Performance Evaluation of a Spontaneous Reload Cache

Alex Interrante-Grant,
Northeastern University,
EECE7352

December 8, 2016

Modern computer processors implement an increasingly complex memory hierarchy in order to balance the trade-offs between cost, power, and performance. One critical element of this hierarchy is the last level cache (LLC) between the CPU and main memory since the time cost of a memory access is high compared to a cache hit at any level. This paper examines a proposed LLC optimization in the form of a novel cache design proposed by Zhang et al. - the Spontaneous Reload cache. This paper shows that the performance benefits of an SR cache are questionable and argue that its implementation (outside of a simulation environment) is not as feasible as originally proposed.

1 Introduction

Often referred to as overcoming the "memory wall," optimizing interaction between the processor and main memory remains one of the most important problems in computer architecture [6]. As a result, there has been a considerable amount of research in the area of cache design. Modern processors implement a multi-layer cache between the processor and main memory. Following the conventions of the memory hierarchy, these caches are typically of increasing size and decreasing access speed as they get closer to main memory. Most research in this area tends to focus on the performance of the last level cache (LLC) as a miss in this cache forces the processor to access significantly slower main memory.

One major focus of LLC research is the cache eviction algorithm [3] [5] [2]. Although many novel and complex cache eviction algorithms are proposed every year, most modern processors still implement some form of least recently used (LRU) or pseudo LRU. As such, many performance increases in processor caches are realized due to increasing the size of the LLC, rather than implementing a novel eviction algorithm. This paper focuses on a novel cache eviction algorithm, implementing it in a simulator and evaluating its performance in comparison with LRU.

1.1 Background and Motivation

Zheng et al. proposed one such novel replacement strategy and cache design which they refer to as a Spontaneous Reload (SR) cache [8]. Traditional caches only reload non-resident data on a cache miss, relying on their chosen cache eviction algorithm to decide which block in an associative cache set to evict to make room for the requested block. An SR cache, on the other hand, takes a more active role in cache maintenance, adding the possibility to “spontaneously” reload cache blocks that are predicted to be used in the near future, evicting those that are predicted not to be used soon. In addition, the SR cache is not limited to updating the cache on a miss; it may spontaneously reload data even on a cache hit.

Zheng et al. introduce two additional pieces of data that must be maintained for every cache block and a metric that can be computed from them:

Idle Count The number of cache accesses since this block was last accessed.

Reuse Interval The interval, in cache accesses, at which this cache block is reused.

Reuse Distance = Reuse Interval - Idle Count

Because these data cannot feasibly be maintained for every block of main memory, an SR cache will use some of its storage to maintain only the idle count and reuse interval (but not the actual data) of some non-resident blocks of main memory. It does this by dividing cache ways (associative sets) into real blocks - those that are resident in cache - and virtual blocks - those whose idle count and reuse interval is tracked but whose data is not resident.

