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

Gage Eads[1], **Juan A. Colmenares**[2]
[1]EECS Department, University of California, Berkeley, CA, USA
[2]Computer Science Laboratory, Samsung Research America – Silicon Valley, CA, USA
geads@eecs.berkeley.edu, juan.col@samsung.com

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 paper, we present NbQ-CLOCK, a simple and effective lock-free variant of the Generalized CLOCK algorithm that is 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. Our 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. We demonstrate that NbQ-CLOCK offers better throughput than other competing algorithms, and its fast* `update` *operation scales well with the number of threads. We also show that for our in-memory key-value store prototype, NbQ-CLOCK provides an overall throughput improvement of as much as 9.20% over the best of the other algorithms.*

Keywords: Replacement algorithm, key-value cache, CLOCK, non-blocking, scalability

1. Introduction

Minimizing the service response time experienced by users is very important for global-scale Internet-based service providers, such as Amazon, Facebook, and Samsung. One way to reduce service response times is with web-object caching, in which recently or frequently accessed remote data is cached locally to avoid slow remote requests when possible.

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. 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 to achieve low round-trip access 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.

A recent analysis [2] finds that Memcached on Linux spends 83% of its processing time in the kernel, primarily in the network layer. However, recent work such as KV-Cache [3] removes this bottleneck by using a lightweight protocol stack together with a zero copy design. Furthermore, non-blocking hash tables can eliminate contention accessing key-value pairs. Thus, we anticipate the bottleneck in high-performance key-value stores shifting to the replacement algorithm, the focus of this paper.

The replacement algorithm is responsible for deciding which item to evict when the cache is full and there are 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*.

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 access of an existing item, and `evict` selects one or more cached items for eviction.

A recent study of Facebook's Memcached traces [4] 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.

CLOCK [5] 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

G. Eads conducted this work during his internship at Samsung Research America – Silicon Valley

page is evicted. Unfortunately, CLOCK's hit rate can suffer because with a single bit reference it cannot differentiate access frequency from access recency. To solve this problem, Generalized CLOCK [6] replaces each reference bit with a 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. For instance, CAR (CLOCK with Adaptive Replacement) [7] assumes the cache has a fixed size c, and CLOCK-pro [8] depends on a fixed total memory size m, and both algorithms operate on uniformly-sized pages. WSClock [9] is based on a fixed circular list. The fixed-size assumption is *not* the case for web-object caches, and this limitation renders CLOCK and existing variants impractical in the web-object caching domain.

In this paper, we present **N**on-**b**locking **Q**ueue-based **CLOCK** (NbQ-CLOCK). It is a lock-free variant of the Generalized CLOCK replacement algorithm suited to caches containing variable number of items with different sizes. To the best of our knowledge, no prior work in literature has evaluated a similar CLOCK variant that could effectively handle such dynamically-sized caches, which are essential to web-object caching. NbQ-CLOCK has been implemented in KV-Cache [3], a high-performance in-memory key-value cache conforming to the Memcache protocol, with encouraging preliminary results.

In Section 5, we evaluate NbQ-CLOCK against other replacement algorithms applicable to Memcached, including Bag-LRU [10]. We focus our evaluation on Memcached [1] because it is widely deployed and has recently received considerable attention [10], [11], [12]. We demonstrate that NbQ-CLOCK's low update latency scales well with the thread count, and that it exhibits better throughput scaling than the other algorithms. Moreover, when used in our 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. We also show that NbQ-CLOCK's cache hit rates are at least as good as those of the other replacement algorithms.

2. An Overview of Memcached

Memcached [1], [12], [13] is a widely deployed web-object caching solution. It is typically deployed in a "side-cache" configuration, in which end users, via client devices, send requests to the front-end web servers. The front-end servers then attempt to resolve each end-user request from one or more local Memcached servers by, in turn, sending `GET` requests to them. If a cache miss occurs, the front-end server handling the end-user request forwards it to the back-end database servers 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.

