NbQ-CLOCK: A Non-blocking Queue-based CLOCK Algorithm for Web-Object Caching

Gage Eads

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-174
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-174.html

October 25, 2013

Acknowledgement

Special thanks to Juan Colmenares for providing insight on replacement algorithms, non-blocking data structures, and technical writing; helping formulate the experiments; developing KVTG; and contributing to the text of this report. Juan Colmenares has been an invaluable mentor during my internship at Samsung and my graduate work at UC Berkeley. I'd also like to thank John Kubiatowicz, Krste Asanovic, Daniel Waddington, Fengguang Song, Jilong Kuang, and Reza Dorrigiv for their thoughtful feedback on early drafts of this report.

This research was conducted during a summer internship at Samsung Research America - Silicon Valley.

NbQ-CLOCK: A Non-blocking Queue-based CLOCK Algorithm for Web-Object Caching

by Gage Eads

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

John D. Kubiatowicz
Research Advisor

Date

* * * * * *

Krste Asanović
Second Reader

Date

Abstract

Major Internet-based service providers rely on high-throughput web-object caches to serve millions of daily accesses to frequently viewed web content. A web-object cache's ability to reduce user access time is dependent on its replacement algorithm and the cache hit rate it yields. In this report, I present NbQ-CLOCK, a novel, lock-free variant of the Generalized CLOCK algorithm particularly suited for web-object caching. NbQ-CLOCK is based on an unbounded non-blocking queue with no internal dynamic memory management, instead of the traditional circular buffer. My solution benefits from Generalized CLOCK's low-latency updates and high hit rates, and its non-blocking implementation makes it scalable with only 10 bytes per-object space overhead.

I compare the solution to existing algorithms, including Intel's Bag-LRU, and demonstrate that NbQ-CLOCK's fast `update` *operation scales well with the number of threads and in a in-memory key-value store prototype, NbQ-CLOCK offers an overall throughput improvement of as much as 9.20% over the best of the other algorithms. In addition, NbQ-CLOCK's hit rate exceeds the next best algorithm's hit rate by as much as 1.40%.*

1 Introduction

Minimizing the service response time experienced by users is very important for global-scale Internet-based service providers, such as Amazon, Facebook, Google, Samsung, and Twitter. Service response time is a performance metric those providers optimize to differentiate themselves from the competition in order to retain existing users and attract new ones.

One way to reduce service response times is with web-object caching. This technique temporarily stores recently or frequently accessed remote data (*e.g.,* popular photos, micro web logs, and web pages) in a nearby location to avoid slow remote requests when possible. It is used in search engines and content delivery networks, as well as (locally) in web browsers.

Web-object caches are often implemented as key-value stores. In general, key-value stores provide access to unstructured data through read and write operations, where each unique key maps to one data object. For web-object caches, URLs are often the keys and web objects the

values. Popular key-value stores used for web-object caching operate purely *in memory*; a prime example is Memcached [1] (summarized in Section 2). In key-value stores of this type, frequently accessed data items are stored in cache servers' volatile RAM. By servicing the cache hits out of main memory, these accesses can have low round-trip times. Only object requests that miss in the cache are routed to the slower, non-volatile storage layer; in this way, the key-value store mitigates the load on the storage layer. Distributed in-memory key-value stores, where cached contents are distributed across a cluster of machines, are crucial for serving frequently accessed data to very large number of clients with a satisfactory end-to-end latency.

Inherent to every cache is the *eviction problem*: When the cache is full, the system needs to decide which items to evict to make room for new incoming items. The approach used to evict items in a web-object cache can dramatically impact the performance of the Internet-based services the cache assists, and that influence on service performance is characterized by the *cache hit rate*.

Replacement algorithms are devised to tackle the eviction problem. In the context of web-object caching, a replacement algorithm's API consists of four operations: `insert` adds a new item to the replacement data structure, `delete` removes an item from the replacement data structure, `update` notifies the replacement algorithm of an existing item that has been accessed, and `evict` selects one or more cached items for eviction.

A recent study of Facebook's Memcached traces [3] shows that large-scale key-value store workloads are read-heavy, and key and value sizes vary. Hence, a replacement algorithm for an in-memory key-value store should support variably-sized objects and allow for low-latency, scalable updates and evictions, as well as high cache hit rates.

The most common replacement algorithms are the *Least Recently Used* (LRU) algorithm and its derivatives. LRU, as its name implies, evicts the least recently accessed item. Its derivatives (*e.g.,* pseudo-LRU and CLOCK) trade-off hit rate in favor of lower space complexity or implementation cost.

The classic LRU as well as most variants (*e.g.,* Bag-LRU [18]) are implemented using a linked list. When an item is accessed, it is moved in the list(s) to indicate its recent access. LRU and its variants typically use locks to control the access of concurrent operations that manipulate the

list(s). Locks can severely limit the scalability of updates and evictions, and increase their latency. Thus, LRU variants often incorporate techniques to mitigate lock contention.

CLOCK [6] is a well-known memory-page replacement algorithm. It maintains a circular buffer of reference bits, one for each memory page, and a buffer pointer ("clock hand"). When a page is referenced, its reference bit is set to indicate a recent access. To evict a page, the clock hand sweeps through the buffer and resets each non-zero bit it encounters, until it finds an unset bit; then, the corresponding page is evicted. Unfortunately, CLOCK's hit rate can suffer because with a single bit reference it cannot differentiate access frequency from access recency (*i.e.,* hot items and those accessed once per clock sweep appear identical). To solve this problem, Generalized CLOCK [17] replaces each reference bit with a counter. A recent access is indicated by a non-zero counter, and access frequency is indicated by the value of the counter.

While their fast update path is ideal for read-mostly web-object caches, CLOCK and its variants assume a fixed number of cache entries, which is *not* the case in the web-object caching domain. This limitation renders CLOCK and existing variants impractical in this domain.

In this report, I present **N**on-**b**locking **Q**ueue-based **CLOCK** (NbQ-CLOCK). It is a novel, lock-free variant of the Generalized CLOCK replacement algorithm particularly suited to caches containing a variable number of items with different sizes. To the best of my knowledge, prior to this work no CLOCK implementation existed that could effectively handle such dynamically-sized caches, which are essential to web-object caching.

NbQ-CLOCK, described in Section 4, circulates the cached items through the eviction logic, instead of iterating over the items like prior CLOCK variants. The efficient and scalable circulation of items is enabled by the use of a non-blocking concurrent queue, which by being unbounded also allows NbQ-CLOCK to handle caches with variably-sized items. NbQ-CLOCK's other appealing attributes are simplicity and low memory-usage overhead.

In Section 5, I empirically evaluate NbQ-CLOCK against other replacement algorithms applicable to Memcached, including Intel's Bag-LRU [18]. I focus on Memcached [1] because it is widely deployed and has recently received considerable attention from industry and the research community [7, 16, 14, 18, 11, 12]. I demonstrate that NbQ-CLOCK's low update latency scales

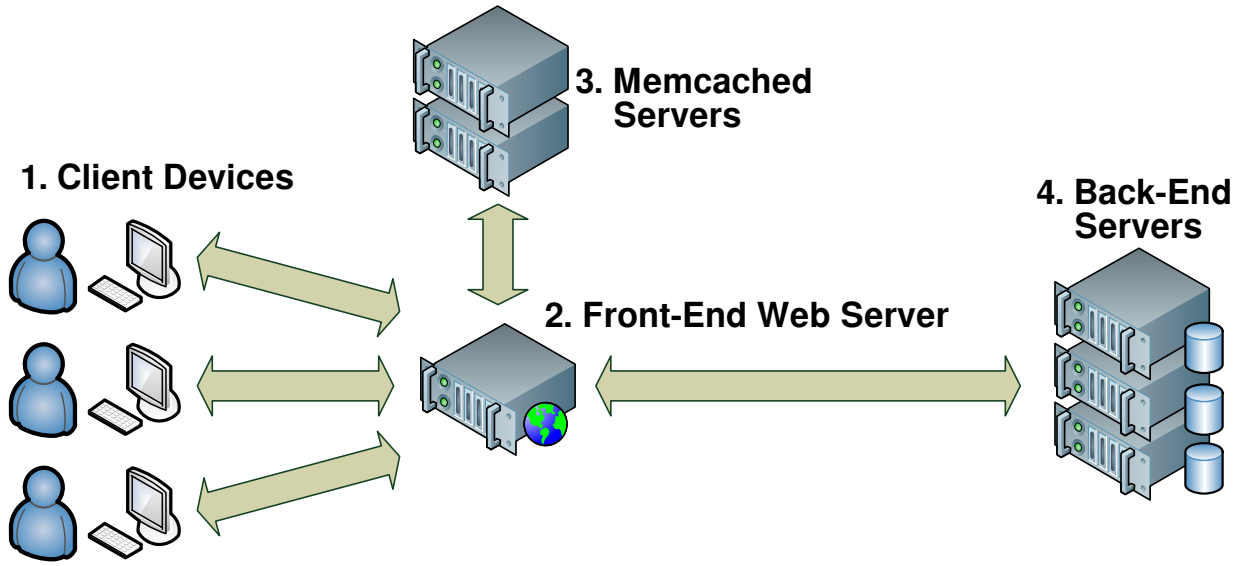

Figure 1: Memcached side-cache deployment.