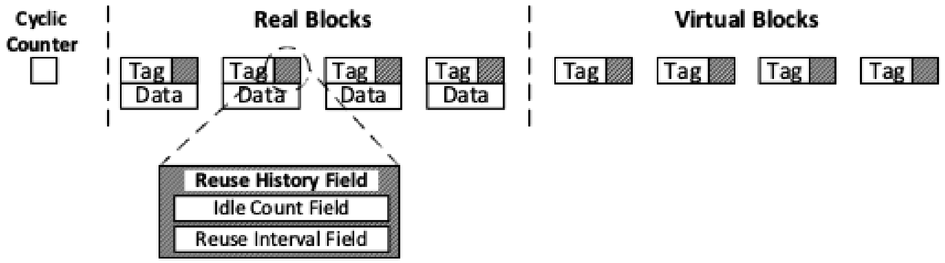


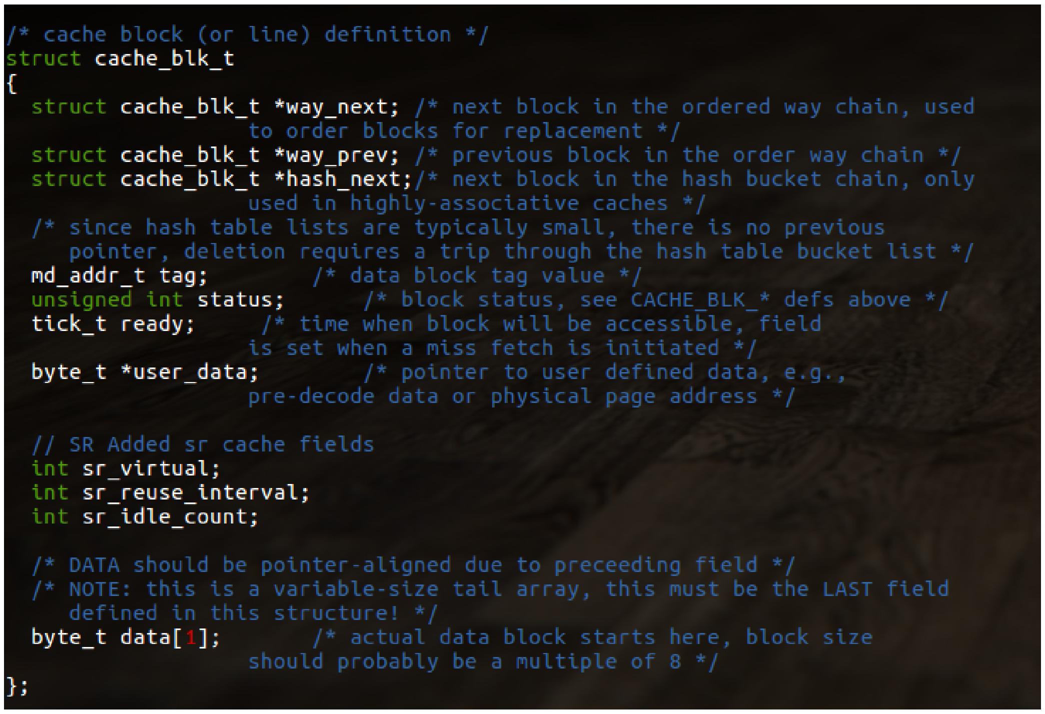
Figure 1: SR Cache Architecture [8]

At every cache access, hit or miss, the SR cache finds the real block with the maximum reuse distance and the virtual block with the minimum reuse distance and, if the minimum virtual reuse distance is less than the maximum real reuse distance, it reloads the cache block. On a cache hit, the hit block's reuse interval is set to its idle count and its idle count is reset to zero. Finally, on a miss, as a default eviction strategy, the SR cache will evict the real block with the greatest reuse distance.

The following implementation and simulations attempt to recreate Zhang et al.'s results but ultimately cast doubt on the feasibility of the SR cache.

2 Implementation

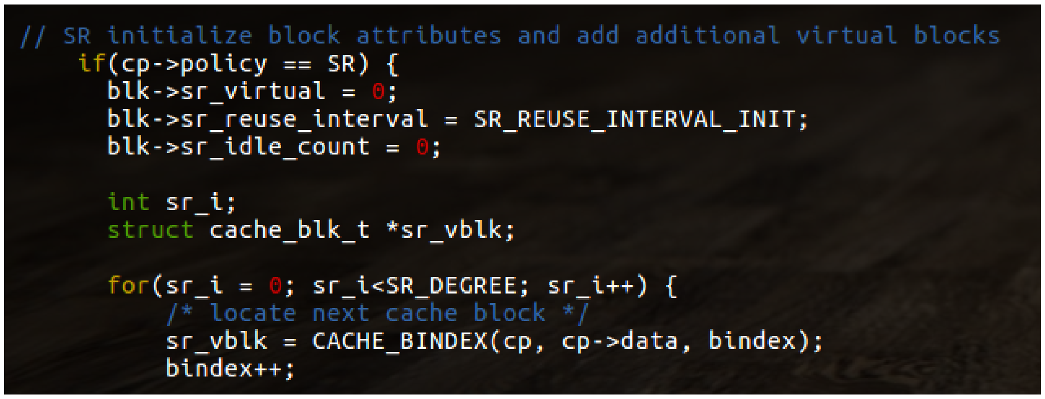
In order to test the effectiveness of an SR cache under various workloads, this analysis uses SimpleScalar [1] with a modified cache module. In order to implement the SR cache, I modified the cache block struct to include three additional fields - `sr_virtual`, `sr_reuse_interval`, and `sr_idle_count`.



```
/* cache block (or line) definition */
struct cache_blk_t
{
  struct cache_blk_t *way_next; /* next block in the ordered way chain, used
                   to order blocks for replacement */
  struct cache_blk_t *way_prev; /* previous block in the order way chain */
  struct cache_blk_t *hash_next;/* next block in the hash bucket chain, only
                   used in highly-associative caches */
  /* since hash table lists are typically small, there is no previous
     pointer, deletion requires a trip through the hash table bucket list */
  md_addr_t tag;        /* data block tag value */
  unsigned int status;      /* block status, see CACHE_BLK_* defs above */
  tick_t ready;     /* time when block will be accessible, field
                   is set when a miss fetch is initiated */
  byte_t *user_data;        /* pointer to user defined data, e.g.,
                   pre-decode data or physical page address */

  // SR Added sr cache fields
  int sr_virtual;
  int sr_reuse_interval;
  int sr_idle_count;

  /* DATA should be pointer-aligned due to preceeding field */
  /* NOTE: this is a variable-size tail array, this must be the LAST field
     defined in this structure! */
  byte_t data[1];       /* actual data block starts here, block size
                   should probably be a multiple of 8 */
};
```