An analysis of Facebook's Memcached traces over a period of several days [4] 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, we describe two existing replacement algorithms used in Memcached [1]: the algorithm shipped with its stock version, and Bag-LRU [10]. In addition, we 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.

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 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 [10]. 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. Bag-LRU keeps track of the two newest bags (needed for updates) and the oldest bag (needed for evictions).

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. 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.

[1] Brad Fitzpatrick, Memcached's original developer, built it with a "scale out, not up" philosophy [14], 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.

If the CAS fails, 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, locking that bag, and choosing the first available item 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:

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

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

4. NbQ-CLOCK

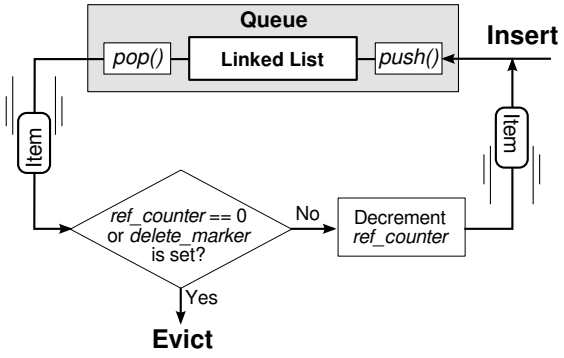

Fig. 1: Process flow of queue-based `evict`, `delete`, and `insert` operations. The data items are the moving elements, as opposed to the clock hand in the traditional CLOCK.

Non-**b**locking **Q**ueue-based **CLOCK** (NbQ-CLOCK) is a lock-free variant of the Generalized CLOCK replacement algorithm [6]. The primary difference between NbQ-CLOCK and previous CLOCK variants is that it circulates the cached items through the eviction logic (see Figure 1), 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 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). We also refer to these nodes as *item descriptors*.

NbQ-CLOCK maintains the traditional CLOCK interface with the following operations:

- `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.
- `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, we include a "delete marker" in each item descriptor. This binary flag is set atomically and indicates whether the corresponding item is deleted from the cache. During the clock sweep operation, the worker thread frees the memory of any items whose delete marker is set.
- `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 1 depicts NbQ-CLOCK's `insert`, `delete`, and `evict` operations. For more details on the design of NbQ-CLOCK, please refer to [15].

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 our non-blocking queue we use Michael and Scott's algorithm [16], but we have made two key optimizations. First, our queue performs no memory allocation in its `init`, `push`, or `pop` operations. Instead, the caller function is responsible for allocating and freeing nodes. 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.

Michael and Scott's algorithm consists of a singly-linked list of nodes, a tail pointer, and a head pointer, where the

[2]or a reference bit for a non-blocking variant of the original CLOCK [5].

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 that their queue performs a memory allocation on each `push` and a de-allocation on each `pop`. Then, 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, our 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. We avoid this problem by taking the allocators out of the queue.

We also remove the modification counters in Michael and Scott's algorithm from our queue. The modification counters protect against the ABA problem,[3] which can corrupt the queue's linked list. 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, we choose not to include modification counters in the NbQ-CLOCK queue.

5. Comparative Evaluation

In this section, we evaluate NbQ-CLOCK against the replacement algorithms Memcached LRU, Bag-LRU, and Static CLOCK, presented in Section 3. We 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 our experiments implements the Generalized CLOCK with 8-bit reference counters. The Static CLOCK, on the other hand, uses an array of packed reference bits.

5.1 Experimental Setup

The test software platform consists of two C++ applications: 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 high-performance shared-memory channels [17].

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. Each worker thread in the key-value store receives 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 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] we restrict our focus to a single replacement instance by choosing object sizes that use a single slab allocator.

Our 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, we 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.

