CPHash: A Cache-Partitioned Hash Table with LRU Eviction

by

Zviad Metreveli

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2011

Certified by . . . .
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

CPHash: A Cache-Partitioned Hash Table with LRU Eviction

by

Zviad Metreveli

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis we introduce CPHash – a scalable fixed size hash table that supports eviction using an LRU list, and CPServer – a scalable in memory key/value cache server that uses CPHash to implement its hash table. CPHash uses computation migration to avoid transferring data between cores. Experiments on a 48 core machine show that CPHash has 2 to 3 times higher throughput than a hash table implemented using scalable fine-grained locks. CPServer achieves 1.2 to 1.7 times higher throughput than a key/value cache server that uses a hash table with scalable fine-grained locks and 1.5 to 2.6 times higher throughput than Memcached.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Assistant Professor

Acknowledgments

Foremost, I would like to thank my advisors, Frans Kaashoek and Nickolai Zeldovich, for all their help and guidance during this project. I've learned a lot while working with them this year.

I would also like to thank MIT for giving me an opportunity to be a part of its truly inspiring community.

Thank you to my girlfriend for being there for me and all her help and care.

Finally, I would like to thank my family. I would like to thank my brother for his friendship and support throughout the years and my parents for always encouraging me to follow my aspirations.

Contents

Chapter 1

Introduction

Hash tables are heavily used data structures in distributed data centers and web servers. This thesis 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. One of the best known distributed applications that uses a key/value cache is MEMCACHED [15]. 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.

With the rapid growth of the World Wide Web and large-scale applications, more scalability is demanded from data structures. Although many data structures and applications are already developed with the scalability requirement in mind, they are usually designed for distributed systems that consist of multiple different machines. The recent emergence of multi-core architectures demands that we rethink scalability not just for scaling across multiple machines but also across multiple cores of the same machine.

This thesis explores the use of *computation migration* to increase data structure performance on multi-core processors. This technique can also be applied to structures other than the hash table. We chose the hash table to demonstrate the benefits of this idea because of its overall simplicity, ease of implementation and relevance.

1.1 Motivation

Since CPU frequencies can no longer be significantly increased due to heat and power dissipation challenges, processors are now becoming more powerful by having more and more separate computation cores. Thus, if we want to get better performance as the number of cores increases, we need to think of ways to make our applications more scalable across multiple computation cores. One of the first steps in this challenge is to rethink the current data structures with the multi-core model in mind.

In a multi-core processor that supports shared memory each core has its own data cache to make memory accesses faster. However, if multiple cores are modifying and reading the same data, it is becoming more and more expensive to keep the caches coherent. Clearly, as the number of cores continues to increase, it will become more and more expensive to access and modify the same data using multiple cores. Also most of the current processors have NUMA architecture, thus localized memory accesses are faster than random memory accesses.

In the 48-core machine used in thesis, a core can access its L1 cache in 3 cycles, its L2 cache in 14 cycles, and the shared on-chip L3 cache in 28 cycles. DRAM access latencies vary, from 122 cycles for a core to read from its local DRAM to 503 cycles for a core to read from the DRAM of the chip farthest from it on the interconnect. Since accesses from caches are much faster than accesses from the DRAM, reducing the number of cache misses can have large impact on overall performance of an application.

1.2 This Thesis

This thesis introduces a new hash table, which we call CPHash, which uses computation migration to avoid unnecessary data transfer between cores and increase performance and scalability by reducing total number of cache misses. Instead of all the cores accessing shared data, CPHash splits the contents of the data structure into multiple parts and assign a core to each particular part of the structure.

CPHash uses message passing to pass the lookup/insert operation to the core that is assigned the data needed for that particular operation. CPHash assumes that computation migration will work well when modifiable data contents are large and the computation description is small. This thesis strives to prove this assumption by providing an implementation of CPHash that uses computation migration and demonstrating its performance gains.

On the 48-core machine we compare the performance of CPHash to the performance of a standard fine grain lock implementation of a hash table and observe benefits due to two main reasons: Decrease in cache capacity misses (for small data sets) and decrease in cache coherency misses (for all data sets). Cache capacity misses are reduced since CPHash avoids data transfers and thus data duplication in different caches. Cache coherency misses are reduced due to caching common partition data that are modified, which in CPHash is the head of the LRU list. Both the lookup and the insert operations access and modify the head of the LRU list.

This thesis also 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 server that uses a hash table with standard fine grain locks. We also compare the performance of CPServer against Memcached. We observe increase in throughput in both cases due to the speedup provided by the CPHash hash table implementation.

1.3 Outline

The remainder of this thesis is structured as follows. Chapter 2 describes the related work. Chapter 3 describes the overall design and implementation of CPHash. Chapter 4 describes CPHash's memory management algorithm for allocating and storing the hash table contents in memory. Chapter 5 describes design and protocol of CPServer. Chapter 6 describes the benchmarking methods and contains a detailed evaluation of the performance gains. In Chapter 7 we discuss future plans for CPHash. Finally, Chapter 8 concludes this thesis with an overall perspective and summary of the achieved results.

Chapter 2

Related Work

There already exists many different techniques to optimize use of caches on multi-core chips. In this chapter we present overview of some of those methods and describe how they differ from the approach taken in this thesis.

2.1 Multi-core Cache Management

Thread clustering [22] dynamically clusters threads with their data on to a core and its associated cache. Chen et al. [7] 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 [8, 21, 14, 19, 24]. Chakraborty et al. [5] 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.

Several researchers place OS services on particular cores and invoke them with messages. Corey [2] 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 [16]. Suleman et al. put critical sections on fast cores [20]. Barrelfish [18] and fos [23] treating cores as independent nodes that communicate using message passing.

The methods described in this thesis focus specifically on data structures and are meant for providing techniques for scaling the data structures on many core processors. Flat combining [11] has the same motivation. The main idea behind flat combining is to let a single thread gain global lock on a data structure and perform all the operations on it that all the other threads have scheduled. 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. This approach is somewhat similar to the approach that we take with CPHash in a sense that there is a server thread that performs all operations and there are client threads that schedule their operations. The main difference is that in CPHash there are multiple dedicated server threads that perform the operations and this server threads are pinned to specific cores. On the other hand in flat combining there is a single thread at any time that acts as a server thread, but any thread can become the server thread.

