2015

Fetch-and-Phi in Memcached

Adam Michael Schaub
Lehigh University

Fetch-and-Phi in Memcached

Adam Schaub

ams314@lehigh.edu

Lehigh University

May 2015

Thesis Signature Sheet

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____

Date

_____

Thesis Advisor

_____

Thesis Co-Advisor

_____

Department Chairperson

Table of Contents

ABSTRACT

Memcached and other in-memory distributed key-value stores play a critical role in large-scale web applications, by reducing traffic to persistent storage and providing an easy-to-access look-aside cache in which programmers can store arbitrary data. These caches typically have a narrow interface, consisting only of gets, sets, and compare-and-set. In the worst case, this interface can cause significant inefficiencies as clients get large data items, perform small changes, and then set the updated items back into the cache.

We present extensions to memcached that allow the system administrator to dynamically load custom code modules into memcached, so that clients may execute code directly on the memcached server. Our system permits both fetch-and-phi operations, which update the cache, and filtering operations, which compute a function over the data in the cache, and return the result to the client without making an update to the cache. We evaluate our extensions on benchmarks based on workload traces from a Cable/Internet service provider, and find that our new functionality provides a means of dramatically reducing network overheads and increasing responsiveness.

1 Introduction

One of the most critical components of interactive distributed systems is an in-memory cache of key/value pairs [1]. This cache can serve many roles, to include providing fast access to information from persistent table stores like HBase [19], BigTable [2], Silt [12], Cassandra [20], and Dynamo [3]; caching results from expensive joins in traditional relational databases [1]; and storing the result of expensive computations.

Memcached [14] is one of the most popular and widely-used key/value caches. Memcached is a lookaside cache, and when a client fails to find data at a memcached server, it must both (a) contact the next tier of the storage infrastructure to acquire the data, and (b) decide whether to update memcached with the data. Memcached uses an LRU policy to age and evict data, and is oblivious to the nature of the data it stores: it treats keys and values as untyped byte arrays. With regard to access control, memcached instances are open: they assume they are protected by a firewall, and never configured to provide service to untrusted clients. Any machine that can send requests to the memcached server can get and set any key/value pair.

In embodying the mantra “do one thing, and do it well”, memcached and other in-memory stores manage to provide high performance and value to a diverse

spectrum of distributed systems, while consisting of a rather small and easy-to-understand code base. At the same time, memcached is compatible with a number of best practices in distributed computing, which ensure good system-wide performance. It is well known that locality of data relative to the site of computation is a critical factor in distributed systems [7, 11, 18], and replicated memcached servers can be installed at network edges, to provide fast access to most data. To improve load balancing [10], keys are hashed to decrease the likelihood that any individual memcached server has a higher-than-average load, and again, replicated memcached servers can be employed.

Despite these benefits, in-memory object caches can be a performance bottleneck. The motivating example on which this work is based is the infrastructure behind the user interface (UI) of Comcast's XFINITY X1 platform. Millions of set-top boxes in a geographic market serve as thin clients, forwarding clicks from a customer's remote control to Comcast logic servers, where the clicks are processed. In response to a click, a logic server will access the cache and/or persistence layer, and then render a new screen that is sent back to the customer. The Comcast servers are configured in three layers: a Cassandra-based persistent store, a memcached layer, and a custom logic layer where each server is assigned hundreds of set-top boxes in a one-to-many relationship. Each

layer is independently fault-tolerant and redundant, typically through replication.

Unlike many bursty workloads, this system experiences predictable spikes in use. Consider the case of digital video recorder (DVR) operations: at a fixed time (e.g., 9:00 PM), a million customers might simultaneously wish to stop recording one program, and begin recording another. Each of the logic servers must, for each of its dedicated set-top boxes, start or stop a physical recording operation, and then update the customer's UI metadata to indicate the new recording status. The UI data must be stored several ways, since the underlying Cassandra storage is denormalized; first, the individual recording is updated, and then the denormalized views of the recordings ("scheduled recordings", "in progress recordings", "completed recordings") are updated.

Consumer habits necessitate that the data is cached for fast access, since the customer is likely to verify that recordings have been initiated, and to frequently browse through the DVR listing; allowing each query to reach the Cassandra database would introduce too much latency. In contrast to DVR operations, these interactive queries are not concentrated at the exact times when programs begin and end. The result is a spike in utilization when DVR recording statuses change, which has come to be known as the "top of the hour" problem. Engineers at Comcast have managed to reduce the spike from orders of magnitude to about

2.5x. However, the bursts are too regular to be mitigated through sharing other servers, since the burst is periodic and relies on data being stored in-memory. In effect, handling the bursts requires a large number of servers that are underutilized most of the time.

In analyzing this problem, we identified a trend that we believe generalizes to many workloads: writes of a datum are preceded by a read of the same datum, and the amount of data modified by a write is a small fraction of the overall object. When coupled with the fact that many objects are large (tens, if not hundreds of kilobytes), an opportunity emerges: if it were possible to update the object directly in the cache, and then return only the updated object, we could halve the number of network round trips, and halve the required bandwidth.

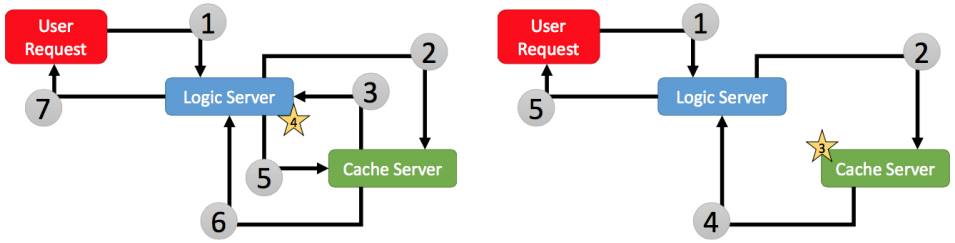


Figure 1: Default system behavior (left) and the behavior with in-cache computation (right). Arrows represent the flow of information (requests and responses), and stars represent computation.

The behavior we wish to achieve is depicted in Figure 1. On the left side, there are two expensive communications, labeled 3 and 5, which correspond to getting a large object O and setting the updated object O′ back into the cache. The

messages sent in steps 2 and 6 are small, representing a get request, and an (optional) acknowledgment. On the right hand side, when the computation (represented by a star) migrates to the cache server, one round trip network communication is avoided, and the only large object transmission occurs at step 4.

In this paper, we extend memcached to allow in-cache computing. Our system is based around the concept of an atomic fetch-and-phi primitive [8], which allows a programmer to instruct memcached to get an object, invoke a function on that object (possibly involving programmer-specified parameters), and set the result back into the cache, all in a single atomic operation that then returns the computed value. We decompose the fetch-and-phi to relax atomicity, to elide the set, or to omit the response. Through evaluation on microbenchmarks that use traces from Comcast's production servers, we show that moving computation into the cache can reduce overheads and increase throughput by more than 30%.