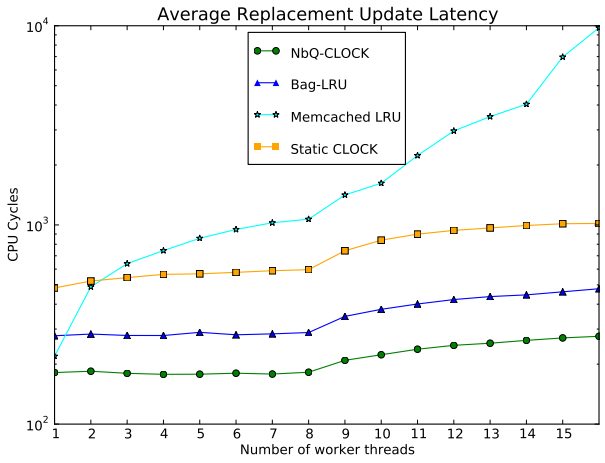

Fig. 2: Scalability of replacement algorithms' `update` operation. Each data point is the average `update` latency over 10 million `GET` requests.

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, we measure the average latency for the `update` operation of the replacement algorithms under evaluation with the worker thread count varying from 1 to 16. The results are shown in Figure 2, with the y-axis (CPU cycles) plotted on a logarithmic scale. Each data point in

[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 n A. In this case, the CAS may succeed when it should not.

[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 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 our Static-CLOCK implementation. We observe that a single atomic CAS operation takes 483 cycles on average on the test platform for a single thread (see Figure 2), 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 [10] 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.

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 comparable to and never worse than the alternative algorithms in terms of hit rate.

We 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% GETs and 30% ADDs. KVTG 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. We evaluate a range of object sizes (key plus value) from 64 B to 4 KB.

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). NbQ-CLOCK's hit rate improvement over Bag-LRU's is more significant in the context of real workloads of web-object cache systems. For instance, considering the workload characterization of live Memcached traffic reported in [4], an additional 1.40% hits of 4.897 billion requests in a day amounts to an additional 68.6 million cache hits per day.

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.

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.

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 our effort to improve this aspect in our 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

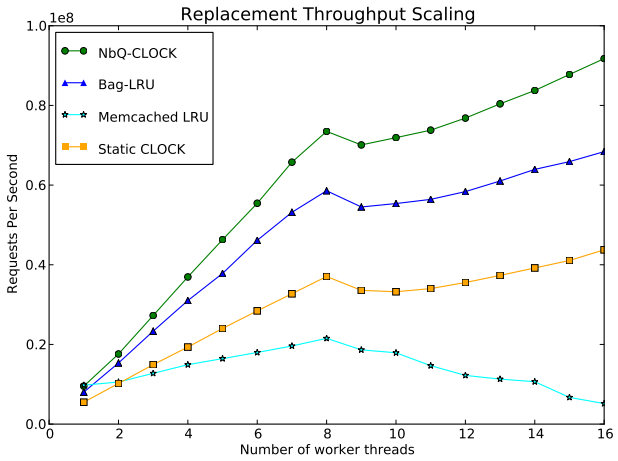

Fig. 3: Throughput scaling of the replacement algorithms when the in-memory key-value store imposes no additional bottlenecks.

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

In the following two experiments, we measure the scalability of the replacement algorithms in isolation and as part of a key-value store. In the first experiment, we 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, we replace the hash table with a pre-allocated array of keys and item descriptors and remove any dynamic memory management. By designing the benchmark in this way, we replicate the behavior of the hash table without any of its scalability bottlenecks.

For this experiment, we define throughput as the time spent in the replacement algorithm operations divided by the number of requests (10 million). We use 70% `GET` and 30% `ADD` operations. We model the key-appearance frequency with a power-law distribution for 10 million unique keys, and use an object size of 128 B. We fill the cache during an initialization phase before running the experiment.

The results are shown in Figure 3. 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 2: 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.

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, in the following 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 the second scaling experiment, we 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. We 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, we pair a KVTG's transmit thread and a key-value store's worker thread on sibling hardware threads.