2.2 Computation Migration

CPHash attempts to move computation close to data, and was inspired by computation migration in distributed shared memory systems such as MCRL [12] and Olden [4] and remote method invocation in parallel programming languages such as Cool [6] and Orca [1].

Chapter 3

CPHash Design

As mentioned in Chapter 1, our goal is to create a scalable hash table that performs well on many core CPUs. To achieve such high scalability we use the idea of computation migration. Figure 3-1 gives top level view of CPHash design. CPHash is split into multiple independent parts, which we call *partitions*. We create a simple hash function to assign each possible key to a different partition. In CPHash all partitions are of equal size. Even though this might not always be the best idea, for simplicity we decided to keep it this way. If needed, partitions can be implemented to have a more flexible size by having more advanced memory management and data eviction algorithms (see Chapter 7 for discussion of such extensions).

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 core.

CPHash is used in an application by having client threads that communicate with the server threads and send queries using message passing (via shared memory). Server threads return query results to the client threads also using message passing.

Section 3.1 below provides a more detailed description of the partition data structure. Sections 3.2 3.3 describe the operation of the server and client threads in more detail. Section 3.4 describes our high performance message passing mechanism that uses buffering and batching. In Section 3.5 we present benefits of computation migration.

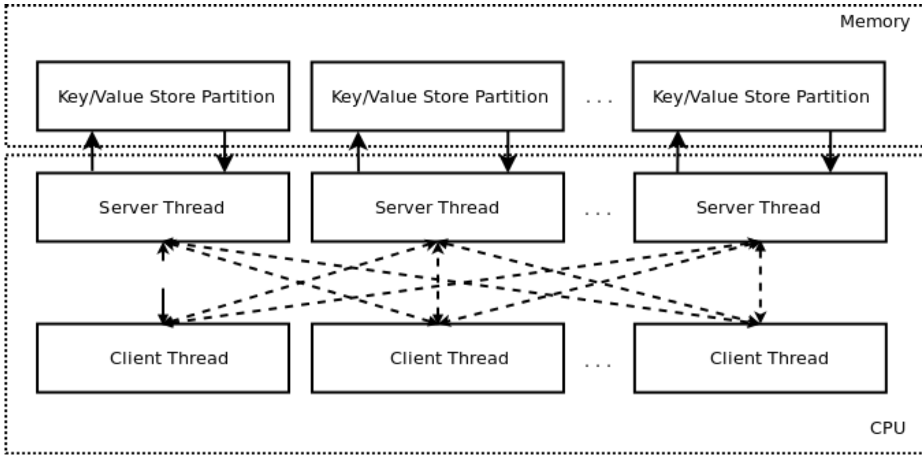

Figure 3-1: CPHash Design

3.1 Data Structure

Every single partition in CPHash is a separate hash table. Figure 3-2 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. We use LRU list to determine which elements to evict from a partition when there is not enough space left to insert new elements.

We pre-allocate the space that a partition can use to store data elements at initialization. Each element stored consists of a `key`, a `pointer` to a value, and a `size`. In CPHash the keys are limited to being 60-bit integer numbers; however, this can easily be extended to support any key size (see Section 5.2 for more details).

3.2 Server Threads

Each server thread is responsible for all the operations that are done on a single partition. The server thread continuously loops over the message queues of each client checking for new requests. When requests arrive, the server thread performs the requested operation and sends its result back to the client.

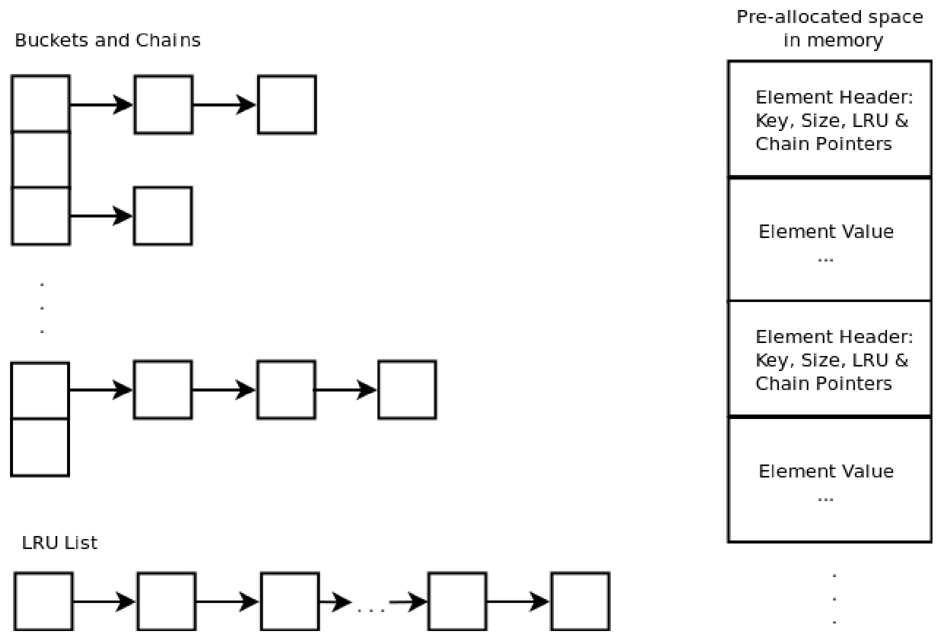

Figure 3-2: Partition Data Structure

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`.

Performing an `Insert` operation is slightly more complicated. CPHash is non-intrusive and supports arbitrary length values; thus, every time an insert operation occurs, space must be allocated in memory for the value to be copied over. In CPHash, the space for the value is allocated by the server thread but the data itself is copied by the client thread. Thus, to perform an `Insert` operation, the server needs to receive the `key` and the `size` of the data. The server thread allocates `size` amount of space 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 server returns a pointer to the allocated space to the client, which the client later fills in with the actual data. Chapter 4 goes into more detail on how exactly the memory management works.

3.3 Client Threads

Applications have client threads that communicate with the server threads to get the queries done. Client threads do not necessarily have to be pinned to a specific core but, to achieve the highest performance in message passing, it is best to keep the client threads attached to a specific core. An example of a client thread in an application would be the client thread in CPServer implementation. The client threads in CPServer gather queries from the TCP connections, route them to the appropriate server threads, gather the results, and send them back to the correct TCP connections. Chapter 5 describes the CPServer implementation in more detail.

3.4 Buffering and Batching

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 buffers – one for each direction of communication. Another possible way to implement message passing could have been to use single value communication. Figure 3-3 gives graphical representation for both designs.

