Is SIEVE better than LRU?

William Hu
Stanford University
Stanford, CA
willhu@stanford.edu

Thomas Li
Stanford University
Stanford, CA
bitset@stanford.edu

Balaji Balachandran
Stanford University
Stanford, CA
balajib@stanford.edu

***Abstract*—Cache eviction algorithms dictate how efficiently a system can use small amounts of fast access memory when the working set size exceeds the size of this memory. Algorithms such as Clock and Cacheus achieve hit rates better than or comparable to LRU but at the cost of greatly increased complexity, preventing adoption in real world systems. Zhang et al. introduce SIEVE, a cache eviction algorithm that is not only simpler than LRU, but also outperforms it and other heavily engineered and tuned algorithms. The authors evaluate SIEVE based on its achieved miss rate reduction compared to FIFO against other caching algorithms such as LRU, Clock, ARC, and TwoQ. Further, SIEVE is also evaluated based on its ability to improve multi-threaded performance through lazy promotion. Motivated by these compelling results, we replicate the SIEVE algorithm and evaluate it based on 1) miss rate reduction compared to FIFO and 2) throughput in multi-threaded settings. We contribute a new cache benchmark for optimizing cache algorithms, and provide an extension to SIEVE that achieves significantly higher throughput, BitSIEVE.**

I. Introduction

Cache eviction algorithms are crucial to efficient cache use. By evicting data that is not likely to be accessed in the future, these algorithms ensure that there is space to store data that are more likely to be accessed, reducing the cost of data movement. Since the introduction of LRU, an algorithm by which the least recently used data is evicted from the cache, numerous other cache eviction algorithms have been proposed. These algorithms can theoretically achieve better miss rates than LRU, but often come at the cost of increased implementation complexity. For example, TwoQ uses two LRU queues to be scan resistant and ARC considers both recency and frequency using an adaptive parameter. Increased complexity prevents integration into real world systems and can often achieve lower overall throughput in production.

Zhang et al. introduce SIEVE, a novel cache eviction algorithm that they claim is both simpler than LRU and achieves state-of-the-art efficiency and scalability [1] through lazy promotion and quick demotion [2]. These two principles contrast the promotion and demotion logic in commonly used algorithms such as LRU, where data that is *hit* is immediately moved to the head of the queue and the eviction target has to traverse the entire queue first. In SIEVE, data that is *hit* remains in place and the eviction target can be in any position of the queue. As a result, it becomes more difficult for stale data to remain resident in the queue and successful hits do not have to incur the cost of taking a write lock on the head of the queue. However, the authors note that SIEVE is not scan-resistant, performing worse than LRU on certain block cache workloads due to the absence of a ghost queue.

Based on these characteristics, we evaluated SIEVE to determine if it is indeed better than LRU. We find that SIEVE achieves comparable performance on the traces used in the paper in terms of 1) miss rate reduction from FIFO and 2) throughput in multi-threaded settings. Further, we implement GhostSIEVE, a scan-resistant variant of SIEVE using a ghost queue to keep track of recently evicted data, and show that this variant outperforms vanilla SIEVE on a block cache workload. We also propose BitSIEVE, a bit parallel extension of SIEVE that is more performant for eviction. To summarize our contributions, we:

1) Show that SIEVE achieves miss rate reductions comparable to the results in the paper and propose GhostSIEVE to improve SIEVE on block cache workloads.
2) Show that SIEVE achieves better throughput compared to FIFO in multi-threaded settings.
3) Propose BitSIEVE, an optimized variant of SIEVE using bit sets, and show that it improves throughput against an optimized normal implementation.

II. Related Works

Next, we describe common cache eviction algorithms.

A. FIFO

FIFO, or first-in-first-out, evicts the piece of data that was first placed in the queue. Demotion and promotion are no-ops.

B. LRU

LRU evicts the piece of data that was least recently used in the queue. On hits, LRU moves the piece of data to the head of the queue, displaying quick promotion. When we need to make space for new data, LRU evicts the piece of data at the tail of the queue.

