CPHASH: A Cache-Partitioned Hash Table

Zviad Metreveli, Nickolai Zeldovich, and 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. Finally, the throughput of CPHASH and CPSERVER scales near-linearly with the number of cores.

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 [16]. 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 and the 3 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.

To understand the value of CPHASH in an application, this paper introduces a MEMCACHED-style key/value cache server, which we call CPSERVER, which uses CPHASH as its hash table. We compare the performance of CPSERVER to the performance of a key/value cache using LOCKHASH, and against MEMCACHED. We observe that the servers based on CPHASH and LOCKHASH scale and perform better than MEMCACHED. In our servers, the hash table lookup is only 30% of the total computation that the server performs to process a request arriving over a TCP connection. CPHASH reduces that by 17% compared to LOCKHASH.

The main contributions of the paper as follows: 1) CPHASH, a scalable, high-performance concurrent hash table that reduce cache-line movement by partitioning the hash table across L1/L2 caches of cores and uses message passing to perform inserts and lookups on a partition; 2) a detailed experimental evaluation (in terms of cache lines moved) explaining under what conditions and why CPHASH can perform better; and 3) a demonstration of CPHASH in a MEMCACHED-style application, including a performance comparison with MEMCACHED.

The remainder of this paper is structured as follows. Section 2 describes the related work. Section 3 describes CPHASH's design. Section 4 describes design and protocol of CPSERVER. Section 5 details our implementation of CPHASH. Section 6 evaluates CPHASH and a fine-grained locking hash table on microbenchmarks. Section 7 evaluates the scalability and performance of CPHASH in an application and compares its performance to MEMCACHED. In Section 8 we discuss future plans for CPHASH. Section 9 summarizes our conclusions.

2 RELATED WORK

CPHASH has a unique design in that it is a shared-memory hash table that uses message passing to reduce cache-coherence traffic. There are many concurrent shared-memory designs for hash tables, including ones that are based on RCU, yet allow for dynamic resiz-

ing [25]. There are also many distributed designs such as Chord [21] or MEMCACHED [16] based on message passing. MEMCACHED clients often choose one of several partitioned MEMCACHED servers to insert or look up a key, based on the key's hash, and the same idea of partitioning MEMCACHED has been applied within a single machine to avoid contention on the single lock protecting MEMCACHED's state [3]. We are unaware, however, of a design of a hash table design that partitions the table across the L1/L2 caches of cores and uses software message passing on a cache-coherent shared-memory multicore machine to reduce hardware cache-coherence traffic, combined with an evaluation investigating when this approach works.

Flat combining [13] has some of the same advantages as message passing through shared memory. Cores post operations to a publication list and the holder of the global lock processes all the posted operations. This way when multiple threads are competing for the global lock only one of them has to acquire it; others can just schedule their operation and wait for the result. CPHASH achieves similar properties by using message passing. Furthermore, it can exploit asynchronous message passing to increase concurrency, and has no notion of a global lock. Finally, flat combining hasn't been applied to hash tables.

Many different techniques to optimize use of caches on multi-core chips exist. Thread clustering [24] dynamically clusters threads with their data on to a core and its associated cache. Chen et al. [11] investigate two schedulers that attempt to schedule threads that share a working set on the same core so that they share the core's cache and reduce DRAM references. Several researchers have used page coloring to attempt to partition on-chip caches between simultaneous executing applications [12, 15, 20, 23, 27]. Chakraborty et al. [9] propose computation spreading, which uses hardware-based migration to execute chunks of code from different threads on the same core to reduce i-cache misses. These techniques migrate threads or chunks of code, while CPHASH uses message passing to invoke remote operations and to reduce cache-coherence traffic.

CPHASH attempts to execute computation close to data so that the coherence protocol doesn't have to move the data, and was inspired by computation migration in distributed shared memory systems such as MCRL [14] and Olden [8], remote method invocation in parallel programming languages such as Cool [10] and Orca [2], and the O^2 scheduler [6]. CPHASH isn't as general as these computation-migration systems, but applies the idea to a single data structure that is widely used in server applications and investigates whether this approach works for multicore processors.

Several researchers place OS services on particular cores and invoke them with messages. Corey [5] can dedicate a core to handling a particular network device and its associated data structures. Mogul et al. optimize some cores for energy-efficient execution of OS code [17]. Suleman et al. put critical sections on fast cores [22]. Barrelfish [19] and fos [26] treat cores as independent nodes that communicate using message passing. CPHASH's message passing implementation is more similar in approach to lightweight RPC on the Firefly multiprocessor [4] than Barrelfish's implementation on Intel's SCC [18], which is not cache-coherent. At a higher level, CPHASH is a specific example of combining advantages of shared-memory and message passing, within a single data structure. From this perspective, the main contribution of the paper is the details of the CPHASH design and when this approach works when applied to hash tables.

3 CPHASH DESIGN

Figure 1 provides a top-level view of CPHASH's design. CPHASH splits a hash table into several independent parts, which we call

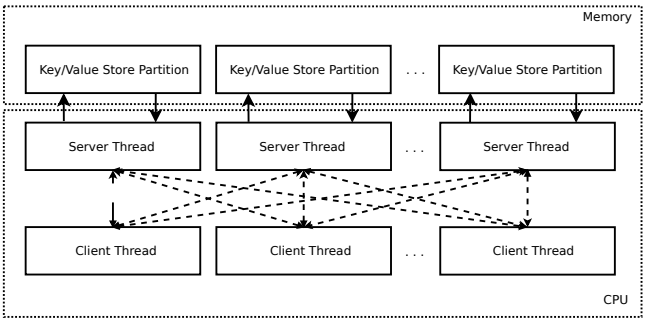

Figure 1: Overview of the CPHASH design.