In the single value communication pattern, space is allocated for each client/server pair and when a client wants to make a request to a server, it modifies this location with its query and waits for the server to respond. When the server is done processing the query it updates the shared location with the result.

The implementation of a single one-way circular buffer consists of the following: a data buffer array, a read index, a write index, and a temporary write index. The buffer uses a single producer – single consumer pattern. 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 temporary write index. When the temporary write index is sufficiently larger than the write index, 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 data and update the read index. The Read Index, Write Index and Temporary Write Index are carefully aligned in memory to avoid any 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 queries in a cache line.

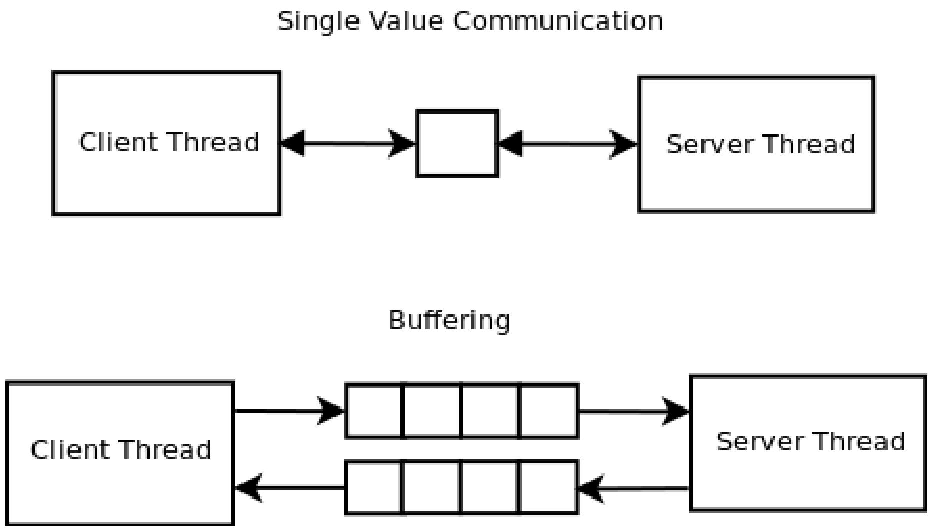

Figure 3-3: Message Passing Designs

There are two major benefits to using buffers instead of single value communication. The first advantage is improved parallelism. With 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 queries for other servers. This way all the servers can stay busy 100% of the time, thus increasing the overall performance. The second reason is the decreased message passing overhead. With single value communication, for every query received, the server would experience a cache miss; however, since the size of the cache line consists of multiple longs (in our test machines it is 64 bytes), with buffering the server can receive multiple requests using only a singe cache miss. Figure 3-3 shows the graphical representation of both designs.

With benefits there are some downsides to using buffers instead of the single value communication pattern. The circular buffer 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 single value communication does not have. Thus, if the client sends requests to the server at a slow rate, single value communication would outperform the buffering implementation. However, if the client has a batch of requests that it needs to complete, buffering will be an advantage. Since we are improving the hash table data structure to target applications that are bottlenecked by the performance of the hash table, there should be no shortage of requests; therefore, buffering is a better design choice for message passing.

3.5 Advantages of Computation Migration

There are several advantages to using computation migration. The first advantage is having a better cache performance due to the fact that each partition is only modified and accessed by a single server thread, i.e. a single core. Since there is only one core accessing and modifying the data, there are no invalidations for data that is written. Furthermore, all frequently accessed structures, such as, the LRU list, buckets array, free list, etc can stay in the cache and can be read and modified fast. This provides a significant overall performance increase. Chapter 6 reports on measurements that quantify the performance increase.

The second advantage is being able to get rid of synchronization mechanisms for shared data access and modification. Since each partition is modified by only a single core, there is no need for any synchronization mechanisms to protect the data from races. In addition to performance benefits, this approach also provides the benefit of ease of implementation. With computation migration the actual data structure operations are single threaded and, thus, no changes are necessary from the single threaded implementation.

The more difficult part of the computation migration implementation lies with message passing; however, the message passing design and implementation can be standardized and the message passing design and implementation can stay exactly the same for many other data structures. Otherwise, to gain good scalability and performance for each specific data structure, different designs for synchronization would be necessary. For more complicated data structures the synchronization design can get very complicated with all possible cases of race conditions and data paths, thus leading to more potential implementation bugs. Computation migration provides an easier way to adopt a previous implementation of a single-threaded data structure into a more scalable one.

Chapter 4

Memory Management

To implement a non-intrusive hash table, in addition to storing keys and pointers to the values, the actual value data needs to be stored. In order to store arbitrary length values, CPHash needs the ability to allocate space in memory when inserting an element and free it when CPHash evicts an element from the hash table. We also need to decide which thread (server or client) should be responsible for data allocation and which thread should be responsible for copying the data into the allocated space. Freeing values is complicated by the fact that each value can be in use in multiple client threads; thus, we need some way of determining when it is actually safe to free and reuse the space used by a data value.

Section 4.1 discusses allocation strategies and provides details on our actual implementation. Section 4.2 discuses strategies for freeing and deallocation. In Section 4.3 we discuss the alternative strategy of reference counting.

4.1 Allocation

The best place to do space allocation is in the server thread since each server is responsible for a single partition and implementing space allocation would be as simple as implementing a space allocator for a single-threaded hash table. However, performing the actual data copying in the server thread would be a bad idea since for large values it would wipe-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. To perform an `Insert` operation, the client sends the `key` and the `size` of the value to the server. The server allocates `size` amount of space in memory and returns the pointer to the allocated memory to the client. 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. The current CPHash implementation this marking is done using atomic operations. In Section 4.3 we discuss an alternative to it using message passing.

There are many different ways to perform data allocation in the server thread. The simplest way is to use C standard `malloc/free` operations. However, in a heavily multi-threaded environment the libc standard `malloc` performs poorly. A better alternative could be to use memory allocators designed for multi-threaded programs such as streamflow [17], or tcmalloc [10] or any other multi-threaded allocator. However, since in CPHash the total space is split equally between partitions, we decided to just pre-allocate all of the available space for each partition and then use the standard single-threaded binning allocator [9] inside that pre-allocated space. This way the server threads will never have to communicate when allocating or freeing space.