well with the thread count, and that it exhibits better throughput scaling than the other algorithms. Moreover, when used in a in-memory key-value store prototype, NbQ-CLOCK offers an improvement on the system's throughput of as much as 9.20% over the best of the other replacement algorithms. I also show that NbQ-CLOCK's cache hit rates exceed the state-of-the-art by as much as 1.40%, a modest relative margin that can have large impact on global Internet-based services that process billions of requests per day.

2 An Overview of Memcached

Memcached [1, 16, 2] is a widely deployed web-object caching solution. Currently used at Facebook, Twitter, YouTube, and Zynga, this open-source in-memory key-value store consists of a hash table coupled with a replacement algorithm. Memcached is typically deployed in a "side-cache" configuration, depicted in Figure 1. In this configuration end users, via client devices (1), send requests to the front-end web servers (2). The front-end servers then attempt to resolve each end-user request from one or more local Memcached servers (3) by, in turn, sending `GET` requests to them. The front-end servers perform protocol-bridging from the web request to the Memcache protocol, and may also perform Memcached-server selection based on the requested object key. If a cache miss occurs, the front-end server handling the end-user request forwards it to the back-end database servers (4) that carry out the computation and IO operations to produce the result. On

receiving the result, the front-end server both sends the answer to the client and updates the cache by issuing a `SET` request to the appropriate Memcached server. Memcache protocol includes additional semantics, such as lifetime of cached entries and explicit modification (*i.e.,* `REPLACE` and `DELETE` requests) of key-value pairs.

On a single server node, a Memcached instance consists of a number of worker threads that serve hash table requests. These include `GET`, `DELETE`, `ADD`, `REPLACE`, and `SET` (which replaces a data item if it exists in the cache; otherwise, it adds a new item). Key and value sizes can vary, and Memcached utilizes multiple slab allocators to optimize for memory efficiency and to minimize fragmentation.

An analysis of Facebook's Memcached traces over a period of several days [3] revealed the following about large-scale key-value store workloads. They are read-heavy, with a `GET`/`SET` ratio of 30:1, and request sizes are seen as small as 2 bytes and as large as 1 Mbytes.

3 Replacement Algorithms for Memcached

In this section, I describe two existing replacement algorithms used in Memcached [1]: the algorithm shipped with its stock version, and Bag-LRU [18]. In addition, I present *Static CLOCK*, a conceivable, but ineffective extension to CLOCK that statically over-provisions for the largest number of items that an instance of an in-memory key-value store (*e.g.,* Memcached) can possibly accommodate in order to support variably-sized objects. I discuss the drawbacks of Static CLOCK in this section and experimentally evaluate the three algorithms against NbQ-CLOCK in Section 5.

3.1 Memcached LRU

The replacement algorithm in the stock version of Memcached is a per-slab class LRU algorithm. Every item in Memcached has a corresponding item descriptor, which contains a pointer for the LRU list. This algorithm uses a doubly-linked list to support arbitrary item removals, which is important for `DELETE` requests and expired items. A global cache lock protects the LRU list from concurrent modifications, and fine-grained locks protect the hash table. Lock contention on the

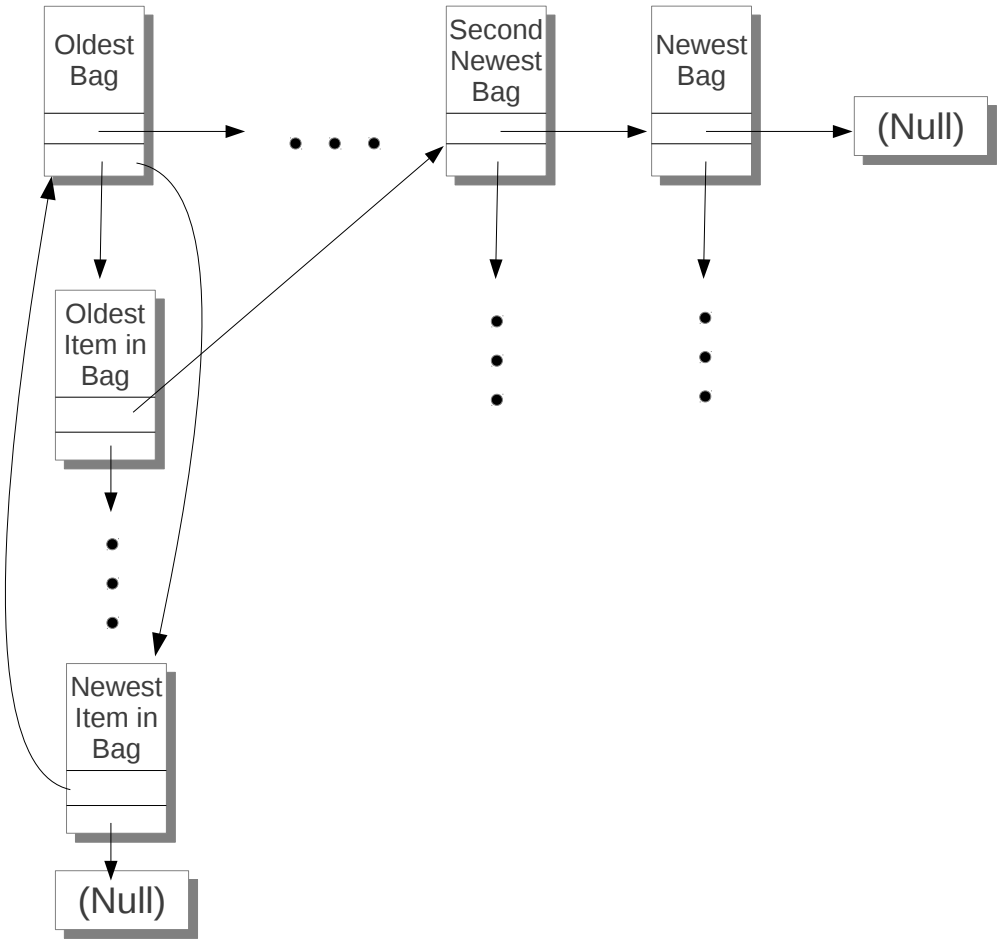

Figure 2: Bag-LRU's data structure [18].

global lock greatly impairs intra-node scalability, though this was not a concern at the time of Memcached's initial development. Instead, the developers sought a simple mechanism to ensure thread-safety.[1] This synchronization solution is clearly unscalable, and there have been many attempts at solving it.

3.2 Bag-LRU

Memcached's poor scalability within the node motivated the development of the Bag-LRU replacement strategy [18]. Bag-LRU is an LRU approximation designed to mitigate lock contention. Its data structure comprises a list of multiple timestamp-ordered "bags", each containing a pointer to the head of a singly-linked list of items (see Figure 2). The developers leveraged both pointer variables from Memcached's LRU for Bag-LRU; the next pointer serves the same purpose, but the back pointer points to the item's bag. Bag-LRU keeps track of the two newest bags (needed for updates) and the oldest bag (needed for evictions).

By design, Bag-LRU enables low-latency, scalable updates and supports parallel eviction operations. Bag-LRU's `update` operation consists of writing the newest bag's address to the item's back pointer and updating the item's timestamp. If the item resides in an older bag's linked list, it is the job of a *cleaner thread* to migrate the item to the list indicated by its back pointer. If the back pointer indicates the newest bag, the cleaner thread instead places the item in the second-newest bag to avoid contention with concurrent `insert` operations.

