INTEL® CORPORATION

Enhancing the Scalability of Memcached

Alex Wiggins – wigginal@engr.orst.edu
Jimmy Langston, Ph.D. – jimmy.langston@intel.com
5/17/2012

Memcached is an open-source, multi-threaded, distributed, Key-Value caching solution commonly used for delivering software-as-a-service, reducing service latency and traffic to database and computational servers. We introduce optimizations that overcome thread-scaling limitations of memcached, enabling effective utilization of high core-count servers. The approach employs Concurrent data structures and a modified cache replacement strategy to improve scalability. These data structures enable concurrent lockless item retrieval and provide striped lock capability for hash table updates. The replacement strategy imposes a relaxed ordering of items based on relative timestamps. Rules for item insert, delete, and cache maintenance guarantee thread safety. A configurable cleaner thread operates autonomously, reducing lock requirements. The optimized application exhibits linear scalability, overcoming the limitations of the open-source version. In testing 16-core servers, throughput improved by 6X and performance/watt by more than 3X, while maintaining service-level agreements. Based on core scaling results, memcached will scale-up on subsequent many-core processors.

Table of Contents

1 Introduction - Memcached and Web Services

Memcached is a Key-Value cache used by cloud and web service delivery companies, such as Facebook [1], Twitter [2], Reddit [3], and YouTube [4], to reduce latency in serving web data to consumers and to ease the demand on database and computational servers [5]. In addition to lowering latency, memcached's scale-out architecture supports increased throughput by simply adding memcached servers. Scaling up, however, is less rewarding as more than 4 cores causes performance degradation [1] [6].

In this paper, we introduce memcached optimizations to utilize concurrent data structures, an updated cache replacement strategy, and network optimizations to increase throughput over 6X versus the current open-source version (v1.6) on Intel® Xeon® E5 server processors while maintaining latency Service-Level Agreements (SLAs) of 1ms or less for Round-Trip Time (RTT). Figure 1 illustrates the 6X speedup of the optimized version over the v1.6 memcached baseline, with the optimized version on 32 logical cores providing 3.15 Million requests per second (RPS), with a median round-trip time(RTT) less than one millisecond.

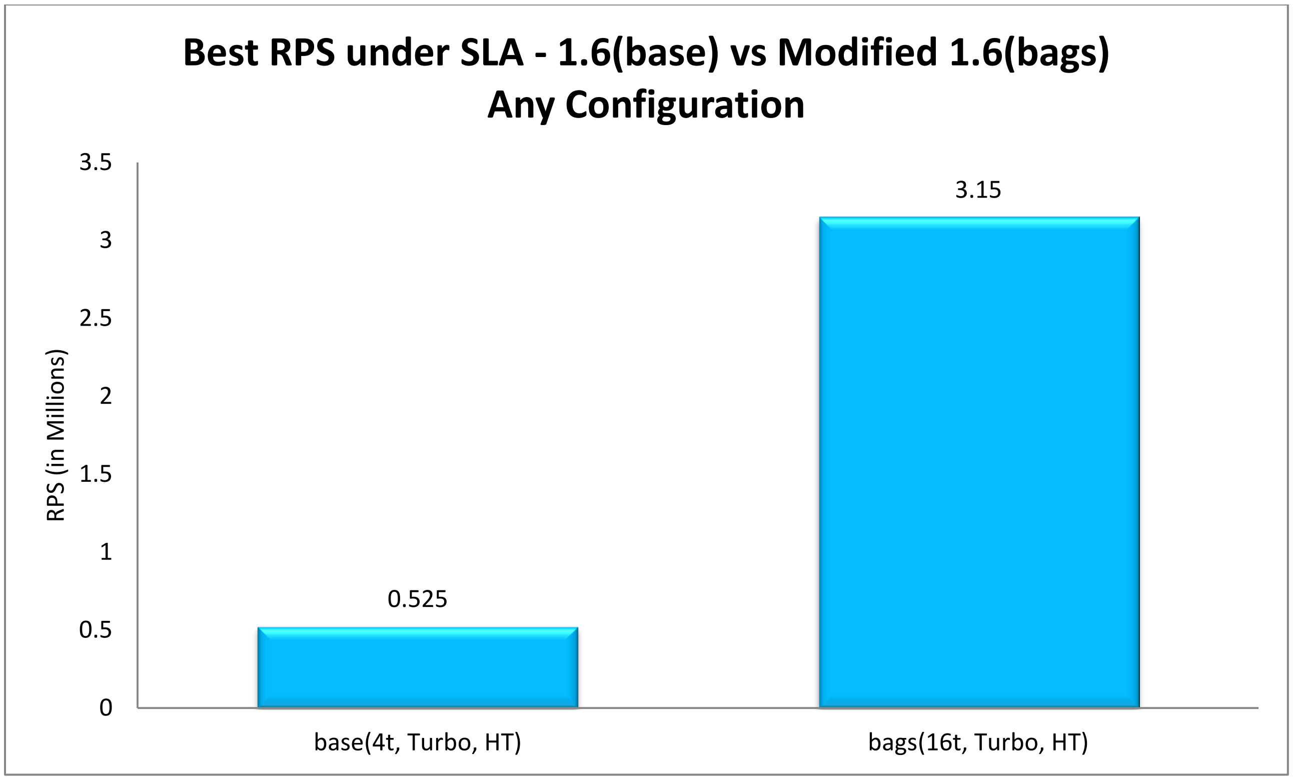


Figure 1 – Best throughput measured using any configuration from the base open-source code version 1.6.0_beta1 and the optimized version

Memcached operates on simple data types of key-value pairs, similar to NoSQL databases, but is not persistent like NoSQL. Memcached stores all the key-value pairs in non-persistent memory, so in the event of failure, all the stored data is lost. Throughout this paper we use the term *cache item* to specify the combination of a key and its associated value. Keys are, by definition, unique values.

In a web-serving architecture, the memcached application sits between the Front-End Web Servers, or Web Tier, and the Back-End Database as shown in figure 2.

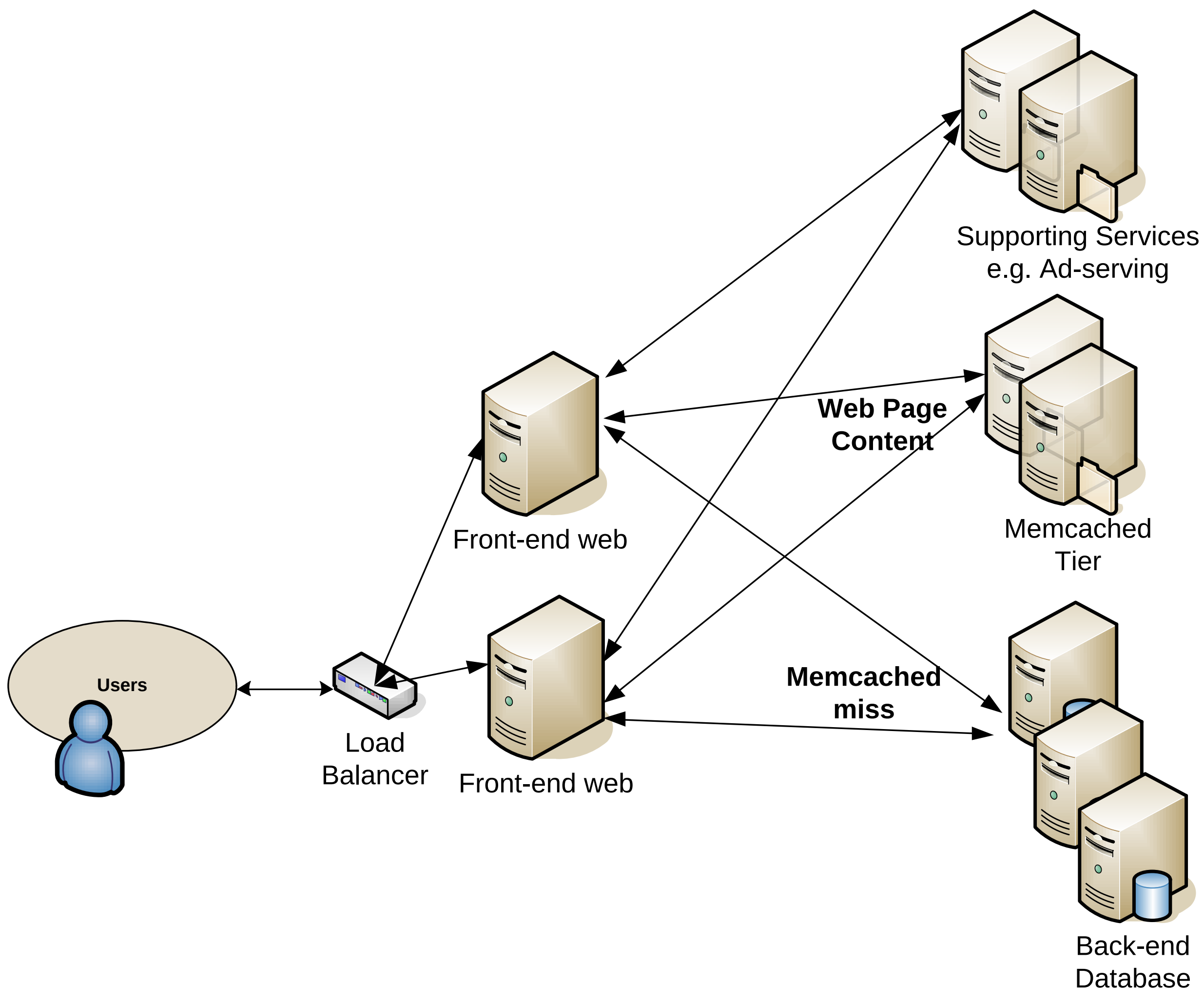


Figure 2 – memcached server sits between the Front-end Web Tier and Back-end Database

Memcached's purpose is to intercept data requests and satisfy them out of its logical cache (i.e. system memory) when possible, avoiding trips to disk storage attached to the back-end database. There is also the potential to avoid compute-intensive tasks by retrieving a pre-computed value from the logical cache. In both cases, the time spent retrieving or computing data results is reduced (i.e. transaction latency). A cluster of servers participates in a memcached logical cache, with each server offering its system memory as a portion of the complete logical cache [7] [8] [9]. For example, a memcached cluster of 10 nodes, with each node having a 128 GB memory footprint, provides a 1.28 TB logical cache for servicing data requests. The logical cache is populated from data in the back-end database, the persistent data store for web-serving, or from computational servers. Cache items are maintained using

Least Recently Used (LRU) policy as well as Time To Live (TTL). When an item is evicted its slot is filled by a more recent item.

A web service request can require multiple trips to memcached, the database, and other services. If the caching strategy is efficient, the number of trips to memcached will be an order of magnitude greater than the required trips to the database (i.e. 10:1). Data retrieval from system memory, as provided by memcached, is an order of magnitude faster than retrieving the same data from the database (microseconds versus milliseconds) and, in most cases, orders of magnitude faster than computing the data on-the-fly. Therefore, avoiding database accesses and computation is necessary to provide acceptable user response times for web service requests.

1.1 Terminology

Throughout the paper, we will reference several items with abbreviations and short-hand.

LRU (Least Recently Used) -- This is the eviction scheme used in memcached to determine which items are removed from the cache to free space for additional cache items.

Base (Baseline) – This is the unmodified 1.6.beta_0 version of Memcached, which can be downloaded directly from memcached.org.

Bags – This is short-hand for the modified version of memcached optimized to increase performance. *Bags* is the modified LRU cache update strategy, called a “bag LRU”. This was the first section of code developed.