C. Clock

Clock approximates LRU by assigning each item a bit of metadata. Upon insertion or a hit, this bit is set to True. A hand traverses the queue to select eviction targets. If a piece of data has its bit set to False, it is evicted and the new piece of data is inserted in its place.

D. TwoQ

TwoQ separates cache entries into two separate queues to distinguish fresh accesses from popular accesses. Newly inserted entries are first placed into a short FIFO-style queue that captures data accessed only once. If an entry is accessed again while in this queue, it is promoted into a larger main queue managed in an LRU-like manner [3]. When eviction is required, entries are first removed from the short queue, allowing the cache to filter out one-time accesses while preserving popular entries.

E. ARC

ARC maintains an adaptive parameter p that balances frequency and recency as well as four different queues: $T1$, $T2$, $B1$, and $B2$. $T1$ and $B1$ are queues that store data that was accessed recently. $B1$ is a ghost queue that only stores keys. $T2$ and $B2$ are queues that store data that was accessed frequently (i.e. if the data is already in $T1$, $B1$, or $B2$, it is inserted into $T2$) and $B2$ is a ghost queue that only stores keys. p is adjusted based on whether data was found in $B1$ or $B2$ and it dictates which queue - $T1$ or $T2$ - to evict data from [4].

III. SIEVE

By comparison to other cache eviction algorithms, SIEVE is similar to CLOCK. A $visited$ bit is maintained per data that is initialized to False upon insertion. When the data is hit, the $visited$ bit is set to True. A hand is maintained that traverses the queue and selects eviction targets. If the $visited$ bit is set to True, it sets it to False and moves on to the next item. If the $visited$ bit is set to False, that item is evicted from the queue. New items, however, are inserted at the head of the queue instead.

Zhang et al. evaluate SIEVE based on its achieved miss rate reduction compared to FIFO and its achieved throughput as a the number of threads increases. They benchmark against a number of existing cache eviction algorithms including the ones previously mentioned. We design two experiments to replicate and verify their claims on these two metrics. Further, we propose slight modifications to improve the SIEVE algorithm for block cache workloads and an optimized implementation using bit sets.

A. Miss Rate Reduction

To compute the miss rate of each cache algorithm, we developed a simple trace parser and cache simulator in Python. The parser relies on the *oracleGeneral* format and the simulator uses the parser to stream requests to a cache eviction algorithm. Each cache algorithm exposes the same interface to the simulator: a get method to request an object ID and a $miss\_ratio$ method to query the achieved miss ratio. A cache algorithm instantiates the get abstract method and updates the $\_num\_requests$ and $\_num\_misses$ fields accordingly. We implement FIFO, LRU, Clock, TwoQ, ARC, and SIEVE, and evalute them against the Twitter KV traces used in the paper [1]. To ensure that execution time is reasonable, we rely on samples of each trace that take 1% of traffic. We successfully replicate the miss rate reductions of a subset of the evaluated cache eviction algorithms and SIEVE in Figure 1, showing that SIEVE indeed achieves state-of-the-art efficiency.

1) GhostSIEVE: The GhostSIEVE algorithm 1 builds upon the SIEVE primitive [1] by adding a ghost queue that makes it more scan resistant. We evaluate SIEVE and GhostSIEVE on block cache traces from Meta's Tectonic Cache Traces and find that GhostSIEVE has slightly better miss rate reduction compared to vanilla SIEVE, achieving -0.0065 compared to -0.0131 at a small cache size and -0.0062 compared to -0.0063 at the large cache size in Figure 2. However, both SIEVE and GhostSIEVE still perform worse than FIFO for these block cache workloads.