Figure 2: SimpleScalar SR Cache Block

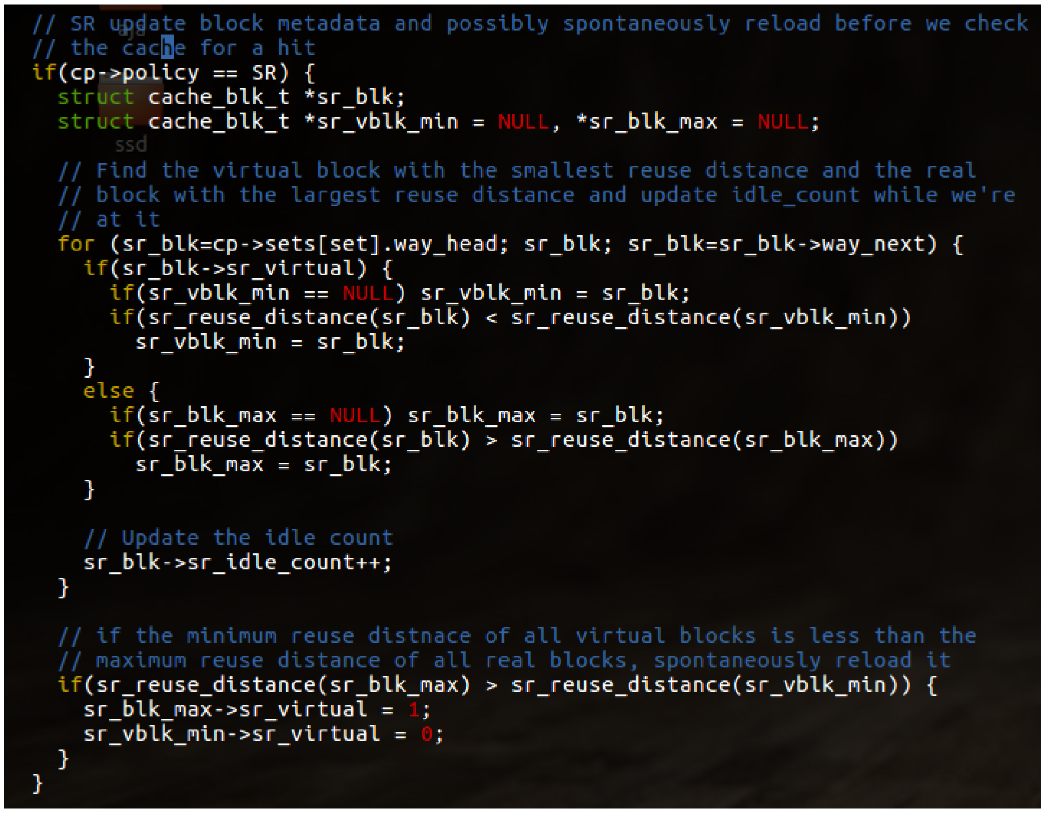
Next, I modified the cache allocation function to allocate additional virtual blocks at a given ratio to real blocks (SR_DEGREE).



```
// SR initialize block attributes and add additional virtual blocks
    if(cp->policy == SR) {
      blk->sr_virtual = 0;
      blk->sr_reuse_interval = SR_REUSE_INTERVAL_INIT;
      blk->sr_idle_count = 0;

      int sr_i;
      struct cache_blk_t *sr_vblk;

      for(sr_i = 0; sr_i<SR_DEGREE; sr_i++) {
        /* locate next cache block */
        sr_vblk = CACHE_BINDEX(cp, cp->data, bindex);
        bindex++;
```

