CPHASH: A Cache-Partitioned Hash Table

Zviad Metreveli
Dropbox / MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

M. Frans Kaashoek
MIT CSAIL

Abstract

CPHASH is a concurrent hash table for multicore processors. CPHASH partitions its table across the caches of cores and uses message passing to transfer lookups/inserts to a partition. CPHASH's message passing avoids the need for locks, pipelines batches of asynchronous messages, and packs multiple messages into a single cache line transfer. Experiments on a 80-core machine with 2 hardware threads per core show that CPHASH has $\sim 1.6\times$ higher throughput than a hash table implemented using fine-grained locks. An analysis shows that CPHASH wins because it experiences fewer cache misses and its cache misses are less expensive, because of less contention for the on-chip interconnect and DRAM. CPSERVER, a key/value cache server using CPHASH, achieves $\sim 5\%$ higher throughput than a key/value cache server that uses a hash table with fine-grained locks, but both achieve better throughput and scalability than MEMCACHED. The throughput of CPHASH and CPSERVER also scale near-linearly with the number of cores.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming—Parallel programming

General Terms Design, Performance

1. Introduction

Hash tables are heavily used data structures in servers. This paper focuses on fixed-size hash tables that support eviction of its elements using a Least Recently Used (LRU) list. Such hash tables are a good way to implement a key/value cache. A popular distributed application that uses a key/value cache is MEMCACHED. MEMCACHED is an in-memory cache for Web applications that store data, page rendering results, and other information that can be cached and is expensive to recalculate. As the number of cores in server machines increases, it is important to understand how to design hash tables that can perform and scale well on multi-core machines.

This paper explores designing and implementing a scalable hash table by minimizing cache movement. In a multi-core processor, each core has its own cache and perhaps a few caches shared by adjacent cores. The cache-coherence protocol transfers cache lines between caches to ensure memory coherence. Fetching lines from memory or from other cores' caches is expensive, varying from one order to two order of magnitude in latency, compared to an L1 fetch. If several cores in turn acquire a lock that protects a data item, and then update the data item, the cache hardware may send several hardware messages to move the lock, the data item, and to invalidate cached copies. If the computation on a data item is small, it may be less expensive to send a software message to a core which is responsible for the data item, and to perform the computation at the responsible core. This approach will result in cache-line transfers from the source core to the destination core to transfer the software message, but no cache-line transfers for the lock, the data, and potentially fewer hardware invalidation messages.

To understand when this message-passing approach might be beneficial in the context of multicore machines, this paper introduces a new hash table, which we call CPHASH. Instead of having each core access any part of a hash table, CPHASH partitions the hash table into partitions and assign a partition to the L1/L2 cache of a particular core. CPHASH uses message passing to pass the lookup/insert operation to the core that is assigned the partition needed for that particular operation, instead of running the lookup/insert operation locally and fetching the hash table entry and the lock that protects that entry. CPHASH uses an asynchronous message passing protocol, allowing CPHASH to batch messages. Batching increases parallelism: when a server is busy, a client can continue computing and add messages to a batch. Furthermore, batching allows packing multiple messages in a single cache line, which reduce the number of cache lines transferred.

To evaluate CPHASH we implemented it on a 80-core Intel machine with 2 hardware threads per core. The implementation uses 80 hardware threads that serve hash-table operations and 80 hardware threads that issue operations. For comparison, we also implemented an optimized hash table with fine-grained locking, which we call LOCKHASH. LOCKHASH uses 160 hardware threads that perform hash-table operations on a 4,096-way partitioned hash tables to avoid lock contention. The 80 CPHASH server threads achieve $1.6\times$ higher throughput than the 160 LOCKHASH hardware threads. The better performance is because CPHASH experiences 1.5 fewer L3 caches misses per operation and the 3.1 L3 misses that CPHASH experiences are less expensive. This is because CPHASH has no locks and has better locality, which reduce the contention for the interconnect and DRAM. CPHASH's design also allows it to scale near-linearly to more cores than LOCKHASH.

The follow sections summarize the design and provide one key performance result. For the interested reader, our technical report [1] provides more detail on CPHASH, as well as a detailed breakdown of its performance.

2. CPHASH Design

CPHASH splits a hash table into several independent parts, which we call *partitions*. CPHASH uses a simple hash function to assign each possible key to a partition. Each partition has a designated server thread that is responsible for all operations on keys that belong to it. CPHASH pins each server thread to its hardware thread. Applications use CPHASH by having client threads send operations to server threads using message passing (via shared memory). Server threads return results to the client threads also using message passing.

2.1 Partitions

Every partition in CPHASH is a separate hash table. Each partition consists of a bucket array, where each bucket is a linked list of hash table elements. Each partition also has an LRU linked list that holds hash table elements in the least recently used order. CPHASH uses the LRU list to determine which elements to evict from a partition when there is not enough space left to insert new elements.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

Each hash table element consists of two parts: a header, which fits in a single cache line and is typically stored in the server thread's cache, and the value, which fits in zero or more cache lines following the header, and is directly accessed by client threads, thereby loading it into client thread caches. The header consists of the *key*, the *reference count*, the *size* of the value (in bytes), and doubly-linked-list pointers for the bucket and for the LRU list to allow eviction.

The ideal size for a partition is such that a partition can fit in the L1/L2 cache of a core, with some overflow into its shared L3 cache. On our test machine with 80 cores, hash table sizes up to about $80 \times 256\text{KB} + 8 \times 30\text{MB} = 260\text{MB}$ see the best performance improvement, at which point CPHASH starts being limited by DRAM performance.