Algorithm 1 GhostSIEVE

Require: The request x, doubly-linked queue T, cache size C, hand p, ghost queue G

if x is in T **then** ▷ Cache Hit
  x.*visited* $\leftarrow 1$
else ▷ Cache Miss
  if $|T| = C$ **then** ▷ Cache Full
    $o \leftarrow p$
    if o is **null then**
      $o \leftarrow$ tail of T
    end if
    while o.*visited* $= 1$ **do**
      o.*visited* $\leftarrow 0$
      $o \leftarrow o$.*prev*
      if o is **null then**
        $o \leftarrow$ tail of T
      end if
    end while
    $p \leftarrow o$.*prev*
    Discard o in T ▷ Eviction
    Insert o in the head of G ▷ Add in Ghost Queue
    if $|G| = C$ **then** ▷ Ghost Queue Full
      Discard the tail of G
    end if
  end if
  is_a_ghost $\leftarrow x$ in G
  if is_a_ghost **then**
    Discard x in G
  end if
  Insert x in the head of T
  x.*visited* $\leftarrow$ is_a_ghost
end if

B. Throughput

To measure the throughput of SIEVE, we built a C++20 cache throughput benchmark from scratch, to have complete control of the design and methodology. The benchmark is designed to make it easy to both iterate on and verify the correctness of cache implementations. We used this to quickly develop BitSIEVE.

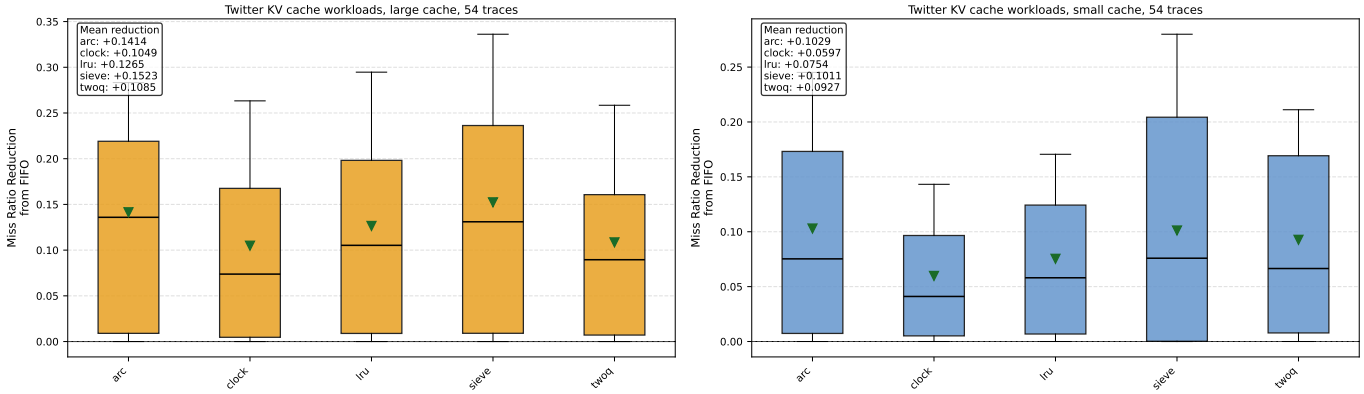


Fig. 1: **Miss rate reduction between ARC, Clock, LRU, SIEVE, TwoQ.** We compare SIEVE and other classical cache eviction algorithms using the same Twitter KV traces from the SIEVE paper [1]. We sample 1% of the traffic to keep execution time reasonable.