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 pin each server thread to its hardware thread. Applications use CPHASH by having client threads that communicate with the server threads and send operations using message passing (via shared memory). Server threads return results to the client threads also using message passing.

The main goal of CPHASH's design is to minimize cache-line transfers to the ones necessary for sending/receiving messages, and for delivering the data needed by the application. The rest of the section describes how CPHASH achieves this goal in detail. Section 3.1 describes a partition. Sections 3.2 and 3.3 describe server and client threads, respectively. Section 3.4 describes the message-passing design, including batching and packing.

3.1 Partitions

CPHASH has one partition for each hardware thread that runs a server thread. Every partition in CPHASH is a separate hash table. Figure 2 shows the partition data structure. Each partition contains a bucket array. Each bucket is a linked list. Keys are placed into different buckets based on a hash function that maps a key to a specific bucket. Each partition also has an LRU linked list that holds 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. Because only a single server uses a partition, no locks are necessary to protect the partition's hash table, the buckets in that table, and the partition's LRU list. Furthermore, the partition data structures will stay in the local cache, because they are read and written from only one core.

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. CPHASH's current implementation limits keys to 60-bit integer numbers; however, this can easily be extended to support any key size. In CPHASH all partitions are of equal size for simplicity. If needed, partitions can be implemented to have different sizes using more advanced memory management and data eviction algorithms. Section 8 discusses both extensions.

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.

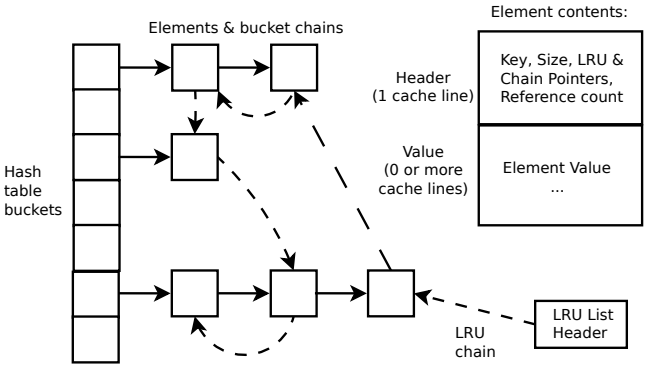

Figure 2: Partition data structures in CPHASH.

3.2 Server Threads

Each server thread performs the operations for its partition. The server thread continuously loops over the message queues of each client checking for new requests. When a request arrives, the server thread performs the requested operation and sends its result back to the client.

CPHASH supports 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. CPHASH is non-intrusive and supports arbitrary length values; thus, for an insert operation 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 so CPHASH can use a standard single-threaded memory allocator. However, performing the actual data copying in the server thread is a bad design since for large values it wipes out the local hardware cache of the server core. Thus, in CPHASH the space allocation is done in the server thread and the actual data copying is performed in the client thread.

Thus, to perform an `Insert` operation, the server must receive the `key` and the `size` of the data. The server thread allocates `size` bytes of memory and removes the existing key/value pair with the given `key` (if it exists) from the partition, to avoid having duplicate keys for two different elements. The allocated space is marked as "NOT READY" and will not be used until it is marked as "READY". The client receives the pointer, copies the data to the location pointed by the given pointer, and marks that space as "READY", by sending a special *Ready* message to the server.

When a server thread evicts or deletes an element from the hash table, the memory allocated for the evicted value must be freed so that it can be reused for new elements. It is incorrect for a server thread to just free the allocated memory when it evicts or deletes an element. The problem is that if a client requests a `Lookup` for some element and receives the pointer to its value, and then the server threads evicts the element from the hash table before the client is done reading its value, the client will have a dangling pointer pointing to memory that might have been reallocated to a new value. To resolve this issue, CPHASH counts references to an element. Each element in the hash table has a reference count. Every time a client requests a `Lookup` of an element, the server thread increases the element's reference count. When the client is done with the item, it decreases the reference count of the element, by sending a `Decref` message to the server. Once the reference count reaches zero, the server thread frees the element's memory.

3.3 Client Threads

Applications have client threads that communicate with the server threads to perform operations. An example of a client thread in an application is the client thread in CPSERVER's implementation. The client threads in CPSERVER gather operations from TCP connections, route them to the appropriate server threads, gather the results, and send them back to the correct TCP connections. Section 4 describes CPSERVER's implementation in more detail. Client threads do not necessarily have to be pinned to a specific hardware thread but, to achieve the highest performance in message passing, it is best to keep the client threads attached to a specific hardware thread.

3.4 Message passing

The design so far arranges that client and server threads can mostly run out of their local caches. The main remaining challenge is a design for messaging passing that minimizes cache-line transfers. CPHASH addresses this challenges in two ways: 1) the design introduces few cache-line transfers per message and 2) the design attempts to hide the cost of these few transfers by allowing client threads to overlap computation with communication to server threads. As a result, as we will see in Section 6, CPHASH incurs about 1.5 cache misses, on average, to send and receive two messages per operation.

CPHASH implements message passing between the client and server threads using pre-allocated circular buffers in shared memory. For each server and client pair there are two arrays of buffers—one for each direction of communication. Another possible way to implement message passing is to use a single buffer per client/server pair, which our original design did; this is more efficient but it allows for less concurrency. Figure 3 gives graphical representation for both designs.

In the single buffer implementation, space is allocated for each client/server pair and when a client want to send a request to a server, it writes the message to the buffer and waits for the server to respond. When the server is done processing the message, it updates the shared location with the result.

The implementation of an array of buffers consists of the following: a data buffer array, a read index, a write index, and a temporary write index. When the producer wants to add data to the buffer, it first makes sure that the read index is large enough compared to the temporary write index so that no unread data will be overwritten. Then it writes data to buffer and updates the temporary write index. When the temporary write index is sufficiently larger than the write index, the producer flushes the buffer by changing the write index to the temporary write index.