The remainder of this paper is organized as follows. In Section 2, we provide a brief overview of the memcached interface and API. Section 3 presents the algorithm for our client-facing modifications to memcached, through which a programmer can execute code on the memcached server. Section 4 details the administrative interface we propose for supporting fetch-and-phi without

completely opening up a server to the possibility of running arbitrary unreliable code. In Section 5, we discuss a workload generator we created. The generator uses traces collected from Comcast servers as the basis for the workloads run by clients. Section 6 presents experiments, and Section 7 discusses related work. Section 8 discusses future work and then conclusions.

2 The Memcached Client API and Extensions

The existing interface to memcached appears in Table 1. Several of these functions can take multiple keys as parameters. For clarity, we omit discussion of this functionality. Three functions comprise the primary API: get takes a key as its parameter, and returns the corresponding object; set takes a key and value, and updates the value stored at that key; delete removes a key/value pair. Each key is stored along with a version number that is incremented by memcached on every set, replace, append, prepend, incr, and delete. By using gets instead of get, a client can observe this value. The cas operation employs this value to achieve a read-modify-write effect, akin to load-linked/store conditional instructions: it takes a key, a new value, and the value of the counter. It updates the pair to the new value if and only if the counter values match. On a successful cas, the counter value is updated.

Command	Purpose
`set`	Store data, possibly overwriting the existing value for the given key; promote the key to the top of the LRU list.
`add`	Store data only if the given key does not exist; promote the key to the top of the LRU list even if the key already existed in the cache.
`replace`	Store the given key/value pair only if the key already exists in the cache.
`append`	Extend an existing value by concatenating the supplied data to the end. Constrained by pre-determined size class of a value.
`prepend`	Extend an existing value by concatenating to the front of the value. Same constraints as append.
`cas`	Compare-and-swap: performs a replace only if the value hasn't changed since the last gets of the key by the client.
`get`	Retrieve and return data.
`gets`	Retrieve and return data, with additional metadata for use in a cas.
`delete`	Remove an item from the cache, if it exists.
`incr`	Add to the value of a key only if the key is an integer.
`decr`	Subtract from the value of a key only if the key is an integer.
`stats [p]`	Interact with statistics information. Uses `p` to request detailed statistics about items or slabs.

Table 1: The complete memcached client API.

When the value stored with a key is an integer, the incr and decr functions can be used to achieve an atomic increment or decrement in a single instruction. Memcached effectively locks the key/value pair, updates the value, and then releases the lock. The same approach can be used when the value is treated as a raw byte array, and the desired operation is an append or prepend. However, there is an additional constraint in this case: memcached uses a slab allocator, such that every object has a size class associated with it. Within the size class, each object is stored with some amount of internal fragmentation, and a prepend or append will fail if the new object cannot fit in the object's existing slab class. The set, add, replace, and cas operations are not subject to this restriction, since they assign the newly provided data to the key, instead of modifying the key in-place.

To this API, we add two new operations:

register(): The register() function provides a mechanism for performing inserts and lookups into a map. The map stores < string, f unction > pairs, where the string is a programmer-visible name. The function returns a boolean, indicating whether or not it succeeded. It takes the following parameters:

Name	Type	Purpose	
name	*String*	in	The name of the function to call
key	*byte*[]	in	The key whose value will be used by the function
params	*byte*[]	in	Parameters to pass to the function being called
atomic	*Boolean*	in	True if the operation should be atomic
update	*Boolean*	in	True if the value should be updated
reply	*Boolean*	in	True if the client expects a reply

Logically, the map is protected with a readers/writer lock, which is held in write mode whenever a register() operation is performed. We discuss the use of the register() function in Section 4, to include discussion of a technique through which the use of a lock to protect the map can be avoided.

Invoke(): The invoke() function is used to execute a registered function on the memcached server. It takes the following parameters:

Name	Type	Mode	Purpose
key	$byte[]$	in	The key with which this function was called
old_val	$byte[]$	in	The value most recently returned by $get(key)$
params	$byte[]$	in	An optional parameter, specified by the caller
ret	$byte[]$	out	The result of the function

The API makes no assumptions about how invoked functions are used. The boolean parameters provide the programmer with knobs for using the function as a filter, a fetch-and-phi, or an atomic fetch-and-phi; and for indicating whether the client expects a reply.

3 The Client Interface

In this section, we discuss the high-level implementation of the invoke function. We then describe usage scenarios and potential pitfalls. We focus on an implementation that executes user code directly in the context of the memcached server process. Section 4.2 discusses a programmer-defined technique for executing code in a more sandboxed setting.

3.1 Implementing Invoke()

Algorithm 1 presents the pseudocode for the invoke() operation. The client issues an invoke request by supplying the name of a function to execute, the key whose data will be used by the function, parameters, and a few flags.

The operation consists of five stages. The first stage entails locating the code

that will be executed. This is achieved via a lookup in the map. Though generally straightforward, we note that the map is not protected by a lock: the register operation is responsible for ensuring that the map isn't modified while it might be accessed by client invoke operations.

In the second stage, we fetch the data currently associated with the key. We can fetch the data along with its current version count (line 7) or else without the version count (line 9), depending on whether the operation will ultimately perform an atomic fetch-and-phi. Note that when update is false, the atomic flag is superfluous. Furthermore, in the common case (where the named function is free of side effects), invokes that do not set are naturally atomic, in that they have a linearization point [9] at the time of the get.

The third stage of invoke() involves calling the requested function, passing the key, current value, and parameters. This produces a status message (success/failure) as well as a new byte array. This byte array may be a new version of the value, or some computed value of some other type. The meaning of the contents of the new_val parameter are user-defined.

The fourth stage is to update memcached so that the new value is associated with the key. This can be done atomically or nonatomically, based on the parameters to invoke(). For brevity, we omit details of how errors at this point are

relayed to the client. The final stage returns the result of the function to the client, if a reply was requested. Otherwise a short acknowledgment of success or failure is returned to the client.

3.2 Usage Scenarios

While our focus is on reducing overhead for a pattern built upon get and set, our invoke() implementation is general enough to support four usage patterns:

Basic Filtering: When the persistent layer is implemented using a NoSQL database, it is likely that there will be denormalized data in the cache. This is a common optimization for non-relational databases, where the programmer must maintain the relationships among data by storing rows in an additional table. It is often easier to over-denormalize, and store very large objects, in order to avoid maintaining multiple denormalizations. In this case, as well as the case where individual objects are large, filtering can dramatically reduce overhead.

At its simplest, filtering entails returning only *part of* the object, in order to reduce bandwidth. For example, a client program attempting to get the next DVR recording to initiate does not need a full list of scheduled recordings; if the recordings are sorted by start time, then it suffices to return the first recording. When all pending recordings are saved as a (serialized) array in memcached, filtering allows the transmission of a specific array entry. To achieve filtering, the

client calls invoke with both atomic and update set to false. Note that a simple filter is an atomic operation, since the value is read atomically through a standard get call.

Filtering with Computation: In the former case, filters were little more than field selectors, serving to limit the amount of data returned to the client. It is equally straightforward to employ a pure computation within the invoked function, and thus to add computation to the filter. This enables, for example, computing statistics over the entries of an array stored in the cache. Furthermore, the function need not be pure. As an extreme example, the function could open another connection to the same memcached server to request additional data.