4.2 Deallocation/Freeing

When the server thread evicts or deletes an element from the hash table, the space allocated for this value must to be freed so that it can be reused for new elements. It would be incorrect for server thread to just free the allocated space when it evicts or deletes the element. The problem is that if a client requests a `Lookup` on some element X and gets the pointer to its value, and then the server threads evicts the element X from the hash table before the client is done processing X's value, the client will have a dangling pointer pointing to unallocated space, potentially causing all kinds of errors. To resolve this issue, CPHash counts references to the elements. 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 given element. When the reference count reaches 0, the space can be safely deallocated. We implemented reference counting using atomic operations.

The deallocation must be done by server threads, otherwise there would be race conditions between allocations and deallocations for a partition. When a client dereferences an element and its reference count becomes zero we need some way to schedule an element for freeing in the server thread. It is worth noting that this is, in general, a highly unlikely scenario, especially if clients process elements quickly, since if an element was just accessed in the hash table, it would become the most-recently used item thus significantly decreasing chances of its eviction before the client is done processing it.

To implement scheduling of elements for freeing, we implemented a lock-free singly-linked list using atomic operations that holds the list of elements that can be safely freed. Scheduled elements are freed in the server thread before the next allocation.

4.3 Atomic Operations VS Message Passing

As mentioned in previous sections we implemented the necessary synchronization for memory management using hardware-supported atomic operators. Another way to implement reference counting could have been using message passing. Instead of the client thread updating the reference count, it would send a message to the appropriate server to update the reference counter. However, in this case, message passing is not the best option for several reasons. Since message passing is implemented using shared memory and without special hardware support, sending a single message is just as expensive as a single write to a memory location. Also in most common cases when a client needs to update the reference count, the counter is already in the client's cache thus making those atomic operations fast. The message passing version could become a more viable option if the hardware had some special support for fast core-to-core communication.

Another alternative way to implement reference counting would be to send a single message to the server to release all pointers per batch. However, that would require the server thread to store all the pointers allocated during the last batch for each client. This would impose a significant overhead on the server's local hardware cache, especially if the batches are large. Also it would provide less flexibility for a client to decide when to release values.

Chapter 5

CPServer Design

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. CPServer has server and client threads as described in Chapter 3; however, it also has clients that connect to the server using TCP connections. To avoid confusion with names of client threads and clients that connect over TCP, we will call the latter TCP clients. Figure 5-1 shows the design of CPServer.

The server threads operate as described in Chapter 3. Client threads monitor TCP connections assigned to them and gather as many requests as possible to perform them in a single batch. Then, as described in Chapter 3, client threads pass the requests to the appropriate server threads using message passing. After the server threads are done and the client threads receive their results back, they write back those results to the appropriate TCP connections.

The CPServer also has a TCP server 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. For our testing needs this simple type of load balancing works fine, however the load balancer could be more advanced for work loads in which the traffic on different connections differ significantly.

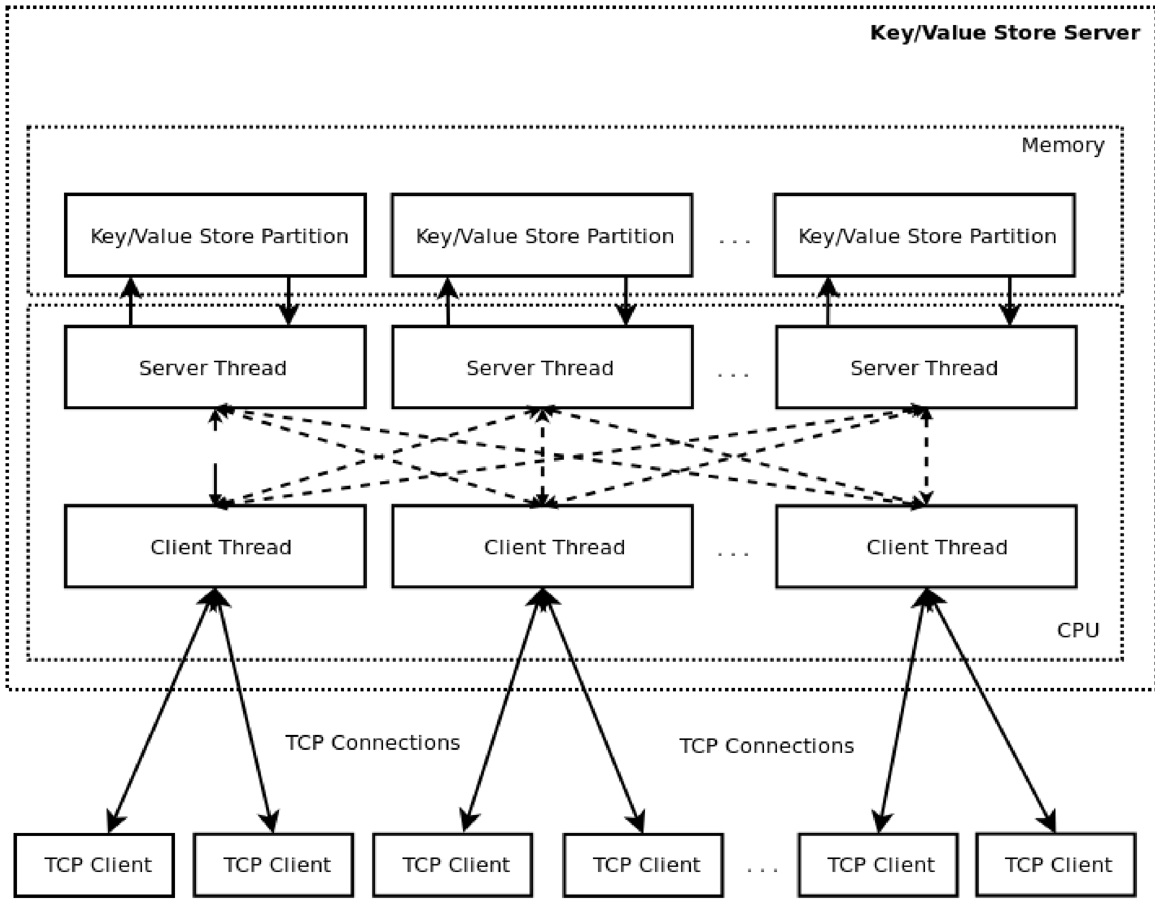

Figure 5-1: CPServer design

5.1 Protocol

CPServer uses a simple binary protocol. Figure 5-2 presents binary format of a request header. Figure 5-3 present binary format of a response.