RTT (Round Trip Time) – This metric is the elapsed time of a Memcached request, including the elapsed time from request from client to server until response back to client.

SLA (Service Level Agreement) – This metric is the maximum RTT allowed to maintain acceptable web service request response time to the User. Response time above the SLA will result in an unacceptable response time for generating the web service request. There can be multiple memcached requests (to memcached logical cache) per web service request (to Front-end Web Tier). For the purpose of this paper, the memcached SLA is 1 millisecond response time to the requesting Front-end Web Tier client.

NIC (Network Interface Card) -- This is the network card used in the system.

2 Current Memcached Architecture

When memcached was introduced in 2003 [5], there were few X86 processors with multiple cores, and multi-socket servers were extremely expensive. Intel® Xeon® dual-core and quad-core processors weren’t released until 2005 and 2009 respectively. Today, dual-socket, 8 and 16-core server systems are the building block of web service delivery, with roadmaps for processors showing increasing core counts [10]. These advanced systems provide ample opportunity for workloads to execute in parallel using multiple threads or processes, yet memcached is unable to do so using the current data structures and software architecture [1] [6].

2.1 Data Structures

Memcached has 4 main data structures, including

- A hash table to locate a cache item.
- Least Recently Used (LRU) list to determine cache item eviction order when the cache is full.
- A cache item data structure for holding the key, data, flags, and pointers.
- A slab allocator, which is the memory manager for cache item data structures.

The hash table data structure is an array of buckets. The array size (k) is always a power of 2 to make finding the correct bucket quicker by taking the value of *2^k-1* and using it as a hash mask. Executing a bit-wise AND (e.g. hash_value & hash_mask) quickly determines the bucket that contains the hash value. The buckets are constructed as single linked-lists of cache items that are NULL-terminated. Figure 3 illustrates this data structure.

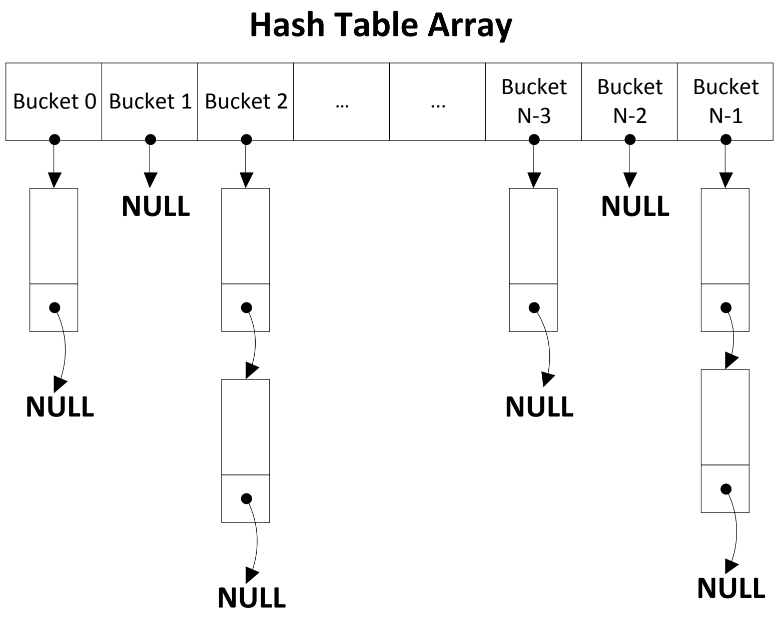


Figure 3 – Data Structure of hash table used to lookup cache items

The LRU, used to determine eviction order, is a double linked-list which holds all cache items in a single-sized slab (i.e. unit of memory allocation, described below). The double linked-list pointers are maintained in each cache item structure, similar to the hash table, and are modified each time the cache item is manipulated. In the event of an eviction, the tail of this list is checked for the oldest cache item, which is removed for reuse. Figure 4 Illustrates the LRU data structure.

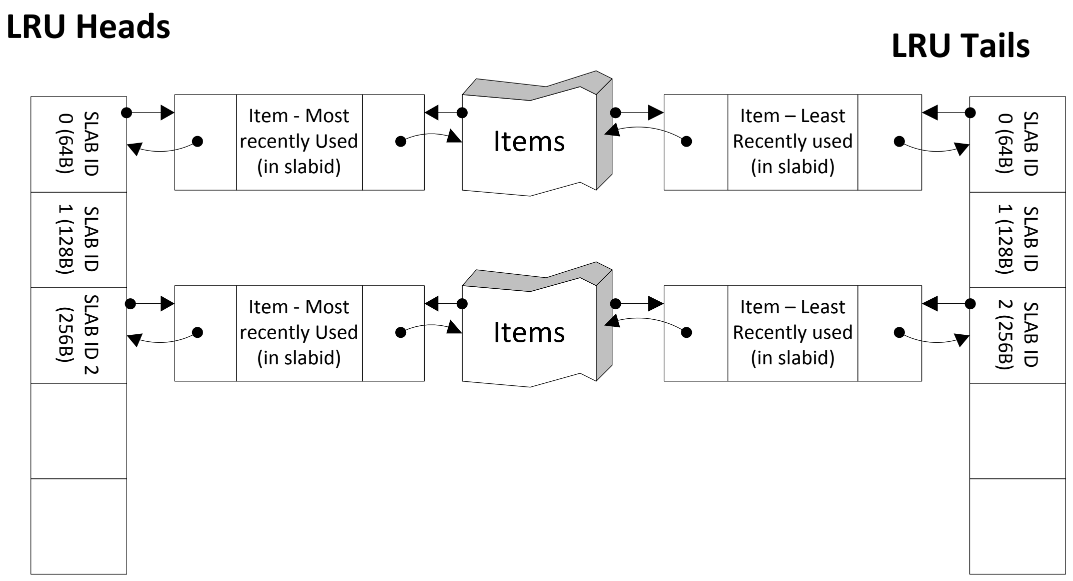

LRU Heads
LRU Tails
SLAB ID 0 (64B)
SLAB ID 1 (128B)
SLAB ID 2 (256B)
Item - Most recently Used (in slabid)
Items
Item – Least Recently used (in slabid)

Figure 4 – Data Structure of the current open-source LRU used to determine cache item eviction order

The cache item data structure holds the key-value pair data. It is an intersection of the following reference data:

- pointers used in the hash table for the single linked-list
- pointers used in the LRU for the double linked-list
- reference counter to determine how many threads are currently accessing a cache item
- flags for cache item status
- the key
- length of the value in bytes
- value

The slab allocator provides memory management for cache items. Since the cache items are relatively small, *allocating* and *freeing* of these small chunks of memory using system calls would be slow and likely cause thrashing. Instead, memcached uses slabs as the unit of memory allocation. A slab is a larger piece of memory that can hold many cache items within it. For example, a slab could be a 1024 byte chunk of memory which will hold 16 cache items that are 64 bytes or less. The slab allocator uses these larger memory allocations, and then maintains a list of free cache items in the slabs. Each time a cache item is requested, the slab allocator checks the size of the value being stored and returns a cache item from a slab large enough to accommodate. In some cases this solution wastes space, but the time saved and memory thrashing avoided offsets this disadvantage.

2.2 Commands

There are three primary commands supported by the memcached server:

1. GET to retrieve a cache item from the cache

2. STORE to add a cache item to the cache
3. DELETE to remove a cache item from the cache

Additionally, there are 15 other specialty requests including stats, replacements, arithmetic, flush, and updates. These code paths generally follow some order of the above primary commands. For instance, a replacement first DELETEs the cache item and then STOREs a new cache item into the hash table and LRU. In analysis, we focus effort on GETs, as this operation constitutes the bulk of the command requests in memcached operation. [11]

For the three primary commands (GET*,* STORE*,* and DELETE), the flow of memcached is as follows:

1. Request arrives at NIC and is processed by *libevent*
2. A worker thread
    - Takes connection and data packet
    - Determines the command and data
3. A hash value is created from the key data.
4. The cache lock is acquired to begin hash table and LRU processing. **(Critical Section)**
    - (STORE only)
        - Cache item memory is allocated.
        - Data is loaded into the cache item data structure (flags, key, and value).
    - Hash table is traversed to GET, STORE, or DELETE the cache item from the hash table.
    - LRU is modified to place move cache item in the front of the LRU (GET, STORE) or remove it (DELETE).
    - Cache item flags are updated
5. Cache lock is released. **(End of Critical Section)**
6. Response is constructed.
7. (GET only) Cache lock is asserted.
    - Cache item reference counter is decremented.
    - Cache lock is released.
8. Response is transmitted back to requesting client (i.e. Front-end Web server).

2.3 Process Flow

Figure 5 documents the process flow of memcached when processing data requests from the perspective of a memcached server in the cluster. Multiple servers work in conjunction to act as single larger logical data cache. In essence, these servers constitute a large, distributed hash table. Reviewing figure 5 from top to bottom, the clients (these are usually in the form of web tier servers building a response for a user or computational servers needing data for calculations) are responsible for submitting operation requests to memcached. When memcached is initiated, a static number of worker threads are created to process these client requests, normally aligned with the number of physical processors in the server system.