2.2 Server Threads

CPHASH server threads support two types of operations: `Lookup` and `Insert`. In the case of a `Lookup`, the message contains the requested `key`. If a key/value pair with the given `key` is found in the partition, then the server thread updates the head of the partition's LRU list and return the *pointer* to the value to the client thread; otherwise, the server returns a null pointer.

Performing an `Insert` operation is slightly more complicated, because memory must be allocated for the value, and the value must be copied into the allocated memory. It is convenient to allocate memory in the server thread, since each server is responsible for a single partition, and can use a standard memory allocator. However, copying the actual data is performed in the client thread, to avoid polluting the cache of the server core.

CPHASH uses reference counting to keep track of outstanding pointers to hash table elements. To drop a reference, clients send a message to the corresponding server core.

2.3 Message passing

CPHASH implements message passing between the client and server threads using pre-allocated circular buffers in shared memory. For each client and server pair there are two circular rings of buffers—one for each direction of communication—along with pointers to the first pending and last pending buffer in the ring. Each buffer can hold several messages. Message senders locate the next available buffer, and append as many messages to it as possible up before advancing the last pending pointer. Receivers poll for pending buffers, and process all messages in a buffer before moving on.

Buffers are a multiple of the cache line size, and each cache line can hold several messages. This allows a single cache line transfer to transmit many messages, thus enabling an amortized message passing cost of less than one cache miss per message. Pending indexes are cache-aligned to avoid false sharing. A ring of buffers allows asynchronous message passing, so that the sender can perform other tasks while waiting for the response.

3. Performance Evaluation

In this section we discuss the performance results that we achieved using CPHASH, and compare it to the performance achieved by LOCKHASH. To evaluate hash table performance, we created a simple benchmark that generates random queries and performs them on the hash table. A single query can be either a LOOKUP or an INSERT operation. The INSERT operation consists of inserting key/value pairs such that the key is a random 64-bit number and the value is the same as the key (8 bytes).

We use an 80-core Intel machine for our evaluation. This machine has eight sockets, each containing a 10-core Intel E7-8870 processor. All processors are clocked at 2.4 GHz, have a 256 KB L2 cache per core, and a 30 MB L3 cache shared by all 10 cores in a single socket. Each of the cores supports two hardware threads (Hyperthreading in Intel terminology). Each socket has two DRAM controllers, and each controller is connected to two 8 GB DDR3 1333 MHz DIMMs, for a total of 256 GB of DRAM.

To evaluate the overall performance of CPHASH relative to its locking counterpart, LOCKHASH, we measure the throughput of both hash tables over a range of working set sizes. Clients issue a mix of 30% INSERT and 70% LOOKUP queries. The maximum hash table size is equal to the entire working set, which means no eviction takes place. We run 10^9 queries for each configuration, and report the throughput achieved during that run.

For CPHASH, we use 80 client threads, 80 partitions, and 80 server threads. The client and server threads run on the first and second hardware threads of each of the 80 cores, respectively. This allows server threads to use the L2 cache space of each core, since client threads have a relatively small working set size. Each client maintains a pipeline of 1,000 outstanding requests across all servers; similar throughput is observed for batch sizes between 512 and 8,192. Larger batch sizes overflow queues between client and server threads, and smaller batch sizes lead to client threads waiting for server replies.

Figure 1 shows the results of this experiment. For small working set sizes, LOCKHASH performs poorly because the number of distinct keys is less than the number of partitions (4,096), leading to lock contention. In the middle of the working set range (256 KB–128 MB), CPHASH consistently out-performs LOCKHASH by a factor of $1.6\times$ to $2\times$. With working sizes of 256 MB or greater, the size of the hash table exceeds the aggregate capacity of all CPU caches, and the performance of CPHASH starts to degrade as the CPUs are forced to incur slower DRAM access costs. At large working sets, such as 4 GB to the right of the graph, the performance of both CPHASH and LOCKHASH converges and is limited by DRAM.

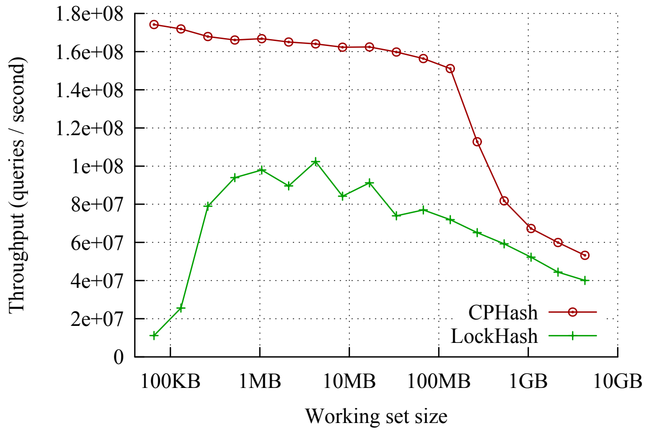

Figure 1. Throughput of CPHASH and LOCKHASH over a range of working set sizes.

Acknowledgments

We thank Robert Morris for helping us improve the ideas in this paper. This work was partially supported by Quanta Computer and by NSF award 915164.

References

[1] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: A cache-partitioned hash table. Technical Report MIT-CSAIL-TR-2011-051, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, November 2011.