To read data, the consumer waits until the read index is less than the write index, then it proceeds to read the buffered data and update the read index. The read index, write index and temporary write index are aligned in memory to avoid false sharing. To decrease the number of cache misses when reading or writing buffers, the client threads flush the buffer when the whole cache line is full and the server threads update the read index after they are done reading all the operations in a cache line. In the common case, the temporary write index lives in the producer's cache, the read index and write index are shared by the producer and consumer (updated whenever an entire cache line worth of messages is produced or consumed), and the cache lines comprising the buffer array are also transferred from producer to consumer.

There are two major benefits to using arrays of buffers instead of single buffers. The first advantage is improved parallelism. With an

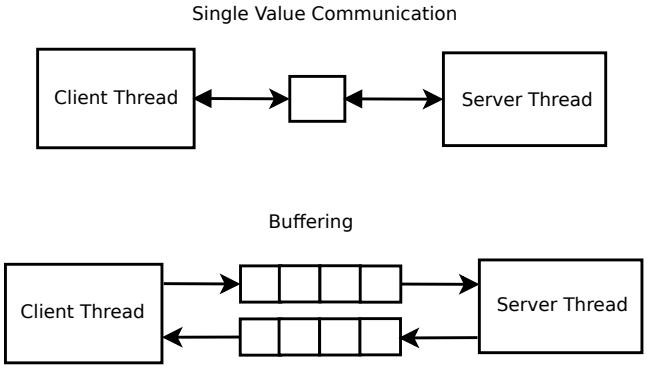

Figure 3: Two message-passing designs.

array of buffers, the client can just queue the requests to the servers; thus, even if the server is busy, the client can continue working and schedule operations for other servers. This way all the servers can stay busy most of the time, allowing applications to achieve better performance.

The second reason is decreased overhead for message passing. With a single buffer, for every message received, the server experiences a cache miss; however, since a cache line can hold several messages (in our test machines a cache line is 64 bytes), with batching and packing the server can receive several messages using only a singe cache miss.

There are some downsides to using arrays of buffers instead of the single buffers. The array implementation requires having extra indices to enable the server and the client to know how much data has been written and how much has been read. Maintaining these indices introduces extra performance overhead that a single buffer does not have. Thus, if the client sends requests to the server at a slow rate, a single buffer outperforms the array implementation. However, if the client has a batch of requests that it needs to complete, batching will be an advantage. Our target applications are bottlenecked by the performance of the hash table, and have no shortage of requests; therefore, buffering is a better design choice for message passing.

4 KEY/VALUE CACHE SERVERS

To demonstrate the benefits of CPHASH in an application, we developed CPSERVER, a MEMCACHED-style Key/Value Cache Server, which uses CPHASH to implement its hash table. For comparison, we also implemented a version of this server in a shared-memory style using fine-grained locking. Section 6 compares these two implementations with MEMCACHED.

4.1 CPSERVER

Figure 4 shows the design of CPSERVER. CPSERVER has server and client threads as described in Section 3. TCP clients on client machines connect to CPSERVER using TCP connections. A client thread monitors TCP connections assigned to it and gathers as many requests as possible to perform them in a single batch. Then, a client thread passes the batch of requests to the appropriate server threads using message passing. After the server thread is done and the client thread receives the results, the client thread writes back those results to the appropriate TCP connections.

The CPSERVER also has an additional thread that accepts new connections. When a connection is made, it is assigned to a client thread with the smallest number of current active connections. The load balancer could be more advanced for work loads in which the traffic on different connections differ significantly.

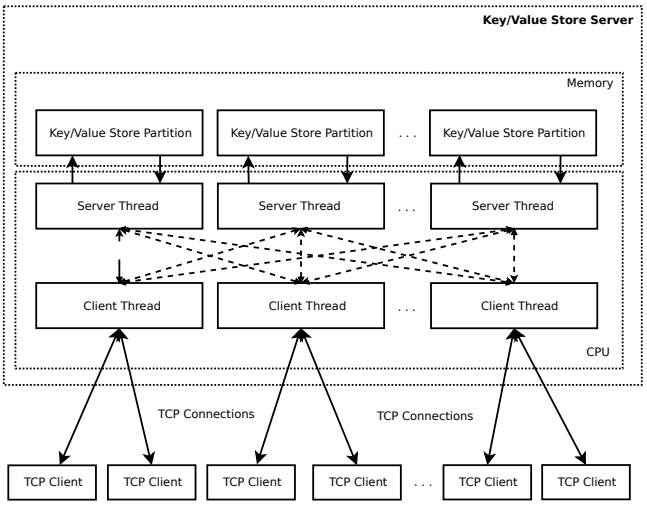

Figure 4: Overview of the CPSERVER design.

CPSERVER uses a simple binary protocol with two message types:

LOOKUP With the LOOKUP request the TCP client asks the server to try to find a key/value pair in the hash table such that the key matches the `hash key` field from the request. If the requested key/value pair is found in the hash table, then the server returns the size of the value, along that many bytes containing the actual value. Otherwise, the server returns a response with a `size` field of zero.

INSERT With the INSERT request the TCP client asks the server to insert a new key/value pair in the hash table. The `hash key` field is the key to be inserted. The `size` field is the size of the value to be inserted in the hash table. The INSERT request header is followed by `size` number of bytes which contain the value to be inserted. The server silently performs INSERT requests and returns no response.