Fetch-and-Phi The compare-and-set mechanism I nmemcached more closely resembles the Load Linked/Store Conditional pair of instructions that are common in RISC processors [8]. Specifically, an arbitrary amount of time can pass between the get and the set, but the pair appears to be atomic as long as there is no intervening update. In memcached, this is achieved by attaching a counter to the pair, and incrementing it on every update. When both atomic and update are true, the behavior of invoke gives precisely the behavior of a load-linked/compute/store-conditional sequence in traditional shared memory synchronization. For functions that are side-effect free, or whose side effects are

not visible to concurrent threads, the result is an atomic fetch-and-phi. Note that setting atomic to false results in a similar effect to implementing a shared counter with loads and stores: atomicity can be violated if there is sharing, but it is safe as long as the underlying pair (or counter) is not shared.

Set-and-Go When the client updates the cache, it often must also send a message to the next level of the storage hierarchy (e.g., a NoSQL database) to ensure the persistence of the new data. Whereas fetch-and-phi is ideal when the client does not have the data on hand, set-and-go serves the case where the client can compute the new value without a preceding query. In this case, rather than send the entire new object to the cache, the client can re-execute the function at the cache, store the result, and not send a reply. If the object is not found, then it remains uncached. For cached objects, this pattern avoids sending large objects after small modifications, an allows the in-cache update to execute concurrently with the client.

3.3 Concerns

Any time code is dynamically loaded into a high-availability process, there are causes for concern. Buggy code can cause the process to crash. Long-running code can keep a thread occupied for an extended period, causing a service degradation or denial of service. Worse, the code may interact with the operating

system to cause the process to block, lose priority, or release needed resources. As we discuss in Section 4, we do not allow clients to load arbitrary code without the intervention of a system administrator. Since memcached servers are typically not publicly accessible, this should provide the minimal needed oversight to avoid the aforementioned problems.

There are three additional concerns that are more specific to our design. We outline them briefly below:

Multiple Keys The memcached client interface allows a single get to request multiple keys, or a single set to update multiple keys. We do not provide this ability, and instead require each invoke to operate on a single key/value. This choice is intended to provide clarity, since the semantics of a multi-key invoke are ambiguous. Should several invokes be executed as a single atomic transaction? Should a single function take several pairs as input, and produce multiple outputs that all are set into the cache? Farther afield, it would even be imaginable that operations could use some keys to locate input objects, and others to identify output objects.

While the above opportunities are appealing, their value depends on an efficient fetch-and-phi. Furthermore, there are additional programming concerns that are outside the scope of this work. For example, any multi-key operation needs

guarantees that the keys are all stored at the same memcached node, and this is not easily guaranteed for common hashing functions. While coalescing multiple independent invoke operations into a single message could decrease network costs, we believe that such extensions are best left as future work, after compelling use cases are identified.

Copying Overheads While our implementation and discussion are focused on memcached, we took care to avoid algorithmic choices specific to memcached, unless obvious alternatives exist (e.g., the technique we use to avoid using a readers/writer lock to protect the map; see Section 4). However, this introduces overhead due to data copying.

Specifically, when a get is performed, the value is copied from the cache to a temporary buffer. In our system, the function is given this buffer as input. The buffer must be manually reclaimed when invoke returns. Similarly, the function may create a new buffer to store the new value of the key. This buffer is not inserted directly into the cache, but instead is copied into a new pair object, after which it must be reclaimed. The alternative is to hold the lock on a pair throughout the invoke operation. However, this technique fails when the function produces a new value that is of a significantly different size than the input value. In this case, the memcached memory slab allocator would require the new value

to be stored in a different slab class. The motivation for direct access to objects in the cache, as opposed to copies, can only be in response to performance concerns. We discuss this topic further in Section 6.

Data Formats Thus far, we have glossed over the actual formats of objects stored in the cache, and the type of the parameter provided by the client during an invoke call. Generally speaking, we leave these as client programmer-defined. However, it is worth noting that in the common case, the object format is a serialization of some other format (e.g., pickling in Python, or Google Protocol Buffers [6]). Furthermore, some memcached client libraries implement compression and decompression [21]. In these settings, the invoked code may need to decompress, deserialize, compute, re-serialize, re-compress, and then perform the set on line 19 or 21. Particularly in the case of compression, this can introduce increased CPU utilization, and possibly slowdown.

Communication While we do not explicitly support communication from within an invoked function, we do not forbid it. This creates a tension. On the one hand, it is possible for a function to maintain a static pool of connections, and attempt to contact other memcached servers, or the next level of the hierarchy, in order to implement a form of write-through caching or atomic multi-object transactions.

We argue that such techniques are beyond the reasonable scope for our

mechanism. For example, to implement a memory hierarchy properly, it would be necessary to also provide a way to execute a set followed by a user-defined function, without performing a get to acquire the data passed to the function and to set. This complexity seems unjustifiable for a general-purpose system like memcached, unless a use case is identified. However, should such a case arise, we believe our work can easily be extended to support it.

4 The Management Interface

The obligation of the management interface is to provide a means through which arbitrary functions can be made available for subsequent use by the invoke() operation. Additional concerns relate to (a) how permission is granted for providing new functions, and (b) how the execution of those functions is sandboxed.

4.1 The Register() Function

We begin by describing the basic infrastructure for registering functions. We take the position that arbitrary clients should **not** be allowed to install arbitrary code into the memcached process. While it is possible to sandbox the execution of code, even sandboxing mechanisms can result in denial of service if an arbitrary function contains an infinite loop. To prevent this, our design requires a machine

administrator in order to register a new function for use with invoke().

We make use of an environment variable to identify the root folder for all code loaded into memcached. An administrator can copy shared object files into this directory, whereas clients cannot. Once a file is loaded into this directory, register() can be called to load the shared object, find a function within it, and add a pointer to that function to a map of available functions. The sanitize() function ensures that any relative path (specified by a client as part of an invoke()) is converted to an absolute path rooted at the folder indicated by the environment variable.

The behavior of register contains only one subtlety: the use of a maintenance mode. In memcached, a handshake mechanism exists through which the cache rebalance thread gains exclusive access to the entire cache. The mechanism entails a lock hierarchy. At the top is a single “assoc_lock”, below which are per-item locks. When a cache rebalance is required, the maintenance thread invokes switch_item_lock(GLOBAL). This sets a flag to indicate that all client operations should use the assoc_lock instead of item locks. The maintenance thread then waits for all in-flight client operations to complete. It thread then acquires the assoc_lock and performs its work. It is possible that new client requests have arrived in the interim, but they now use the assoc_lock, and thus do not overlap

with the rebalance. When the rebalance completes, the maintenance thread invokes switch_item_lock(GRANULAR) to return to using per-item locks. The key feature of this mechanism is that in the common case, threads only require fine-grained item locks, but during rebalancing, the absence of concurrency prevents the maintenance thread from acquiring these locks.

By connecting the registration of new functions with this mechanism, we can ensure that the map is never modified while a client invoke is in progress. This avoids the need for a readers/writer lock for the map, which would introduce a bottleneck for simultaneous invoke calls.