The lock-free `insert` operation places the item in the newest bag's list. To do so, a worker thread repeatedly attempts to append the item to the tail of the list using the atomic compare-and-swap (CAS) operation. If the CAS fails, one or more other threads successfully appended to the list, so the worker thread traverses the bag's list until it finds a `NULL` next pointer and retries.

An eviction requires grabbing a global eviction lock to determine the oldest bag with items in it, and then locking that bag. Once the bag is locked, the evicting thread chooses the first item available to evict. Bag-LRU requires locks for evictions and deletes to prevent the cleaner thread from simultaneously removing items, which can result in a corrupted list.

3.3 Static CLOCK

There are two ways to extend CLOCK in order to support dynamically-sized objects. These are:

[1]Brad Fitzpatrick, Memcached's original developer, built it with a "scale out, not up" philosophy [8], at a time when multi-core chips were just entering the market. Scale-up, however, is important for Memcached deployments to rapidly serve web-objects stored on the same node in parallel.

- Statically pre-allocate a bit buffer that is large enough to support the worst-case (*i.e.,* most) number of items. I refer to it as *Static CLOCK*.
- Replace the underlying data structure with a list. This is the alternative I pursue in this report and the solution is described in Section 4.

For completeness, I discuss and evaluate Static CLOCK, and show that the list-based approach is much more effective for web-object caching.

Static CLOCK is more complex than a pre-allocated bit buffer. Figure 3 depicts the data structures needed for Static CLOCK. In the likely case that the cache fills before all bit-buffer entries are used, the clock-sweep procedure needs some mechanism to differentiate used and unused buffer entries. Likewise, this approach requires a way to map from a bit-buffer entry to the corresponding item descriptor. Both problems can be solved with a pre-allocated pointer buffer, with one pointer-buffer entry per bit-buffer entry. An invalid bit-buffer entry is then indicated by a `NULL` value in the pointer-buffer entry. Furthermore, each item's descriptor contains the index of its corresponding entries into the two buffers to provide direct access to the entries for the `update` and `delete` operations. In other words, the index in the item's descriptor acts as a "back pointer" to the buffer entries. In total, the replacement strategy overhead is 65 bits times the worst-case number of items, plus 8 bytes for every cached item's index (assuming 8-byte pointers).

Static CLOCK's operations are as follows:

- `insert` searches for a `NULL` entry in the pointer buffer, then fills the pointer entry with the item descriptor's address using an atomic CAS instruction. If the CAS succeeds, the `insert` operation saves the item's array index in the item descriptor; otherwise, `insert` finds a different `NULL` entry and retries.
- `update` sets the item's reference bit, using the index stored in the item descriptor.
- `evict` performs the classic clock-sweep procedure – *i.e.,* sweeping the clock hand along the reference-bit buffer, resetting each reference bit until an unset bit is encountered – but only on valid items (with a non-`NULL` pointer entry). When an item with a unset reference bit is found, the `evict` operation sets the item's pointer entry to `NULL` using an atomic CAS. If the CAS succeeds, the item is evicted; otherwise, the operation continues the clock-sweep procedure.

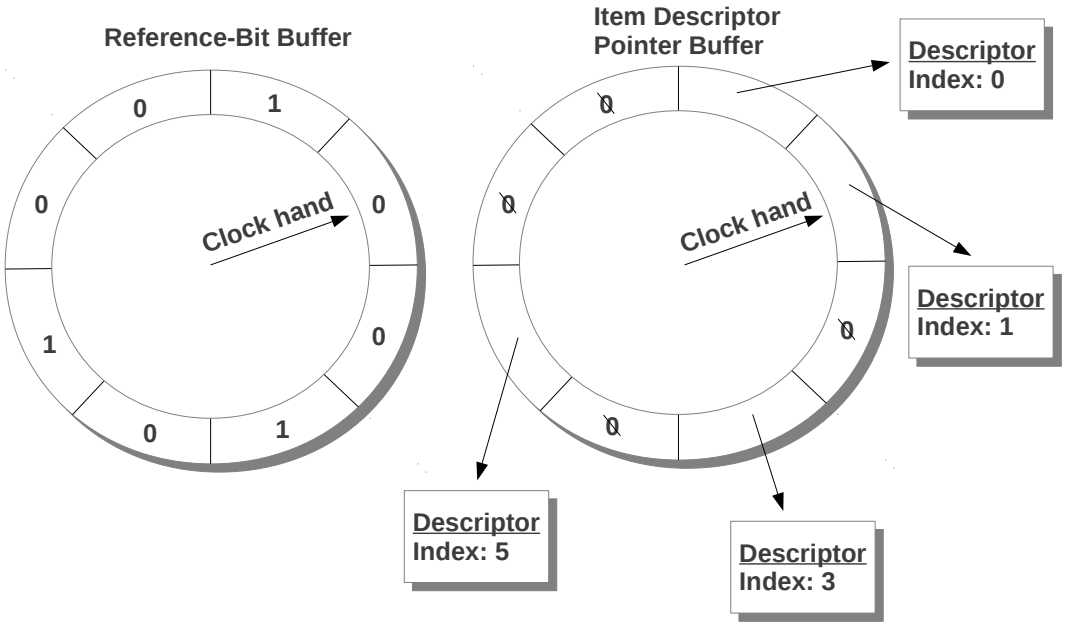

Figure 3: Static CLOCK's data structures. In addition to the circular buffer of reference bits, Static CLOCK uses a buffer of item-descriptor pointers to indicate whether a given CLOCK entry is occupied by an item. An entry in the buffer of pointers is non-`NULL` if and only if the CLOCK entry is in use, in which case the entry points to the item's descriptor.

- `delete` uses the index stored in the item's descriptor to find the item's pointer-buffer entry and atomically sets the entry to `NULL`.

4 NbQ-CLOCK

Non-**b**locking **Q**ueue-based **CLOCK** (NbQ-CLOCK) is a novel lock-free variant of the Generalized CLOCK replacement algorithm [17]. The primary difference between NbQ-CLOCK and previous CLOCK variants is that it circulates the cached items through the eviction logic (see Figure 4), instead of iterating over the items (by moving the clock hand). The efficient and scalable circulation of items is enabled by the use of a non-blocking concurrent queue, as opposed to the traditional statically allocated circular buffer in prior CLOCK variants. Moreover, the use of an unbounded queue allows NbQ-CLOCK to handle a dynamically-sized cache (containing variable number of items with different sizes), which is key for web-object caching (*e.g.,* Memcached [1]).

NbQ-CLOCK's non-blocking queue is based on a singly-linked list, and its algorithmic details

