ECE 565 – Fall 2024 – The Shepherd Cache

Aidan Prendergast, Timothy Hein, Jacob Chappell
Group #32

Abstract

L2 Caches suffer from deeply decreased temporal locality resulting from increasingly effective L1 caches. As a result, more complicated victimization and replacement scheme must be put into effect as the spatial and temporal locality of accesses becomes increasingly fuzzy down the cache hierarchy. We implement and validate the Shepherd Cache replacement policy proposed by Rajan and Govindarajan, which utilizes a FIFO within a frame to evaluate access patterns more similarly to OPT. We assess their claims of how far a Shepherd Cache can bridge the gap between LRU and OPT and its continued relevance by evaluating performance on a modern workload, emulated with the SPEC2017 suite.

Introduction

Modern computer systems require a high-efficiency memory hierarchy to keep pace with multiple high-speed cores consuming immense quantities of program instructions and program data. This requires optimized hit times, miss rates, and miss penalties at each level of a cache hierarchy. Although slower than the datapath, memory can provide enough data by constructing a hierarchy of caches that take advantage of the spatial and temporal patterns present in access streams, referred to as locality. Continued improvements to Level 1 Cache implementations have “absorbed” ever more of this locality, passing increasingly random streams of L1 misses to the L2 cache [3]. Thus, a Least-Recently Used scheme has shown lesser performance, meriting a more involved approach to close the distance to optimal omniscient replacement.

Design

The Shepherd Cache revolves around hybridizing the ways of a cache set into a "Shepherding" FIFO that characterizes the access pattern of the cache and a Main Cache which victimizes blocks based on the FIFO's approximation of the optimal replacement policy. In the proposing paper [1], the Shepherd Cache was achieved using metadata corresponding to each set within the cache. These additional fields kept track of if the block was in the FIFO half, where it was in the FIFO ordering, and what order cache blocks had been earliest accessed in since the Shepherd block had been inserted. More specifically, the paper used a set of counters to implement a FIFO Queue/Relative Order for each SC way that ordered first accesses and physically allocated certain set ways as Shepherd Cache ways. These were also ordered as a FIFO, and, upon victimization, the blocks would be passed down the SC ways.

Within the Gem5 simulator, the possible implementation approaches are more limited. The simulator does not have per-set metadata (shared across ways) for its replacement architecture. So instead of storing a matrix per-set, we opt to store a row per-way regarding its earliest access time with respect to each shepherd cache block. In total, our design has data per-way including Valid Bit, SC Bit, Insertion Timestamp, and for each Shepherd Cache way a Touched Bit and a Touched Timestamp. The SC bit is used to track what is in the Shepherd Cache and what is in the Main Cache, while the Touched Bit and Timestamp are used to form the imminence matrix.

A replacement policy in the Gem5 simulator requires four defined functions: ‘reset’, ‘invalidate’, ‘touch’, and ‘getVictim’. These correspond to when a block is added to cache, when it is removed from cache, when it is accessed (via read or write) within the cache, and when the cache row is polled for an eviction candidate. Further constraining design of the shepherd cache, the reset, invalidate, and touch functions all pass only the single block in consideration, not the cache row. As a result, the shuffle functionality that would typically occur on the addition of a cache block instead must occur during the selection of a victim, as it is the only function that can determine the state of the cache row.

The ‘invalidate’ function merely sets the valid bit for the block to false. The rest of the algorithm considers such a block to have no other useful data. The ‘reset’ function sets a block to valid, adds it to the Shepherd Cache by setting the SC bit, marks its insertion time (to track its age relative to other SC blocks), and zeros its row of SC timestamp data. Later, we guarantee that the Shepherd Cache will never contain the maximum number of elements when this function is called, as the ‘getVictim’ function will always be called to find a suitable location prior to the insertion of a block. The ‘touch’ function toggles all touch flags in the row and provides a touch timestamp.

The ‘getVictim’ function of the replacement policy takes on most of the burden of the algorithm, as it has access to all ways of the cache simultaneously. It first determines the number of SC blocks, passing this value to each block individually. It then determines the oldest SC block, the only element of the Shepherd Cache ever a valid target for eviction. It picks a candidate by first searching for an invalid/empty block, then by searching for an untouched block in the oldest SC block's imminence ordering, and then by picking the youngest block in the imminence ordering. Additionally, if the SC is full, it will shift out the oldest block into the Main Cache to make room for the incoming block. Old and young are determined by comparing the touch timestamps and insertion timestamps of blocks.

A unique downside of this structure is that it degrades under multicore conditions. Coherence invalidations will never invoke the 'getVictim' routine but may invalidate Shepherd Cache blocks. Resultingly, there may be circumstances where the imminence matrix has columns that do not correctly align with their corresponding entries, as the FIFO order was quietly disrupted in a manner that the matrix could not be updated. Thus, we did not consider a multicore test for the Shepherd Cache in this state.