The results are shown in Figure 4. 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). 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%. Because the rest of the cache is the limiting factor for five or more worker

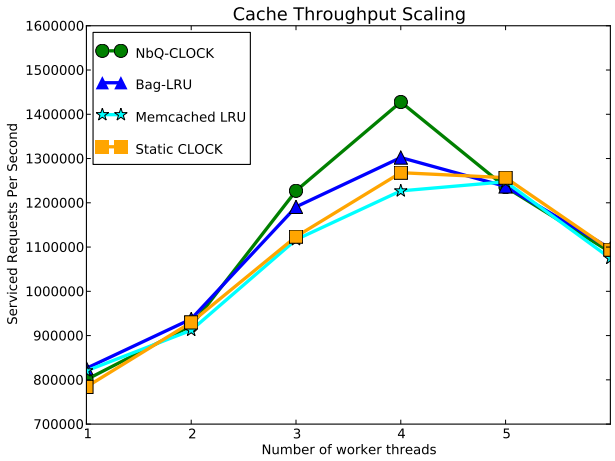

Fig. 4: Throughput scaling of a realistic in-memory key-value store with different replacement algorithms.

threads, we do not show results beyond six threads.

6. Related Work

Nb-GCLOCK [18] 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 [11] 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).

7. Conclusion

This paper presents NbQ-CLOCK, a replacement algorithm for web-object caches. Its performance exceeds state-of-the-art algorithms like Bag-LRU in terms of overall system throughput and number of items stored, while offering hit rates at least as good as those of the alternative algorithms. Furthermore, NbQ-CLOCK's simplicity is beneficial when developing a multi-threaded key-value store, as it requires less debugging and testing effort than more complicated alternatives.

NbQ-CLOCK has been implemented in KV-Cache [3], a high-performance in-memory key-value cache conforming to the Memcache protocol. Our preliminary experimental results show that, when servicing traffic consisting of 70% `GET`s and 30% `SET`s, KV-Cache experiences negligible degradation in total system throughput, whereas the performance of Intel's Bag-LRU Memcached [10] is severely affected by the presence of `SET` requests. Besides storing and replacing data items, `SET` requests trigger KV-Cache's eviction logic once the amount of memory used to store key-value pairs exceeds a predetermined threshold. Thus, these results are an encouraging indication of NbQ-CLOCK's effectiveness as part of this full system.

One future direction for this work is to adapt higher-performance CLOCK variants (*e.g.,* [7], [8], [9]) to web-object caching with the ideas presented in this paper. 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. Additionally, we also believe that NbQ-CLOCK could be used in other domains, especially when the cache size and object size vary. We plan to explore these opportunities in the future.

References

[1] "Memcached," http://www.memcached.org.

[2] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the 9th European Conference on Computer Systems (Eurosys'14)*, April 2014.

[3] D. Waddington, J. Colmenares, J. Kuang, and F. Song, "KV-Cache: A scalable high-performance web-object cache for manycore," in *Proceedings of the 6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'13)*, December 2013, pp. 123–130.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 53–64, June 2012.

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

[6] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems*, vol. 3, no. 3, pp. 223–247, September 1978.

[7] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, March 2004, pp. 187–200.

[8] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*, April 2005, pp. 323–336.

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

[10] A. Wiggins and J. Langston, "Enhancing the scalability of Memcached," Intel Corporation, Tech. Rep., May 2012. [Online]. Available: http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached

[11] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, April 2013, pp. 371–384.

[12] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, April 2013, pp. 385–398.

[13] C. Aniszczyk, "Caching with Twemcache," https://blog.twitter.com/2012/caching-twemcache, 2012.

[14] B. Fitzpatrick, "Distributed caching with Memcached," *Linux Journal*, vol. 2004, no. 124, pp. 5–, August 2004.

[15] G. Eads, "NbQ-CLOCK: A non-blocking queue-based CLOCK algorithm for web-object caching," Master's thesis, EECS Department, University of California, Berkeley, October 2013. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-174.html

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

[17] K. Kim, J. Colmenares, and K.-W. Rim, "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 (ISORC'07)*, May 2007, pp. 29–40.

[18] M. Yui, J. Miyazaki, S. Uemura, and H. Yamana, "Nb-GCLOCK: A non-blocking buffer management based on the Generalized CLOCK," in *Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE'10)*, March 2010, pp. 745–756.