4.2 LOCKSERVER

To evaluate the performance and scalability of CPHASH, we created LOCKSERVER, which does not use message passing. It supports the same protocol, but uses a shared-memory style hash table, which we name LOCKHASH, with fine-grained locks. To make the comparison fair, LOCKHASH also has n LRU lists instead of 1 global one, by dividing the hash table into n partitions. Each partition is protected by a lock to protect the LRU list for that partition. The client threads of LOCKSERVER process queries by first acquiring the lock for the appropriate partition, then performing the query, updating the LRU list and, finally, releasing the lock. LOCKSERVER also supports a random eviction policy, in which case it acquires a per-bucket lock instead of a per-partition lock to perform all operations. Overall, LOCKHASH and LOCKSERVER are also highly optimized to minimize the amount of cache lines transferred, subject to the design constraints of the lock-based approach.

5 IMPLEMENTATION

We have implemented CPHASH, LOCKHASH, CPSERVER, and LOCKSERVER in a common C/C++ framework on Linux. Both of the hash tables implement the same API, allowing the same benchmark or TCP server to drive either hash table. Furthermore, both CPHASH and LOCKHASH use the same code for implementing a single hash table partition; the only difference is that LOCKHASH

acquires a lock to perform an operation on a partition, and CPHASH uses message-passing to send the request to the appropriate server thread. The total implementation is approximately 4,400 lines of code.

To precisely evaluate performance, we developed a 500-line C++ profiling library that uses `rdtsc` and `rdpmc` to access the hardware timestamp counter and performance counters. To access performance counters from user-space, we use a 300-line Linux kernel module.

To avoid bottlenecks in the Linux TCP stack and network driver from limiting the throughput of our TCP servers and clients, we use Intel's ixgbe Linux driver, and route its interrupts to 64 different cores, along with other patches described by Boyd-Wickizer et al [7].

6 PERFORMANCE EVALUATION

In this section we discuss the performance results that we achieved using CPHASH, and compare it to the performance achieved by the LOCKHASH design. 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). The benchmark is configured using several parameters:

- Number of client hardware threads that are issuing queries.
- Number of partitions (which, for CPHASH, is also the number of server hardware threads).
- Working set size of queries issued by clients, in bytes (i.e., amount of memory required to store all values inserted by clients). This corresponds to the number of distinct keys used by clients.
- Maximum hash table size in bytes (meaningful values range from $0\times$ to $1\times$ the working set size).
- Ratio of INSERT queries.
- Size of batch.

We first evaluate the overall performance of CPHASH under several representative workloads, then explore in detail the reason for the performance differences between CPHASH and LOCKHASH, and finally study how other parameters affect the performance of CPHASH relative to LOCKHASH.

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.

Although we report results from an Intel system, we have also evaluated CPHASH and LOCKHASH on a 48-core AMD system (also consisting of eight sockets). The performance results on the AMD system are similar, but we focus on the Intel results in this paper both for space reasons, and because the Intel machine has more cores and hardware threads.

6.1 Overall performance

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. Other parameters are fixed constants for this experiment, as follows. Clients issue a mix of 30% INSERT and 70% LOOKUP queries. The maximum hash table size is equal to the entire working set. 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.

For LOCKHASH, we run 160 client threads, one on each of the 160 hardware threads. We use 4,096 partitions to avoid lock contention, which we experimentally determined to be optimal: a smaller number of partitions incurs lower throughput due to lock contention, and a larger number of partitions does not increase throughput.

Figure 5 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.

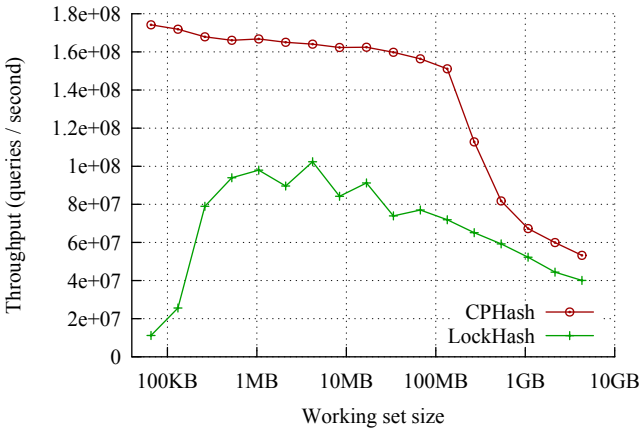

Figure 5: Throughput of CPHASH and LOCKHASH over a range of working set sizes.

6.2 Detailed breakdown

To understand the precise reasons why CPHASH is faster than LOCKHASH, we now examine where time is spent in both hash tables under the 1 MB working set configuration from the previous section.

With CPHASH, a single operation takes an average of 1,126 clock cycles, as measured on the client thread (including time spent waiting for the server thread). CPHASH's server threads spend 59% of the time processing INSERT and LOOKUP operations; the

rest of the time is spent polling idle buffers. A single INSERT or LOOKUP operation typically translates into two messages to a server thread: one message to allocate space or look up a key, and a second message to release the reference count on the result (if found). Server threads spend an average of 336 clock cycles handling each message. However, even though the server threads are idle for 41% of the time, reducing the number of server threads leads to a reduction in throughput for two reasons: as server thread utilization goes up, the average wait time for clients increases, and the L2 caches on cores with two client threads (as opposed to one client and one server thread) are underutilized.

With LOCKHASH, a single operation takes an average of 3,664 clock cycles to execute (note that, while this time is 3.25× the CPHASH operation latency, LOCKHASH has twice as many client threads issuing operations, resulting in 1.63× lower throughput). Even though LOCKHASH is slower than CPHASH, it is still highly optimized, and all of the operations and cache misses it incurs are necessary by its design.

The increased latency of operations under LOCKHASH is due to two factors. First, LOCKHASH performs more memory accesses that miss in L2 and L3 caches (due to shared data structures), and second, the memory misses incurred by LOCKHASH are more expensive than those incurred by CPHASH (due to higher contention on the memory system). Figure 6 summarizes the cache misses observed on average for one operation under both CPHASH and LOCKHASH, and Figure 7 further breaks down the causes for the cache misses. The overall latency of an operation under LOCKHASH is less than the sum of cache miss latencies due to out-of-order execution and pipelining.