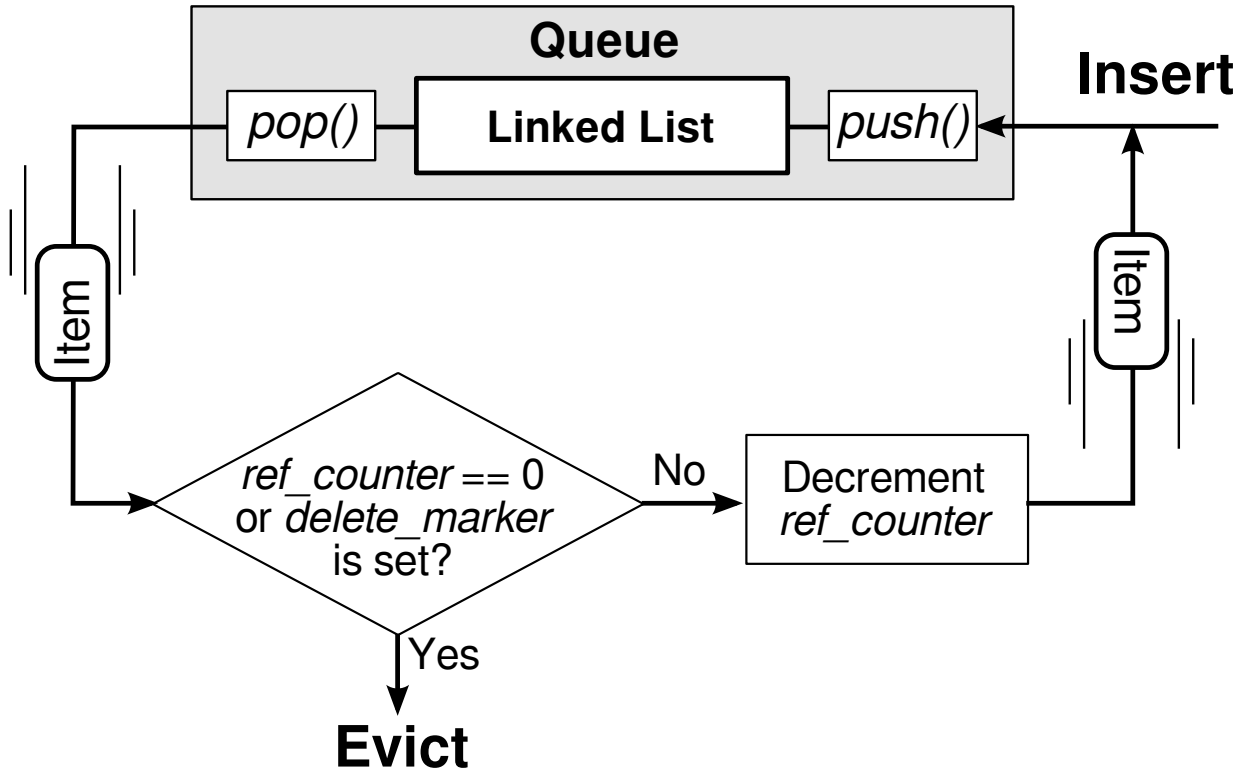

Figure 4: Process flow of queue-based `evict`, `delete`, and `insert` operations. The data items are the moving elements, as opposed to the clock hand in the traditional CLOCK.

are presented next in Section 4.1. NbQ-CLOCK stores bookkeeping information in each linked-list node; a node includes:

- a reference counter,[2]
- an atomic delete marker, and
- a pointer to the next node.

In addition, each linked-list node contains a pointer to a unique data item (*e.g.,* a key-value pair). I also refer to these nodes as *item descriptors*.

NbQ-CLOCK maintains the traditional CLOCK interface and its operations are:

- `insert` pushes an allocated list node into the queue. The node's reference counter is initialized to zero and the delete marker to `false`.
- `update` increments the node's reference counter.

[2]or a reference bit for a non-blocking variant of the original CLOCK [6].

- `delete` removes an item from the queue. The non-blocking queue can only remove items from the head of the queue, but web-object caches must support the ability to delete arbitrary objects. To support arbitrary object deletions, each item descriptor contains a "delete marker". This binary flag is set atomically and indicates whether the corresponding item is deleted from the cache (*e.g.,* as the result of a `DELETE` request for Memcached). During the clock sweep operation, the worker thread frees the memory of any items whose delete marker is set, regardless of the reference count value.
- `evict` pops the head of the queue, and if the node's reference counter is zero or its delete marker is set, evicts the item. If the item is not suitable for eviction, its reference counter is decremented and it is recycled back into the queue.

Figure 4 depicts NbQ-CLOCK's `insert`, `delete`, and `evict` operations.

As a performance optimization, the implementation of `evict` collects the popped items in a linked list, and pushes the head of this list back onto the queue once an item is evicted. Such an optimization will deplete the queue if all reference counters are non-zero; therefore, one must limit the size of such lists of popped items to a bound B, where B is at most $(queue\_size \,/\, thread\_count) - 1$. Then, the `evict` operation can perform at most $\lceil (number\ of\ items\ swept \,/\, B) \rceil$ push operations per sweep. The drawback of collecting popped items is that the resulting item ordering after re-inserting in the queue may differ slightly from what existed prior to the sweep (particularly if two or more threads are performing `evict` simultaneously), which causes the implementation to deviate slightly from the original CLOCK algorithm. Regardless, I do not believe this has any significant impact on NbQ-CLOCK's hit rate.

My original implementation of `update` atomically modified the reference counters to ensure that no increments or decrements were lost in a data race. If counter modifications are lost, NbQ-CLOCK may make different replacement decisions than the original CLOCK algorithm. However, the likelihood that a data race occurs is very small. In the x86 ISA, non-atomic increment and decrement operations have the smallest possible data race window: a single assembly instruction. Even if two or more threads execute that instruction simultaneously, a data race requires that they update the same item's counter in a cache containing possibly *millions* of items. Also, it is unclear whether any different replacement decisions as a result of a race would be worse than CLOCK.

Further, atomic operations on the test platform (described in Section 5.1) incur a performance overhead. In a single-threaded experiment, I measured an average of 167 CPU cycles to complete non-atomic increments vs. 338 CPU cycles for their atomic counterparts. One source of this extra latency is the hardware thread flushing its load and store queues before executing the atomic operation [9]. Since the goal is a replacement algorithm with *low-latency* overhead, particularly for `updates`, I choose to use non-atomic counter updates.

4.1 Underlying Non-blocking Queue

NbQ-CLOCK is operates on top of a non-blocking concurrent queue. Hence, the clock hand is not an explicitly maintained variable, but instead is implicitly represented by the head of the queue.

For the non-blocking queue I use Michael and Scott's algorithm [15], but I have made two key optimizations. First, the queue performs no memory allocation in its `init`, `push`, or `pop` operations. Second, NbQ-CLOCK's `push` and `pop` methods operate on nodes, not the data items themselves, so that one can re-insert a popped node at the tail with no memory management overhead. These optimizations primarily benefit NbQ-CLOCK's `evict` operation.

Listing 1 shows the queue's `init` operation. The algorithmic details of the non-blocking `push` and `pop` operations are presented in Listings 2 and 3, respectively.

```
1 void init(Node* node) {
2   Q->head = Q->tail = node;
3   Q->head->next = NULL;
4 }
```

Listing 1: The `init` operation for the non-blocking queue.

```
1  void push(Node* node) {
2    node->next = NULL;
3    while(true) {
4      tail = Q->tail;
5      next = tail->next;
6      if (tail != Q->tail)
7        continue;
8      if (next == NULL) {
9        if (CAS(&tail->next, next, node))
10         break;
11     } else {
12       CAS(&Q->tail, tail, next);
13     }
14   }
15   CAS(&Q->tail, tail, node);
16 }
```

Listing 2: The `push` operation for the non-blocking queue.

```
1  Node* pop() {
2    data_t head_data;
3    while (true) {
4      head = Q->head;
5      tail = Q->tail;
6      next = head->next;
7      if (head != Q->head)
8        continue;
9      if (head == tail) {
10       if (next == NULL)
11         return NULL;
12       CAS(Q->tail, tail, next);
13     } else {
14       if (next == NULL)
15         continue;
16       head_data = next->data;
17       if (CAS(Q->head, head, next))
18         break;
19     }
20   }
21   head->data = head_data;
22   return head;
23 }
```

Listing 3: The `pop` operation for the non-blocking queue.

Michael and Scott's algorithm consists of a singly-linked list of nodes, a tail pointer, and a head pointer, where the head always points to a dummy node at the front of the list. Their algorithm uses a simple form of snapshotting, in which the pointer values are re-checked before and during the CAS operations, to obtain consistent pointer values.

One manner in which NbQ-CLOCK's queue and Michael and Scott's algorithm differ is their queue performs a memory allocation on each `push` and a de-allocation on each `pop`. However, if a clock sweep operation inspects n items before finding one suitable for eviction, it pops n list

nodes and pushes $(n-1)$ list nodes – resulting in n memory frees and $(n-1)$ memory allocations. On the contrary, the non-blocking queue takes the memory management logic out of `push` and `pop` so that every `evict` incurs the minimum amount of memory management overhead: a single de-allocation. Further, most memory allocators use internal locking, in which case the queue is not fully non-blocking. I avoid this problem by taking the allocators out of the queue.

I also remove the modification counters in Michael and Scott's algorithm from the queue. The modification counters protect against the ABA problem[3], which can corrupt the queue's linked list. In the queue, there are four code paths that can experience the ABA problem: in Listing 2 between lines 4 and 12, in Listing 2 between lines 4 and 15, in Listing 3 between lines 4 and 17, and in Listing 3 between lines 5 and 12. In each code path the ABA problem occurs if, between reading the value that is compared in the CAS and executing the CAS, the compared value is cycled around the queue such that the CAS succeeds, but the swapped value is no longer valid. However, for sufficiently large queues, the likelihood that other threads can cycle through the queue in such a short time is effectively zero. Since key-value stores typically cache millions of items, I choose not to include modification counters in the NbQ-CLOCK queue.