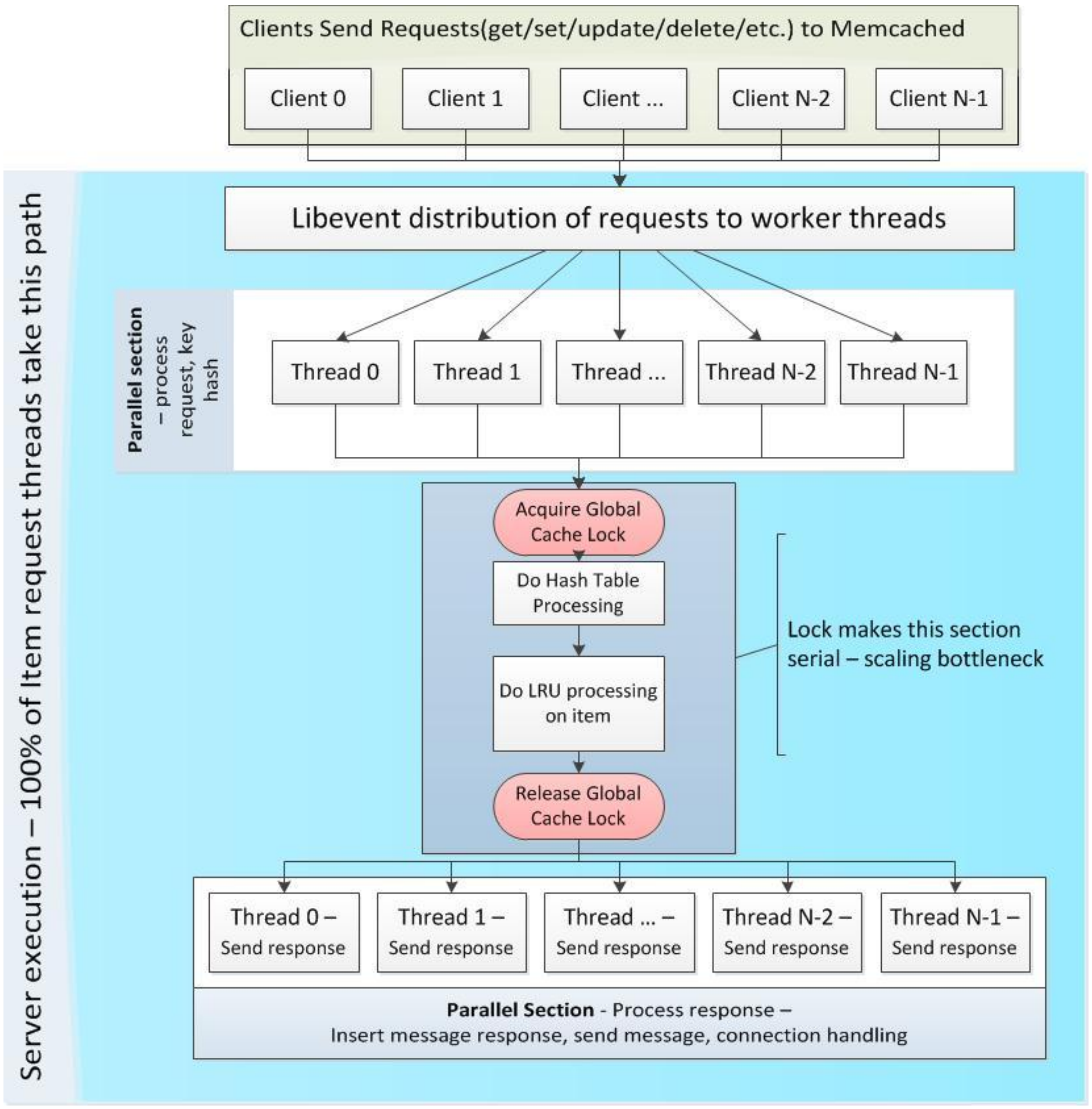

Clients Send Requests(get/set/update/delete/etc.) to Memcached
Client 0
Client 1
Client ...
Client N-2
Client N-1
Server execution – 100% of Item request threads take this path
Libevent distribution of requests to worker threads
Parallel section – process request, key hash
Thread 0
Thread 1
Thread ...
Thread N-2
Thread N-1
Acquire Global Cache Lock
Do Hash Table Processing
Do LRU processing on item
Release Global Cache Lock
Lock makes this section serial – scaling bottleneck
Thread 0 – Send response
Thread 1 – Send response
Thread ... – Send response
Thread N-2 – Send response
Thread N-1 – Send response
Parallel Section - Process response – Insert message response, send message, connection handling

Figure 5 – Process flow for cache item operations (STORE/GET/DELETE) in open-source v1.6

Client requests are distributed to worker threads for processing. For a GET request, each thread must complete a hash table lookup to identify where the value data resides associated with the key. Additionally, to indicate that the key-value is recently accessed and update the eviction order, it is moved to the front of the Least Recently Used (LRU) list. Unfortunately, both of these activities require protection (i.e. locks) for these shared data structures. After the data is located, it is retrieved from system memory by the worker thread and transmitted back to the requesting client.

Hash table thread safety is provided by a single global cache lock that is asserted for hash table access. Additionally, the LRU linked-list, maintained to manage cache evictions, uses the same global cache lock when cache items' LRU linked-list is manipulated to ensure thread safety.

Threads are initiated by *libevent*, which is a portable network protocol wrapper. Threads wait in *libevent* for a request, and upon receiving one, load the data from the packet and decode the command. Assuming a request for a cache item (GET/STORE/DELETE/ARITHMETIC), the thread computes a hash value from the key and acquires the global cache lock. The thread will hold the lock until the command is completed, then release the lock and create a response. Doing this, memcached is effectively serialized to ensure data consistency and thread safety.

2.3.1 Hash Table Lookup

After acquiring the lock, the hash table traversal and/or manipulation is executed. Serialized hash table entry and exit eliminates concern for thrashing pointers and corrupting the linked-list bucket during a STORE or DELETE. If we leave the hash table implementation as-is and remove the locks, there is a scenario where two cache items "close" to each other (i.e. adjacent in the linked-list) are being removed and, upon completion of the removals, the linked-list no longer points to cache items in the correct chain. This is illustrated in Figure 6, with each row representing a time progression.

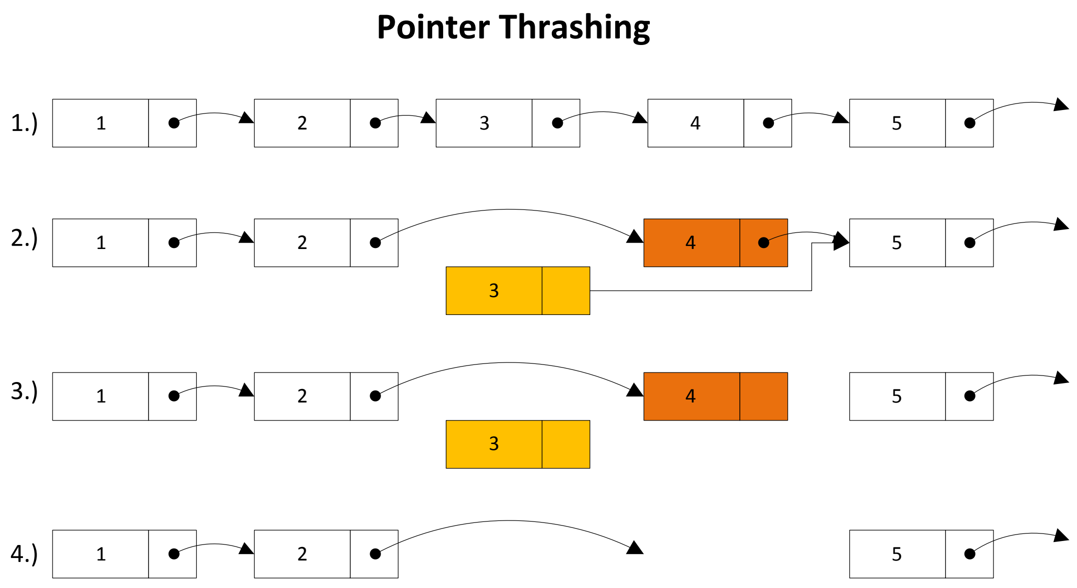


Figure 6 - Linked-list corruption from two threads removing cache items concurrently.

In the second row, one thread is removing the yellow cache item (#3) and one is removing the orange cache item (#4). They both modify the pointer of the previous cache item. In row 3, they both NULL out the *next* pointer. Finally, in the fourth row, the cache items are removed and the remaining hash chain is left with a gap in the middle of the chain, causing item #5 to be lost from the hash chain.

The lock is also required for GET requests so they are not traversing a bucket while the linked-list is changing. Traversing a linked-list during update could cause the thread to follow a pointer into nonexistent memory, causing a segmentation fault or a miss of cache item that does exist. Refer to Figure 3 for the layout of the hash table data structure.

2.3.2 LRU Manipulation

Upon completion of the hash table operation, the cache item is either added (STORE) or removed (DELETE) from the double linked-list structure (Figure 4) used for the LRU eviction scheme. When processing a GET request, the cache item is removed from its current spot in the LRU and added to the head of the list. The cache lock is held for all LRU manipulation as two cache items being removed from the same location, or inserted at the head simultaneously, can cause list corruption. This is the same corruption described above with the hash table linked-list, except with an additional pointer to corrupt.

2.3.3 Other Serial operations

The Hash table lookup and LRU Manipulation operations are the primary code segments protected by the global cache lock, but there are others. Cache evictions, allocations, and flushes also require holding the global cache lock. These operations access the either the LRU or hash table, so simultaneous access by multiple threads can cause corruption. Additionally, cache item flags that are modified by these operations are protected by the global cache lock to ensure thread safety. An example of flag use is the ITEM_LINKED flag, which needs to be cleared when removing a cache item from the hash table. The global cache lock ensures only one worker thread can manipulate this flag at a time.

2.3.4 Parallel operations

The bulk of the process flow for memcached operation requires protection by the global cache lock. Outside of this functionality, there are some operations that are completed in parallel, including: network transmission (both receive and transmit), packet decode, key hashing, and building of the data response.

2.4 Performance Bottleneck

The global cache lock is a performance bottleneck for more than four threads, as discussed in [1] and [6]. This behavior was also confirmed in testing (Figure 7 and Table 1). Figure 7 illustrates that application throughput degrades for open-source memcached past 4 cores. The lock is coarse-grained; that is, time spent in serial execution is substantial, and contention for the lock among worker threads is high.

To verify this, Intel® Vtune™ Amplifier XE[1] was used to produce stack traces on this code and indicated that a majority of the execution time spent in a lock (i.e. waiting for the global cache lock). Looking deeper into the stack trace, we determined the global cache lock was causing contention. To quickly confirm the problem, we removed the global cache lock (which we knew would be unsafe, but effective) and got a substantial throughput increase on a read-only workload.

Abundant research illustrates shared lock contention can cripple a program's parallel performance. Reference [12] provides an in-depth analysis of contention and how it affects a program and degrades

[1] Vtune is a trademark of Intel Corporation in the U.S. and other countries.

performance. We concluded that avoiding the contention for the shared global cache lock would improve parallel performance and application scalability.

Percent	Function	DSO(Dynamic Shared Object)
60.40%	_spin_lock	[kernel.kallsyms]
1.40%	ixgbe_poll	/lib/modules/…/ixgbe/ixgbe.ko
1.10%	net_rx_action	[kernel.kallsyms]
1.10%	__pthread_mutex_lock_internal	/lib64/libpthread-2.12.so
1.10%	assoc_find	/usr/local/lib/memcached/default_engine.so
1.10%	copy_user_generic_string	[kernel.kallsyms]
1.00%	tcp_sendmsg	[kernel.kallsyms]
0.90%	irq_entries_start	[kernel.kallsyms]
0.80%	__pthread_mutex_unlock	/lib64/libpthread-2.12.so
0.70%	__GI_vfprintf	/lib64/libc-2.12.so
0.70%	__audit_syscall_exit	[kernel.kallsyms]
0.60%	ip_route_input	[kernel.kallsyms]
0.60%	ixgbe_resume	/lib/modules/…/ixgbe/ixgbe.ko
0.50%	tcp_ack	[kernel.kallsyms]

Table 1 - Captured "perf top" from a server running memcached with 8 threads

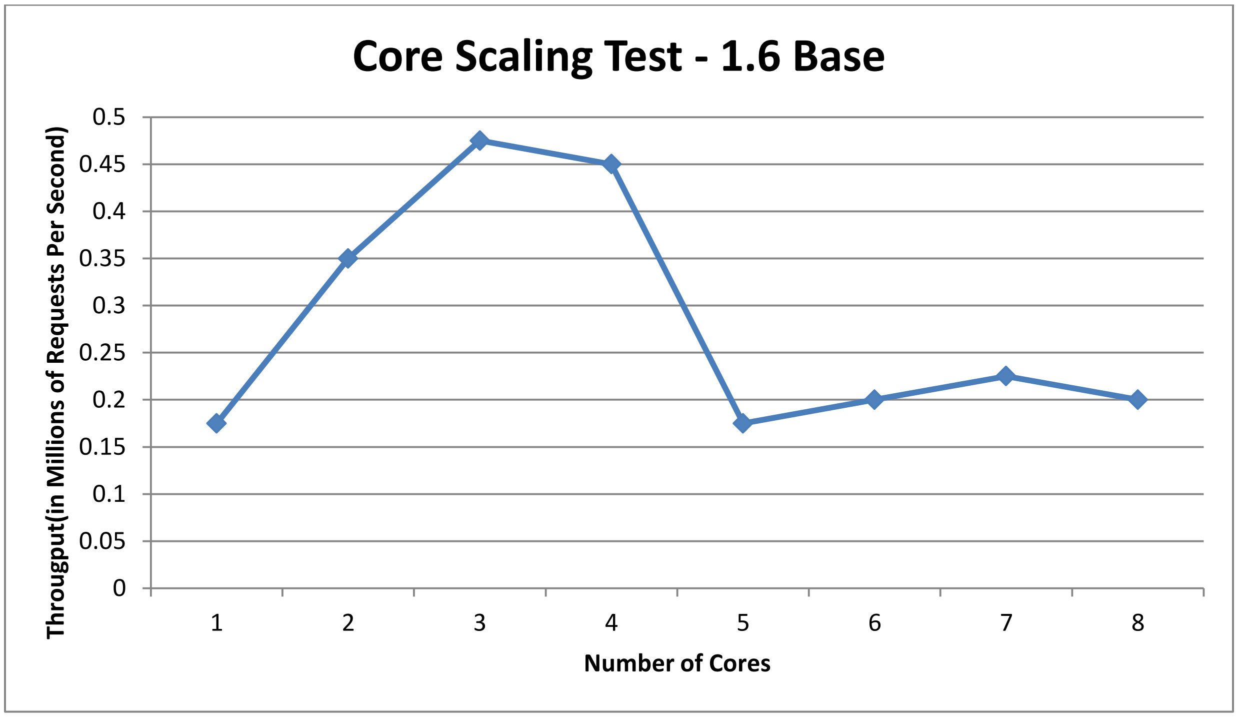


Figure 7 - Scaling performance of Base (version 1.6) memcached showing performance peak around 3-4 cores

3 Memcached Optimizations

This section outlines the optimizations to memcached to improve application scalability for multi-threaded execution on many-core servers. After examining the current memcached architecture, we investigated techniques to minimize or eliminate the global cache lock, the primary performance bottleneck.