32 bits operation type	32 bits size	64 bits hash key

Figure 5-2: CPServer request header

Two operation types currently supported are: LOOKUP and INSERT.

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. The `size` field is unused for the LOOKUP request, thus it can be set to any value.

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` amount of bytes which describe the value to be inserted.

The actual supported size of the hash key in CPHash is 60 bits, thus the most significant 4 bits of the `hash key` field will always be ignored by the server.

Figure 5-3: CPServer response

The responses for currently supported operations are the following:

LOOKUP For the LOOKUP requests, if a key/value pair is found in the hash table such that the key matches the `hash key` provided in the LOOKUP request, then the size of the value and the actual value data are returned. Otherwise if such a key/value pair can not be found in the hash table, a response with a `size` of 0 is returned.

INSERT The INSERT requests are silent, thus for them there is no response returned from the server.

5.2 Handling Any Size Keys

In our current implementation only 60 bit hash keys are supported. This can easily be extended to any size keys without modifying CPServer. The main idea to support any size keys is to use the 60 bit hash of the given any size 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, we 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 we would compare the key string to the actual key that we wanted to lookup and if return the value. If the key strings do not match, this would mean we got hash collision since their hash values match but the strings itself do not. In this case we would just return that the value was not found. The chance of collision with 60 bit keys would be very small, especially considering the fact that the hash table is stored in memory thus it can not have more than couple billion elements in it.

To perform the INSERT operation we would calculate the hash key from the key string and insert a key/value pair in the hash table where the key would be our calculated hash key and the value would be a combined string of both the key and the value.

Chapter 6

Performance Evaluation

In this chapter we discuss the performance results that we achieved using CPHash and CPServer. In the following sections we first discuss our alternative implementations that were developed for comparative benchmarking. Then we present our evaluation of the scalability and performance of CPHash. Finally we provide the benchmark results for CPServer

6.1 Alternative Hash Table and Key/Value Cache Server Implementations using Locks

To evaluate the performance and scalability of CPHash, we created an alternative implementation of the hash table that does not use computation migration. This version is implemented in a traditional shared memory style with scalable fine-grained locks. In this alternative implementation, each partition is protected by a lock and there are no server threads. The client threads process queries by first acquiring the lock for the appropriate partition, then performing the query, updating the LRU list and, finally, releasing the lock. We call this implementation LockHash. We use this alternative implementation for comparison with CPHash to show that computation migration provides much better scalability and performance than having scalable locks.

In addition to developing the alternative hash table implementation we also developed an alternative key/value cache server implementation that uses LockHash as its hash table instead of CPHash. We call this alternative server implementation LockServer.

6.2 Hash Table Performance

We created a simple benchmark that tests various aspects of the hash table implementations. The benchmark 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 number and the value is the same as the key (8 bytes).

The benchmark can be configured using several parameters:

- Number of Partitions (i.e. number of server threads)
- Number of Client Threads/Cores
- Size of Batch
- Maximum cache size in bytes
- Hash INSERT ratio over total number of queries
- Maximum Value of Hash Key
- Number of iterations

We use a 48 core AMD64 machine for our testing. This machine has eight six-core processors of type AMD Opteron(tm) 8431. Each core has a 512KB L2 cache and each six-core processor has a unified 6MB L3 cache.

6.2.1 Scalability

The first experiment evaluated the scalability of the CPHash implementation. We ran our benchmark with an equal number of server and client threads varying from

3 to 24 (i.e. using 6 to 48 cores – half of the cores for the server threads, and the other half for the client threads), with a total hash table size of 10 MB, with 30% INSERT ratio, with keys ranging from $0..2^{17}$, and for 10^8 iterations. We also ran our LockHash implementation with the same exact parameters using 6 to 48 cores. Figure 6-1 shows the throughput per core for CPHash and LockHash on a 10 MB hash table.

We also ran the same tests but with a hash table size of 1 GB with keys ranging from $0..2^{24}$. In this case, we ran the benchmark for 10^9 iterations to make sure that the hash table was full for most of the iterations. Figure 6-2 shows the throughput per core for CPHash and LockHash on the 1 GB hash table.

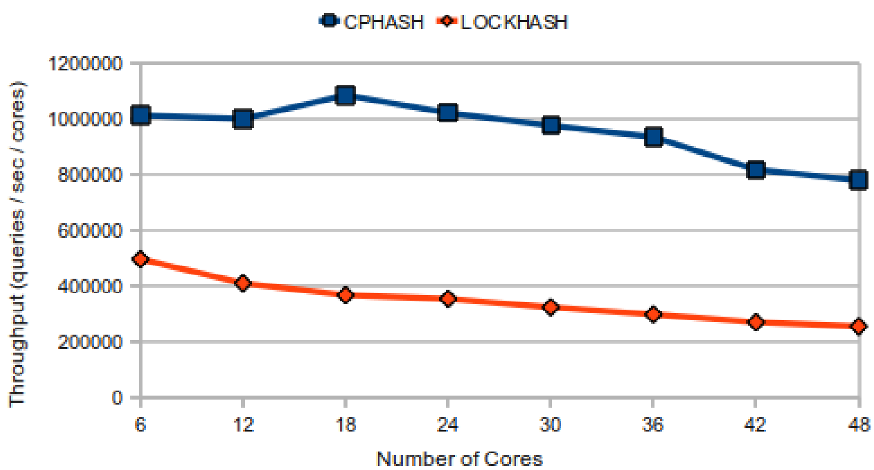

Figure 6-1: Throughput per core on 10 MB hash table

These graphs show that the CPHash implementation has better scalability than the LockHash implementation. However, we do not get a complete linear speedup for small hash tables. The reason for this is the message passing overhead. This overhead is more significant for smaller hash tables since most other memory operations are cache hits, thus, message passing takes a significant percentage of the total computation time. On the other hand CPHash achieves great scalability for larger hash table. It achieves a super-linear speedup in Figure 6-2 due to the fact that as the number of cores increases, its combined L2 and L3 cache space is larger, resulting in fewer cache misses per query and higher throughput per core.