	CPHASH client	CPHASH server	LOCKHASH
Cycles per op.	1,126 cycles	672 cycles	3,664 cycles
# of L2 misses	1.0 misses	2.5 misses	2.4 misses
L2 miss cost	64 cycles		170 cycles
# of L3 misses	1.9 misses	1.2 misses	4.6 misses
L3 miss cost	381 cycles		1,421 cycles

Figure 6: Performance of a single hash table operation under CPHASH and LOCKHASH, including the total cycles spent per operation, the number of cache misses incurred for each operation, and the average latency for each such miss. "L2 miss" indicates memory accesses that missed in the local L2 cache, but hit in the shared L3 cache or a neighbor's L2 cache on the same socket. "L3 miss" indicates memory accesses that missed in the local L3 cache, and went to DRAM or another socket.

Design	Function	L2 misses	L3 misses
LOCKHASH	Spinlock acquire	0.1	0.9
	Hash table traversal	2.0	2.4
	Hash table insert	0.4	1.2
	Total	*2.4*	*4.6*
CPHASH client thread	Send messages	0.1	0.5
	Receive responses	0.5	0.4
	Access data	0.4	0.9
	Total	*1.0*	*1.9*
CPHASH server thread	Receive messages	1.4	0.4
	Send responses	0.4	0.4
	Execute message	0.7	0.4
	Total	*2.5*	*1.2*

Figure 7: Detailed breakdown of cache misses in CPHASH and LOCKHASH for an average operation, with a 1 MB working set size, 1 MB hash table capacity, LRU eviction, and an INSERT ratio of 0.3. Every operation performs a hash table traversal, and an average operation performs 0.3 hash table inserts.

As can be seen in the detailed cache miss breakdown in Figure 7, CPHASH incurs about 1.5 cache misses, on average, to send and receive two messages per operation. This is achieved through batching: CPHASH can place eight lookup messages (consisting of an 8-byte key), or four insert messages (consisting of an 8-byte key and an 8-byte value pointer) into a single 64-byte cache line. CPHASH also incurs additional cache misses to access the data (either to read it for lookup, or to modify it for insert), since CPHASH servers return pointers to values rather than copying the data directly.

LOCKHASH spends the bulk of its cache misses on hash table traversal, even though the hash table is configured to store an average of one element per bucket. A hash table that was more dense (i.e., had more elements per bucket on average) would see more significant performance improvements with CPHASH over LOCKHASH.

In LOCKHASH, each operation incurs the cost of acquiring the lock. CPHASH's idea of batching could, in principle, also be applied to a lock-based design, by batching multiple operations into a single lock acquisition and LRU update. However, LOCKHASH has a larger number of partitions (4,096) compared to CPHASH (80), in order to avoid partition lock contention. As a result, effective batching for LOCKHASH would require a larger number of concurrent operations, in order to accumulate multiple operations for each of the 4,096 partitions. On the other hand, CPHASH can batch requests to just 80 distinct server threads. Batched locking would also not be able to achieve the locality of accesses to hash table structures observed in CPHASH.

LOCKHASH uses a spinlock to protect each hash table partition from concurrent access. Although the spinlock is not scalable, it performs better than a scalable lock. For example, Anderson's scalable lock [1] requires a constant two cache misses to acquire the lock, and one more cache miss to release. In contrast, an uncontended spinlock requires one cache miss to acquire and no cache misses to release. Contended spinlocks can become expensive, but even contended scalable locks need to be avoided (e.g., by increasing the number of partitions) to maintain good performance.

6.3 Effect of LRU

As described above, one of the reasons why CPHASH wins over LOCKHASH is due to the fact that CPHASH servers can keep LRU pointers stored in their local cache, as opposed to LOCKHASH which must incur cache misses to update LRU pointers. To evaluate the importance of LRU to CPHASH, Figure 8 reports the throughput of CPHASH and LOCKHASH for the same configuration and range of working set sizes as before, except with a random eviction policy instead of LRU (this configuration also avoids maintaining any LRU data structures). As can be seen from the figure, the relative performance benefit of CPHASH is less, but still significant (1.45× at 4 MB). With a random eviction policy, CPHASH client threads incur the same number of cache misses per operation, but LOCKHASH threads incur fewer misses (a total of 1.2 L2 misses and 3.6 L3 misses at 1 MB), which accounts for the performance improvement of LOCKHASH. CPHASH server threads also incur fewer misses due to not having to maintain LRU pointers, but this does not affect CPHASH throughput because it was not bottlenecked by server threads.

6.4 Effect of eviction and write ratio

To understand how CPHASH performs with a different mix of queries, this sub-section explores the effect of eviction (i.e., a working set that exceeds the capacity of the hash table), and the effect of the fraction of INSERT queries on the performance of CPHASH and LOCKHASH.

Figure 9 illustrates the throughput for a range of total hash table capacities for a workload with a 128 MB working set size (using

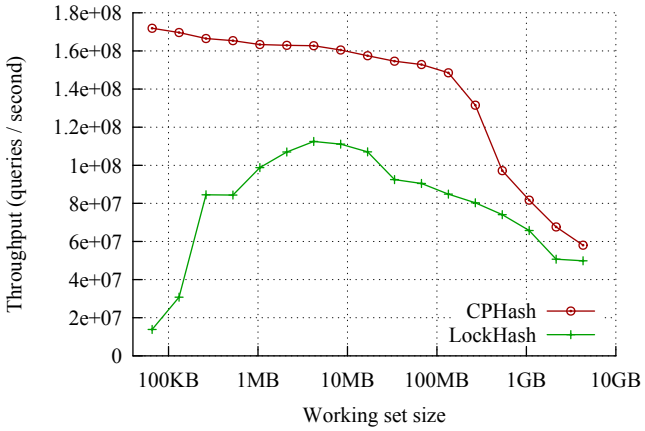

Figure 8: Throughput of CPHASH and LOCKHASH over a range of working set sizes with a random eviction policy instead of LRU.