Performance goals include:

- Equivalent performance regardless of number of keys (i.e. 1 key versus 1 million key lookups)
- Updated cache replacement strategy for LRU Eviction should provide equivalent (or better) hit rates. The expectation is a 90%+ hit rate for cached value lookups in the synthetic workload we are using.
- LRU correctness for cache management; that is, the correct value is evicted based on its LRU status.

To measure progress versus these goals, we have baseline measurements using a synthetic workload to replay a set of transactions and measure the hit rate. The optimizations described in this section have been implemented as a new "engine" in the Open Source Memcached version 1.6. As of July 2012, this optimized version is available in source code for download from GitHub at https://github.com/rajiv-kapoor/memcached/tree/bagLRU.

3.1 Data Structures

For our modifications two data structures are modified: the hash table and the LRU. The following optimizations were implemented:

- Hash table locking mechanism changed to allow for parallel access
- Bag LRU – The data structure is changed to an array of different sized LRU's with Single linked-list bags (i.e. collections) of cache items.

While we are changing the algorithms in the hash table, we make no physical changes to the hash table data structures. We implement *striped locks* which are required to modify the hash table buckets. A striped lock is a fine-grain set of locks (versus the global hash table lock) implemented on sections of the hash table. For this implementation, we first create and initialize a set of Z locks, where Z is a power of 2. Keeping this a power of 2, a bitmask is used to calculate the lock number quickly. To traverse the linked-list, the lock guarding that bucket is acquired the performing a bit-wise AND of the value *Z-1* with the bucket to-be-locked. This protocol provides a shared lock for every Zth bucket (Figure 8), reducing contention for hash table locks. In testing, Z=32 was a suitable value for the number of locks.

The LRU is where the bulk of the data structure changes are required. To build a more parallel LRU to take advantage of the parallel hash table, we researched ideas for parallel LRU data structures. There are two good ideas for lock-mitigating LRU algorithms and data structures described in [13] and [14].

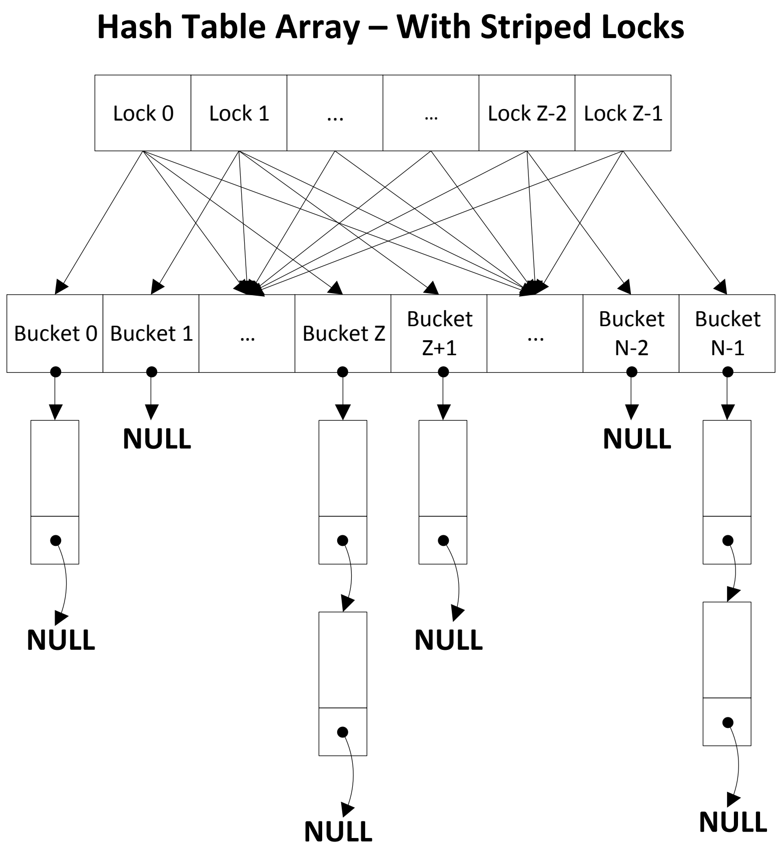


Figure 8 - Addition of the striped locks that guard each bucket

Taking inspiration from both of these, and the concept of a Bag LRU, we implement single linked-list "bags" of cache items. Each "bag" contains cache items with similar insertion or last-touched times, and can be inserted atomically. There is a cache item "clean-up" functionality required with this model; a *cleaner thread* exists for this purpose. It evicts expired/invalid cache items and performs maintenance on the bags. A more detailed explanation of the new data structure including diagrams and logic is in section 3.5.

3.2 Commands

There are no changes to any of the memcached commands or protocols described in section 2.2. The difference is solely in the methodology for command execution.

3.3 Process Flow

Figure 9 relates the updated process flow with modifications. From top to bottom, the client, libevent, and worker thread functionality remains the same. The serialized operations for hash table lookup and LRU processing are modified to operate in a parallel fashion.

DELETE and STORE operations now use a parallel hash table approach with striped locks (explained earlier) (Figure 8), and GET operations now execute non-blocking and parallel.

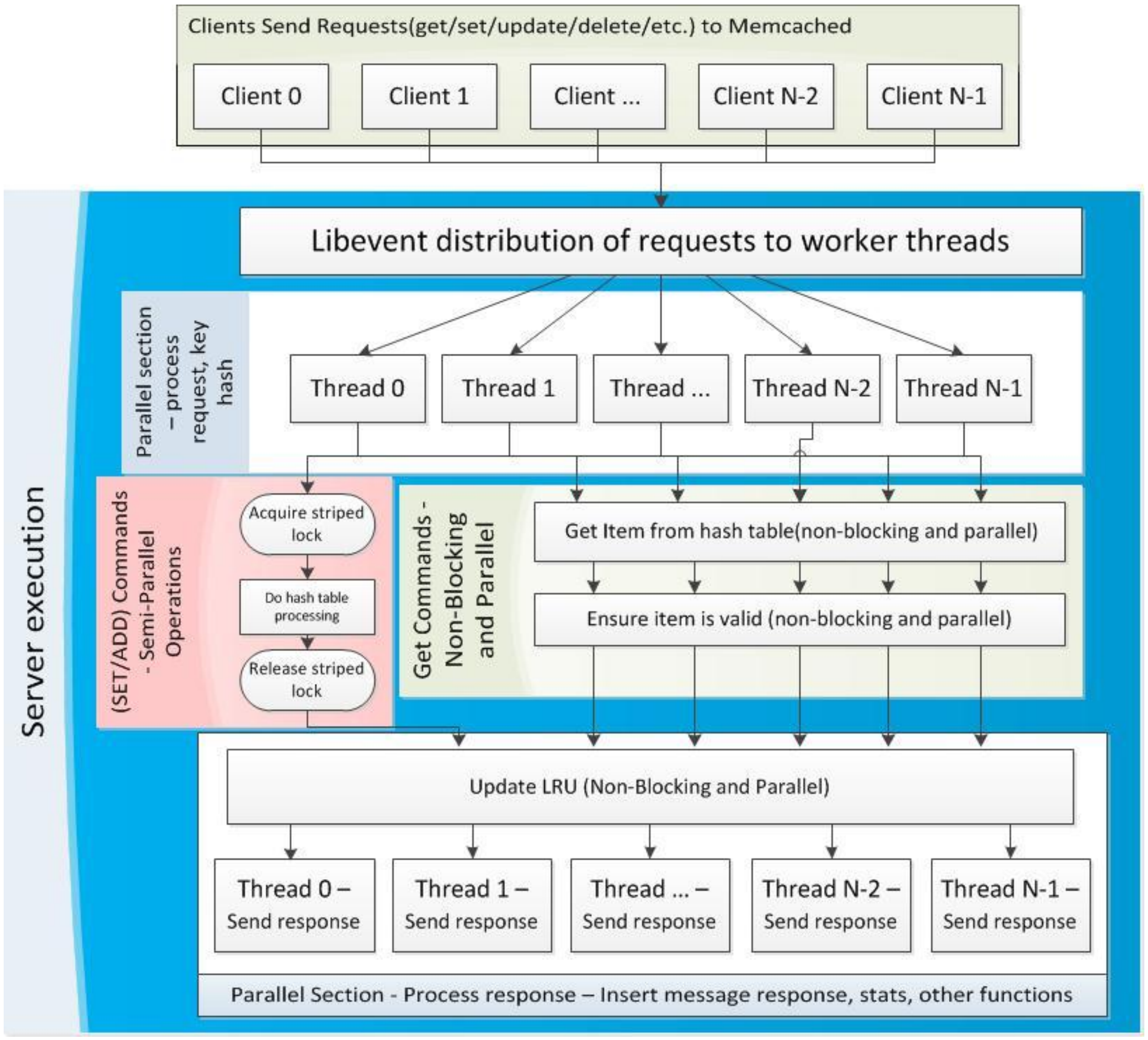

Clients Send Requests(get/set/update/delete/etc.) to Memcached
Client 0
Client 1
Client ...
Client N-2
Client N-1
Server execution
Libevent distribution of requests to worker threads
Parallel section – process request, key hash
Thread 0
Thread 1
Thread ...
Thread N-2
Thread N-1
(SET/ADD) Commands - Semi-Parallel Operations
Acquire striped lock
Do hash table processing
Release striped lock
Get Commands - Non-Blocking and Parallel
Get Item from hash table(non-blocking and parallel)
Ensure item is valid (non-blocking and parallel)
Update LRU (Non-Blocking and Parallel)
Thread 0 – Send response
Thread 1 – Send response
Thread ... – Send response
Thread N-2 – Send response
Thread N-1 – Send response
Parallel Section - Process response – Insert message response, stats, other functions

Figure 9 - Process flow for cache item operations (STORE/GET/DELETE) in optimized version of 1.6.0_beta1

The algorithmic changes, described below, prevent hash chain corruption and maintain pointers of cache items being removed. In the event a thread is pre-empted on a removed cache item, it will likely continue in the correct bucket. There is a slight nuance to the unblocked GET. If a thread is preempted for a long duration on a cache item, and that cache item is moved, the GET can return a “false negative” (i.e. memcached will return “no cache item found” when the cache item is actually in the hash table).