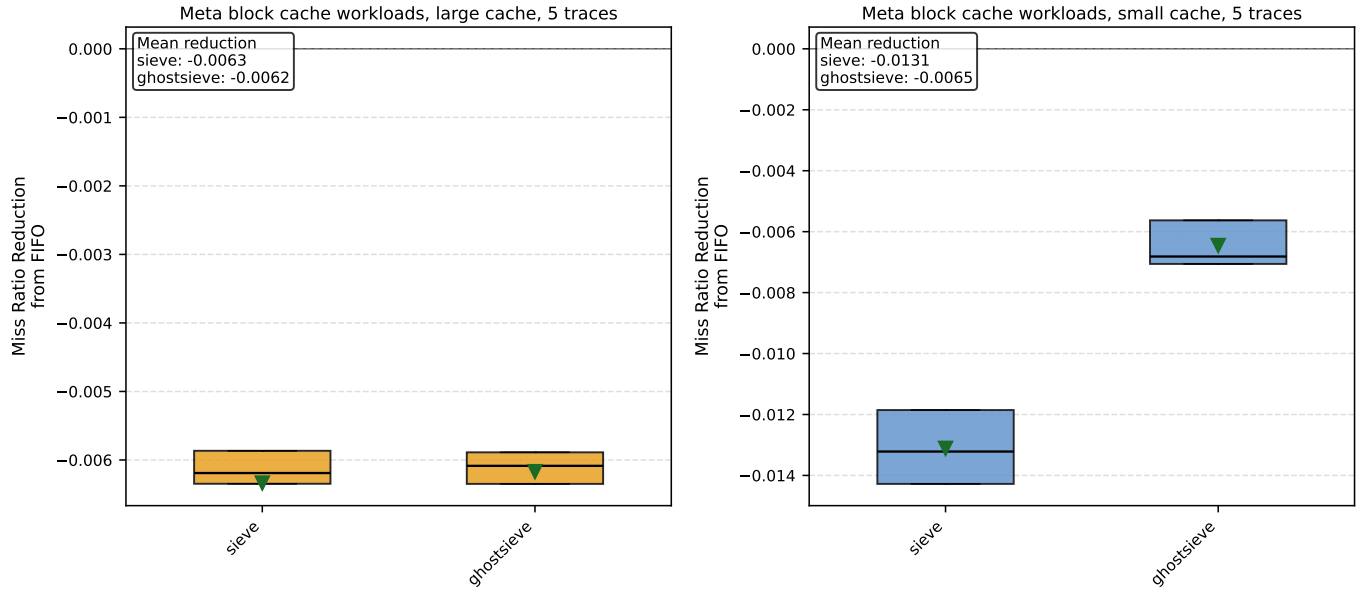


Fig. 2: **Miss rate reduction between SIEVE and GhostSIEVE.** We compare SIEVE and GhostSIEVE using Meta's Tectonic Cache Traces, traces captured over 5 days from one of Meta's block storage clusters.

1) Design: The first thing to consider is the input format. Our input format is simply the list of requests to the cache, in the form of 64-bit keys. We don't consider object sizes, since no algorithm we evaluate makes decisions based on object sizes[1]. Therefore, the values we consider are also 64-bit integers, and we refer to them as "tokens" to make it clear that we aren't considering real objects. To convert arbitrary keys to integers, we apply a polynomial string hash; we have verified that there isn't any collisions on our processed traces.

We preprocess existing traces into our format, which makes running the benchmark far faster, and the input sizes much smaller; we provide the code for preprocessing Twitter Twemcache, and both Meta KV traces.

[1]Although developing such an algorithm could be interesting.

We provide a simple interface for each cache implementation, consisting of a single function - Query$(k, miss)$, where $miss$ is a function from 64-bit keys to 64-bit tokens. The implementation is responsible for returning $miss(k)$, either by caching the value, or by calling $miss$. Clearly the latter case counts as a cache miss. With this interface, we can trivially verify the correctness of the cache by comparing the query value to the expected one. And this implementation clearly allows the most generality of implementation: at the cost of being unable to have finer grained measurements of cache subprocesses. But this is not measured in the original paper, and we also believe that we don't gain much insight at looking at internals; ultimately, the overall performance is what matters.

Our benchmark supports several configuration options:

namely a set of threads choices, a set of cache size proportions (relative to the working set size), and a set of scaling policies.

An interesting choice is how to scale the input to run on multiple threads (which share a cache). We provide several policies:

1) INTERLEAVE: the threads process the queries in an interleaved fashion; each query is processed once by some thread.
2) TRANSFORM_SPACE: each thread will process all queries, but we'll transform the key into a new space by applying a hash function on a combination of the key, and the thread's id.
3) REPLICATE: each thread processes all queries, and we don't transform the keys.