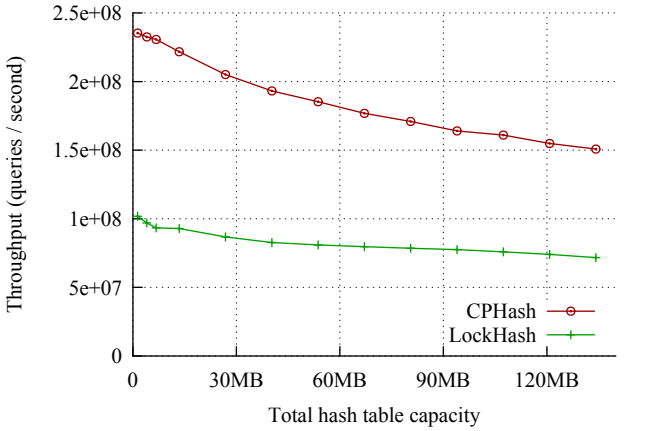

Figure 9: Throughput of CPHASH and LOCKHASH over a range of hash table capacities, for a 128 MB working set size.

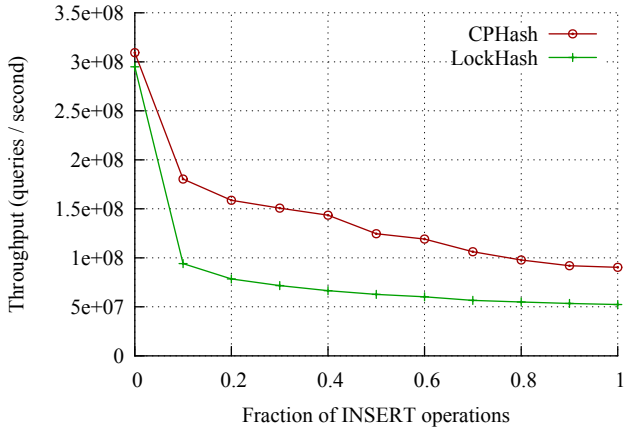

Figure 10: Throughput of CPHASH and LOCKHASH over a range of fractions of INSERT operations, for a 128 MB working set size and hash table capacity.

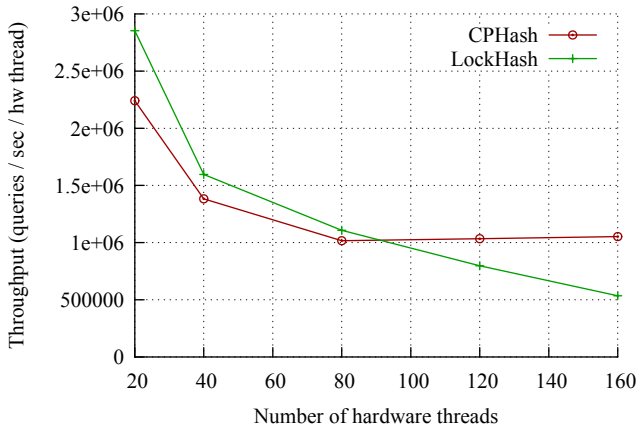

Figure 11: Throughput of CPHASH and LOCKHASH on a varying number of hardware threads.

LRU and a 0.3 ratio of INSERT operations). The throughput of both hash table implementations goes up as the total hash table capacity is reduced, because a larger fraction of the hash table fits in the processor caches, and a larger fraction of LOOKUP responses are null. However, CPHASH maintains a consistent throughput advantage over LOCKHASH for all hash table capacities.

Figure 10 illustrates the throughput for a range of ratios of INSERT operations, using a 128 MB working set size, 128 MB hash table capacity, and LRU eviction policy. A higher fraction of INSERT operations reduces throughput, because INSERT operations are more expensive to process: they require first looking up the given key in a hash table, evicting it if already present, and then inserting a new element. In the degenerate case of zero INSERT operations, the hash table is noticeably faster because the hash table remains empty and none of the LOOKUP operations succeed. This illustrates that CPHASH's performance advantage is not sensitive to the ratio of INSERT requests.

6.5 Effect of additional cores and threads

To understand how well our hash table designs scale with the number of cores, we measured their throughput when running with different numbers of cores on our experimental machine. Since our machine is not fully symmetric (e.g., some cores share a single L3 cache), we only varied the number of sockets: either all of the cores and hardware threads in a given socket were in use, or none of them were. We configured our benchmark to use a 1 MB working set size, 1 MB hash table capacity, 30% INSERT fraction, and LRU eviction scheme.

Figure 11 shows the per-socket throughput of CPHASH and LOCKHASH for varying numbers of sockets (each of which has 20 hardware threads). On the left side of the graph, both CPHASH and LOCKHASH perform better on a single socket than on a multi-socket configuration. This is because cache misses within a single socket are less expensive. As the number of cores increases past one socket, the throughput of LOCKHASH degrades, because cache misses incurred by LOCKHASH become increasingly more expensive. On the other hand, CPHASH scales better with the number of cores past one socket, since it incurs fewer expensive cache misses between cores. Infact, CPHASH scales near-linearly with the number of cores past one socket.

On smaller numbers of cores, CPHASH is slightly slower than LOCKHASH—since the aggregate capacity of caches across all cores is not significantly greater than the cache capacity accessible from a single core—but its throughput advantage grows with the number of cores.