The pros and cons of this were weighed, and, since memcached is a cache for data that should be persistent elsewhere (e.g. Database Tier); the extra throughput realized from removing the locks outweighs the possibility of a false negative. If there is a case where the false negative presents a problem, you could use the striped locks, which minimally decrease performance. However, the expectation is, as core counts increase, that striped locks could again become a point of contention.

LRU processing is now non-blocking and parallel, as described above in section 3.2, and below in section 3.5. In bags, a single linked-list of cache items is stored so they can be inserted atomically, removing the need for a lock while placing a new cache item into a bag. There is still a Compare-and- Swap (CAS) command required, which could be a point of contention if only STOREs are performed. However, optimizing for the majority of requests (i.e. GETs), the Bag LRU is implemented with no locks or CAS required for GETs.

3.4 Parallel Hash Table Lookup

The first performance impediment encountered when executing a memcached transaction is the serialized hash table lookup. Previous work on increasing parallel performance of memcached has focused on a partitioned hash table in which threads access only specific portions (i.e. partitions) of the cache. This works well for removing lock contention and increasing performance as seen in [16]. The disadvantage of this approach is that multiple frequently accessed cache items in a single partition can overwhelm the threads available to service that partition. Similarly, traffic patterns that favor a partition can be starved for processing power. Overall performance of a partition is limited to the threads that can access that cache without causing lock contention.

This implementation optimizes throughput for the majority of traffic, GET operations. Reference [11] shows that traffic on the majority of Facebook's cache servers are GET requests. This balance is as high as 99.8% GETs in some cases, with only a few caching tiers being write-dominated. This is consistent with the performance testing done by [15] and [16] as both testing methodologies only measure throughput for GET requests.

With this in mind, a parallel hash table is designed that doesn't require any locks when processing GET requests. Lock contention, no matter how small, impacts performance (transaction latency, overall throughput under Latency SLA), as shown in [12]. In the future, as CPUs execute faster and core-counts increase, lock contention will increase. The new designs accounts for these trends by completely eliminating lock contention for GETs.

3.4.1 GET Lock Removal

In order to remove locks on GETs, there are two main cases to address: expansions and hash chain modifications (i.e. insertions and removals).

An *expansion* is required when the hash table has a much larger amount of cache items then buckets. In this case, adding more cache items will increase the length of the hash chain, increasing the time required to find a cache item and RTT. Instead of suffering this time penalty, a new hash table with twice the amount of buckets is allocated, as follows:

- all cache items are taken from the old table
- the key is hashed
- the cache item is inserted into the new table

***Assoc_get* logic:**

```
1.  mask hash to get the bucket
2.  If (!expanding)
3.      Check hash chain bucket – return cache item if found
4.      If (cache item not found)
5.          If(expanding && expansion bucket is greater than the bucket)
6.              Find bucket in new larger table
7.              Wait for old bucket to be empty
8.              Check new hash bucket – return cache item if found
9.  else if expanding
10.     Ensure bucket is in old table
11.     Check hash chain bucket – return cache item if found
12.     If (cache item not found)
13.         If(expansion bucket is greater than the bucket)
14.             Find bucket in new larger table
15.             Wait for old bucket to be empty
16.             Check new hash bucket – return cache item if found
```

To accomplish cache item retrieval, a check for an expansion is required. If there is no expansion in progress, the GET request executes normally and returns the cache item, if found. However, there is the possibility the expansion started after the initial check and moved the cache item. In this case, the process waits for the old bucket (the bucket where the item would be located in the previous hash table) to be empty and then checks the new hash table and bucket for the cache item.

If the hash table is currently being expanded, there is a check to see if the old bucket has been moved already. If the old bucket has not been moved, it is checked for the cache item. Like before, if the cache item isn't found, a check is required to see if a "bucket move" occurred on the old bucket. If so, the new hash table will then be checked for the item.

The second case to address, hash chain modifications, is handled in the STORE and DELETE code. As long as pointers are changed in correct order, GETs will have no problem with hash chain traversal.

3.4.2 STORE/DELETE Lock Removal

The STORE and DELETE commands are very similar to the current open-source implementation. To ensure safety in hash chains and multiple inserts/deletes on the same chain or cache item, STOREs and DELETEs use the striped locks that we described earlier (Figure 8).

***Assoc_delete* logic:**

```
1.  Determine bucket of cache item
2.  Get the bucket lock
3.  If(expanding)
4.      If(expansion bucket is greater than the bucket){
```

```
5.              Release the bucket lock
6.              Determine new bucket
7.              Get bucket lock
8.          Delete cache item from bucket if found
9.          Release bucket lock
10. Else if not expanding
11.         Delete cache item from bucket if found
12.         Release bucket lock
```

The striped lock is required for all hash chain manipulation, including moving buckets during an expansion. This makes STOREs and DELETEs simpler since they get the bucket lock and then check for an expansion. If the expansion is moving the bucket the STORE or DELETE is going to manipulate, they are locked out until the old bucket is empty and know to check the new one. If the move started after the lock was acquired, then the expansion waits until the lock is released and can be acquired by the expansion to move the bucket. The logic is identical for contending STOREs, DELETEs, and MOVEs as they are all locked out of the buckets until the manipulations are complete.

***Assoc_insert* logic:**

```
1.  Determine bucket of cache item
2.  Get the bucket lock
3.  If(expanding)
4.          If(expansion bucket is greater than the bucket){
5.              Release the bucket lock
6.              Determine new bucket
7.              Get bucket lock
8.          Search bucket, if cache item not found, insert into bucket
9.          Release bucket lock
10. Else if not expanding
11.         Search bucket, if cache item not found, insert into bucket
12.         Release bucket lock
```

3.5 Parallel LRU Eviction

A parallel replacement for the double linked-list LRU implementation is required to determine cache item eviction order. The concept behind the "bag" implementation is to group cache items into "bags" based on their timestamp. In these bags, a single linked-list of cache items is stored so they can be inserted atomically to the end of the list using a CAS. This removes the need for a lock while placing a new cache item into a bag although the CAS could be a point of contention if only STOREs are performed. Again, optimizing for the majority (GET requests), the Bag LRU is implemented with no locks or CAS for GETs.

The new design also needs a way to evict old cache items and maintain new bag LRU structures. To accomplish this, we implement a *cleaner thread* to run in the background. This thread removes expired cache items, invalid cache items, and performs maintenance on the LRU. Work that GET requests were doing in the old version (i.e. Removal on expirations) is moved to this background thread, increasing the throughput of GET requests and decreasing overhead per GET request transaction. The cleaner thread also provides a benefit of increasing hit rates of the cache. Cache items inserted with quick expirations are now cleaned up within a few minutes of expiring, instead of taking space until aging out of the cache.

3.5.1 Linked-list structure

The linked-list structure reuses the *previous* and *next* pointers from the current LRU, so no extra memory allocation is required. The *next* pointer still points to the next cache item in the new single linked-list and the *previous* pointer is used to mark the cache item's bag membership which is the *newest bag* at the time the cache item was last accessed. The cleaner thread uses the residence pointer while cleaning the bags.

3.5.2 Additional Data structures

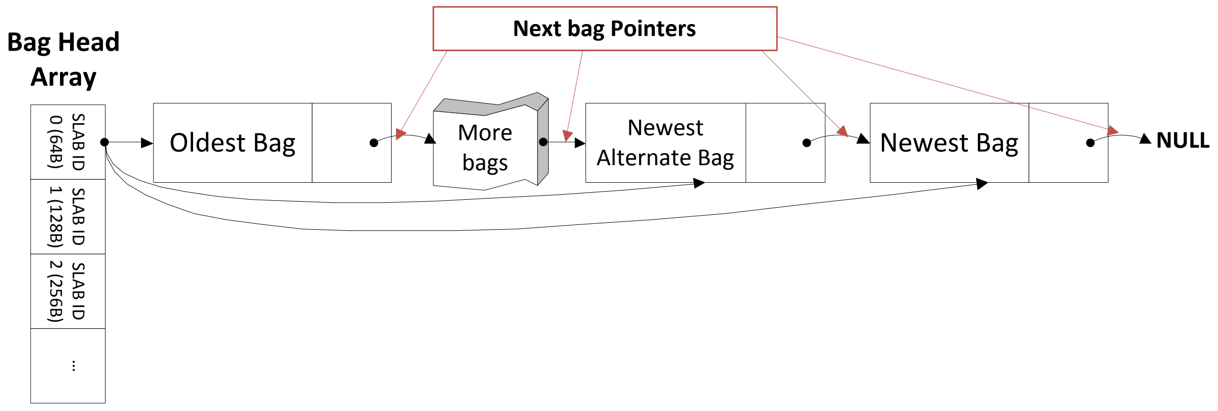


Figure 60 - Bag data structure

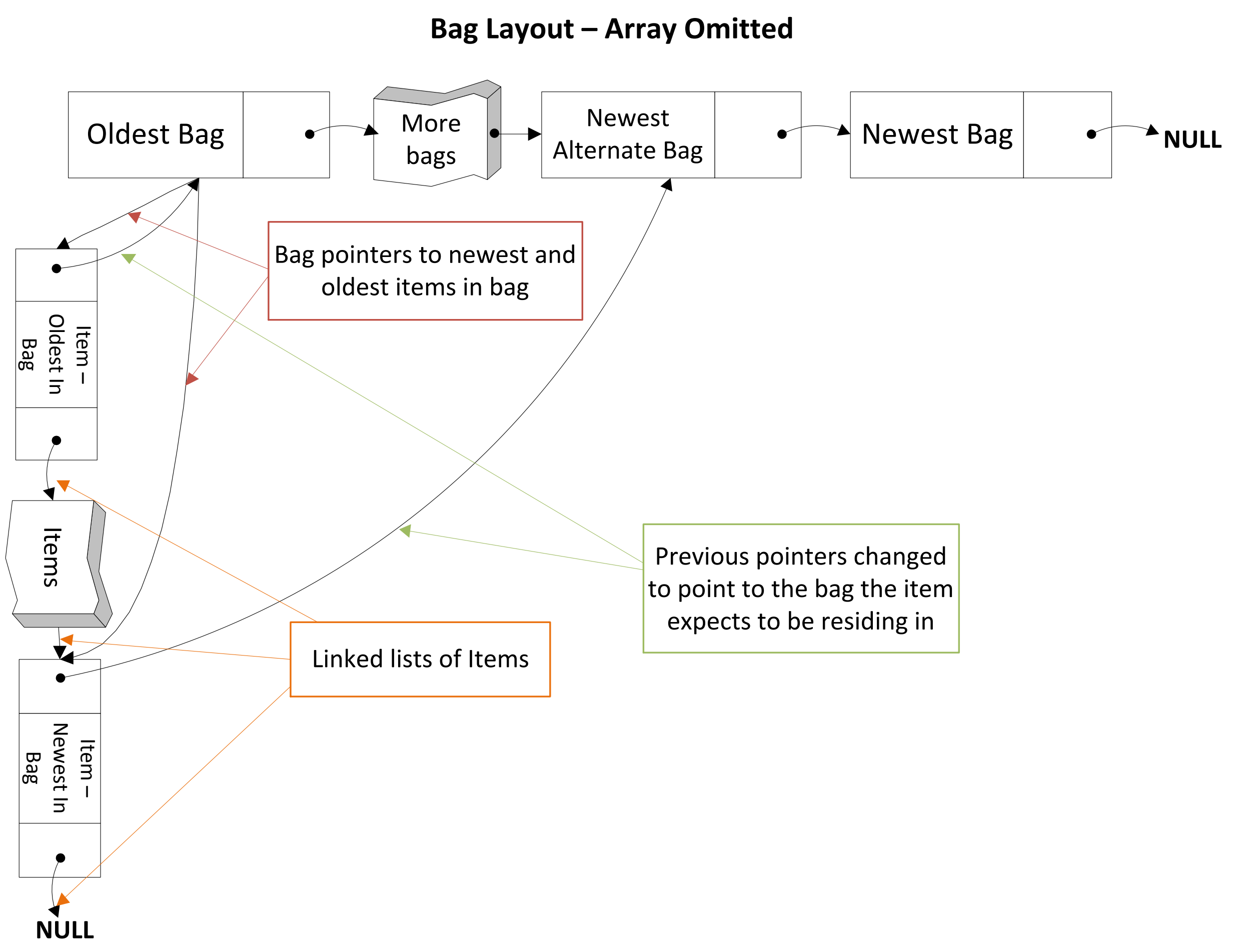

Bag Layout – Array Omitted
Oldest Bag
More bags
Newest Alternate Bag
Newest Bag
NULL
Bag pointers to newest and oldest items in bag
Item – Oldest In Bag
Items
Previous pointers changed to point to the bag the item expects to be residing in
Linked lists of Items
Item – Newest In Bag
NULL

Figure 11 – Bag List data structure

There are two data structures required for the bag LRU, an array of LRUs for bag identification, and the actual bag data structure.