Methodology

The original Shepherd Cache proposal used the SPEC2000 benchmark suite to test their replacement policy [1]. Our testing has recruited the SPEC2017 benchmark suite to see if this augmentation still holds up against a more modern workload. We use a fork of the Gem5 simulator (the official version, not the course version) and an in-order processor with identical cache configurations to those listed in the paper to test the Shepherd Cache. We compared our shepherd cache implementation to an LRU replacement policy with 8- and 16-way associativity.

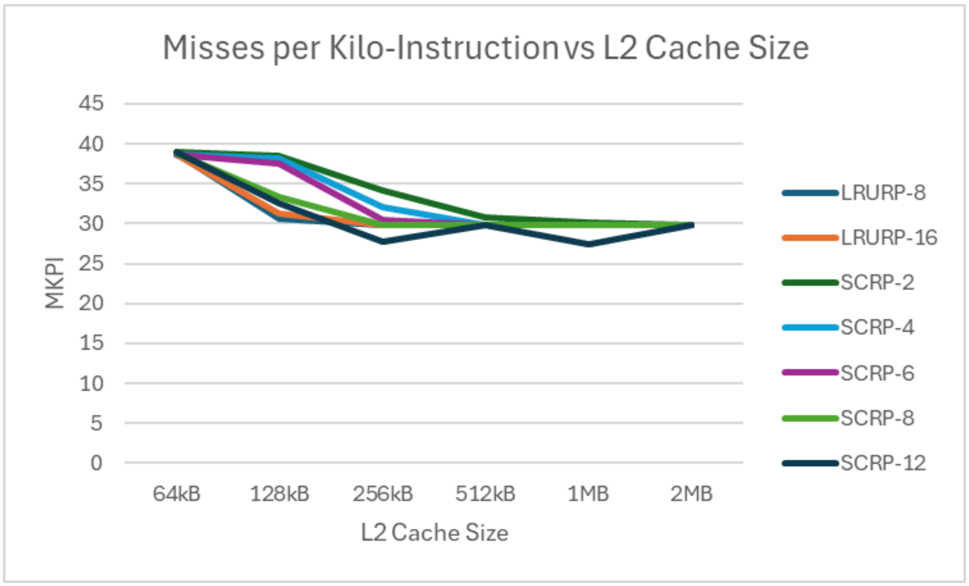
To establish a better understanding of the Shepherd Cache we also swept across a set of L1 and L2 capacity values and made the following observations. The claim of the Shepherd Cache paper is that the L1 cache absorbs all the temporal locality, justifying a different replacement policy for the L2. However, we observed that the L1 cache capacity used in the shepherd cache paper (16kB) is not enough to capture all temporal locality and instead the L1 performance started to experience diminishing returns closer to 64kB. We concluded to run our benchmarks on both 16kB and 64kB capacities to measure the differences.

Another observation was that their L2 cache capacity (4MB) was far beyond the point of diminishing returns, particularly since the replacement policy is stressed by having a lower capacity. Specifically, miss rates flattened out with cache size, likely due to working sets being much smaller than the cache. In our benchmarks we leaned farther into the low end of L2 capacities, ranging from 128KB to 2MB.

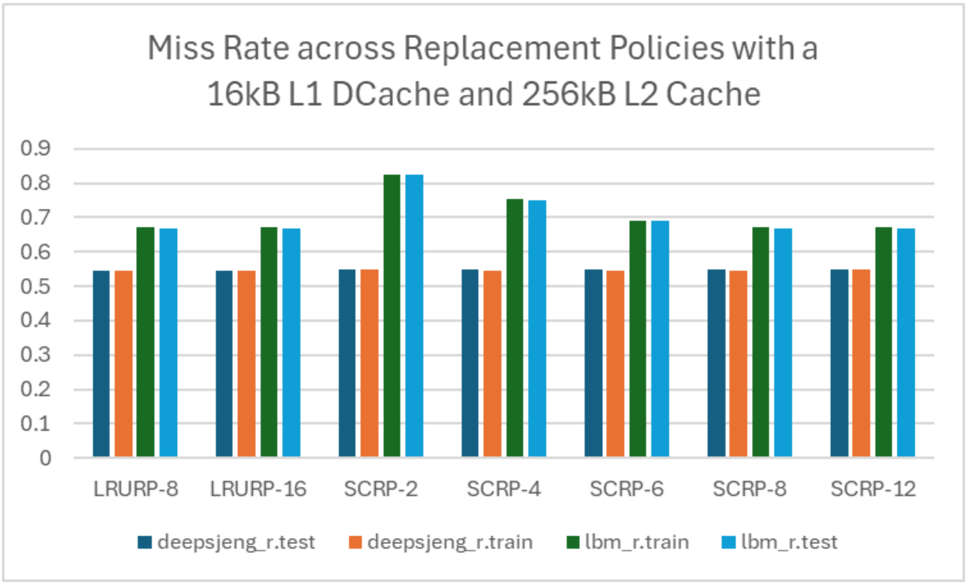
We validated our implementation by comparing an 8-way LRU cache to a 16-way Shepherd Cache with 8 main cache blocks. The 8-way LRU establishes a hard lower bound where, if the Shepherd Cache were to perform worse with the same number of main cache blocks, we would find serious issue with the design of the SC algorithm. We ran our benchmarks using the Gem5 simulator on a 64-thread processor. Each program ran for one billion instructions, ensuring measurement of the cache statistics in a steady state.

