Progress Report: Cache Replacement, Application Performance, and Relations to DSM

Zhengyi Chen

October 9, 2023

(Cache) Replacement Strategies

- There have been significant development in (CPU) cache replacement strategies in the last decades.
- e.g., RRIP(++)[1] and more recently (various) ML-*derived* heuristics.
- Also popular is studying adequate cache replacement strategies for distributed systems (though more stagnant).
- There are many variables within each cached system (whether CPU or distributed FS, etc.) that affect which strategy is more *efficient* in operation.
- Moreover, different applications (e.g., threads) exhibit different access patterns which may be better served by one strategy than another.[2]

[1] Jaleel et al., "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)".

[2] Sethumurugan, Yin, and Sartori, "Designing a Cost-Effective Cache Replacement Policy using Machine Learning".

Notable (i.e., Encountered) Strategies

- LRU family
- FIFO family
- Adaptive Replacement Cache
- CPU-LLC Intended: Dynamic Insertion Policy, Re-Reference Interval Prediction, Signature-based Hit Predictor, . . .
- ML-derived: Reinforcement Learned Replacement, LeCaR, Cache Replacement Problem as Markov Decision Process[3], . . .

[3] Gu et al., "Distributed cache replacement for caching-enable base stations in cellular networks".

Notable (i.e., Encountered) Strategies

- The performance of replacement strategies correlate strongly to the context of their operation.
- For example, LRU is theoretically better-performing than FIFO *in their most textbook implementations* but recent studies[4] have shown that FIFO can outperform LRU in practice (CDNs, for example, where even cache bookkeeping structures can be costly).
- To summarize, **The (dynamic) choice of replacement algorithm in any system is of practical concern!**

[4] Eytan et al., "It's Time to Revisit LRU vs. FIFO"; Yang et al., "FIFO Can Be Better than LRU: The Power of Lazy Promotion and Quick Demotion".

LRU & FIFO family – Patches and Applications

- The state-of-the-art implementations of LRU or FIFO is far-cry from their textbook implementations.
- This is so that they can capture both *recency* and *frequency*: we desire to use both to predict/assume the *re-reference interval* of a given entry.
- e.g., Linux uses `LRU_GEN` which is a multi-queue LRU strategy wherein each queue(generation) represents a ”similar” level of access recency and is evicted in batch.
- The kernel developers wanted a *fast and reasonably good* replacer as opposed to an optimal one.
- Likewise, Yang, et.al.[5] shows that FIFO with *lazy promotion* and *quick demotion* outperforms textbook LRU.

[5] Yang et al., “FIFO Can Be Better than LRU: The Power of Lazy Promotion and Quick Demotion”.

LRU_GEN and Access Patterns

The LRU_GEN algorithm specifically makes stronger protection of pages for memory accesses through PT than through FD:

- Heap/Stack/Text access misses have higher cost – executables perform blocking I/O at memory access, less likely for file access.
- They are also more likely to miss, as their in-kernel dirty bits are approximated.
- Finally, they can be reasonably assumed to more likely exhibit temporal locality.

Nevertheless, the algorithm is capable to dynamic adjustment on re-faults – **the data model of programs can be file-based or object-based**. The same algorithm can deviate in fault rates on different programs on the same node.

Machine Learning as Analytic Tool: RLR, etc.

- Large distributed systems (e.g., CDNs) can afford to perform machine learning for cache replacement tasks[6]: run-time is much faster than I/O so some cycles could be afforded.
- For page replacement in kernel, we can't really afford to run anything costly (Amir).
- ML analysis[7] shows how different (computation-intensive) programs exhibit distinct access patterns.

[6] Gu et al., "Distributed cache replacement for caching-enable base stations in cellular networks".

[7] Sethumurugan, Yin, and Sartori, "Designing a Cost-Effective Cache Replacement Policy using Machine Learning".

Machine Learning as Analytic Tool: RLR, etc.

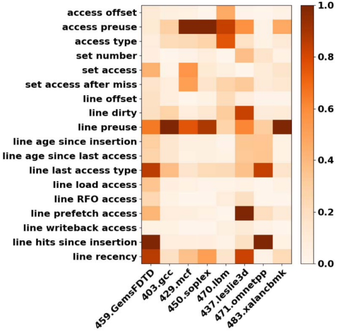

Fig. 3. Heat map of neural network weights. The y-axis shows features representing LLC state, and the x-axis shows the benchmarks used in the agent simulation. The features with high magnitude of weights are (considering at least three benchmarks) access preuse, line preuse, line last access type, line hits since insertion, and line recency. [8]

P.S. *preuse*: set access since last access to address/line.

[8] Sethumurugan, Yin, and Sartori, "Designing a Cost-Effective Cache Replacement Policy using Machine Learning".

DSM, Applications, and Memory (Contention)

The speedup of applications on DSM systems is negatively correlated to shared memory contention.
Take TreadMarks[9] for example:

- *Jacobi* is a solver for linear system of equations via the *successive over-relaxation* method. The memory access pattern should be map-reduce-like: the problem is parallelized w/ partial matrices for each node with immutable storage of the relevant matrices? TreadMarks achieves $\sim 7x$-speedup on a 8-node system over one single-core node.
- *Water* is a parallel N-body molecular dynamics simulator that requires at least $O(\frac{N}{2})$ communications per processor. TreadMarks only achieves $\sim 4x$-speedup with around 47% time used for blocking communications.

[9] Cox et al., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems".

DSM, Applications, and Memory (Contention)

- It's kinda difficult to compare statistics from different DSM systems.
- Even with the same programs being run, different parameters makes for different program behaviors wrt. contention, etc.
- Logically speaking, the more contention on the same address, the less speedup is possible for the system[10].
- Should cache replacement strategies be aware of how contended a page may be to prevent it from e.g., being swapped out?

[10] Lara et al., "The effect of contention on the scalability of page-based software shared memory systems".

Hardware-based Dynamic Strategy Selection: DIP

Hardware-based replacement strategies can provide low-cost inspirations for software replacement strategies.

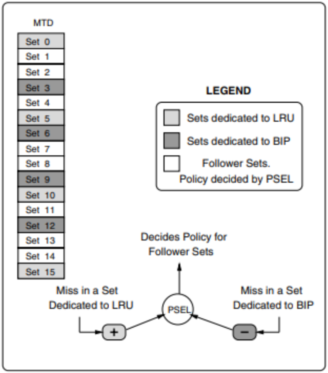

Figure 10: DIP via Set Dueling [11]

Problem: How can this be scaled for multiple strategies?

[11] Qureshi et al., "Adaptive Insertion Policies for High Performance Caching".