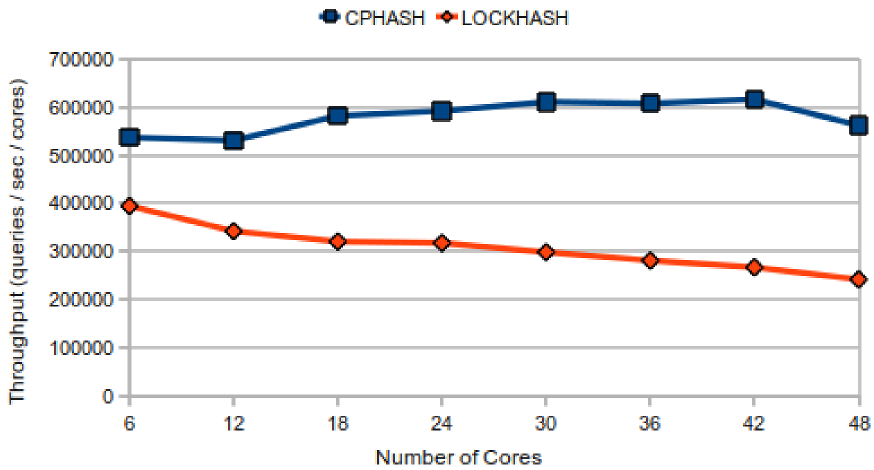

Figure 6-2: Throughput per core on 1 GB hash table

As the number of cores increases, message passing becomes more expensive due to the hardware architecture. If the cores are physically farther apart from each other, the message passing cache miss will take longer to complete. To prove this hypothesis we ran our benchmark with a small empty hash table, and no INSERTs. In this scenario message passing takes most of the computation time. Figure 6-3 shows the declining throughput as the number of cores increases.

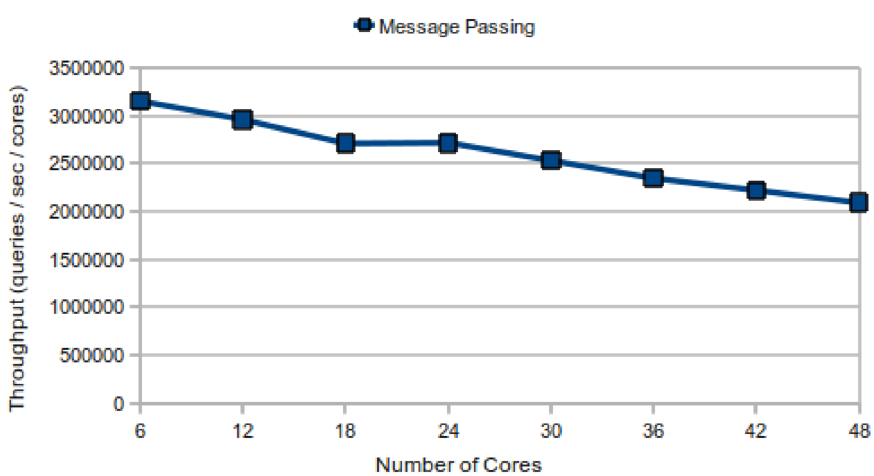

Figure 6-3: Message Passing Throughput per core

6.2.2 Performance

Figure 6-4 shows the throughput gains of CPHash compared to the LockHash implementation for the tests described in the previous section, as well as for a 40 MB hash table.

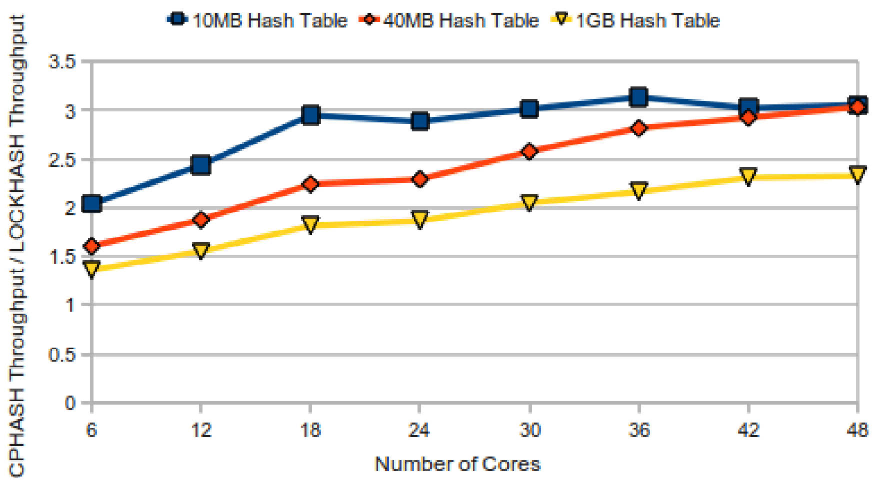

Figure 6-4: CPHash throughput vs LockHash throughput

The performance of CPHash does not depend on the actual size of the hash table but rather on the number of elements in it. This is because the server threads never access or modify the values in the hash table but just the meta data (buckets, chain lists, LRU list etc). In our benchmarks the value of each element has the same size (8 bytes), therefore, the number of elements in the hash table is proportional to the size of the table.

The results shown in Figure 6-4 are consistent with the scalability graphs. CPHash's throughput per core decreases after all the hash table meta data fits in the combined hardware caches of the server cores. This is the main reason why we do not see any increase in throughput ratio after the number of cores reaches 18, for 10 MB hash table. The current bottleneck for scalability on small hash tables is the CPHash's message passing implementation.

6.2.3 Summary

The benchmark 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 some benefits through batching and caching the common partition data (such as an LRU list).

There is another scenario when CPHash is beneficial for large hash tables. Even though most of the time the load on the hash table may not be significant and the performance is acceptable, there might be certain peak times when a specific small subset of the hash table experiences heavy load (i.e. most of the queries involve operations on keys in that subset). If this subset is small enough so that its meta data fits into the combined caches of server cores, then the CPHash implementation can significantly improve performance.

6.3 Key/Value Cache Server Performance

We tested the performance of our CPServer against the LockServer and Memcached. For these benchmarks we used a 16 core AMD64 machine as our server. This 16 core machine has four quad-core processors of type AMD Opteron(tm) 8350. Each core has a 512KB L2 cache, and each quad-core processor has a unified 4MB L3 cache.

We developed a benchmark client for the Key/Value Cache Server. We used our 48 core AMD64 machine as a load generator to run the benchmark clients. The load generator machine and the server machine were connected with 10 GBit Ethernet network. We made sure in our benchmarks that we were generating enough load to bottleneck the servers, so that the speed of a single benchmark client would not affect the performance. We also made sure that the servers were CPU and Memory bottlenecked and that the network was not the limiting factor. To achieve this, in addition to having 10GBit Ethernet network, we used a patch for the Linux kernel network driver [3] to make it more scalable.