4.2 Sandboxing and Reliability

For many environments, the requirement of administrator access to install code suffices to provide a secure and reliable environment: new functions are expected to be simple, so as not to affect the CPU load on the memcached server, and hence they ought to be easy to statically analyze or verify through a code review.

However, we recognize that in some circumstances this may prove insufficient. For example, in an environment where the memcached clients are written in a high-level language (e.g., Python), the requirement to write C code that operates on pickled objects may be burdensome. We contend that this can be resolved

through a level of indirection in the new code, without further impact on the memcached code.

As a concrete example, we propose that an administrator might install a Python process that listens on a named pipe for messages of the form (function, data, parameters). In response to any message, the process will consult a local map of functions, find the appropriate one, and execute it on the data and parameters that are provided. It will then write its response to the pipe. If the registered C function simply writes its parameters to the pipe and then awaits a reply, then the same (modulo Python function name) C code can be registered for each Python function. This process is somewhat more cumbersome than registering C code directly, and it suffers from the overheads of both (a) inter-process communication between memcached and the Python process; and (b) overheads from the Python interpreter. However, it generalizes, and the same approach can be leveraged for arbitrary managed languages. By using a timeout when reading from the named pipe, the C code running in memcached can simply return an error whenever erroneous code causes the Python process to crash. A daemon can re-start the Python process periodically.

The purpose of this example is merely to demonstrate that additional sandboxing can be introduced, if needed, to prevent erroneous code from crashing the

memcached server. We believe that the open source community is best suited to providing the infrastructure for creating this sandboxing on a per-language basis, using a generic methodology such as that described above. Such an approach will also allow environment-specific optimizations (e.g., using a pool of named pipes, and a pool of independent Python interpreters, to avoid serialization on the Python interpreter's global lock).

5 A Top of the Hour Workload

As discussed in Section 1, the original motivation for this work was a regular usage spike observed on Comcast servers. At the beginning of every hour, and again on the half hour, requests to the memcached servers would increase dramatically. Increases in response time from the memcached servers caused cascading delays, since UIs could not be rendered until memcached responded. A combination of scaling out and rewriting interface code resolved much of the problem, but left a system that is underutilized most of the time.

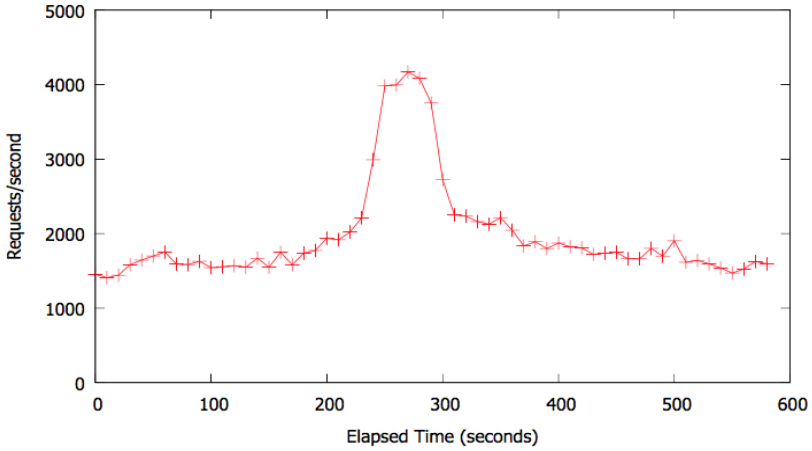


Figure 2: Requests per second during a top-of-the-hour spike.

To understand this workload better, we collected statistics from a Comcast server over a 10-minute period that included a spike. On ten-second intervals, we collected statistics from the server about the number of operations performed on each of its memory slabs. As seen in Figure 2, the workload exhibits a burst of activity.

This burst is significant for several reasons. First, during the burst, the number of set operations increases dramatically. Prior to the surge, the workload is 61% gets and 39% sets. During the surge, it becomes 40% gets, and after the surge, it has 55% gets. During the surge, there are two primary operations: among small objects, there is an order of magnitude increase in the number of sets, as logic servers mark DVR recordings as started or stopped, and then overwrite a single-recording object of under 1000 bytes. The logic servers are able to cache these

objects, and thus they need not perform a get before the set. However, they must send the entire object, even though only 4 bytes change. The second operation is an update to very large objects (over 10K bytes). These objects represent denormalized rows, storing each customer's set of recordings and their corresponding states. When the small-object set occurs, the row becomes stale, and must be updated to indicate the new recording state.

Without tracing each individual TV viewer's click behavior, it is not possible to determine the number of sets that (a) were part of a get/set pair, and (b) could be replaced with a fetch-and-phi. However, there is strong evidence that these operations were concentrated on large objects, where the ratio of gets to sets remained relatively constant, though the numbers increased during the surge.

Similarly, the set operations on small objects can be optimized: rather than sending an updated object, the logic servers can invoke a function that performs the necessary modification to the object, and does not send a reply. Again, we cannot precisely determine the frequency of these operations in the trace. However, they roughly correspond to the increase in sets of small objects during the surge.

To re-create this behavior, and to create similar behaviors, we implemented a memcached client that is heavily parameterized, so that it can produce workloads

of the same shape and operation mix as described above. Our client is a Java program that uses the spymemcached client library to interface to the memcached server. We extended spymemcached to support invoke and register functions, in addition to the standard interface.

Like many real-world deployments, this workload at Comcast uses Google protocol buffers to serialize and deserialize data, so that arbitrary objects can be provided to memcached as byte streams. There are two primary object types in the Comcast workload, which we generate and populate with anonymized data. The first is a small object (roughly 480 bytes, though the exact size depends on the length of a few strings) that describes a specific recording of a single show, to include the state of the recording. The second is a large object (typically 10K bytes, though the trace we captured included objects as large as 200K bytes) storing an array of recording objects (e.g., the list of all scheduled recordings).

Our client is multi-threaded, but since it mirrors a one-to-many relationship between set-top boxes and logic servers, each client thread accesses a unique set of objects at the memcached server. Using the Comcast trace as a guide, we pre-populate memcached with objects of 31 different size classes. We populate the cache to 60% full, to prevent evictions during experimentation.

Since our traces were measured by querying a memcached server at 10-second

intervals, we do not have fine-grained information about the number of gets and sets that could be replaced with calls to invoke. To compensate for this, our memcached client is parameterized. It alternates between periods of low activity, during

which a fixed number of operations are performed per second, and bursty periods, where it attempts to execute as many operations as possible. Parameters govern the number of operations that are sets, gets, or invokes, and the object sizes from which randomly selected elements will be chosen for use with those operations. We are also able to control the length and frequency of bursts.