We produce several direct statistics: 1) the wall time to process the input 2) the number of queries 3) the number of hits 4) a set of latency samples. From this, it's straightforward to postprocess statistics such as latency percentiles, miss rates, throughput, etc. To measure throughput, we used `std::chrono::high_resolution_clock`, and we sample every 1024-th query for latency using rdtsc. This achieves about 10ns granularity, which is sufficient to see latency differences between our implementations.

We support several cache implementations: LRU, FIFO, and several SIEVE variants. We also provide a simple interface to add sharding to any cache implementation.

2) Design Comparison: Our design differs significantly with the original paper in our choice of input scaling. The original paper essentially uses our TRANSFORM_SPACE, arguing that "to better emulate real-world deployment", we should scale up the working set and cache sizes with the number of threads. We partially agree with this design: there clearly are workloads, like cloud services, where users can have highly distinct query sets, and hence the working set size could scale with the number of users. But, in workloads like serving web content, we argue that scaling up users wouldn't necessarily increase the working set size, since new users would likely have similar usage patterns to existing ones.

Thus, we added the REPLICATE and INTERLEAVE policies, to better capture those workloads.

3) Scaling Replication: We run our evaluations on bare metal, on the *AMD EPYC 9R45* processor running at a fixed 2.6 GHz.

Since our benchmark supports TRANSFORM_SPACE, it's easy to replicate the paper's throughput graph (Figure 6); we achieve a similar scaling to the paper, and demonstrate the lack of scalability for LRU. We note that we were only able to replicate this type of scaling on workloads with very high hit rate. On workloads with lower hit rate, SIEVE scales poorly, as a lock must be used to sweep the hand. Comparatively, FIFO can be implemented easily with a lock-free queue, and can achieve much better scaling on lower hit rate workloads. SIEVE also scales worse on our other scale policies. Thus, we believe that the original paper's claims on scalability should come with some caveats: it's true that SIEVE is more scalable than LRU, but its scalability depends on having a high hit rate. And other algorithms like FIFO also don't require locking on hits, and thus are similarly scalable.

C. BitSIEVE

We present a bit parallel extension of SIEVE. We will provide a data structure that can approximately provide the same structure as SIEVE's list, but can do eviction operations much faster.

Let $W = 64$ be the word size of the system, and let $K > 0$ be an integer parameter. We'll store a linked list of "blocks". In each block, we'll store two bitsets of length WK: one for the visited bit, and another denoting whether the given index is occupied. We'll also store an integer array of length WK for keys.

Now let's consider how to insert into the structure. We'll maintain a pointer head_p, which will point to the first block in our list. We'll also have an integer head_i which stores the number of elements we added into the first block. If $\text{head}_i = WK$, we'll create a new block, and link it before head_p, and reset head_i to be zero. Then we will add our element into the current first block, at index head_i, by setting the occupied bit to be one, and writing the key.

Now let's consider eviction. At construction, we'll create the first block, and we'll initialize a pointer hand_p that points to it, and an index into the block hand_i. When we evict, we'll consider a block at a time, starting from index hand_i in block hand_p. In that block, we need to do two things: 1) find the smallest occupied index j where $j \geq \text{hand}_p$ and $\text{vis}_j = 0$, and if it doesn't exist, return WK. And 2) for all occupied k where $\text{hand}_p \leq k < j$, set its visited bit to be 0. This is just the SIEVE algorithm's procedure, and can be done in $O(K)$ word operations (Listing 1). We'll iterate the procedure until we find a valid index to evict. Then, the question is: how to remove it from the middle of the list? Imagine, if all we did was set the occupied bit to be zero. This would maintain SIEVE's behavior correctly, as we would effectively delete that index (our insertions never reuse indices). But, our memory usage would explode, as we would need to keep adding new blocks while old blocks become sparser. And simply deleting blocks when they become empty isn't sufficient, as we can still degrade to very sparse, yet being visited, blocks. This would be extremely bad, as the memory usage would become proportional to $\text{capacity} \cdot WK$, which is at least 64 times the capacity!