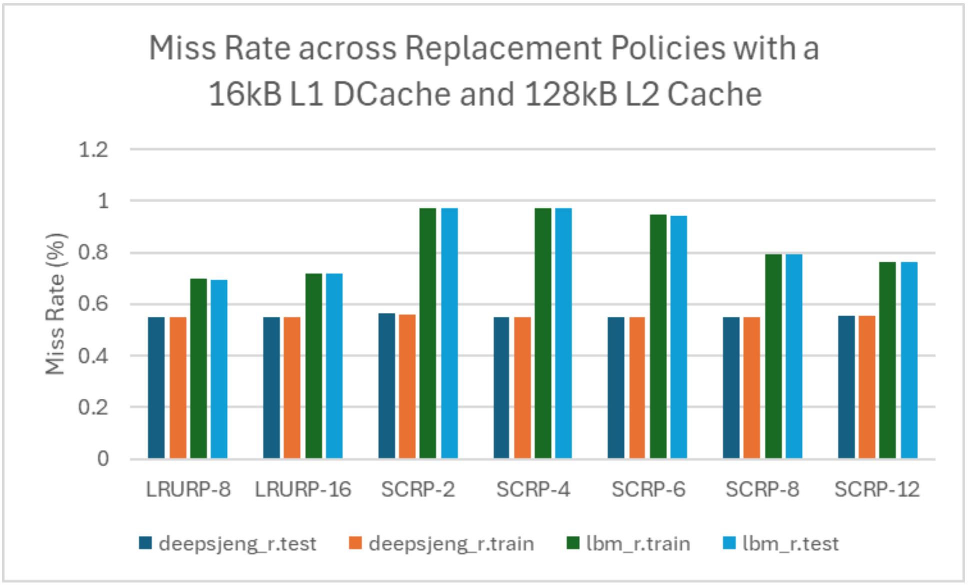
Results

We found little to no improvement in the best case from the Shepherd Cache (at most ~0.09% lower miss-rate than LRU). We compared Shepherd Caches with different SC allocation sizes and found that the SC-12 replacement policy performed best, averaged across all benchmarks in terms of average misses per kilo-instruction (MPKI), which is the metric used in the Shepherd Cache paper. The best MPKIs for the 64kB and 128kB were by the LRU with 38 and 30 respectively. The best MPKIs for the 256kB, 512kB, 1MB, and 2MB were all Shepherd Caches with 27, 29, 27, and 29 respectively.

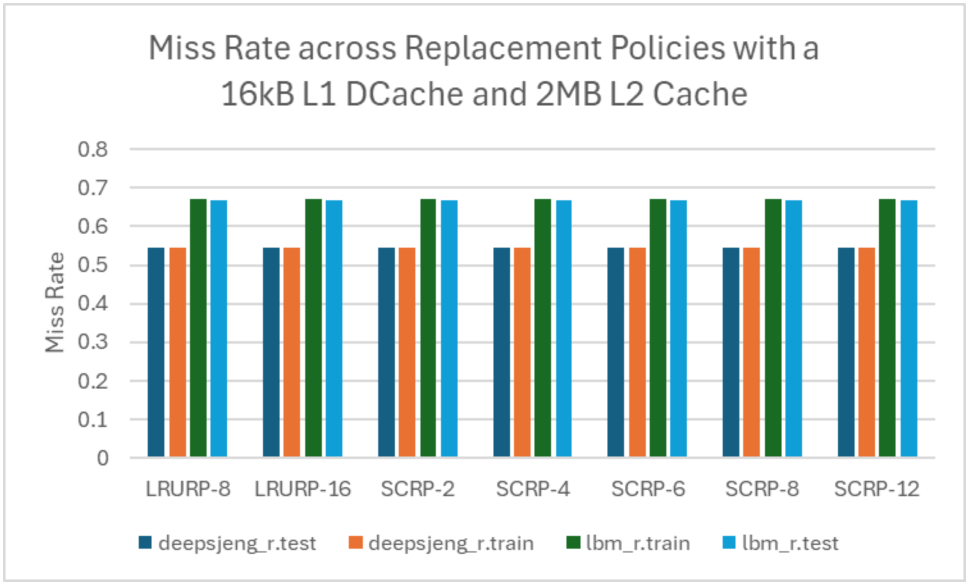


Observing that 128kB and 256kB had the most interesting distribution of performance, we examined the miss rate of the replacement policies at those specific cache sizes. We found that, in the best cases, they performed about on-par with LRU on our suite of modern workloads. The Shepherd Cache had a higher miss rate than the 16-way LRU by about 2.5% for a 128kB L2 capacity and tied (0.1%) for a 256kB L2 capacity.