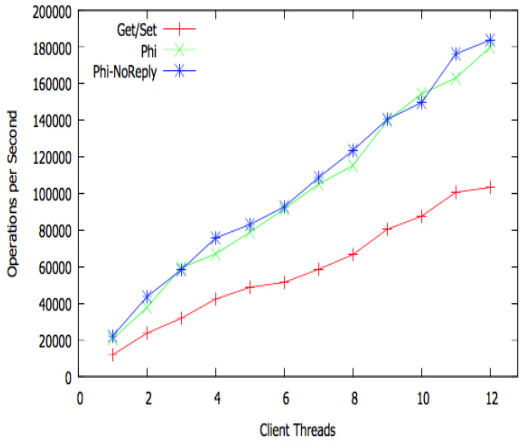
6 Evaluation

In this section, we evaluate the performance of our extensions to memcached on a number of synthetic workloads. We ran memcached on a system with two Intel Xeon 5650 chips and 12 GB of RAM. This system has a total of 24 hardware threads (12 cores). Unless otherwise specified, we generated requests to this system from an identically configured machine with a single Intel Xeon 5650 (6 cores/12 threads) and 6 GB of RAM. The machines were connected via a switched 1GBps network fabric. The software stack on both machines included Ubuntu Linux 13.10, GCC 4.8.1, memcached 1.4.20, Oracle Java 1.8.0_11, and

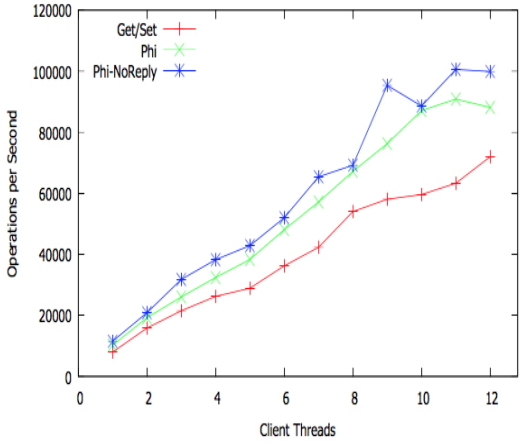
spymemcached 2.11.4. All experiments were run five times, and the average is reported.

6.1 Microbenchmark Performance

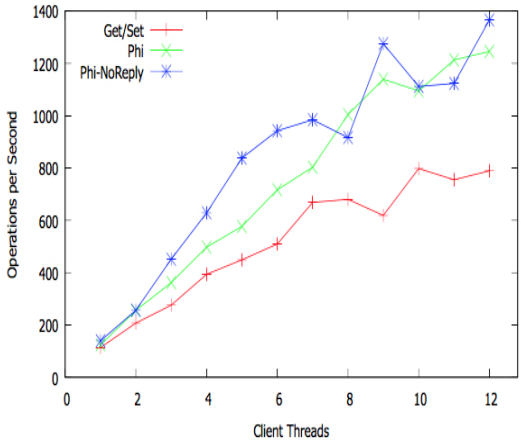
Our first experiment is a best-case study for fetch-and-phi. We did not use the workload described in Section 5. Instead, we populated the memcached server with a set of objects of the same size. Each thread of the client accessed a disjoint set of objects. The workload consists of getting the object, performing an O(n) operation that modifies the object one byte at a time, and then setting the object back into the cache. There were no cache evictions during the experiment.



(a) Local, 256 Byte Objects

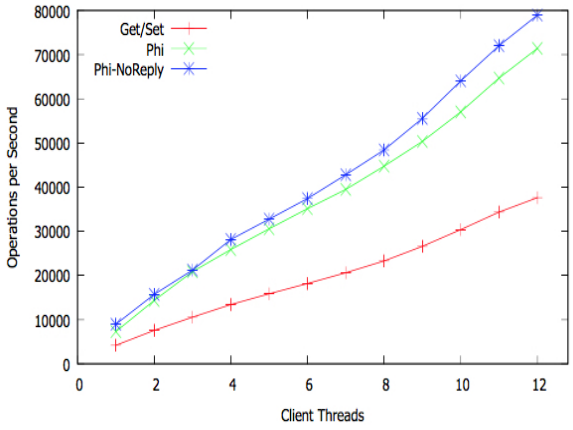


(b) Local, 4K Byte Objects

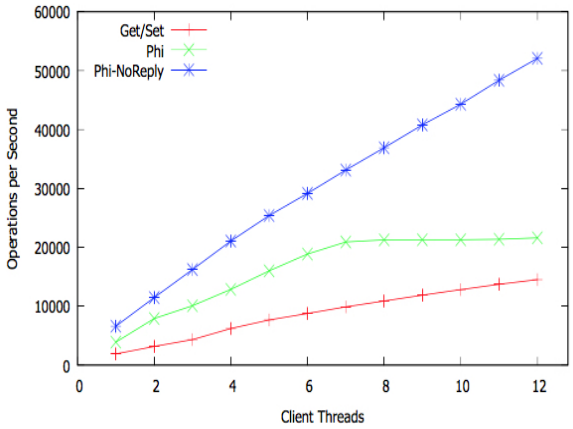


(c) Local, 64K Byte Objects

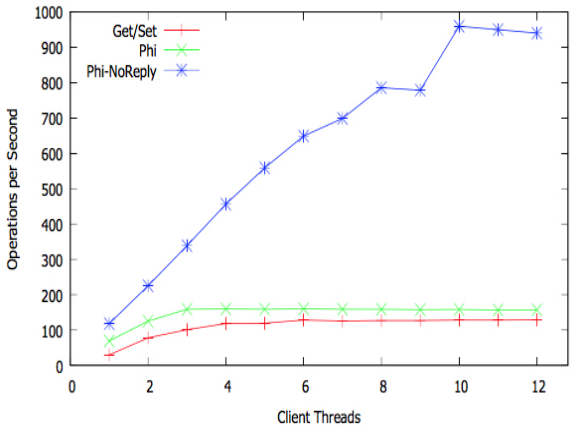
Figure 3: Microbenchmark performance for varying object sizes, with the client and server running on the same machine.



(a) Remote, 256 Byte Objects



(b) Remote, 4K Byte Objects



(c) Remote, 64K Byte Objects

Figure 4: Microbenchmark performance for varying object sizes, with the client and server running on separate machines.

Figures 3 and 4 compare the performance of this workload as we vary three parameters. Bars labeled “Get/Set” are the baseline: they correspond to a configuration in which the client gets the object, modifies it, and then sends it back to the server. “Phi” corresponds to the case where the client uses invoke to request that the server perform the operation and then send the new object back to the client. In “Phi-NoReply”, the server modifies the object and stores it back to the cache, but only sends an acknowledgment to the client; the new version of the object is not returned. In the “Phi” and “Phi-NoReply” experiments, we set the atomic flag to true, so that updates were achieved as a compare-and-set. In additional experiments, we found the cost of atomicity to be negligible for all workloads. We leave as future work analysis of whether there exist workloads that would favor non-atomic fetch-and-phi.

We ran this experiment at three object sizes: 256 bytes, 4K bytes, and 64K bytes. We also considered two client configurations. In the first configuration, labeled “remote”, we execute the client on a separate machine from the machine running memcached. In the second configuration, labeled “local”, we run the client and server on the same machine. This experiment isolates performance improvements that come from reduced network communication.