5 Comparative Evaluation

In this section, I evaluate NbQ-CLOCK against the replacement algorithms Memcached LRU, Bag-LRU, and Static CLOCK, presented in Section 3. I focus on *update latency scalability*, *cache hit rate*, and *throughput*, which are performance metrics important to high-throughput, low-latency, read-mostly web-object caches.

The version of NbQ-CLOCK used in the experiments implements the Generalized CLOCK with 8-bit reference counters. The Static CLOCK, on the other hand, uses a reference bit per data item. In order to reduce Static CLOCK's memory-usage overhead, its reference-bit buffer is implemented as a packed array. This is an attempt to make Static CLOCK competitive in terms of the number of data items it can hold and thereby its hit rate. Consequently, since the test platform does not support bit-granularity writes, Static CLOCK's `update` operation is a read-modify-write

[3]The ABA problem can occur when, between reading a shared value of A and performing a CAS on it, another thread changes the A to a B and then back to an A. In this case, the CAS may succeed when it should not.

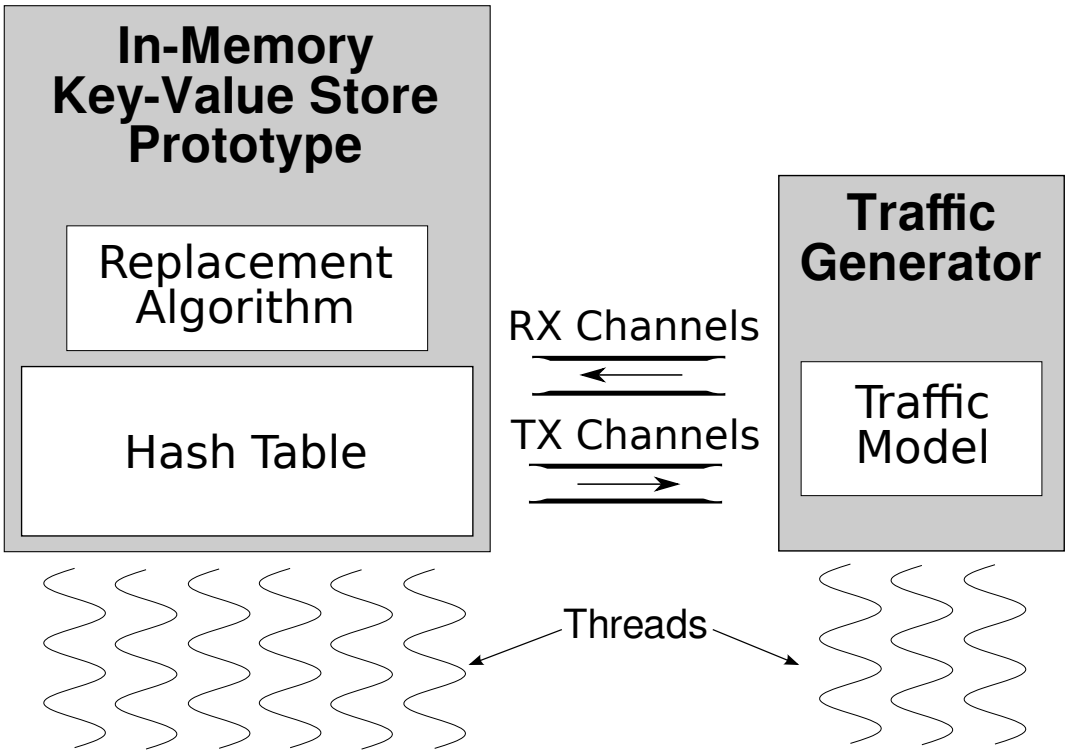

Figure 5: Structure of the test software platform.

loop using the atomic CAS instruction.

5.1 Experimental Setup

The test software platform consists of two C++ applications (see Figure 5):

- an *in-memory key-value store prototype*, which is Memcached-protocol conformant, and
- a *client traffic generator*, called KVTG.

The two applications run in Linux 3.2.0 and are connected through sockets, which enables both local and distributed tests. For the purposes of this report, I run both applications on the same machine and connect them via high-performance shared-memory channels [13].

The key-value store prototype primarily comprises a hash table and an item-replacement algorithm, plus the logic necessary to receive and parse client requests and generate and transmit responses. When in operation, each worker thread in the key-value store receives Memcached-protocol conformant requests on a receive (RX) channel, performs the hash-table and replacement

operations necessary to satisfy the requests, and transmits the results across a transmit (TX) channel. The hash table uses a bucket hashing with 2^{20} buckets and a fine-grained locking solution with 2^{12} locks (*i.e.,* 2^8 buckets protected per lock). I design the prototype to work with any replacement algorithm that implements an interface (*i.e.,* a C++ pure abstract class), whose virtual methods include `add_item()`, `delete_item()`, `update_item()`, and `evict_item()`. In this manner, I decouple the hash-table logic from the replacement algorithm.

The key-value store prototype, like Memcached [1], allows us to set the cache size, which is the memory limit for keys, values, and item descriptors (including replacement-algorithm bookkeeping). Also similar to Memcached, the prototype uses 7 slab allocators with sizes ranging in powers of two from 64 B to 4 KB. While per-slab class replacement logic is necessary in a deployment scenario,[4] I restrict my focus to a single replacement instance by choosing object sizes that use a single slab allocator.

KVTG is an application that generates Memcached-style requests (`GET`, `SET`, `DELETE`, etc.) according to user-supplied probability distributions. It allows creation of traffic with similar statistical properties of realistic workloads in production environments (*e.g.,* [3]). The aspects of KVTG that are configured by distributions are key appearance frequency, request type, request inter-departure time, and value size. KVTG uses a pair of transmit and receive threads for every key-value store prototype worker thread. As KVTG's companion, there is also a Python script capable of generating sets of unique keys according to a given key-size distribution.

The experimental platform is a quad-socket server containing the Intel E5-4640 2.4GHz CPU (8 cores, 16 hardware threads per socket) with 20 MB of last-level cache, and 128-GB DRAM overall. To minimize performance variability across test runs, I disable Turbo Boost, redirect interrupts to unused cores, fix each CPU's frequency to 2.4 GHz, and affinitize software threads to cores isolated from the Linux scheduler.