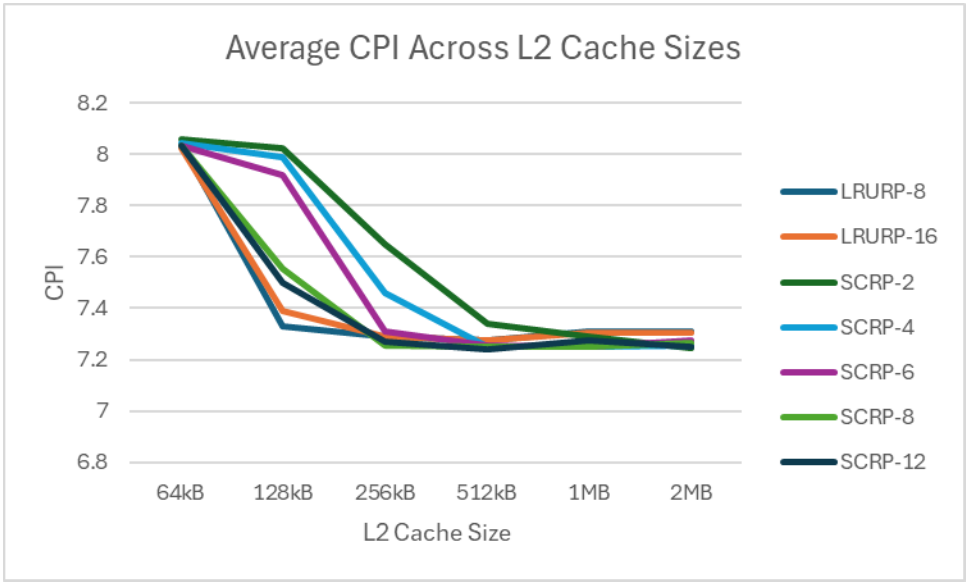




With the small L2 Cache sizes, we observed a dramatic change in miss rate amongst the Shepherd Cache, with SC-12 blocks coming closest to LRU. To see if this trend continued for large caches sizes, we ran the same set of simulations with a 2MB L2 cache. The 2MB Shepherd Cache had the same miss rate as the 16-way LRU (about one hundredth of a percent difference).



Here we found the miss rate reaching the point of diminishing returns with all policy miss rates approximately equal, reinforcing the conclusion that the Shepherd Cache does little to improve the overall performance.



This is backed up by the CPI data, that shepherd cache does perform the best with an average CPI of 7.24 across the largest two caches.

Conclusions

For most benchmarks, our Shepherd Cache Replacement Policy proves to have some configuration that beats out 16-way associative LRU in terms of miss rate. That said, dramatically better performance as seen in the original paper was not replicated on SPEC2017. We find the original paper dubious, using obscuring metrics in “Misses-per-Kilo-Instruction” over miss rate, and describing improvements in terms of “Improvement between LRU and OPT”. While these create effective context for how substantial certain gains are, they also mislead the impact on performance such an implementation has. We failed to observe any performance improvement close to the 7% reported [1].

Overall, this project served as a great learning experience and introduced us to more end-to-end project development. Working from scratch provided a far better understanding of the Gem5 simulator and the steps the course repository had taken to get to its current state. This approach did make certain aspects of simulation far more difficult but proved rewarding in the end. We are proud of our final product and thrilled our Shepherd Cache implementation works as well as it does.

References

[1] K. Rajan and R. Govindarajan, "Emulating Optimal Replacement with a Shepherd Cache," *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Chicago, IL, USA, 2007, pp. 445-454, doi: 10.1109/MICRO.2007.25.

[2] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," in *IBM Systems Journal*, vol. 5, no. 2, pp. 78-101, 1966, doi: 10.1147/sj.52.0078.

[3] W. A. Wong and J. . -L. Baer, "Modified LRU policies for improving second-level cache behavior," *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, Touluse, France, 2000, pp. 49-60, doi: 10.1109/HPCA.2000.824338.

[4] N. P. Jouppi and S. J. E. Wilton. 1994. Tradeoffs in two-level on-chip caching. SIGARCH Comput. Archit. News 22, 2 (April 1994), 34–45. https://doi.org/10.1145/192007.192015