pro can at least hold its own in most speed comparisons with the Dapustor data-center-oriented drive. Of course the Samsung does not have the endurance claimed by the Dapustor so it would not be appropriate for heavy write loads.

Large (32K) tranches provide much better performance on SSD, even for short records, than small (256B) tranches, in contrast to Optane Persistent Memory or DRAM, where small tranches are much faster.

Records up to the tranche size limit can be retrieved with exactly one random access, which is the maximum possible performance achievable with any algorithm.

Comments on SSD performance results, part 2

The most unexpected result of these tests was that accessing hash table entries in the order in which they were stored on an SSD improved retrieval/update speed by a factor of from 10x to 30x or more.

This was surprising to me because the **logical** locations where the records are stored are scattered as evenly as possible throughout the file. My error was in not taking into consideration the Flash Translation Layer, which implies that if the SSD is empty when the file is constructed, direct records will likely be stored in **physical** sequential order, resulting in sequential-level performance when rereading those records in the same order.

What about computational storage?

Applicability to computational storage

Computational storage is an ideal application for this new hash table because it is directoryless. This means that a table of any size can be accessed optimally without being limited by the size of available DRAM in the SSD or HDD computational storage controller.

The ability to read records back at millions of transactions per second when done in the same order as the records were stored may also have significant benefits for computational storage.

Notes

Notes

1. For a more detailed description of an earlier implementation of the heterogeneous hash table described here, see **US Patent #11,254,590**; other US and foreign patents pending.
2. The reason that it's impossible to be more specific about VM overhead is that there doesn't seem to be a way to specify which socket to use for vCPUs, at least on Hyper-V. Thus, the program could be running on a socket with a longer path to persistent memory, requiring a round-trip over the UPI for each persistent memory access.

Notes, *continued*

3. While Windows results are consistent with the theoretical performance hit due to running the program on the “wrong” socket, for some reason this effect isn’t reproducible in a native Ubuntu installation.
4. “One random access to the table” doesn’t mean that only one random access to memory is occurring during the lookup. With random access to very large memory maps, the TLB (translation lookaside buffer) is being repopulated for virtually every user memory access request. This can take several additional random memory accesses to page table structures with 4K pages, typical in Windows. Setting “transparent huge pages” on Ubuntu reduces this overhead by mapping larger sections of memory for each page table entry.

To learn more

You can contact me at sheller@2misses.com for documentation including more detailed performance results.

See www.2misses.com for the latest updates, including the user manual.

Please take a moment to rate this session.

Your feedback is important to us.