Figure 3: SimpleScalar SR Virtual Block Allocation

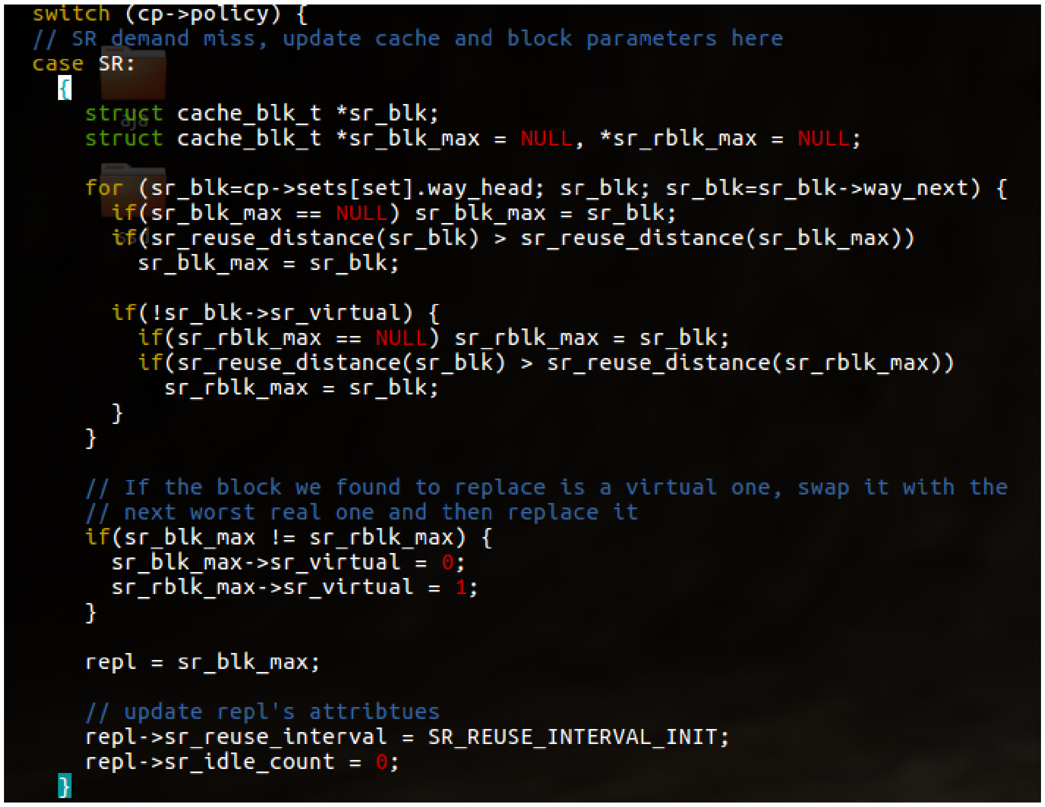
Then, I implemented the spontaneous reload functionality by finding the virtual block with the smallest reuse distance and the real block with the largest reuse distance (updating the idle count in the process). If the smallest reuse distance virtual block is less than the greatest reuse distance real block, the virtual block is marked as real and the real block is marked as virtual - acting as a simulated reload operation.



```
// SR update block metadata and possibly spontaneously reload before we check
// the cache for a hit
if(cp->policy == SR) {
  struct cache_blk_t *sr_blk;
  struct cache_blk_t *sr_vblk_min = NULL, *sr_blk_max = NULL;

  // Find the virtual block with the smallest reuse distance and the real
  // block with the largest reuse distance and update idle_count while we're
  // at it
  for (sr_blk=cp->sets[set].way_head; sr_blk; sr_blk=sr_blk->way_next) {
    if(sr_blk->sr_virtual) {
      if(sr_vblk_min == NULL) sr_vblk_min = sr_blk;
      if(sr_reuse_distance(sr_blk) < sr_reuse_distance(sr_vblk_min))
        sr_vblk_min = sr_blk;
    }
    else {
      if(sr_blk_max == NULL) sr_blk_max = sr_blk;
      if(sr_reuse_distance(sr_blk) > sr_reuse_distance(sr_blk_max))
        sr_blk_max = sr_blk;
    }

    // Update the idle count
    sr_blk->sr_idle_count++;
  }

  // if the minimum reuse distnace of all virtual blocks is less than the
  // maximum reuse distance of all real blocks, spontaneously reload it
  if(sr_reuse_distance(sr_blk_max) > sr_reuse_distance(sr_vblk_min)) {
    sr_blk_max->sr_virtual = 1;
    sr_vblk_min->sr_virtual = 0;
  }
}
```