The benchmark client has following configuration parameters:

- Size of Batch
- Hash INSERT ratio over total number of queries
- Maximum Value of Hash Key
- Size range for Hash Value
- Number of iterations

First we compared the performance of CPServer to LockServer. We ran the servers with a 16 GB hash table. Before starting any benchmarking we made sure to completely fill up the hash table with random data. We set the batch size to 1000, and the size range for the hash values to $8-128$ bytes. We run our tests for 80 million iterations.

We varied the INSERT ratio from 0% to 100% with 20% steps. We tried three different Hash Key ranges: $0..2^{16}$, $0..2^{23}$, and $0..2^{28}$. A 16 GB hash table can hold around 160 million elements when the size of values are 8 to 128 bytes. Therefore when running with Hash Key ranges of $0..2^{16}$ or $0..2^{23}$, the hit rate is 100%. On the other hand when running with a key range of $0..2^{28}$ it is around 60%-70%. Figure 6-5 shows the throughput gains of CPServer compared to LockServer.

There are four major factors that affect the speedup: Hit Rate, INSERT ratio, size of the hash table (or the range of hash keys), and the size of the hash values. Smaller Hit Rate and higher INSERT ratio result in larger speedups for CPServer. This is because hash INSERTs are silent (i.e. the server does not return any response for them), and LOOKUP misses just return a single 32 bit value. Thus, for hash INSERTs and LOOKUP misses, most of the computation time for a query is taken by the hash table operation and not by the writes to the TCP connection buffer.

Larger hash values work against the CPServer implementation. If hash values are large most of the computation time during LOOKUP query hits is spent on writing back the value to the TCP buffer. For very large hash values the LockServer can

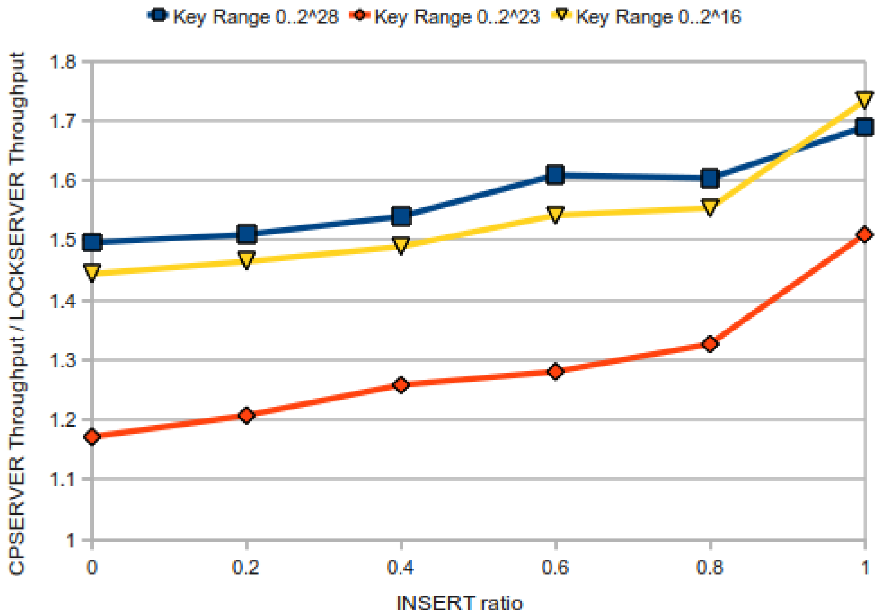

Figure 6-5: CPServer throughput vs LockServer throughput

outperform the CPServer since it has more threads writing to the buffers which results in better performance.

The size of the hash table or the range of the hash keys affects performance since, as we showed in the previous section, if the meta data of all the elements that are accessed can fit into the combined caches of the server cores, the hash operations in CPHash will be faster than in LockHash.

Figure 6-5 confirms our hypotheses. With key ranges of $0..2^{16}$ or $0..2^{23}$ the hit rate is 100%; however, the meta data of 2^{16} elements fits into the combined caches of 8 server cores, resulting in a higher speedup. For the key range of $0..2^{28}$ the hit rate is around 65%; therefore, there is less time spent on writing buffers and more time spent on the actual hash operations resulting in a higher overall speedup.

6.3.1 Performance against Memcached

We compared the performance of CPServer to Memcached. We ran 16 Memcached servers with 1 GB size limit each (16 GB total) on our server machine. We extended the benchmark client with the `libmemcached` library [13] to run same the benchmarks for Memcached as the ones described in the previous section. However, since the Memcached protocol supports batching only for LOOKUPs, we performed the tests with only LOOKUP operations. As in previous test runs we made sure to fill up hash table server with some random elements before running any benchmarks.

We ran tests with two different hash value size ranges: 8 to 128 bytes and 8 to 512 bytes. Tables 6.1 and 6.2, show the results for the two scenarios described.

Since the Memcached protocol has a higher overhead per each request and response sent, we also compared the performance of CPServer with hash value ranges of 72 to 192 bytes to Memcached with value ranges of 8 to 128 bytes. The results for this test run are provided in Table 6.3.

The tables give total running times for CPServer and Memcached (in seconds) and also provide the average hit rate per each LOOKUP operation. Figure 6-6 shows the throughput gains of CPServer compared to Memcached based on values provided in the tables.