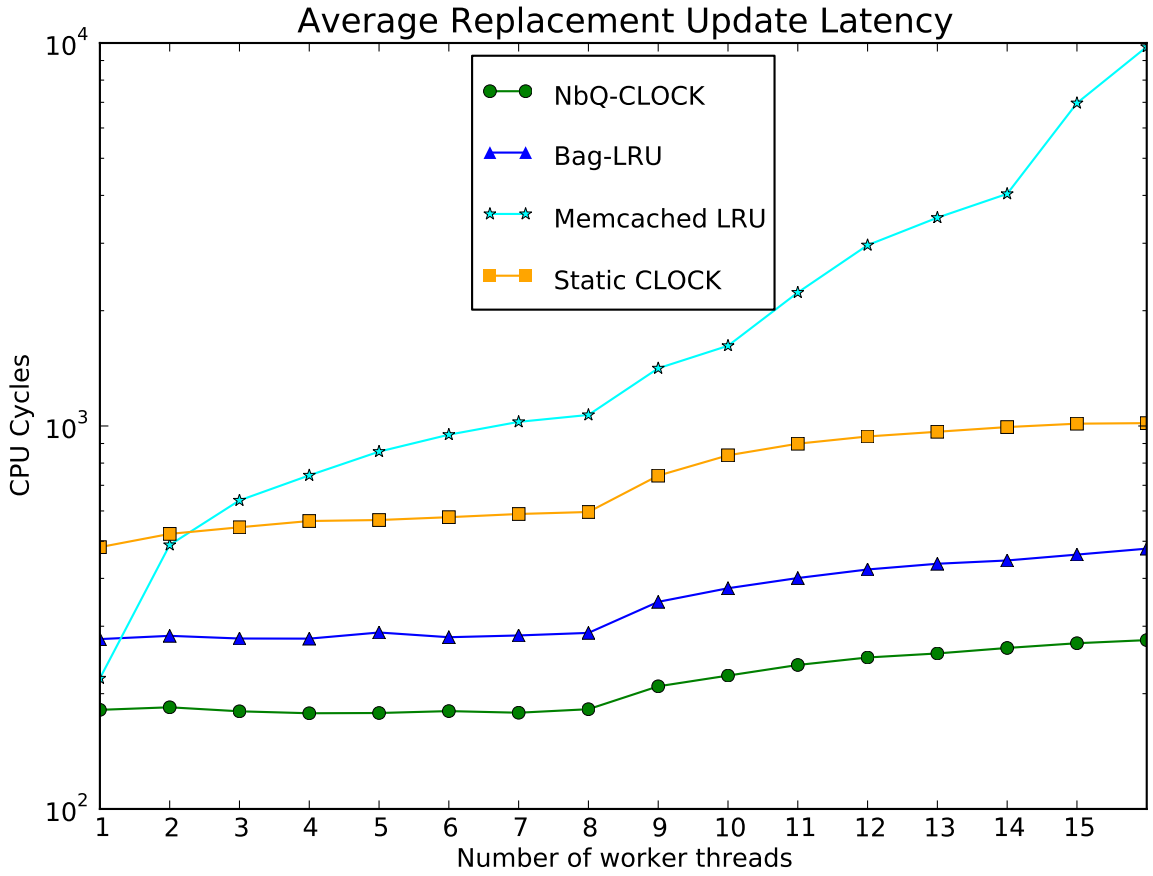

Figure 6: Scalability of replacement algorithms' `update` operation. Each data point is the average `update` latency over 10 million `GET` requests.

5.2 Update Latency Scaling

Update latency is the time to update a cached item's entry in the replacement data structure in the event of a cache hit. It is crucial for server performance that this process – the common case in a read-heavy workload – scales well.

In this experiment, I measure the average latency for the `update` operation of the replacement algorithms under evaluation with the worker thread count varying from 1 to 16. Each worker thread is pinned to a separate hardware thread, with the first eight threads on one CPU socket and the other eight on another CPU socket. I choose to use one hardware thread per core to avoid performance crosstalk with its sibling hardware thread.

The results are shown in Figure 6, with the y-axis (CPU cycles) plotted on a logarithmic scale. Each data point in the figure was calculated over 10 million `GET` requests, with key frequency determined by a power-law probability distribution. NbQ-CLOCK's update latency scales significantly better than Memcached LRU. Heavy contention for the global lock seriously hinders LRU's scalability, resulting in a 44.6x increase in mean update latency from 1 to 16 threads, while NbQ-CLOCK's mean update latency increases by 2.11x in that range.

Interestingly, Static CLOCK's `update` performs worse than NbQ-CLOCK despite both being CLOCK variants. The main reason is the read-modify-write loop with CAS in the `update` operation of the Static-CLOCK implementation. I observe that a single atomic CAS operation takes 483 cycles on average on the test platform for a single thread (see Figure 6), but the likelihood of CAS failing and the loop repeating grows with the number of threads, resulting in the observed update-latency increase. NbQ-CLOCK's non-blocking `update` operation, requiring a single increment operation, scales much better.

The Bag-LRU solution [18] scales well, but its average update latency is between 1.5x-1.73x that of NbQ-CLOCK. This is because, besides writing the back pointer, the `update` operation must also check if the newest bag is full and, if so, atomically update the global newest bag pointer.

[4] When a request causes an eviction, the evicted item must occupy enough dynamic memory to satisfy that request. To find an appropriately-sized item quickly, each slab class must have its own replacement logic.

The performance dip between 8 and 9 worker threads occurs in all algorithms and appears to be a memory-system artifact resulting from the 9th thread running on a second socket.

5.3 Cache Hit Rate

Another metric fundamental to a web-object cache is hit rate. Besides having lower-latency `update` operations, this experiment shows that NbQ-CLOCK is also superior to the alternative algorithms in terms of hit rate.

For this experiment, the key-value store prototype and traffic-generator program each run on a single thread. The two threads are pinned to separate hardware threads on different cores of the same CPU socket. As before (Section 5.2), I use one hardware thread per core to avoid performance crosstalk between sibling hardware threads.

I set a memory limit of 1 GB for slab allocators for keys, values, item descriptors, and replacement algorithm overhead. The key space is modeled by a standard normal distribution of 30 million items and the requests are 70% `GET`s and 30% `ADD`s. The traffic generator issues 15 million `SET` requests during the "warm up" phase, in which no data is collected. In the subsequent phase, 15 million requests are sent according to the key and request-type distributions and hit-rate data is collected.

I evaluate a range of object sizes (key plus value) from 64 B to 4 KB. For a fair comparison with NbQ-CLOCK and Bag-LRU, I count the memory allocated for the Static CLOCK's circular buffers against the memory limit.

Table 1 shows that NbQ-CLOCK's cache hit rates exceed the next best algorithm by as much as 1.40% (for 4 KB objects). While the relative improvement of NbQ-CLOCK's hit rate over Bag-LRU's may not seem significant, its significance becomes evident when considering realistic workloads of web-object cache systems. For instance, considering the workload characterization of live Memcached traffic reported in [3], an additional 1.40% hits of 4.897 billion requests in a day amounts to an additional 68.6 million cache hits per day. Furthermore, this workload characterization was generated from five server pools within one of many datacenters; the impact of an improved replacement algorithm compounds in a global Memcached deployment.

Object Size (bytes)	NbQ-CLOCK	Bag-LRU	Static CLOCK	LRU
	Number of items stored (millions)			
64	**5.263**	5.113	4.503	5.089
128	**4.006**	3.919	3.336	3.905
256	**2.711**	2.671	2.198	2.664
512	**1.647**	1.632	1.306	1.629
1024	**0.922**	0.918	0.721	0.916
2048	**0.491**	0.489	0.380	0.489
4096	**0.253**	**0.253**	0.196	**0.253**
	Hit rate			
64	**82.81%**	82.34%	81.32%	82.29%
128	**80.86%**	80.76%	80.34%	80.76%
256	**78.23%**	78.13%	77.14%	78.13%
512	**75.26%**	**75.26%**	74.06%	75.25%
1024	**72.20%**	72.05%	70.90%	72.07%
2048	**68.92%**	68.32%	67.29%	68.32%
4096	**65.60%**	64.20%	63.32%	64.19%

Table 1: Number of stored data items and hit rate for a 1 GB cache, with a standard normal key distribution, across various object sizes.

NbQ-CLOCK's higher hit rates for small objects can be attributed to its superior space efficiency. Compared to Bag-LRU, NbQ-CLOCK has six fewer bytes per item due to its use of one pointer for its singly-, not doubly-, linked list, plus the single-byte reference counter and single-byte delete marker. For smaller object sizes, NbQ-CLOCK's bookkeeping space advantage is more significant relative to the object size, allowing it to store more objects than the alternatives – 150 thousand more than Bag-LRU for 64 B objects, for instance.

Static CLOCK has poor space efficiency, despite my effort to improve this aspect in the implementation. The reason is that it requires a statically allocated reference-bit array and item pointer array for every possible item in the cache. In this experiment, 249.67 MB (25% of the memory limit) is dedicated to Static CLOCK's arrays, and the hit rate suffers as a result.

For larger object sizes, the space overhead advantage of NbQ-CLOCK is insignificant with respect to the object size – NbQ-CLOCK, Bag-LRU, and LRU can store nearly the same number of items for 2 KB and higher. In the case of 4 KB objects, NbQ-CLOCK, Bag-LRU, and LRU can only store a small fraction (0.84%) of the 30 million items (due to the 1 GB memory limit), so the eviction algorithm's intelligence is the main factor affecting the hit rate.

The reason for NbQ-CLOCK's hit-rate advantage for 1 KB and larger objects is that NbQ-CLOCK, as a variant of Generalized CLOCK, considers not only access recency, but also access frequency. As a result, frequently accessed items tend to persist longer in a cache with Generalized CLOCK than in one with LRU. This is particularly important when the number of key-value pairs the cache can store is small compared to the total set of pairs, and key appearance is governed by a power-law distribution (as in this experiment). For this distribution, a small fraction of the keys appear much more frequently than the rest, and NbQ-CLOCK keeps the key-value pairs in that fraction longer than the LRU variants do.

5.4 Throughput Scaling of Pure Replacement Algorithms

In this experiment, I evaluate the performance of NbQ-CLOCK, Memcached LRU, Bag-LRU, and Static CLOCK in the best case – *i.e.,* when the rest of the cache imposes no bottlenecks. This allows one to accurately measure the scalability (or lack thereof) of each replacement algorithm in

the cache.