Figure 4: SimpleScalar Cache Spontaneous Reload

Finally, I modified the default behavior on a miss for replacement block selection such that the block with the greatest reuse distance is selected to be replaced.



```
switch (cp->policy) {
// SR demand miss, update cache and block parameters here
case SR:
  {
    struct cache_blk_t *sr_blk;
    struct cache_blk_t *sr_blk_max = NULL, *sr_rblk_max = NULL;

    for (sr_blk=cp->sets[set].way_head; sr_blk; sr_blk=sr_blk->way_next) {
      if(sr_blk_max == NULL) sr_blk_max = sr_blk;
      if(sr_reuse_distance(sr_blk) > sr_reuse_distance(sr_blk_max))
        sr_blk_max = sr_blk;

      if(!sr_blk->sr_virtual) {
        if(sr_rblk_max == NULL) sr_rblk_max = sr_blk;
        if(sr_reuse_distance(sr_blk) > sr_reuse_distance(sr_rblk_max))
          sr_rblk_max = sr_blk;
      }
    }

    // If the block we found to replace is a virtual one, swap it with the
    // next worst real one and then replace it
    if(sr_blk_max != sr_rblk_max) {
      sr_blk_max->sr_virtual = 0;
      sr_rblk_max->sr_virtual = 1;
    }

    repl = sr_blk_max;

    // update repl's attribtues
    repl->sr_reuse_interval = SR_REUSE_INTERVAL_INIT;
    repl->sr_idle_count = 0;
  }
```

Figure 5: SimpleScalar SR Cache Replacement Algorithm

The following section explains the benchmarks used to evaluate performance and presents experimental results.

3 Simulation

3.1 Methodology

The following five benchmarks were selected from SPEC CPU 2000 [9] and SPEC CPU 2006 [10] to represent a diverse set of workloads to run on our SR cache implementation.

Name	Description
bzip2	In memory data compression
go	Play the game go - an artificial intelligence task
hmmer	Protein sequence analysis
mcf	Network simplex vehicle scheduling
milc	Gauge field generation - floating point

Table 1: Selected Benchmarks

These applications were cross-compiled for SimpleScalar under the Alpha architecture. A number of them were already cross-compiled and binaries were available online [7].

SimpleScalar's `sim-cache` program was used to evaluate the cache performance. The L1 cache configuration remained constant (see Table 2) and the L2 associativity, block size, and replacement algorithm were varied according to Table 3.

Name	Size	Associativity	Block Size	Replacement
L1 Data	32k	4-way	32	LRU
Instruction	32k	4-way	32	LRU

Table 2: L1 Cache Configuration

Test	Size	Associativity	Block Size	Replacement
1	2M	4-way	32	LRU
2	2M	4-way	32	SR
3	2M	8-way	32	LRU
4	2M	8-way	32	SR
5	2M	16-way	32	LRU
6	2M	16-way	32	SR
7	2M	4-way	64	LRU
8	2M	4-way	64	SR
9	2M	8-way	64	LRU
10	2M	8-way	64	SR
11	2M	16-way	64	LRU
12	2M	16-way	64	SR

Table 3: L2 Data Cache Configuration Tests

While the original proposal for this project suggested prefetching would be evaluated, SimpleScalar does not support prefetching at the moment, so varied block sizes will be examined instead. As is demonstrated below, varying block sizes yielded noteworthy results.