The first data structure is the bag LRU list, an array of bag heads, with one per slab id (Figure 10). These structures are referenced in a global variable and initialized when the LRU is initialized. Each bag head in this array maintains:

- A pointer to the newest bag
- An alternate to the newest bag
- The oldest bag in the bag list
- A count of the number of bags for statistics

The newest bag is the bag that is currently filled with newly-allocated cache items. The oldest bag is where the cleaner thread starts cleaning bags and where eviction threads evict cache items from. The newest alternate bag is used by the cleaner thread to avoid locks and contention with *inserts* into the newest bag. Any cache item that already exists in a bag, but has been accessed recently and needs to be moved to the newest bag by the cleaner thread to maintain eviction order (the one currently being filled), will be placed into the newest alternate bag.

The second data structure created is a bag, which maintains a single linked-list of cache items (Figure 11). Each bag has:

- A pointer to the newest and oldest cache items in the bag
- A counter for the number of cache items in the bag
- A lock
- Bag open and close timestamps that are currently set, but not used.

When cleaning a bag, the cleaner thread begins its operation from the *oldest* cache item in the bag. The evictions done by worker threads begin in the same location. The *newest* cache item pointer is to allow for cache items to be added to the end of the bag quickly when being inserted or as bags are merged. The *count* is for determining when to open a new bag and when to merge an old bag into a newer one. This *count* can also be used for statistics and debugging as needed. The reason for a lock on each bag is to ensure thread safety between the cleaner thread and evictions done by worker threads. If an eviction thread and a cleaner thread are moving cache items in the same bag, it is possible for one of the threads to corrupt a bag (we have shown linked list corruption in Figure 6). Since the cleaner thread spends very little time in each bag, it is unlikely that there is contention on this lock between cleaner and worker threads. Also, since worker threads are serialized higher up in the call stack, so there is no contention on this lock between these threads.

3.5.3 Operations on the Data Structures

This section reviews operations on the new data structures and algorithms for initialization, inserting cache items, and cleaning the LRU List.

First, all added locks need to be initialized and an initial bag is created for each slab size to be used. After all the structures are initialized, the cleaner thread is created and launched in a cleaning loop.

Bag LRU initialization

```
1. Initialize the global eviction lock
2. Initialize the cleaner locks – (currently only one lock could be one cleaner thread per slab id)
3. Create the array of bag head structures
4. For each slab id
5.     Initialize the first bag
6.     Initialize the bag lock
7.     Point the bag array head to this as the newest bag
8.     Increment the bag counter
9. Create and launch the cleaner thread
```

In the cleaning loop, the thread continually checks to see if bags are full and adds new bags to those slabs as needed. Periodically, the cleaner thread traverses every cache item in the bags, removing invalid or expired cache items, and merging bags below a minimum cache item count threshold.

Cleaner thread loop

```
1.  Loop forever
2.      Acquire cleaner lock
3.      For each slab id
4.          if the newest bag is full
5.                  create and initialize a new bag
6.                  point the newest bags next bag pointer to the newly initialized bag
7.                  set a bag holder = newest bag
8.                  Newest bag = newly initialized bag
9.                  Newest alternate = bag holder
10.                 atomic increment of bag count
11.     Every Nth iteration, clean the bags
12.     Release cleaner lock
13.     Sleep for a given interval
```

When a cache item is first inserted into a LRU, it goes into the newest bag available for that cache item's slab size. Once the bag is located, the cache item is placed at the end of the *newest* cache item in the newest bag using a CAS. If this fails, the thread traverses the chain for that bag until it finds a NULL pointer (indicating the end of the bag), and tries the CAS again. This continues until the CAS passes indicating a successful insertion.

Inserting New Cache item not already in LRU

```
1.  New cache item next pointer = NULL
2.  If(current open bag is empty)
3.      If( CAS (current bags newest cache item, NULL, new cache item)){
4.          New cache item previous pointer = current bag
5.          Current bag newest cache item = new cache item
6.          Atomic increment bag cache item counter
7.          Return
8.  Swap cache item = current bags newest cache item
9.  While( !CAS(swap cache item next pointer, NULL, new cache item)){
10.     While(swap cache item next pointer != NULL)
11.         Follow the swap cache item next pointer until you find a null and try the swap again
12. New cache item previous  pointer = current bag
13. Current bag newest cache item = new cache item
14. Atomic increment bag cache item counter
```

For a GET, since cache item already exists in the LRU, we are simply touching the cache item to update its timestamp and location in the LRU. The touch copies the new timestamp into the cache item *last accessed* field and points the cache item's *previous* pointer (the bag the cache item should now reside in) to the newest bag in that slab size.

Updating Cache item already in the LRU

1. Cache item previous pointer = current bag
2. Update cache item timestamp

3.5.3.1 Cleaning the LRU list

Within the context of cleaning the LRU list, there are several operations to support; including cache item reordering, merging buckets, cache item eviction, and dumping the cache.

As the cleaner thread traverses the bag lists, it checks the cache item *previous* pointer. If the pointer points to the current bag it is cleaning, it leaves the item in its current location. If the item points to different bag, that means that the item has been touched since it was first inserted. In this case, the cleaner thread will remove the item from its current bag, and insert it into the bag it should be residing in. As described above, if this is the *newest bag* in the LRU size, it will be placed into the *newest alternate* to avoid contention.

Merging two buckets

```
1.  Older buckets newest cache item next pointer = newer buckets oldest cache item
2.  Newest buckets oldest cache item = older buckets oldest cache item
3.  For each cache item we added
4.      If the cache item was pointing to the old bag
5.          Point it to the new bag
6.  Increment the newer bags counter by the amount of cache items in the older bag
```

The logic to merge two buckets is to concatenate the two linked-lists together, put the combined linked-list into the newer of the two bags. Then it must ensure every cache item that had its *previous* pointer pointing to the old bag now points to the newer bag and reclaim the older bag.

Evicting a cache item

```
1.  Get the eviction lock for the slab id
2.  Get the oldest bag from the slab id
3.  Find the first bag in the list that is before the newest alternate and has cache items in it
4.  if we can't find the previous bag, add two new bags to this slab id, then check those two bags
5.  if we still can't find a bag
6.      stop trying to evict cache items just return a failure
7.      release eviction lock
8.  lock the bag we found
9.  starting at the oldest cache item in the bag
10. try to find an cache item with a refcount of 1 or 0 or a delete locked cache item in the first N
    cache items of the bag
11.     if cache item is found evict it and free it
12.     try and get the newly free cache item from the slab allocator
```

```
13.     if we evicted an cache item but didn't get it from the allocator –
14.         evict another cache item and try again
15.     else
16.         stop trying to evict cache items just return a failure
17. unlock the bag
18. release eviction lock
```

To evict a cache item from the cache, the worker threads are serialized with a global eviction lock to ensure correctness. The bag is then locked to avoid conflicts with the cleaner thread. After this is done, the thread is the only thread touching the bag and can remove any cache item that meets the eviction criteria (which is the same as the open-source version).

Dumping the cache

```
1. get cleaner lock
2. get eviction lock
3. add new bags so we can evict without effecting other threads
4. for each slab id
5.     for each cache item
6.     If (cache item older than the cache dump time)
7.         Free cache Item
8. release eviction lock
9. release cleaner lock
```

Dumping the cache is the same as the eviction of a cache item as before, but this time all cache items older then the time specified in the command are evicted.

3.6 Removal of global cache locks

With functionality in place to manage parallel hash lookups, LRU ordering, and parallel cache evictions, the global cache locks are nearly ready to be removed from the GET/STORE/DELETE code paths. Management of reference counts must by changed; instead of a lock to change the reference count, atomic operations are used to increment and decrement the count. Upon completion, the global cache lock/unlock pairs are removed from the operations, as the global lock is no longer necessary.

1. ~~Lock (global_cache_lock)~~ ← Remove these locks
2. . . . . . (GET/STORE/DELETE)
3. ~~Unlock (global_cache_lock)~~ ← Remove these unlocks

4 Test Environment Configuration and Metrics

With the coding and debugging of the new memcached architecture complete, a setup and testing methodology is defined to determine performance gain.

4.1 Hardware Configuration

This section reviews the hardware configuration of all the systems being used.

4.1.1 System under Test (i.e. SUT)

Based on early evaluation, a 1 GbE (Gigabit Ethernet) NIC is easily saturated, so a 10GbE NIC is required to remove the network as a performance bottleneck.

Memcached is primarily run in large server farms where Total Cost of Ownership (TCO) is important. The objective is to maximize performance/watt results to decrease the TCO. The test system is a power-optimized 2-Socket Open Compute 2.0 System Board manufactured by Quanta Computer.

System Hardware:

- Dual-Socket Open Compute 2.0 System board (half width) in 1.5U chassis[2]
- Quantity 2 -- Intel® Xeon® E5-2660 2.2Ghz Processors (95W TDP) [10]
- 128 GB DRAM – Quantity 16, 8 GB DDR3 PC3 12800 ECC
- Intel® "Niantic" 82599 10Gb Direct Attach SFP+ Ethernet Controller [10]
- Arista[3] 7124 Switch – 24 Port SFP+[4]

4.1.2 Load generators

With the open-source version of memcached on the previous generations of servers, a single load generator is sufficient. However, with the modifications, an increase in the number of clients is required to stress a single SUT. Using one client, it is possible to get approximately 800K requests per second (RPS); therefore, we configure four clients for driving the load. All load generators are configured identically and confirmed individually to be capable of generating 800K+ RPS with the following Hardware configuration:

- Dual-Socket Open Compute 2.0 System board (half width) in 1.5U chassis[5]
- Quantity 2 -- Intel® Xeon® E5-2660 2.2Ghz Processors (95W TDP) [10]
- 16 GB DRAM – Quantity 2, 8 GB DDR3 PC3 12800 ECC
- Intel® "Niantic" 82599 10Gb Direct Attach SFP+ Ethernet Controller [10]

4.2 Software (SW) Configuration

This section reviews the SW configurations for the SUT and client systems, network optimizations, and power monitoring configuration.