But, if we are willing to change a bit of SIEVE's properties, we can solve this issue. The idea is that we'll bound the sparsity of each block: when a block gets too sparse, we'll delete it, and reinsert its elements to the head. It turns out that reinserting when the occupancy drops below 50% is sufficient, and performs well. And since each block is at least 50% full, we'll use at most 2x memory. Based on our evaluations, even though we are changing the order of the elements, this doesn't degrade the miss rate performance by much, while still offering a significant speedup over a normal SIEVE implementation.

1) BitSIEVE Evaluation: Like the original SIEVE, to preserve the algorithm's properties, we can only have a single hand, and thus necessarily need a lock for eviction. So it doesn't scale very well on miss heavy workloads. Therefore, we evaluated on a single thread, versus a self-written optimized SIEVE implementation and a fast ring-buffer based FIFO implementation (Figure 3). We use $K = 8$, and we can see very impressive results: BitSIEVE has comparable performance to FIFO, and is much faster than the normal SIEVE - at a 15% (of working set) cache capacity, we achieve a 79% speedup, and our lowest speedup (at 50% capacity) is still 44% (Figure 5a). BitSIEVE also only has less than 1% lower hit rate on the trace, and is still significantly higher than FIFO. And despite BitSIEVE's reinsertion behaviour, its tail latency is low, showing comparable or lower latency than FIFO and SIEVE at all percentiles (Figure 4).

IV. Conclusion

Cache eviction algorithms are crucial to the performance of a cache. While these algorithms have generally evolved over time to include more complexity based on the characteristics of their workloads, Zhang et al. introduces a simple modification of classical cache eviction algorithms in SIEVE [1]. The authors claim that it achieves the state-of-the-art in terms of hit rate and throughput. Further, its simplicity enables its use as a powerful caching primitive to create more complex algorithms. We validate that their claims generally hold true on the traces that they have use in their evaluations, measuring SIEVE in terms of its achieved miss rate reduction compared to FIFO and its throughput in multi-threaded settings. We also extend SIEVE to create GhostSIEVE, a scan-resistant variant of SIEVE that performs better on block cache workloads. Finally, we propose a bit parallel extension of SIEVE that is strictly more performant in the eviction case. We find that SIEVE is indeed a powerful caching primitive comparable to LRU.

Acknowledgment

This research was carried out as part of a course project for *CS244c: Advanced Networking and Distributed Systems* at Stanford University. We would like to acknowledge the support of our professors, Keith Winstein and David Mazières, for their continual support and guidance as we explored the efficacy of cache eviction algorithms. Their input on experimental methodology and evaluation techniques helped shape the direction of this work.

Appendix

```
struct Block {
  T vis[K], in[K];
  int nxt_and_clr(int st) {
    for (int b = st >> 6; b < K; b++) {
      if (b == st >> 6) {
        int shift = st & 63;
        T any = (~vis[b] & in[b]) >> shift;
        if (any) {
          int out = std::countr_zero(any) + st;
          T skip = (any & -any) - 1;
          vis[b] &= ~(skip << shift);
          return out;
        } else {
          vis[b] &= (T(1) << shift) - 1;
        }
      } else {
        T any = ~vis[b] & in[b];
        if (any) {
          vis[b] &= ~((any & -any) - 1);
          return std::countr_zero(any) + b * W;
        } else {
          vis[b] = 0;
        }
      }
    }
    return BLOCK_SZ;
  }
};
```

Listing 1: The source code to implement the "next and clear" operation

References

[1] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, "SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1229–1246. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

[2] J. Yang, Z. Qiu, Y. Zhang, Y. Yue, and K. V. Rashmi, "Fifo can be better than lru: the power of lazy promotion and quick demotion," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, ser. HotOS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 70–79. [Online]. Available: https://doi.org/10.1145/3593856.3595887

[3] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, p. 439–450.