There are two trends that emerge from this experiment. First, by contrasting the

remote and local experiments, we see that the most significant savings come from the reduction in round-trip network communication. Using either of the “Phi” approaches, the number of trips is halved. When objects are small, the difference between returning the object, and returning an acknowledgment, is insignificant, and the two “Phi” curves are indistinguishable, though both significantly better than “Get/Set”. As object sizes increase, the additional bandwidth savings from sending a simple acknowledgment grows. At the largest object size, halving the number of round-trip communications saves 25% over “Get/Set”.

6.2 Top-of-the-Hour Performance

Our next set of experiments is based on the traces discussed in Section 5. We generated a variety of workloads that used Comcast protocol buffers. Parameters included the distribution of operations per buffer size, the number of regular gets, and the number of get/set pairs. We also varied the duration and frequency of bursts. During a burst, the client executes as many requests as possible; during non-burst periods, the client performs a fixed number of requests per second.

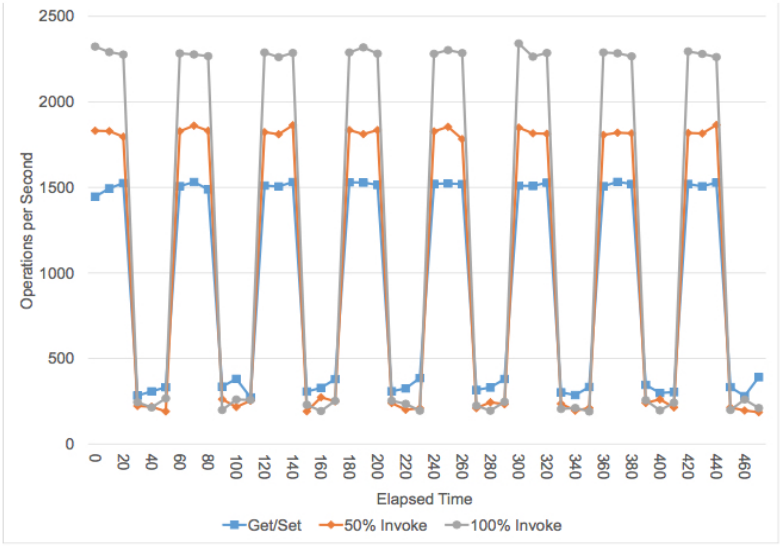
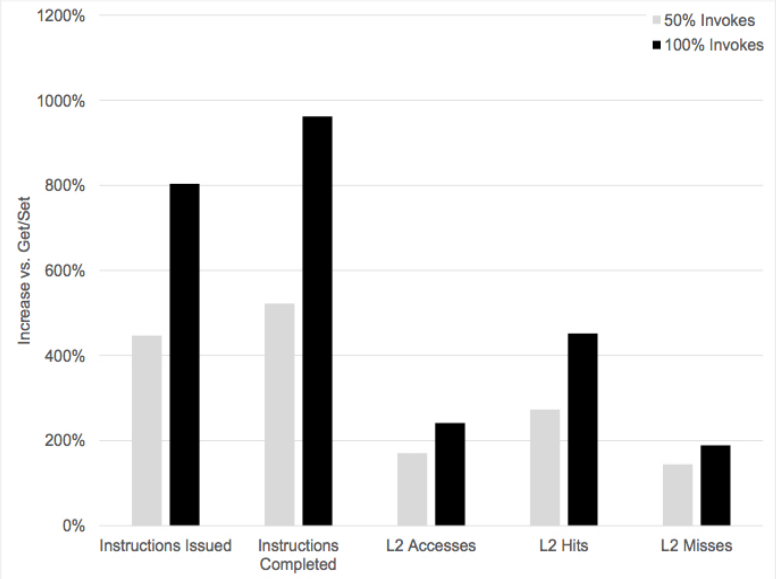


Figure 5: Throughput for 11KB objects with varying use of *invoke*. The workload included 44% gets/56% updates during periods of high utilization, and 62% gets/38% updates during periods of low utilization.

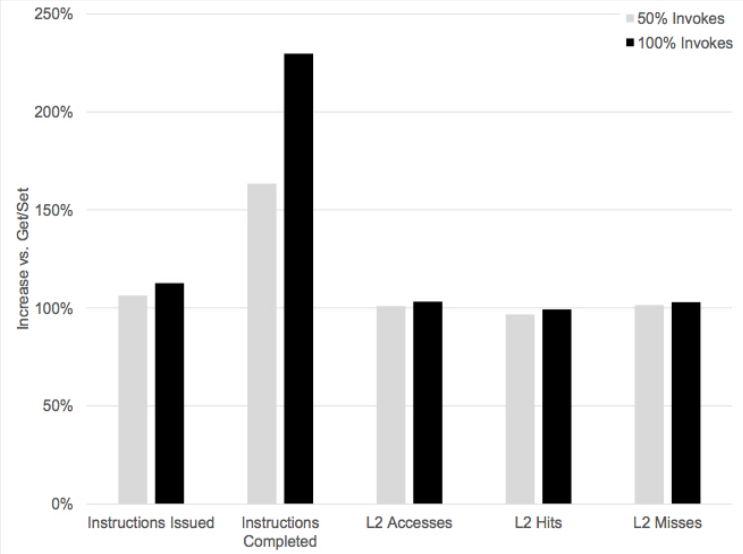
In Figure 5, we show one such experiment, which is performed on a cache populated with 10K byte protocol buffers. We oscillated between 30-second bursts, and 30 seconds of non-burst behavior. The figure shows the total number of operations. An operation is either an unpaired get, or a get/set pair. In this manner, an invoke counts equal to a pair, and captures the notion of an operation from the client's view. In keeping with the workload trace, get operations comprised 44% of the workload during bursts, and 62% of the workload during low-utilization periods.

This result, which is representative of experiments with varying buffer sizes,

shows two key performance trends. The first is that even for a modestly large object size, and a computationally expensive operation (while we update on the order of 16 bytes, there is an O(n) overhead to de-serialize the protocol buffer before operating on it, and then another O(n) overhead to serialize it to a byte array before setting it back in the cache), we still achieve a speedup of close to 40%. The second trend is that the benefit is linear in the ratio of get/set pairs that are replaced with invoke operations.



(b) PMU measurements when the workload operates on protocol buffers



(c) PMU measurements when the workload operates on objects in a custom format

Figure 6: Performance metrics during a period of high utilization.

To gain a deeper understanding of this performance improvement, we re-ran the workload and requested statistics from the CPU performance monitoring unit (PMU). Figure 6b presents this data. The CPU utilization increases by 50% on average, with significant increases in instructions issued, instructions retired, cache accesses, cache hits, and cache misses. Of particular importance, we see that there is a higher ratio of cache hits, and a higher incidence of instructions issued in the same amount of time. These results show that operating on the object immediately after retrieving it has good locality, and also that we are using the CPU more effectively, since we are spending less time making system calls and performing network I/O.

We were, however, surprised by the shear magnitude of the increase. We conducted an additional experiment, presented in Figure 6c, where we used a custom object format that did not require deserialization and re-serialization. Several of the PMU statistics dropped precipitously, and throughput increased even further (we do not present throughput numbers, since the technique does not generalize). This raises a concern: the object format, and the cost of converting between byte-array and object representations, can play a significant factor in overall performance. As the cache server performs more computation on behalf of clients, care is needed. The savings in bandwidth and round-trip

communication, which is enjoyed by client and server, can be offset by increased overhead to operate on the data at the server.

