




FL^{ec}C: a Fast Lock-Free Application Cache

André J. Costa¹ , Nuno M. Pregoça¹ , and João M. Lourenço¹ 

NOVA University Lisbon — FCT & NOVA LINCS, Portugal
aj.costa@campus.fct.unl.pt nuno.pregoica@fct.unl.pt
joao.lourenco@fct.unl.pt

Introduction. When compared to blocking concurrency, non-blocking concurrency can provide higher performance in parallel shared-memory contexts, especially in high contention scenarios. This paper proposes FL^{ec}C, an application-level cache system based on Memcached [**memcached**], which leverages redesigned data structures and non-blocking (or lock-free) concurrency to improve performance by allowing any number of concurrent writes and reads to its main data structures, even in high-contention scenarios. We discuss and evaluate its new algorithms, which allow a lock-free eviction policy and lock-free fast lookups. FL^{ec}C can be used as a plug-in replacement for the original Memcached, and its new algorithms and concurrency control strategies result in considerable performance improvements (up to 6×).

FL^{ec}C Design. Memcached [**memcached**] has three main separate data structures: a hash table with singly linked-list buckets, for fast lookups; a doubly linked-list to implement a Least Recently Used (LRU) eviction policy; and a slab allocator for fast and efficient memory allocation. These separate data structures are incompatible with non-blocking concurrency control as they allow unwanted interleavings to occur. For example, if we simply substituted Memcached’s blocking hash table and LRU by their non-blocking counterparts, unwanted interleavings would make it possible for cached items to be in present in one of the structures and absent in the other. Maintaining correctness in such setting would require additional work, which in turn would negatively impact performance. FL^{ec}C does not have a separate data structure to implement an eviction policy, but has it embedded into its hash table instead. To this end, we took inspiration from CLOCK, a known Operating Systems page replacement algorithm. CLOCK [**corbator:multics-paging**] associates a bit (which we refer to as the CLOCK value) to each cache entry: the value 0 represents a not recently used item, while the value 1 represents a recently used item. Cache eviction using a very fine-grained approach, based in associating a CLOCK value to each item in FL^{ec}C, would require the eviction procedure to traverse all hash table buckets. As each hash table bucket is implemented as a linked-list, with items that are not stored continuously in memory, each step in the list traversal would likely need to fetch data from main memory. This means that traversals would not be cache effective and, thus, exhibit lower performance. In FL^{ec}C we opted for a medium-grained approach, associating instead a CLOCK value to each hash table bucket. This means that the eviction is cache friendly, as it traverses continuous blocks of memory. Since the hash table expansion occurs when the number of

cache items is $1.5 \times$ the number of hash table buckets, we know that each CLOCK value represents at most 1.5 items. Furthermore, FL^{ec}C's CLOCK values are not limited to just one bit, so that its eviction policy can distinguish between mildly and highly popular items. Through a hash table with embedded CLOCK eviction, we can now substitute FL^{ec}C's hash table blocking linked-list buckets by non-blocking linked-lists. To do so, we make use of Harris' pragmatic non-blocking linked-list algorithm [harris2001pragmatic]. To be able to reclaim memory while utilizing a non-blocking concurrency control strategy, we need to employ a memory reclamation scheme. FL^{ec}C's memory reclamation scheme is based on DEBRA [debra], due to its flexibility and performance. DEBRA is a general memory reclamation scheme that does not assume the overlying algorithm knows when it is out of memory, meaning the memory reclamation scheme progresses even when no memory needs to be reclaimed. Since FL^{ec}C is a caching system, it *must* know when it is out of memory. For this reason, FL^{ec}C deviates from DEBRA by only progressing the memory reclamation scheme when it is absolutely necessary, minimizing the amount of work required to reclaim memory. Finally, having a blocking hash table expansion algorithm partially nullifies some of the progress condition advantages that come with having non-blocking concurrency control. Thus, FL^{ec}C implements a non-blocking hash table expansion algorithm, opposed to Memcached's stop-the-world like expansion.

Evaluation. Our evaluation aims to clarify some characteristics of FL^{ec}C and its intermediate step, where Memcached eviction policy was substituted by a hash table with embedded CLOCK eviction (MemCLOCK):

- What is the impact on both performance and hit-ratio of having an approximated LRU eviction policy (MemCLOCK) rather than a strict one?
- How does each system perform under varying degrees of contention?

Contention was mediated by a few characteristics of the system, its environment and the workload it is being subjected to. In particular, the system's contention degree was dependent on the item size, item access frequency, and network bandwidth.

Figure 1 depicts the throughput of Memcached, MemCLOCK, and FL^{ec}C, under read-intensive (99% reads) workloads and varying degrees of skewness (the higher the α value, the more skewed the access distribution is). These tests were performed with small items, so network communication is not a performance bottleneck.

From Figure 1a, it becomes evident that FL^{ec}C consistently provides higher throughput under every workload. Figure 1b clarifies the relative performance between the three systems, emphasizing the impact of FL^{ec}C's new design and concurrency control strategy.

Our experimental results show that the CLOCK-based eviction policy used in both MemCLOCK and FL^{ec}C does not significantly impact the hit-ratio. In terms of performance and latency, MemCLOCK exhibits throughput and latency similar to the original Memcached. FL^{ec}C, on the other hand, can obtain up to $6 \times$ improvement in throughput and up to $1/6$ of the latency, w.r.t. Memcached and

when under very high contention scenarios; $\approx 1.2\times$ improvement under medium contention scenarios; and equivalent (to Memcached) performance under low contention scenarios.

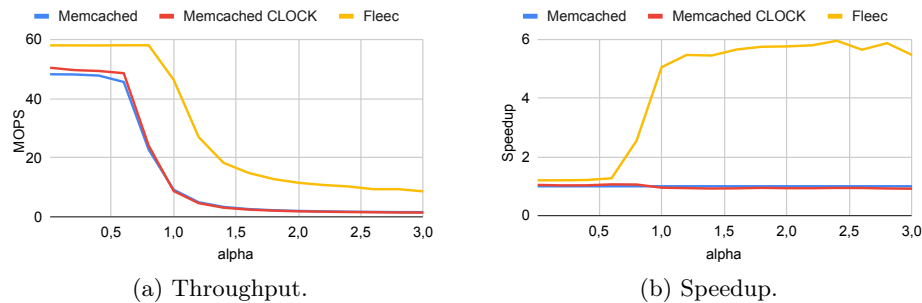


Fig. 1: Throughput and Speedup varying zipfian α , for a read-intensive (99% reads) workload.

Conclusion. FL^{eC} performs significantly better than Memcached under high- and mid-contention scenarios, while presenting no performance degradation under low contention scenarios. Thus, FL^{eC} can easily be used as a plug-in replacement of Memcached, with no identified disadvantages.

Acknowledgments. This work was supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT.IP.