[4] N. Megiddo and D. S. Modha, "Arc: a self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, ser. FAST'03. USA: USENIX Association, 2003, p. 9.

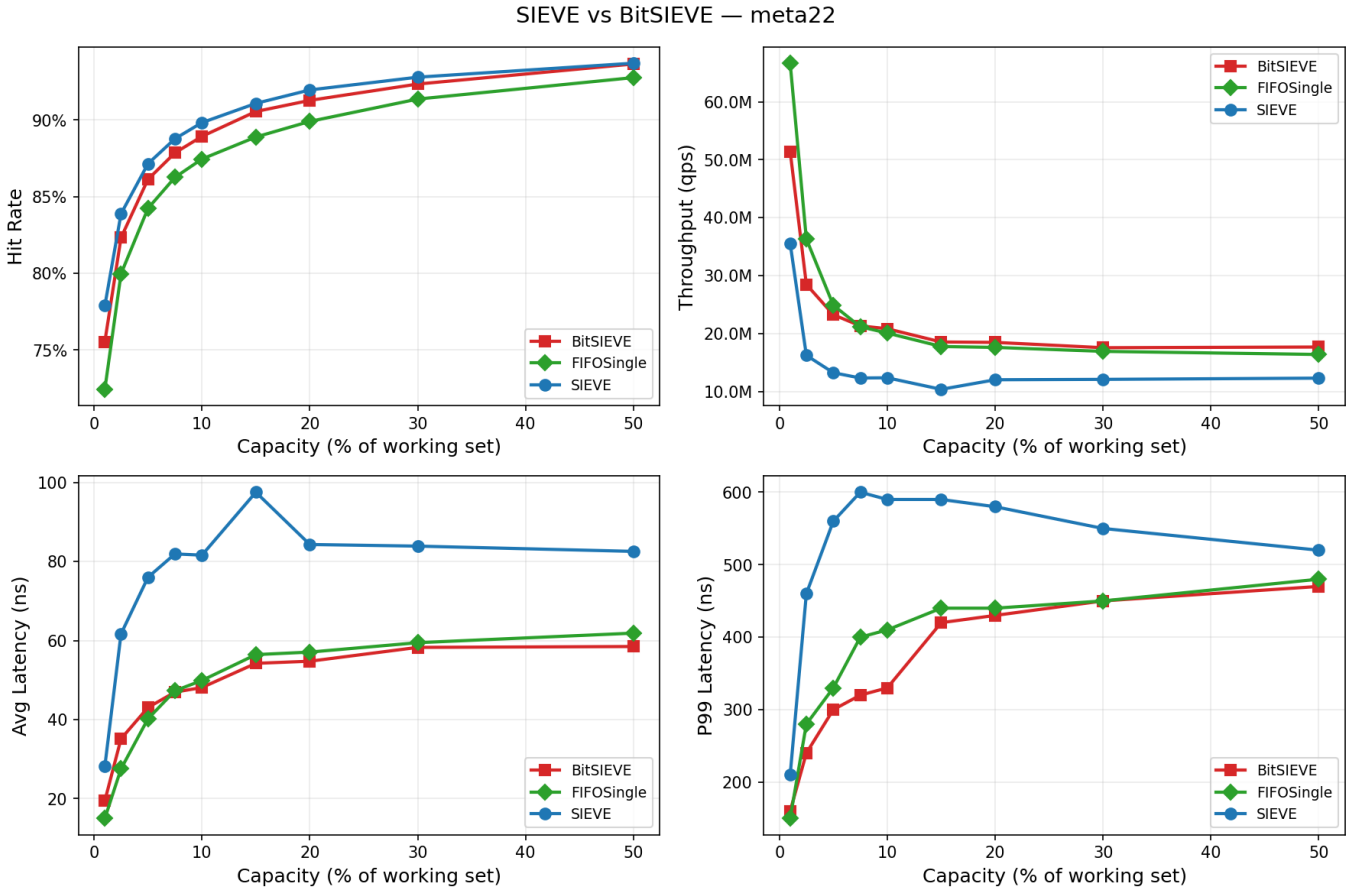


Fig. 3: **BitSIEVE Evaluation versus SIEVE and FIFO on the Meta KV 2022 Trace**