6.3 Filtering Microbenchmark

While our focus has been on using invoke to perform atomic fetch-and-phi, there are a number of other uses. One of particular interest is filtering, where the result of an operation is not set back into the cache. One can think of filtering as providing a way to limit the size of the payload returned to the client, or as a mechanism for performing simple queries directly against memcached.

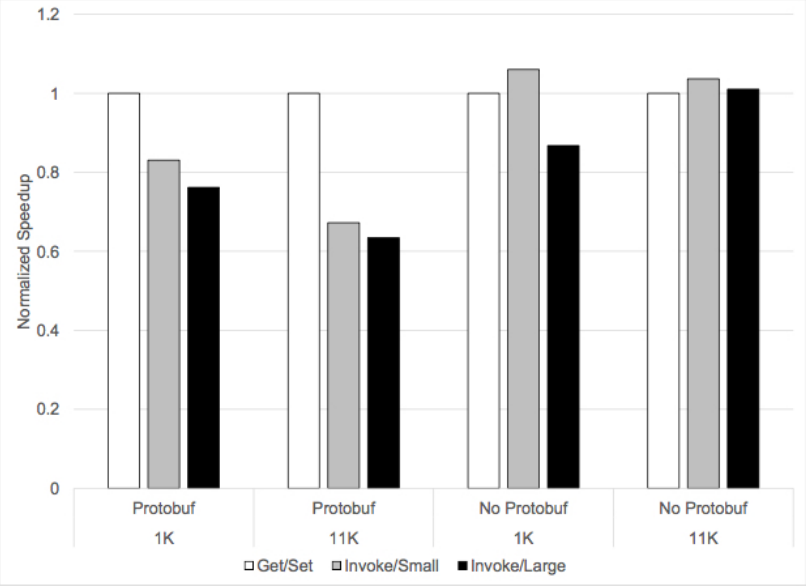


Figure 7: Performance of the filtering microbenchmark.

In Figure 7, we compare the performance of filtering on four workloads. These workloads are parameterized by whether the objects in the cache require deserialization before they are accessed, by the size of the objects that are accessed, and by the amount of data that is accessed. We operated on two object sizes, 1K bytes and 10K bytes, and considered both “small” queries, which returned a 4 byte field, and “large” queries that returned several hundred bytes.

In general, filtering did not perform well, except in the case where the amount of data to return was small and the objects did not require deserialization. In our implementation, we never operate directly on an object stored in the cache. Instead, we lock the object, copy it, unlock it, and then pass the copy of the object to the function being invoked. Even when we used a raw object representation and could avoid the overhead of protocol buffer serialization and deserialization, this copying dominated for small filter operations. Since this experiment is something of a best case for filtering, we conclude that it may be necessary to operate on objects directly in order to achieve efficient filtering. We leave further study of this topic for future work.

6.4 Space and CPU Utilization Implications

In addition to serialization and deserialization of objects, some systems employ compression at the client before sending data to memcached. For example, this

behavior is the default in the spymemcached client library for objects above a user-tunable threshold. In this subsection, we explore the implications of compression on performance. We gathered objects of a variety of sizes, and then evaluated the overhead to serialize, deserialize, compress, and decompress the objects. We also report compression ratios.

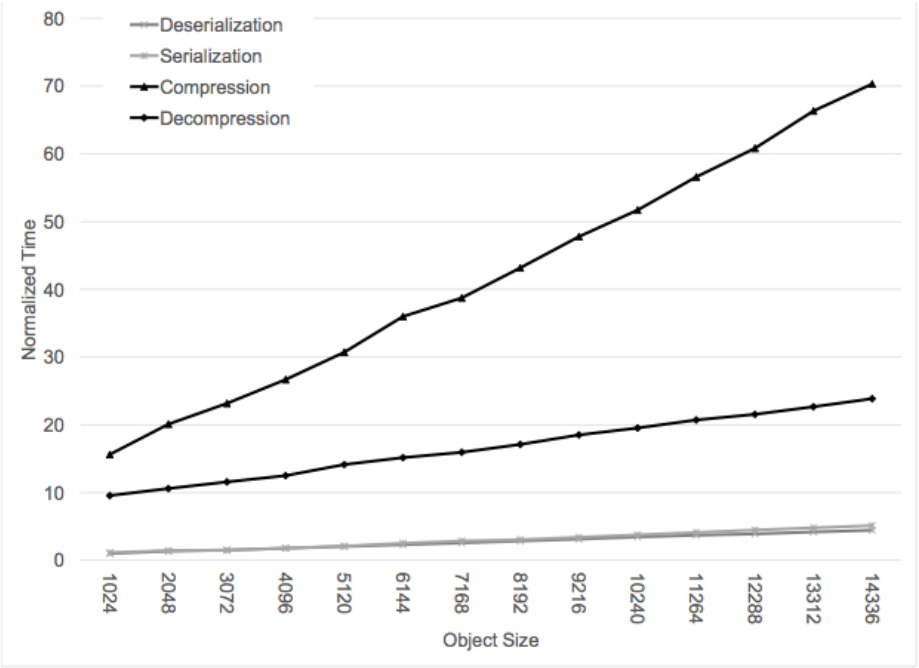


Figure 8: Computational cost of compression and serialization.

Figure 8 presents the average time to serialize, deserialize, compress, and decompress the protocol buffers used in our workloads. All results are normalized to the time to deserialize a 1024-byte object. For reference, the

average deserialization time for such objects is 9.2 microseconds.

Deserialization is marginally faster than serialization, with both operations scaling sub-linearly in the size of the object: a 14K byte object deserializes in only 4.4x the time of a 1K byte object. However, the costs for compressing and decompressing are both significantly higher. Not only is compression slowest, its cost increases most rapidly with the size of the object, though still at a linear rate. From this result, we argue that storing compressed data in the cache is likely to nullify any gains to be achieved by using fetch-and-phi.

This raises an interesting question: what impact can be expected when compression is disabled? While we disabled compression in all of our experiments, it is enabled by default in the spymemcached client library, and is applied for objects above a threshold. The motivation is that compression can reduce both network bandwidth and the amount of RAM needed at the memcached server.

Object Size	Reduction	Object Size	Reduction
1024	19%	8192	33%
2048	23%	9216	34%
3072	27%	10240	35%
4096	29%	11264	35%
5120	30%	12288	35%
6144	31%	13312	36%
7168	32%	14336	36%

Table 2: Compression of protocol buffers in our workload

Table 2 presents the average size reduction we observed when compressing objects from the traces we gathered. Savings begin at 19%, and rise steadily to 36%. Beyond 14K bytes, the compression rate remains constant.

The tradeoff between using compression and using fetch-and-phi is nuanced. If fetch-and-phi can halve the number of round-trip communications, then the bandwidth savings will be greater than the 36% per-round-trip savings from compression. However, this savings does not apply to get operations.