4.2.1 Test System Configuration

The Following software is installed to run the Memcached tests:

[2] http://www.opencompute.org
[3] Other brands and names are the property of their respective owners.
[4] http://www.aristanetworks.com/en/products/7100s
[5] http://www.opencompute.org

- CentOS (Community ENTerprise Operating System), version 6.2 – Linux Kernel 2.6.32[6]
- Libevent 1.4.14b-stable
- Intel® ixgbe 3.8.14 NIC driver
- Memcached-1.6.0_beta1 and Memcached-1.6.0_beta1-bags

After SW install, memcached is run with the following command line

- Memcached –p 11211 -u nobody –t <*thread*> -m 128000 –d

With this command, *thread* is the number of worker threads to run in Memcached. For all tests, the only variable modified for testing memcached is the number of threads; this workload driver and input parameters are utilized to determine peak throughput and scalability.

4.2.2 Client System Configuration

The client systems are configured with the following software:

- CentOS (Community ENTerprise Operating System), version 6.2 – Linux Kernel 2.6.32[7]
- Intel® *ixgbe* (10 GbE) version 3.8.14 NIC driver[8]
- Mcblaster[9]

With the operating system and network driver software installed, mcblaster is utilized to drive a synthetic cache lookup workload and measure performance (i.e. transaction throughput, latency/response time) of the memcached instance. There are two steps involved in this testing. First, the keys and values must be loaded into the memcached instance, using:

- ./mcblaster –p 11211 -t 16 -z 64 –k 1000000 –r 10 –W 75000 –d 20 <memcached host>

This command loads the cache with 1 million keys, all with 64 Byte values. In the next phase of testing, all clients send requests for those keys, in random order, simultaneously at a given rate to measure throughput and latency at that rate.

- ./mcblaster –p 11211 -t 16 -z 64 –k 1000000 –r <rate> –W 0 –d 120 <memcached host>

With these tests, the *rate* is increased as memcached's average RTT is monitored. Maximum throughput is the highest *rate* achieved without exceeding the 1ms average RTT SLA.

Using previous work on measuring performance of memcached [15] and [16] as a guide, all measurements are gathered using GET requests. Many key-value caching servers are GET-dominated [11] and the number of STOREs in the workload mix has little impact on GET performance [16].

[6] http://www.centos.org/
[7] http://www.centos.org/
[8] http://sourceforge.net/projects/e1000/files/ixgbe%20stable/ - 3.8.14 was replaced by 3.8.21 (2-24-2012)
[9] https://github.com/fbmarc/facebook-memcached-old/tree/master/test/mcblaster

4.2.3 Network Setup

For maximum throughput, there are two different networking configurations depending on the Intel® Hyper-Threading (HT) Technology setting (enable/disable).

The following setup is used when Intel® HT Technology was disabled:

- Reduce the Ring buffer size to 96 for both rx (receive) and tx(transact, send)
    - Goal: Reduce latency (RTT)
    - Command
        - *ethtool –G ethX rx 96*
        - *ethtool –G ethX tx 96*
- Disable generic receive offload
    - Goal: Reduce latency (RTT)
    - Command: *ethtool –K ethX gro off*
- Affinitize NIC interrupts to CPUs by executing the set_irq_affinity.sh[10] shell script
    - Goal: Reduce network interrupt contention
    - Command: *./set_irq_affinity.sh ethX*
    - Note: Ensure that the *irqbalance* service is disabled in the kernel or it will overwrite these changes. (killall irqbalance)

These settings provide best throughput when combined with the latest ixgbe driver (3.8.14 when the test was run). Flow director (included with the ixgbe driver) is used to steer incoming packets evenly across the physical CPU cores distributing the load of network interrupts. This allows CPUs to maintain balanced utilization across cores.

With Intel® HT Technology enabled, the optimal network configuration uses 1 memcached thread per physical CPU and 2 NIC queues which are now affinitized to the additional 16 logical cores (Table 2). Using process affinity, this configuration allows a majority of the NIC handling to be off-loaded to the logical cores adjacent to the Memcached threads, providing a boost in performance vs. Intel® HT Technology disabled.

	Socket 0							
	Phys. Core 0	Phys. Core 1	Phys. Core 2	Phys. Core 3	Phys. Core 4	Phys. Core 5	Phys. Core 6	Phys. Core 7
Logical Core 0	Memcached 0	Memcached 1	Memcached 2	Memcached 3	Memcached 4	Memcached 5	Memcached 6	Memcached 7
Logical Core 1	RxTx IRQ 0 & 16	RxTx IRQ 1 & 17	RxTx IRQ 2 & 18	RxTx IRQ 3 & 19	RxTx IRQ 4 & 20	RxTx IRQ 5 & 21	RxTx IRQ 6 & 22	RxTx IRQ 7 & 23

	Socket 1							
	Phys. Core 0	Phys. Core 1	Phys. Core 2	Phys. Core 3	Phys. Core 4	Phys. Core 5	Phys. Core 6	Phys. Core 7
Logical Core 0	Memcached 8	Memcached 9	Memcached 10	Memcached 11	Memcached 12	Memcached 13	Memcached 14	Memcached 15
Logical Core 1	RxTx IRQ 8 & 24	RxTx IRQ 9 & 25	RxTx IRQ 10 & 26	RxTx IRQ 11 & 27	RxTx IRQ 12 & 28	RxTx IRQ 13 & 29	RxTx IRQ 14 & 30	RxTx IRQ 15 & 31

Table 2 – Logical Cores share Memcached and Network traffic in an Intel® HT Technology memcached setup

[10] This script is provided in the ixgbe driver download.

4.2.4 Power measurements

For testing, power is monitored with a Yokogawa WT210 digital power meter[11]. The wattage is collected every 5 seconds while the system is under load. The power is then averaged to give the power readings listed in the charts in the next section.

As a note, all of these servers were run with a video card and a less-than-optimal cooling solution. We expect less power consumption per server in a data center environment versus the numbers reported in this paper.

5 Findings

To summarize the performance gains, Figure 12 provides the maximum server capacity (i.e. RPS) for the SUT while maintaining the Latency SLA of <1ms RTT (Round-trip time). Consistent with our research, these measurements are taken with GET requests.

Testing confirmed that the optimized memcached performs best with one thread per physical processor, so optimized test results use 16 threads to align with 16 physical cores in the SUT. Intel® HT Technology and Turbo mode both provide performance upside when enabled.

The two open-source baseline results illustrate no performance gain from increasing 8 to 16 threads (i.e. lack of scalability due to lock contention). The optimized "Bags" results show increasing performance gains as we add Intel® HT Technology (+8%) and Turbo (+23%). The performance improvement with Intel® HT Technology is attributed to additional logical processors handling the network interrupts concurrently with worker threads on the same physical core, removing the need for context swaps. Turbo mode provides an increase in processor frequency, contributing a significant performance improvement as well. Our best throughput result, at 3.15M RPS, is achieved with Intel® HT Technology and Turbo enabled (+31%), with each feature providing an additive increase in performance. When we compare the best optimized result (3.15M) with an identically-configured baseline (.525M), we realize a 6X performance gain.

Figure 13 relates the average round-trip time (RTT) for the four optimized memcached cases from Figure 12. It is interesting to note that there is a point (260 us, 285 us, 215 us, and 205 us, respectively) where RTT degrades immediately past the 1 ms SLA.

The reality is that, regardless of RPS load, the response time SLA must be managed to less than 300 us to avoid the point on the curve where RTT degrades; however, our "capacity" or RPS per server provides consistent RTT under the 300us threshold, allowing each server to be fed millions of RPS to maximize per-server throughput.

Figure 14 shows the maximum RPS throughput with a median RTT < 1ms SLA as core counts increase. Turbo was enabled and Intel® HT Technology disabled for these tests. As core counts are increased,

[11] http://tmi.yokogawa.com/products/digital-power-analyzers/digital-power-analyzers/wt210wt230-digital-power-meters/#tm-wt210_01.htm

memcached scales linearly due to parallel data structures and algorithms operating on many-core processors. We expect this linear scaling to continue on next-generation many-core processors, as there is no blocking required between threads for execution of the GET requests.

