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

Gage Eads¹, Juan A. Colmenares²
¹EECS Department, University of California, Berkeley, CA, USA
²Computer Science Laboratory, Samsung Research America – Silicon Valley, CA, USA
ageads@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
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. 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.

1Brad 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-blocking Queue-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,
- 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
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 KV TG. 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.

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 \( A \). 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 distri-
bution. 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 aver-
age 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.

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.

<table>
<thead>
<tr>
<th>Object Size (bytes)</th>
<th>NbQ-CLOCK</th>
<th>Bag-LRU</th>
<th>Static CLOCK</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of items stored (millions)</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Hit rate</td>
<td>82.81%</td>
<td>80.86%</td>
<td>78.23%</td>
<td>75.26%</td>
</tr>
<tr>
<td>Number of items stored (millions)</td>
<td>0.253</td>
<td>0.253</td>
<td>0.253</td>
<td>0.253</td>
</tr>
</tbody>
</table>

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
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 bytes. The key-appearance frequency is modeled by a power-law distribution of 10 million keys and the requests are 70% GETs and 30% ADDs. 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
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 Memcached protocol. Our preliminary experimental results show that, when servicing traffic consisting of 70% GETs and 30% SETs, 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