Similarly, without programmer intervention, it appears that disabling compression will result in decreased capacity at the memcached server, since larger objects will be stored for the same workload. The impact can be even more severe than anticipated, since memcached uses a slab allocator: larger objects may spill into larger slab classes, and hence incur more internal fragmentation. On the other hand, if an application aggressively employs filtering along with fetch-and-phi, it may be possible to perform less denormalization of data. This, in turn, can

reduce the number of objects stored in the cache, as well as the number of objects stored in the persistence layer. We leave further exploration of this relationship as future work.

7 Related Work

There has been substantial research into distributed key/value stores, both persistent and in-memory. Due to the performance-critical nature of these systems, they are increasingly adopting complex low-level systems techniques to achieve peak performance [4, 13]. In some cases, these systems are also tailored to specific workloads, such as Facebook's TAO [1]. TAO's cache is not a lookaside, but rather a write-through cache, tightly bound to an underlying MySQL-based persistent store. We believe that our work, which studies the performance of fetch-and-phi in an unmanaged language, is complementary to these efforts. In all cases, there is an awareness that decreasing communication bandwidth and lowering the latency of accesses to the cache layer is crucial to overall system performance.

The filtering mechanism we discussed in this paper is similar to several prior proposals [2, 16, 19]. While these generally require the cache to be aware of the object layout, so that they can return specific fields of an object, there is no obstacle preventing such systems from supporting more complex filtering

operations. Our work takes the opposite approach, assuming nothing about the object format and leaving it up to the user-provided code to deserialize a copy of the data and compute over it. The best solution for real-world systems and applications is likely to fall somewhere in between: a limited set of operations, but optimized for a workload whose data layout is known by the cache so that copying can be avoided.

The first system we are aware of that supports arbitrary computation directly in the key/value store is Comet [5]. In Comet, objects can either be in an unknown format, or else Lua objects. In the latter case, objects can have triggers attached to them, which execute in response to gets and sets. Our work differs from Comet in a number of regards. From a performance perspective, Comet provides persistence, and hence there is much more room to mask the latency of object serialization and deserialization. In that regard, our work can be thought of as providing a lower bound on the best-case latency. More importantly, Comet focused on the security of extensions and the reliability of the overall system. Whereas we studied memcached, which is rarely shared among applications, Comet was intended to support web services with untrusted clients. Thus security of extensions was a more significant factor than in our work.

Another system which bears relation to our work is OOlong [15]. OOlong used

the analogy of database triggers to describe a technique for performing computation on a key/value store. OOlong provides some features that are more general than fetch-and-phi, such as allowing the get of object O to cause an update to some other object K. Unfortunately, we are not aware of any performance results for this system.

8 Conclusions and Future Work

In this work, we studied the impact of supporting client-requested computation within the context of a memcached server. Our extensions to memcached, which are based on the concept of a fetch-and-phi operation, have a minimal footprint (under 400 lines of code) and provide an orthogonal mechanism supporting fetch-and-phi and filtering. Our implementation also supports operations that do not send the new object as part of the response. Using traces from Comcast as the basis for our evaluation, we showed that fetch-and-phi operations have the potential to reduce overhead by over 30%.

There are several exciting directions for future work. First, we showed that the cost of deserializing byte streams into objects was a critical overhead. Enabling operations (especially read-only operations) directly on the serialized form of objects could provide a significant performance boost. However, even this is unlikely to make low-cost filters practical. A second appealing research direction

is to consider performing computation (again, especially readonly operations) directly on the object as it is stored in the cache. This may require new approaches to locking inside of memcached, to prevent deadlocks with concurrent multi-object gets and sets. One appealing approach may be to use transactions [17]. Third, we have not yet studied multi-key operations (i.e., a generalization of fetch-and-phi to multiword compare-and-swap). Among other challenges, this introduces the need to ensure deadlock-freedom. However, when coupled with careful selection of hash functions, this could lead to substantial improvements in filtering, as a single request could aggregate data from multiple keys and provide it back to the client in a single message.

Another direction to study is security and reliability. Our mechanism defaults to extensions provided as unmanaged C code, loaded into the memcached process as shared objects. Sandboxing this code would increase reliability, though the overheads may be too high. Similarly, it would be possible to run extensions in an interpreter, though it is not clear that running an interpreter within the memcached process would be any more reliable than running untrusted client code. Whatever reliability mechanism is employed at an installation, we believe our work will provide a useful baseline for measuring performance overheads.

Acknowledgments

We thank John McCann for many helpful discussions during the conduct of this research. We also thank Sree Kotay and Paul Bosco, who introduced us to the top-of-the-hour problem. Thanks to Michael Spear for his guidance and aid.

This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. This work was also supported through a Comcast TechFund grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of Comcast.

9 References

[1] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage

System for Structured Data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, Seattle, WA, Nov. 2006.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, Oct. 2007.

[4] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, Apr. 2014.

[5] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. Levy. Comet: An Active Distributed Key-Value Store. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[6] Google Inc. Protocol Buffers, 2014. https://developers.google.com/protocol-buffers/.

[7] P. K. Gunda, L. Ravindranath, C. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[10] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the EuroSys2013 Conference*, Prague, Czech Republic, Apr. 2013.

[11] C. Kozyrakis. Resource Efficient Computing for Warehouse-scale Datacenters. In *Proceedings of the Design, Automation, and Test in Europe Conference*, Grenoble, France, Mar. 2013.

[12] H. Lim, B. Fan, D. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[13] H. Lim, D. Han, D. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, Seattle, WA, Apr.

2014.

[14] memcached.org. Memcached, 2014. http://memcached.org/. [15] C. Mitchell, R. Power, and J. Li. Oolong: Programming

Asynchronous Distributed Applications with Triggers. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[16] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, and B. Rinald. YCSB++ : Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Cascais, Portugal, Oct. 2011.

[17] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[18] M. Saad and B. Ravindran. HyFlow: A High Performance Distributed Software Transactional Memory Framework. In *Proceedings of the International ACM Symposium on High Performance and Distributed Computing*, San Jose, CA, June 2011.

[19] The Apache Software Foundation. Apache HBase, 2014.

http://hbase.apache.org/.

[20] The Apache Software Foundation. The Apache Cassandra Project, 2014.

http://cassandra.apache.org/.

[21] YourKit, LLC. spymemcached, 2014.

https://code.google.com/p/spymemcached/.

Vita

Adam Schaub was born in Richmond, Indiana on October 20th, 1991. At the age of 5, he moved with his parents Barry and Linda Schaub to Middleburg Pennsylvania, where he graduated Midd West High Schaub. From 2010-2014 Adam attended Lehigh University and obtained a B.S. in Computer Engineering in May 2014. He is currently finishing an M.S. in Computer Science from Lehigh University, to be earned in May 2015. During his time at Lehigh, Adam has been a Teaching Assistant for ECE 33 (Introduction to Computer Engineering) and ECE 138 (Digital Systems Laboratory).

Fetch-and-Phi in Memcached

Adam Schaub

ams314@lehigh.edu

Lehigh University

May 2015