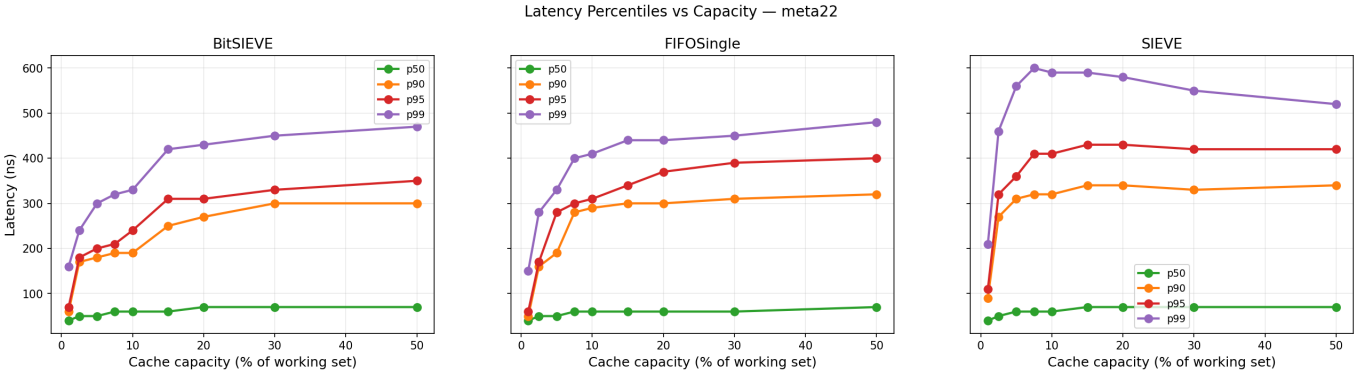
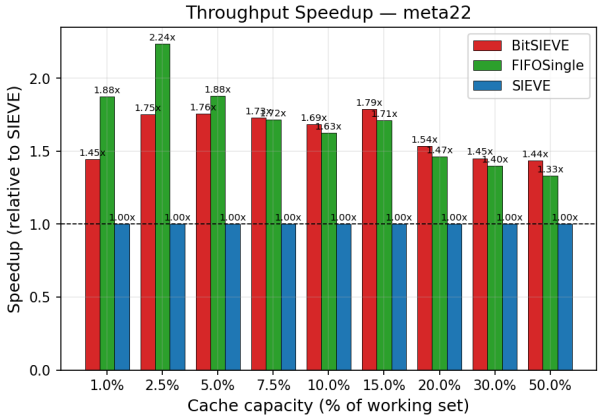


Fig. 4: **BitSIEVE Latency versus SIEVE and FIFO on the Meta KV 2022 Trace**



(a) BitSIEVE Speedup on the Meta KV 2022

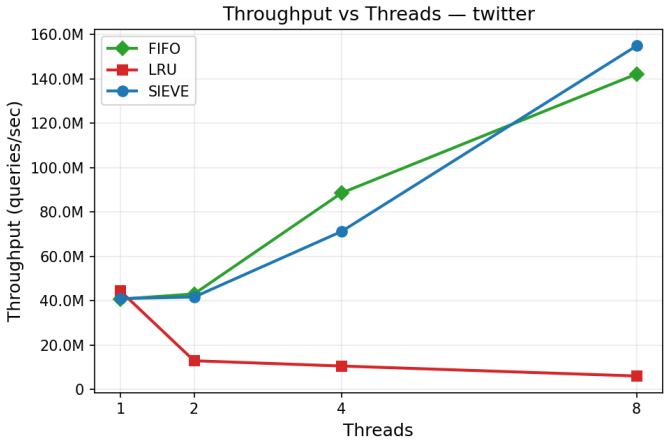


Fig. 6: **Replication of SIEVE paper throughput scaling**