Table 6.1: Speedup with hash value size range of 8 to 128 bytes

Key Range	CPServer	Memcached	Speedup	Hit rates
$0..2^{28}$	13.563	35.618	2.626	0.655 vs 0.817
$0..2^{23}$	22.102	39.741	1.798	1.0 vs 1.0
$0..2^{16}$	16.719	39.512	2.363	1.0 vs 1.0

Table 6.2: Speedup with hash value size range of 8 to 512 bytes

Key Range	CPServer	Memcached	Speedup	Hit rates
$0..2^{28}$	27.587	41.753	1.514	0.688 vs 0.750
$0..2^{23}$	43.532	46.251	1.062	1.0 vs 1.0
$0..2^{16}$	39.845	46.388	1.164	1.0 vs 1.0

Table 6.3: Speedup with larger hash value size range for CPServer

Key Range	CPServer	Memcached	Speedup	Hit rates
$0..2^{23}$	23.240	39.741	1.710	1.0 vs 1.0
$0..2^{16}$	19.329	39.512	2.044	1.0 vs 1.0

The results achieved are similar to the results achieved against LockServer. Smaller hit rate, smaller values and smaller key range all benefit the CPServer

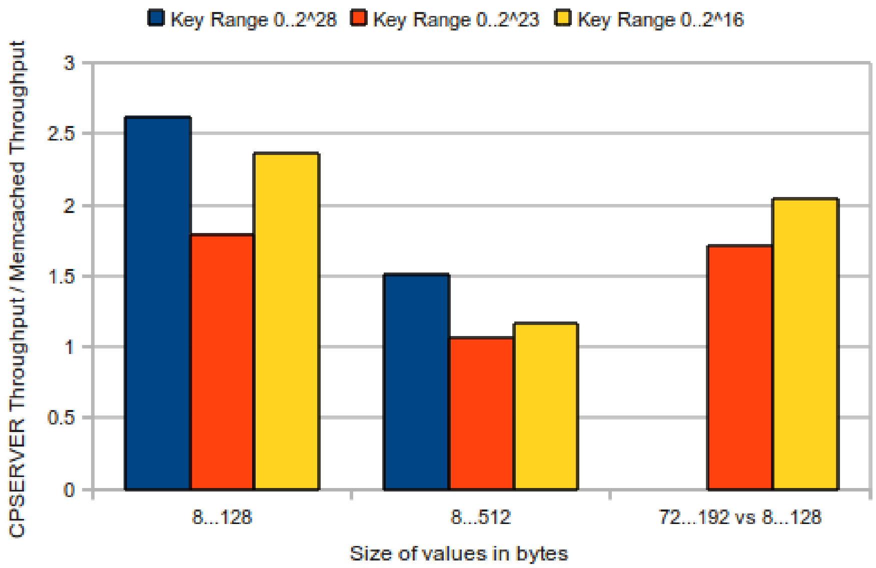

Figure 6-6: CPSERVER throughput vs MEMCACHED throughput

implementation for the same reasons as described in the previous section. On the other hand, as can be noticed in the second row of Table 6.2, when the hit rate is 100% and the value range is 8 to 512 bytes, CPSERVER has almost no advantage over MEMCACHED, especially when the size of hash table is much larger than size of combined L2 and L3 caches of all the server cores.

The results indicate that CPSERVER outperforms MEMCACHED in several specific scenarios; however, a performance evaluation with a load specific to a real-world application or web-server is needed for a more comprehensive comparison of the two server implementations.

Chapter 7

Future Work

CPHash demonstrates that by using computation migration, message passing and careful memory management techniques it is possible to create a scalable hash table implementation for the multi-core NUMA architecture machines. However, there is still room for improvement of CPHash's implementation.

7.1 Dynamic Adjusting of the Server Threads

One issue with the current design is that a fixed number of cores must to be dedicated to run the server threads. A better approach would be to have an algorithm that would dynamically decide 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 less power and CPU resources when the workload is small. Saving power is essential for any data center due to reduced cost. Using less CPU resources would make 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.

We have not tried implementing dynamic adjustment of server threads due to time constraint reasons; however, we tried a different approach to avoid wasting the CPU resources. We tried utilizing CPUs with Intel's HyperThreading technology to run the server and the client threads on two separate logical cores of the same physical core. The problem we discovered with this approach is that since the logical cores share the L2 cache, the client thread can easily pollute the whole cache thus nullifying most of the benefits of the CPHash design. This is especially true for an application such as CPServer, since the connection buffers themselves can take most of the space in the cache.

7.2 Message Passing

The scalability and performance of message passing is important for CPHash. The current implementation is simple and does not use any hardware specific operations to further speedup the communication between the server and the client threads. One possible improvement is to forcefully flush the caches when the buffer is full, this way the overhead of sending the message would shift more from the receiver towards the sender. Such an approach could be beneficial to decrease the time spent on reading the message buffers in server threads. Such an approach could potential provide more scalable message passing implementation.

Another idea to improve the current message passing implementation is to change the design of the circular buffer to eliminate the read and write indexes to further decrease average cache misses spent on message passing per each query.

Chapter 8

Conclusion

In this thesis we introduced CPHash − a scalable fixed size hash table implementation that supports eviction using an LRU list, and CPServer − a scalable in memory key/value cache server implementation that uses CPHash as its hash table. Experiments on a 48 core machine showed that on a small hash table CPHash has 3 times higher throughput than a hash table implemented using scalable fine-grained locks. On a large hash table CPHash had 2 times higher throughput than a hash table implemented using scalable locks. CPServer achieved 1.2 to 1.7 times higher throughput than a key/value cache server that uses a hash table with scalable fine-grained locks, and 1.5 to 2.6 times higher throughput than Memcached.

The improved performance is due to the reduced number of cache misses per operation. The number of cache coherency misses is reduced due to caching common partition data that are modified, which in CPHash is the head of the LRU list. For small hash tables, the number of cache capacity misses is reduced, since CPHash avoids the data duplication in local hardware caches by transferring the computation between the cores instead of the data.

Bibliography

[1] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Trans. Comput. Syst.*, 16(1):1–40, 1998.

[2] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[4] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proc. 5th PPoPP*, 1995.

[5] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 283–292, 2006.

[6] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Cool: An object-based language for parallel programming. *Computer*, 27(8):13–26, 1994.

[7] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM, 2007.

[8] Sangyeun Cho and Lei 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.

[9] D. Lea. A memory allocator, May 2011. `http://g.oswego.edu/dl/html/malloc.html`.

[10] Google, Inc. Google performance tools, May 2011. http://code.google.com/p/google-perftools/.

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[12] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. Dynamic computation migration in dsm systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[13] LibMemcached, May 2011. http://www.libmemcached.org/.

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

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

[16] 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.

[17] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 84–94, New York, NY, USA, 2006. ACM.

[18] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca 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.

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

[20] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yal e N. Patt. Accelerating critical section execution with asymmetric multi-core archite ctures. In *ASPLOS '09: Proceeding of the 14th international conference on Archite ctural support for programming languages and operating systems*, pages 253–264, 2009.

[21] 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 archit ecture*, pages 27–23, June 2007.

[22] David Tam, Reza Azimi, and Michael 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, New York, NY, USA, 2007. ACM.

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

[24] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Compute r systems*, pages 89–102, 2009.