Finally, all of the benchmarks so far have exploited both of the hyperthreads on each core of our experimental machine. To understand the extent to which our two different hash tables benefit from hardware multi-threading, we measure their throughput in three different

configurations. The first is using both hardware threads in each of the 80 cores, in 8 sockets. The second is using both hardware threads in just 40 of the cores, in 4 sockets. The third is using only one of the hardware threads in each of the 80 cores, in 8 sockets. Figure 12 summarizes the results of this experiment, for a 1 MB working set workload with 1 MB hash table capacity and LRU eviction.

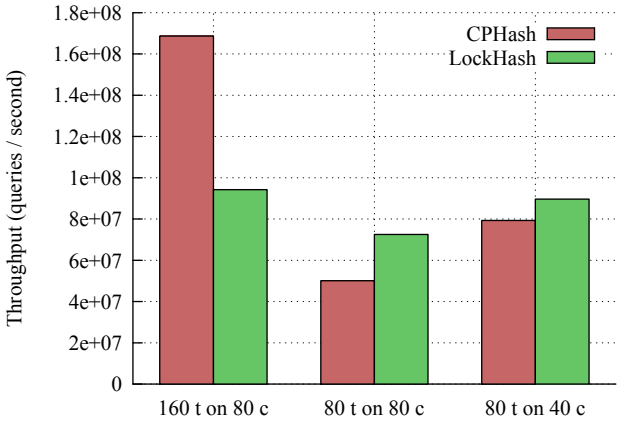

Figure 12: Throughput of CPHASH and LOCKHASH in three different configurations of hardware threads (t) and cores (c).

As can be seen from this experiment, both LOCKHASH and CPHASH perform *better* when using hardware threads that are sharing a core, on a smaller number of sockets, and *worse* when using hardware threads each with a dedicated core, on a larger number of sockets. The reason is that both hash table implementations are sensitive to the cost of transferring cache lines between hardware threads, and having the hardware threads run within fewer sockets (where each socket has a shared L3 cache) reduces this latency.

In the 160 thread / 80 core configuration, LOCKHASH gains little additional throughput compared to the 80 thread / 80 core configuration, because it is already bottlenecked by the socket's aggregate memory access throughput. On the other hand, CPHASH's fewer cache misses allow it to take advantage of the additional hardware threads without being bottlenecked by the socket's memory subsystem.

6.6 Discussion

The experimental results show that CPHASH is scalable and provides increased throughput, especially when the hash table meta data fits in the server cores' combined hardware caches. When the hash table meta data is much larger than the combined caches, CPHASH still provides benefits through batching and caching the common partition data (such as an LRU list). Furthermore, the results show that the benefits are attributable to CPHASH's design: there are no cache line transfers per operation for locks, and CPHASH avoids additional cache line transfers because it has better locality; the core that processes the request has all the metadata in its local cache.

As the number of cores grows in future processors, we expect the aggregate cache space across all of the cores to similarly increase, thereby extending the range of working set sizes for which CPHASH provides a performance improvement over LOCKHASH. Furthermore, in future processors, the number of DRAM interfaces per core will fewer, because more cores will be packed on a single core, and thus taking advantage of local caches will be even more important.

7 APPLICATION PERFORMANCE

To evaluate CPHASH in an application, we measure the performance of CPHASH and LOCKHASH in a simple key-value cache server with a TCP interface, which we call CPSERVER and LOCKSERVER respectively. This server spends about 30% of its time performing hash table operations (for most workload parameters), and spends the other 70% of the time receiving queries and sending responses over TCP. Thus, if CPHASH improved the performance of hash table operations by 1.6× (as we saw in the microbenchmarks from the previous section), this would translate into a 11% performance improvement for the overall application.

To benchmark CPSERVER and LOCKSERVER, we use a 48-core AMD machine to generate load by issuing queries over a TCP connection. The server runs on the same Intel machine as above, and the two machines are connected with a 10 Gbps Ethernet link. In all experiments, the server machines (and not the client machines) are the bottleneck.

Figure 13 shows the throughput of CPSERVER and LOCKSERVER for a range of working set sizes (with a matching hash table capacity, LRU eviction, and 30% INSERT rate). CPSERVER is about 5% faster than LOCKSERVER (out of a maximum possible 11%). CPSERVER does not achieve an 11% speedup because TCP connection processing increases cache pressure and interconnect traffic, making cache misses for CPHASH more frequent and more expensive.

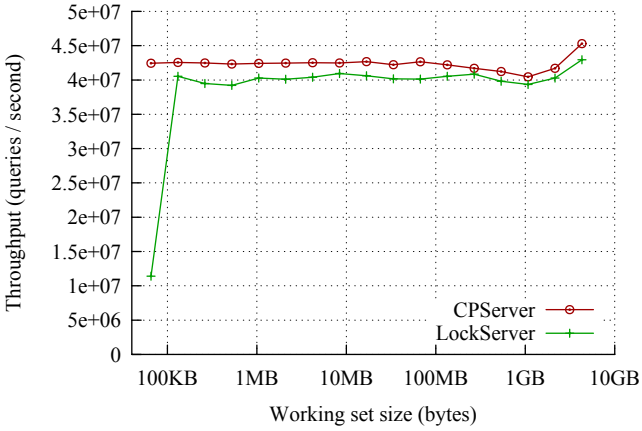

Figure 13: Throughput of CPSERVER and LOCKSERVER over a range of working set sizes.