3.2 Results

3.2.1 bzip2

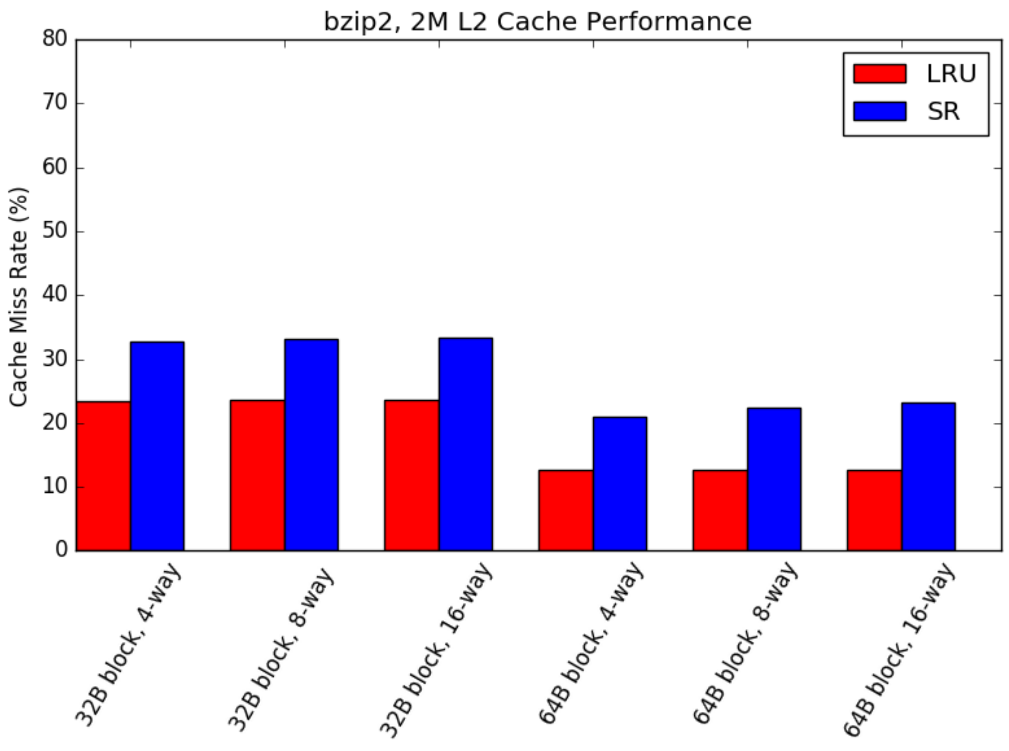


Figure 6: bzip2 Results

The bzip2 benchmark is a modified version of bzip that handles the majority of its data in memory so this benchmark gives us a relatively good idea of how programs that operate on large volumes of data in memory will perform. This likely explains why both algorithms benefit from a larger block size. Across the board SR performs worse than LRU, likely because bzip2 doesn't reuse much data - after the data is compressed, bzip2 is done with it. Spontaneous reloads here are likely to hurt performance.