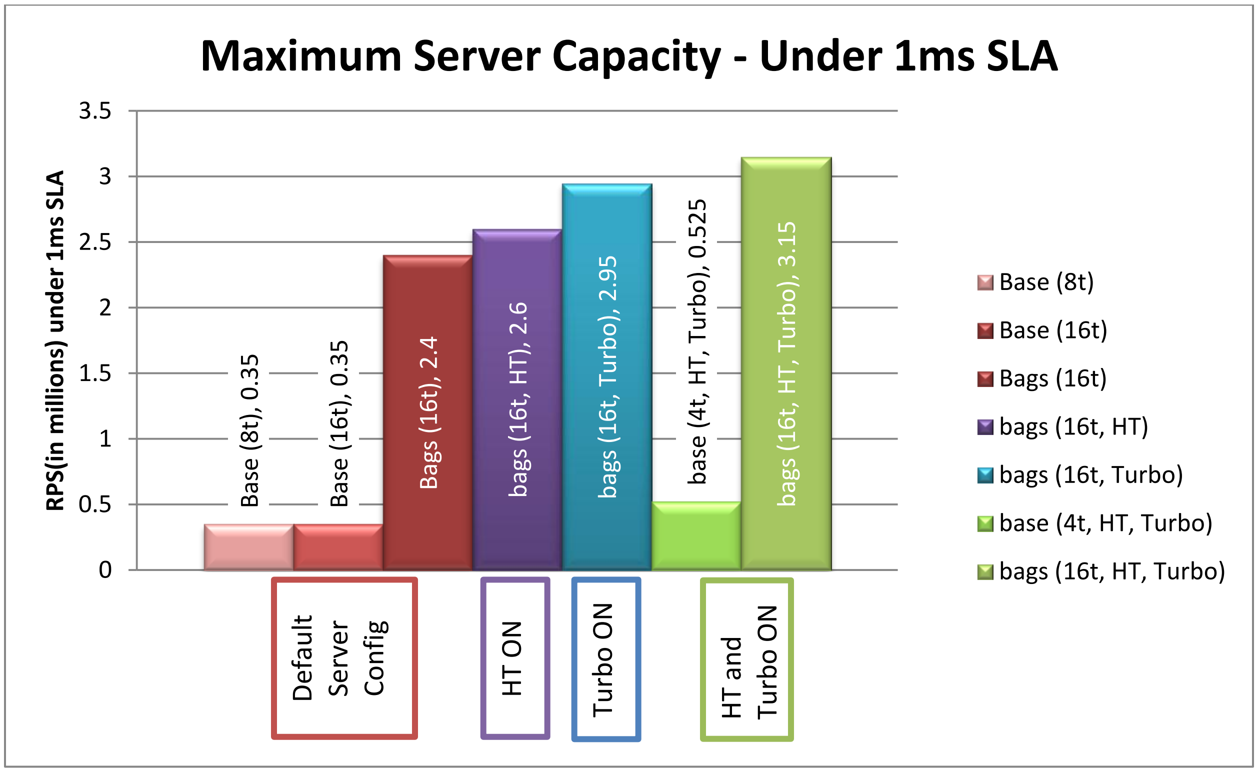


Figure 72 - Maximum GET requests while maintaining <1ms RTT SLA

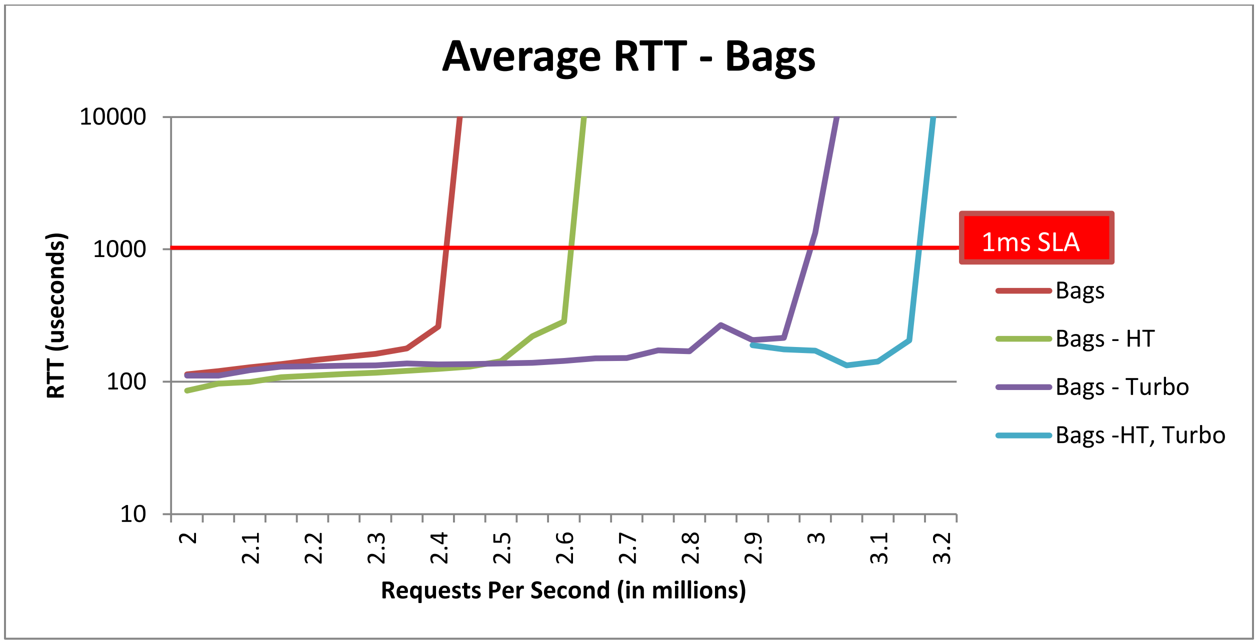
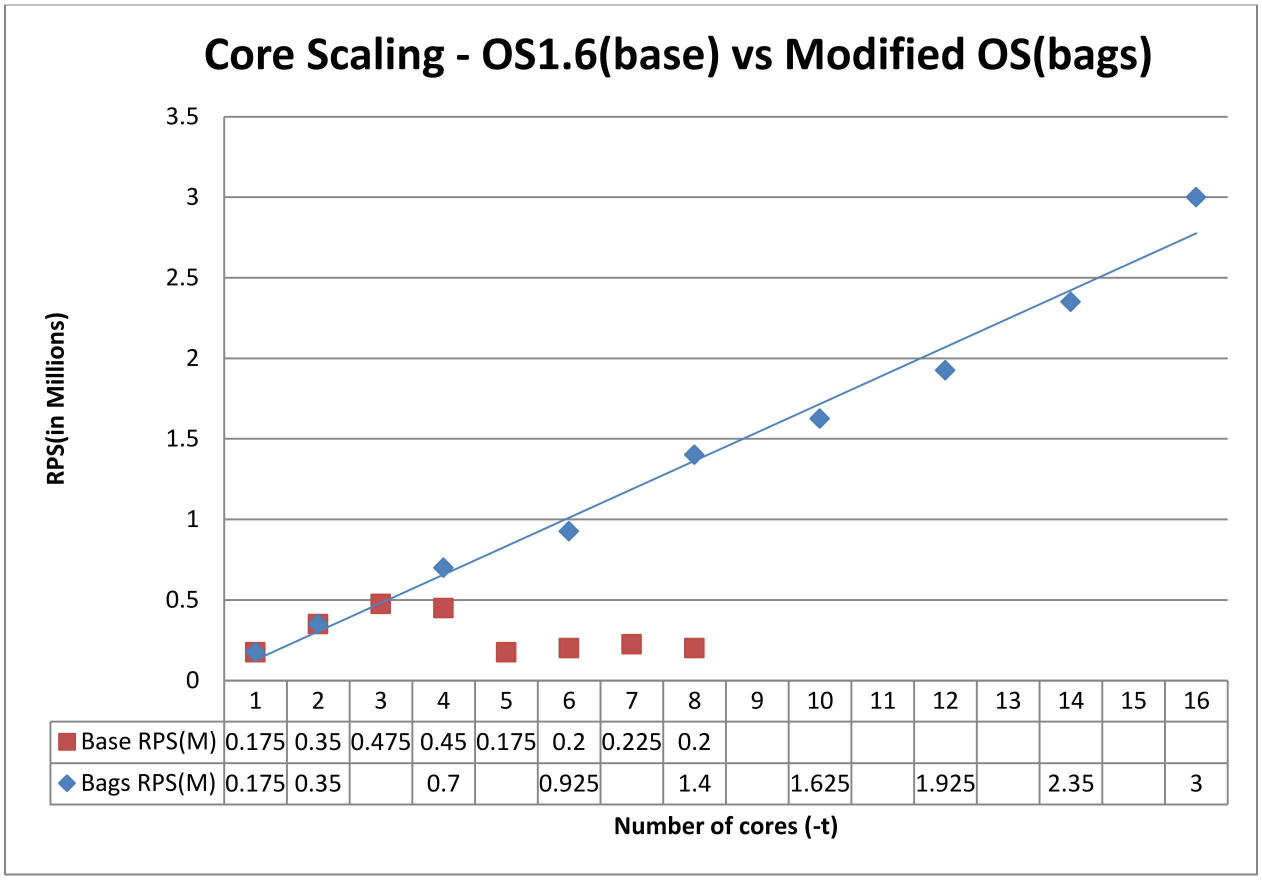


Figure 13 - Median RTT as GET RPS rate is increased



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
🟥 Base RPS(M)	0.175	0.35	0.475	0.45	0.175	0.2	0.225	0.2								
◆ Bags RPS(M)	0.175	0.35		0.7		0.925		1.4		1.625		1.925		2.35		3

Figure 14 - Maximum throughput with a median RTT < 1ms SLA as core counts increase

Table 3 adds power metrics to the best baseline and optimized memcached results at maximum load under the SLA. In addition to the 6X performance delta, performance per Watt improves 3.4X over the baseline. In the data center, where power and cooling is a large portion of the total cost of ownership, maximizing the work completed per watt invested is critical. These per-server power numbers would be lower in a data center environment with no graphics card and a better cooling solution than our SUT.

Binary	Max RPS (under 1ms)	Power (Watts)	RPS/Watt
OS1.6 (HT, Turbo, 4t)	0.525	159.2	3298
Bags (HT, Turbo, 16t)	3.15	284.7	11066
DELTA	6X		3.4X

Table 3 - Power measurements at the wall while running at max throughput < 1ms SLA

6 Conclusions and Future Work

Optimized memcached increases throughput by 6X increase and performance per watt by 3.4X over the baseline. Extrapolating these numbers from the half-width server board to a full chassis (i.e. two boards), the SUT configuration supplies 6.3 Million RPS per 1.5U server with 256GB of memory, with capability to scale to 1 TB (32 GB * 16 DIMM slots) of memory per 1.5U chassis. The speedup is attributed to the redesign of the hash table and LRU to utilize parallel data structures. These data structures allow for the removal of all locks for GET requests, and mitigate the contention of locks for SET and DELETE requests, enabling linear speedup when measured from 1 to 16 cores on a 2-socket Intel® system.

Future research should examine memcached on 4-socket servers and micro-server platforms to determine if the bag version will continue to scale with additional cores, and if an increase in performance per watt occurs with the amortization of the power drawn from shared components.

A second area for research is the cache replacement policy. With updated cache item ordering and eviction, as well as the addition of a cleaner thread, new replacement policies could be assessed (e.g. LFU, Adaptive Replacement Cache, and MQ). A key feature improvement of the cleaner thread is its autonomous nature, allowing cache management without worker threads' involvement. Assessing new replacement policies can now be completed without modifying the flow of the program, potentially increasing the cache hit rate.

7 Acknowledgment

The authors recognize the contributions of project team members; including Rajiv Kapoor (Intel Corporation), George Chaltas (Intel Corporation), Alex Duyck (Intel Corporation), Dean Chandler (Intel Corporation), and Mike Bailey (Oregon State University)

8 References

[1] P. Saab, "Scaling memcached at Facebook," 12 December 2008. [Online]. Available: https://www.facebook.com/note.php?note_id=39391378919&ref=mf. [Accessed 1 April 2012].

[2] Twitter Engineering, "Memcached SPOF Mystery," Twitter Engineering, 20 April 2010. [Online]. Available: http://engineering.twitter.com/2010/04/memcached-spof-mystery.html. [Accessed 1 April 2012].

[3] Reddit Admins, "reddit's May 2010 "State of the Servers" report," Reddit.com, 11 May 2011. [Online]. Available: http://blog.reddit.com/2010/05/reddits-may-2010-state-of-servers.html. [Accessed 1 April 2012].

[4] C. Do, "YouTube Scalability," Youtube / Google, Inc., 23 June 27. [Online]. Available: http://video.google.com/videoplay?docid=-6304964351441328559. [Accessed 1 April 2012].

[5] memcached.org, "memcached - a distributed memory object caching system," Memcached.org, 2009. [Online]. Available: http://memcached.org/about. [Accessed 27 February 2012].

[6] N. Gunther, S. Subramanyam and S. Parvu, "Hidden Scalability Gotchas in Memcached and Friends," in *Velocity 2010 Web Performance and Operation Conference*, Santa Clara, 2010.

[7] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.,* vol. 2004, no. 1075-3583, p. 124, 2004.

[8] B. Fitzpatrick, "LiveJournal's Backend A history of scaling," August 2005. [Online]. Available: http://www.slideshare.net/vishnu/livejournals-backend-a-history-of-scaling. [Accessed 1 April 2012].

[9] N. Shalom, "Marrying memcached and NoSQL," 24 October 2010. [Online]. Available: http://natishalom.typepad.com/nati_shaloms_blog/2010/10/marrying-memcache-and-nosql.html. [Accessed 1 April 2012].

[10] Intel Corporation, "ARK - Your Source for Intel Product Information," Intel.com, 2012. [Online]. Available: http://ark.intel.com/. [Accessed 2012 1 April].

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *ACM SIGMETRICS/Performance 2012 Conference*, London, 2012.

[12] J. Preshing, "Locks Aren't Slow; Lock Contention Is," Preshing on Programming, 18 November 2011. [Online]. Available: http://preshing.com/20111118/locks-arent-slow-lock-contention-is. [Accessed 1 April 2012].

[13] B. Agnes, "A High Performance Multi-Threaded LRU Cache," codeproject.com, 3 February 2008. [Online]. Available: http://www.codeproject.com/Articles/23396/A-High-Performance-Multi-Threaded-LRU-Cache. [Accessed 1 September 2011].

[14] M. Spycher, "High-Throughput, Thread-Safe, LRU Caching," Ebay Tech Blog, 30 August 2011. [Online]. Available: http://www.ebaytechblog.com/2011/08/30/high-throughput-thread-safe-lru-caching/. [Accessed 1 September 2011].

[15] D. R. Hariharan, "Scaling Memcached – Harnessing the Power of Virtualization," in *VMWorld 2011*, Las Vegas, 2011.

[16] M. Berezecki, E. Frachtenberg, M. Paleczny and K. Steele, "Many-Core Key-Value Store," in *Green Computing Conference and Workshops (IGCC), 2011 International*, Orlando, 2011.

[17] S. Hart, E. Frachtenberg and M. Berezecki, "Predicting Memcached Throughput using Simulation and Modeling," Orlando, 2012.