To remove all bottlenecks, I replace the hash table with a pre-allocated array of keys and item descriptors (see Figure 5), and remove any dynamic memory management. Rather than allocating memory for an item descriptor on each `insert` and freeing on each `evict`, I use a "present" boolean flag to indicate whether or not the item is cached. Thus, if the cache operation is a `GET` on a present (cached) item, then the worker thread performs `update`. And, if the cache operation is an `ADD` on a new (uncached) item, the worker thread performs `evict` then `insert`. By designing the benchmark in this way, I replicate the behavior of the hash table without any of its scalability bottlenecks.

For this experiment, I define throughput as the time spent in the replacement algorithm operations divided by the number of requests (10 million). I use 70% `GET` and 30% `ADD` operations. I model the key-appearance frequency with a power-law distribution for 10 million unique keys, and use an object size of 128 B. I fill the cache during an initialization phase before running the experiment. In addition, the key-value store prototype uses the same thread configuration as that in the latency scaling experiment (Section 5.2).

The results are shown in Figure 7. Each cache stores between 33% and 40% of the 10 million objects, depending on the replacement algorithm. By storing at least 1/3 of the keys whose appearance is modeled by a power-law distribution, 91% of all `GET`s are successful. On the other hand, successful `ADD`s - wherein the key does not already exist - occur the other 9% of the time. NbQ-CLOCK, with its low-latency updates (276 cycles average for 16 threads), outperforms the other replacement algorithms. The other algorithms perform as one would expect from Figure 6: Bag-LRU performs better than Static CLOCK, and Memcached LRU scales poorly, particularly after crossing the socket boundary. As with the update latency scaling experiment in Section 5.2, the performance dip between 8 and 9 worker threads appears to be a memory-system artifact resulting from the 9th thread running on a second socket.

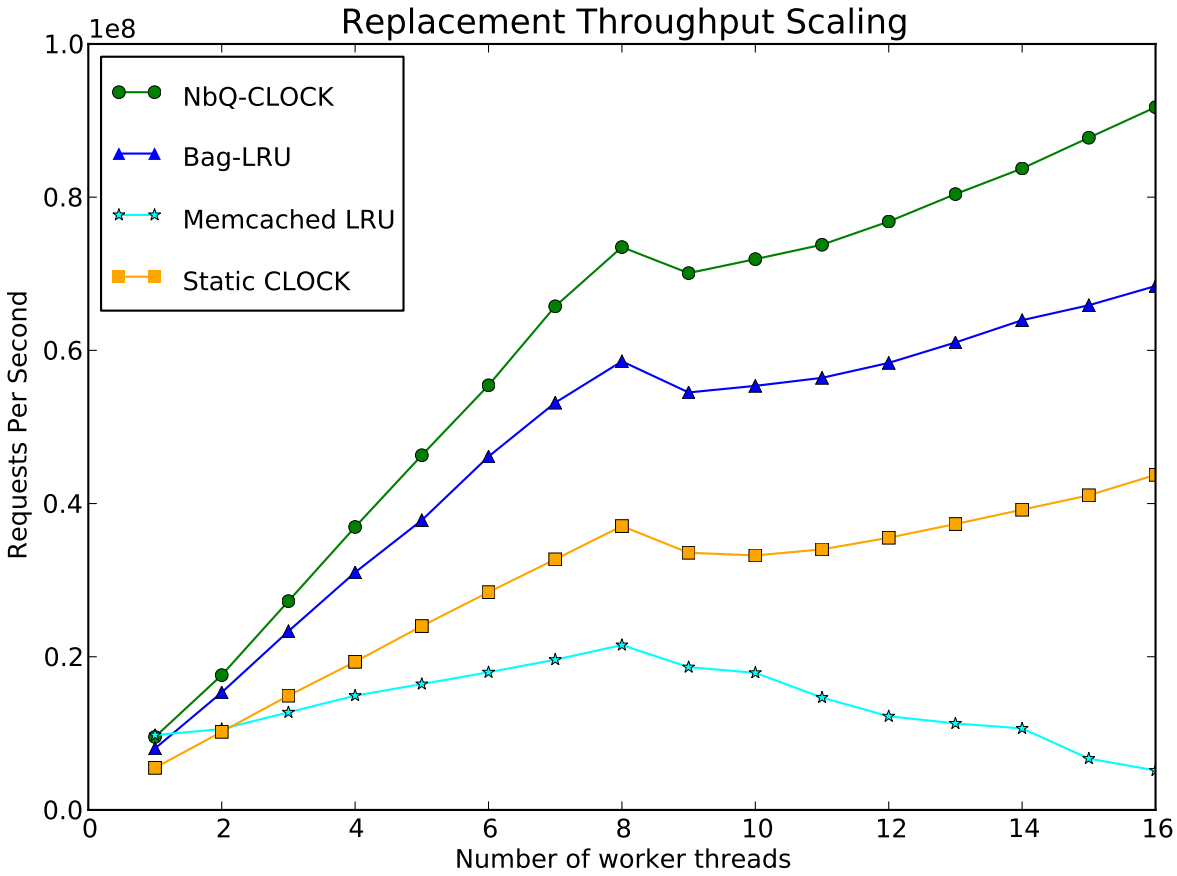

Figure 7: Throughput scaling of the replacement algorithms when the in-memory key-value store imposes no additional bottlenecks.

5.5 Throughput Scaling of an In-Memory Key-Value Store

While hit rate and update-latency scalability give important insight into the performance of a replacement algorithm, those pieces in isolation can only tell part of the story. To capture the effect of a replacement algorithm on the performance of an in-memory key-value store, one must measure the entire system's throughput. Unlike in the previous benchmark, the key-value store prototype *is not* fully optimized for scalability; there is lock contention in the hash table and memory-management operations. Thus NbQ-CLOCK's inherent advantages are hindered by the rest of the cache.

For this experiment, I set a memory limit of 1 GB for slab allocators for keys, values, item descriptors, and replacement algorithm overhead, and use an object size of 128 bytes. The key-appearance frequency is modeled by a power-law distribution of 10 million keys and the requests are 70% `GET`s and 30% `ADD`s. KVTG issues 40 million `SET` requests during the "warm up" phase to ensure the cache is well populated; in this phase no data is collected. In the subsequent phase, 20 million requests are sent to each worker thread according to the key-appearance and request-type distributions. I evaluate each replacement algorithm for up to 6 worker threads. For up to 4 worker threads the key-value store prototype uses the same thread configuration as that in the latency scaling experiment (Section 5.2). However, for 5 and 6 worker threads, I pair a KVTG's transmit thread and a key-value store's worker thread on sibling hardware threads.

The results are shown in Figure 8. As expected, there is much less differentiation in throughput between the four replacement algorithms with the key-value store prototype. For one and two worker threads, and again with five and six worker threads, there is little difference between the algorithms. As opposed to the previous experiment, the latency for a single request is a function of the replacement algorithm *and* the rest of the cache (*i.e.,* the hash table and memory management). Thus, for one and two worker threads, the performance difference among the algorithms is less significant than in Figure 7. As the number of worker threads increases and the performance difference in the replacement algorithm grows, the throughput difference becomes more pronounced. With four worker threads, NbQ-CLOCK's throughput exceeds the next best by 9.20%. For five or more worker threads the (unoptimized) cache does not scale, which is evident when comparing the performance ceiling in Figure 8 with the lack thereof in Figure 7. Because the rest of the cache is the limiting factor for five or more worker threads, I do not show results beyond six threads.