To ensure that CPSERVER and LOCKSERVER achieve competitive absolute performance, we compare their throughput with that of MEMCACHED. Since MEMCACHED uses a single lock to protect its state, we ran a separate, independent instance of MEMCACHED on every core, and configured the client to partition the key space across these multiple MEMCACHED instances. As shown in Figure 14, the results indicate that even this partitioned MEMCACHED configuration is significantly slower than both CPSERVER and LOCKSERVER. We only measured the performance of MEMCACHED to 10 cores, using one hardware thread per core, since MEMCACHED does not benefit from hardware multi-threading, and achieves even lower throughput on multiple sockets. Overall, this result suggests that the performance of CPSERVER and LOCKSERVER is competitive. Note that LOCKSERVER achieves better performance than CPSERVER for low number cores but that CPSERVER outperforms LOCKSERVER for larger number of cores, as shown above.

8 FUTURE WORK

We would like to improve the CPHASH implementation in two ways: dynamically changing the number of server threads and supporting keys with different lengths.

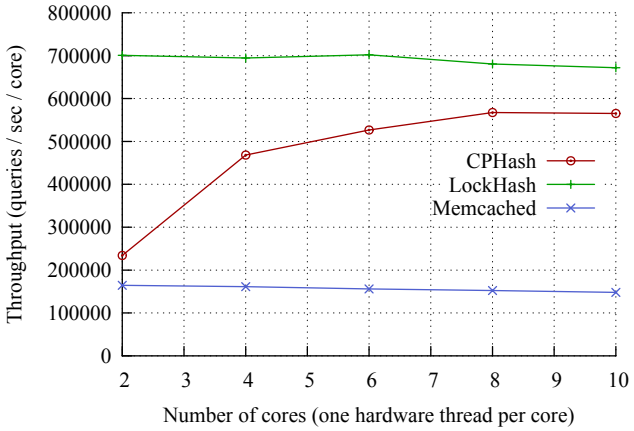

Figure 14: Throughput of CPSERVER, LOCKSERVER, and MEMCACHED for a range of cores (using one hardware thread per core), with a 128 MB working set and hash table capacity.

8.1 Dynamic Adjusting of the Server Threads

The current implementation dedicates a fixed number of cores to run server threads. A better approach would be to have an algorithm that dynamically decides on how many cores to use for the server threads, depending on the workload. Such dynamic adjustment of the server threads would make it possible to use fewer cores when the workload is small, making it possible to run other services on the same machine when there is not much load on the hash table.

Dynamic adjustment of the server threads could also provide higher performance. If the CPU resources needed by the client threads to generate the queries is less than the resources needed by the server threads to complete the queries, then it is better to dedicate more cores to run the server threads than to the client threads. On the other hand, if the client threads need more CPU resources to generate the queries, it is better to dedicate fewer cores to run the server threads and use more cores for the client threads. In our experiments we chose statically the number of servers and clients threads that resulted in the best performance.

8.2 Handling Any Size Keys

The current implementation supports only 60-bit hash keys, which can easily be extended to any size keys without modifying CPSERVER. The main idea to support any size keys is to use on INSERT the 60-bit hash of the given key as a hash key and store both the key and the value together as a value. Then to perform the LOOKUP of a certain key, CPHASH would first calculate the hash key and lookup the value associated with it. If such a value exists it would contain both the key string and the value string in it. Then before returning the value CPHASH would compare the key string to the actual key that the client wanted to lookup, and, if there is a match, return the value. If the key strings do not match, this would mean a hash collision since their hash values match but the strings itself do not. In this case, CPHASH would just return that the value was not found; since CPHASH is a cache, this doesn't violate correctness. Furthermore, the chance of a collision with 60 bit-keys is very small, especially considering the fact that the hash table is stored in memory; it cannot have more than several billion elements.

9 CONCLUSION

This paper introduced CPHASH, a scalable, concurrent hash table for key/value caches. Unlike traditional shared-memory implementation using locking, CPHASH partitions the hash table across server cores, and clients perform operations on a partition by sending a message through shared memory to the right partition. The message system provides asynchronous message passing to increase parallelism between clients and servers, and packs multiple messages in a single cache-line to reduce cache-line transfers. Experiments with CPHASH and an optimized fine-grained locking implementation, LOCKHASH, show that CPHASH achieves 1.6× to 2× higher throughput, because it has 1.5 fewer L3 misses per operation and the 3 misses CPHASH experiences are less expensive. These improvements are attributable to the design of CPHASH, which avoids locks and has better locality. CPHASH also scales better to larger number cores than LOCKHASH. Experiments with CPHASH and LOCKHASH in a MEMCACHED-like application show that both CPHASH and LOCKHASH-based servers are faster and scale better than MEMCACHED, and that CPSERVER outperforms LOCKSERVER by 5% at 160 hardware threads.

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] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.

[2] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Trans. Comput. Syst.*, 16 (1):1–40, 1998.

[3] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2nd International Green Computing Conference*, Orlando, FL, July 2011.

[4] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[6] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verità, Switzerland, May 2009.

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Vancouver, BC, Canada, 2010.

[8] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[9] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–292, 2006.

[10] R. Chandra, A. Gupta, and J. L. Hennessy. Cool: An object-based language for parallel programming. *Computer*, 27(8): 13–26, 1994.

[11] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, 2007.

[12] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, Thira, Santorini, Greece, 2010.

[14] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in dsm systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, 1996.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems. In *International Symposium on High-Performance Computer Architecture*, pages 367–378, February 2008.

[16] Memcached. Memcached, May 2011. `http://www.memcached.org/`.

[17] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, pages 26–41, May-June 2008.

[18] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the single-chip cloud computer. In *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC)*, Ettlingen, Germany, 2011.

[19] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[20] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *41st International Symposium on Microarchitecture (MICRO)*, pages 258–269, November 2008.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2001.

[22] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, Washington, DC, 2009.

[23] D. Tam, R. Amzi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on Interaction between operating systems and computer architecture*, pages 27–23, June 2007.

[24] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, Lisbon, Portugal, 2007.

[25] J. Triplett, P. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the Usenix Annual Technical Conference*, 2011.

[26] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[27] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 89–102, 2009.