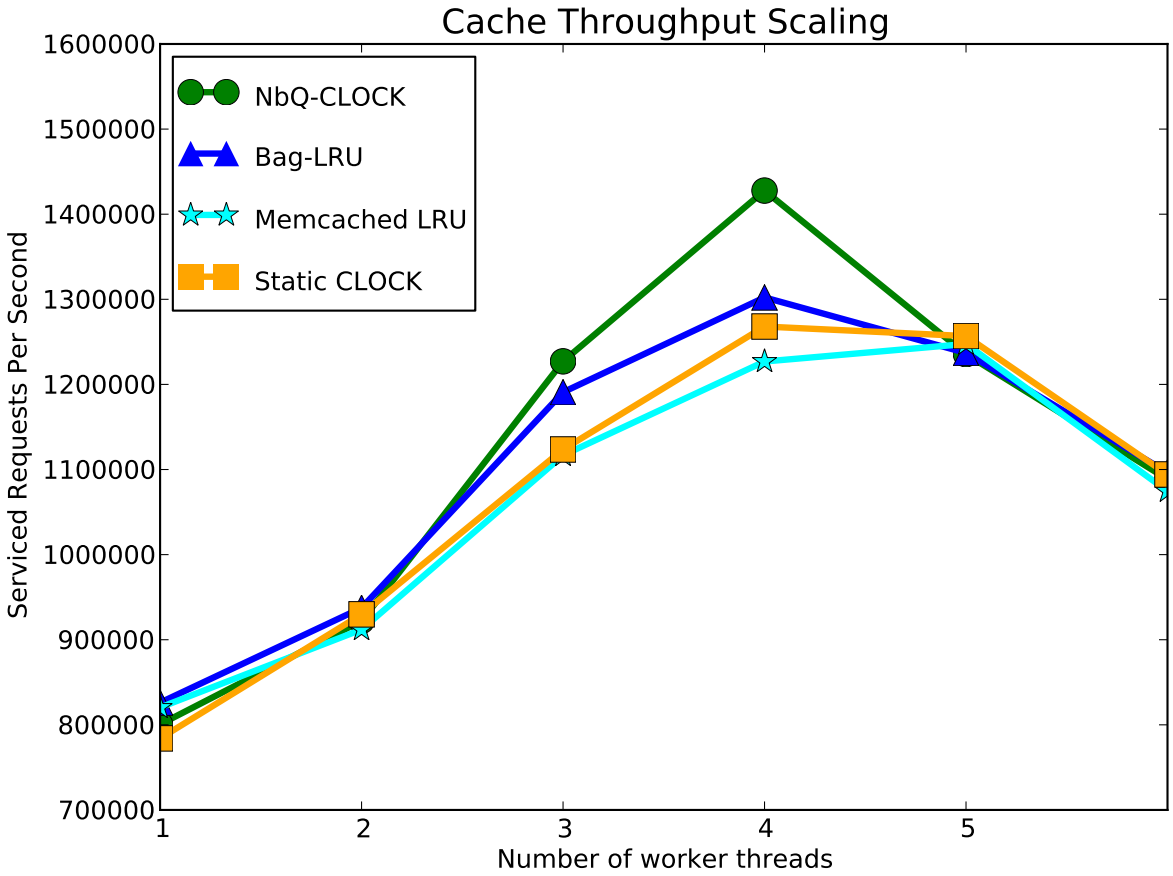

Figure 8: Throughput scaling of a realistic in-memory key-value store with different replacement algorithms.

6 Related Work

Nb-GCLOCK [19] is a non-blocking variant of the Generalized CLOCK algorithm intended for memory-page replacement in operating systems. Nb-GCLOCK assumes a fixed (typically 4KB) cache-object size, and as such is based on a statically allocated circular buffer. This assumption typically does not hold for in-memory key-value stores (*e.g.,* Memcached), where keys and values have *variable size*, which makes Nb-GCLOCK unsuitable for most web-object caching scenarios.

MemC3 [7] is a Memcached implementation that uses CLOCK for its replacement algorithm. However, MemC3 makes the (generally incorrect) assumption that the object size is fixed. Further, MemC3 does not consider the possibility of slab class re-balancing, which requires either pre-allocating a large enough CLOCK buffer for the worst case (*i.e.,* re-balancing all memory to a given slab) for every slab, or dynamic CLOCK buffer resizing. However, maintaining correctness in the presence of dynamic CLOCK buffer resizing is impossible without introducing locks, and statically pre-allocating the buffer introduces an unwieldy space overhead (discussed further in Section 4).

Bag-LRU [18], discussed in Section 3.2, is a pseudo-LRU algorithm that divides the LRU list into a list of 2 or more timestamp-ordered "bags", each of which contains a linked list of item descriptors. This replacement scheme optimizes for cache hits, which require a single non-atomic write to update the item's bag. Evictions are, however, serialized behind a bag lock for the oldest bag. Further, the algorithm has a higher per-item memory overhead than NbQ-CLOCK, resulting in less memory for cached items.

7 Conclusion

This report presents NbQ-CLOCK, a new replacement algorithm for web-object caches. This algorithm's performance exceeds state-of-the-art algorithms like Bag-LRU in terms of hit rate, items stored, and overall system throughput. While NbQ-CLOCK's hit rate was typically within 1% of the alternative algorithms' hit rates, a mere 1% improvement can have a large impact on global Internet-based services that process billions of requests per day. Furthermore, NbQ-CLOCK's

simplicity is beneficial when developing a multi-threaded key-value store.

In the future, I plan to optimize and improve the scalability and performance of the key-value store prototype. Doing so will allow us to compare replacement algorithms (as in Section 5.5) with higher thread counts, and determine the relative contribution of the replacement algorithms to the latency of a higher-performing system. The first two improvements are to replace the prototype's fine-grained lock-based hash table with a non-blocking variant and to use scalable slab allocators.

Another future direction for this work is to adapt higher-performance CLOCK variants (*e.g.,* [10, 5, 4]) to web-object caching with the ideas presented in this report. CLOCK has well-documented deficiencies that these variants overcome. For instance, they can cope with scans, self-tune to a given workload, better measure access frequency, and in general outperform CLOCK.

Finally, I also believe NbQ-CLOCK could be used in other domains, especially when the cache size and object size vary. I plan to explore these opportunities in the future.

8 Acknowledgments

Special thanks to Juan Colmenares for providing insight on replacement algorithms, non-blocking data structures, and technical writing; helping formulate the experiments; developing KVTG; and contributing to the text of this report. Juan Colmenares has been an invaluable mentor during my internship at Samsung and my graduate work at UC Berkeley. I'd also like to thank John Kubiatowicz, Krste Asanović, Daniel Waddington, Fengguang Song, Jilong Kuang, and Reza Dorrigiv for their thoughtful feedback on early drafts of this report.

This research was conducted during a summer internship at Samsung Research America - Silicon Valley.

References

[1] Memcached. `http://www.memcached.org`.

[2] ANISZCZYK, C. Caching with Twemcache. https://blog.twitter.com/2012/caching-twemcache, 2012.

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review 40*, 1 (June 2012), 53–64.

[4] BANSAL, S., AND MODHA, D. S. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), FAST'04, pp. 187–200.

[5] CARR, R. W., AND HENNESSY, J. L. WSCLOCK: A simple and effective algorithm for virtual memory management. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (1981), SOSP'81, pp. 87–95.

[6] CORBATÓ, F. J. A paging experiment with the Multics system. In *In Honor of P. M. Morse*. MIT Press, 1969, pp. 217–228.

[7] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (April 2013), NSDI'13, pp. 371–384.

[8] FITZPATRICK, B. Distributed caching with Memcached. *Linux Journal 2004*, 124 (August 2004), 5–.

[9] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. No. 325384-047US. June 2013.

[10] JIANG, S., CHEN, F., AND ZHANG, X. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference* (April 2005), USENIX'05, pp. 323–336.

[11] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable Memcached design for InfiniBand clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2012), CCGRID'12, pp. 236–243.

[12] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: predictable low latency for data center applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), SoCC'12, pp. 9:1–9:14.

[13] KIM, K., COLMENARES, J., AND RIM, K.-W. Efficient adaptations of the non-blocking buffer for event message communication between real-time threads. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing* (May 2007), ISORC'07, pp. 29–40.

[14] LIM, K., MEISNER, D., SAIDI, A. G., RANGANATHAN, P., AND WENISCH, T. F. Thin servers with smart pipes: designing SoC accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (June 2013), ISCA'13, pp. 36–47.

[15] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (May 1996), PODC'96, pp. 267–275.

[16] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (April 2013), NSDI'13, pp. 385–398.

[17] SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems 3*, 3 (September 1978), 223–247.

[18] WIGGINS, A., AND LANGSTON, J. Enhancing the scalability of Memcached. Tech. rep., Intel Corporation, May 2012.

[19] YUI, M., MIYAZAKI, J., UEMURA, S., AND YAMANA, H. Nb-GCLOCK: A non-blocking buffer management based on the Generalized CLOCK. In *Proceedings of the IEEE 26th International Conference on Data Engineering* (March 2010), ICDE 2010, pp. 745–756.