3.2.2 go

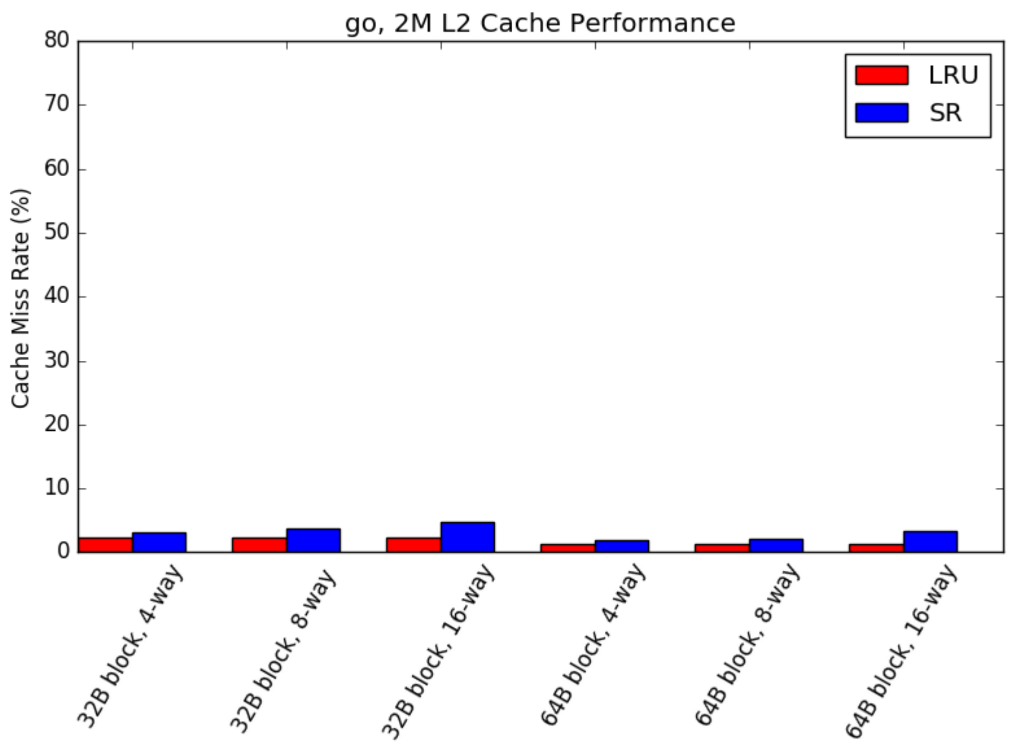


Figure 7: go Results

Both algorithms here perform fairly well with this benchmark, again with marginal improvements with increase block size. SR performed worse than LRU again for much the same reason as before. Although the go benchmark likely rereferences data (pieces on the board, etc.) it does not do so in a predictable fashion. In other words, a block's reuse distance is not likely to be constant so spontaneous reloads based on predicted reuse distance, again, probably hurt performance. As the associativity grows, so do the number of virtual blocks that the SR cache could spontaneously reload from, so the SR cache actually does not benefit from increased associativity here.

3.2.3 hmmer

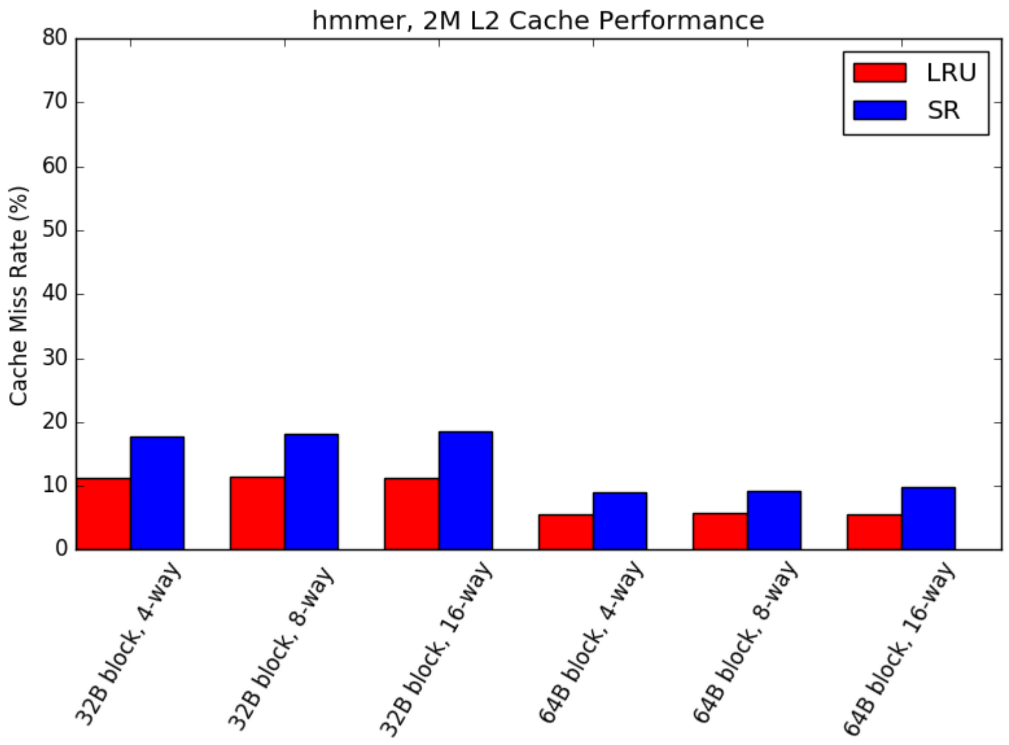


Figure 8: hmmer Results

The hmmer benchmark searches large gene sequences - storing large amounts of data in memory and searching through it. While both applications benefit from increased block size, we see the same performance gap between LRU and SR. As with the previous benchmarks, this is probably because the access patterns are not regular or easily predictable, so spontaneous reloads end up having a negative effect on performance.

3.2.4 mcf

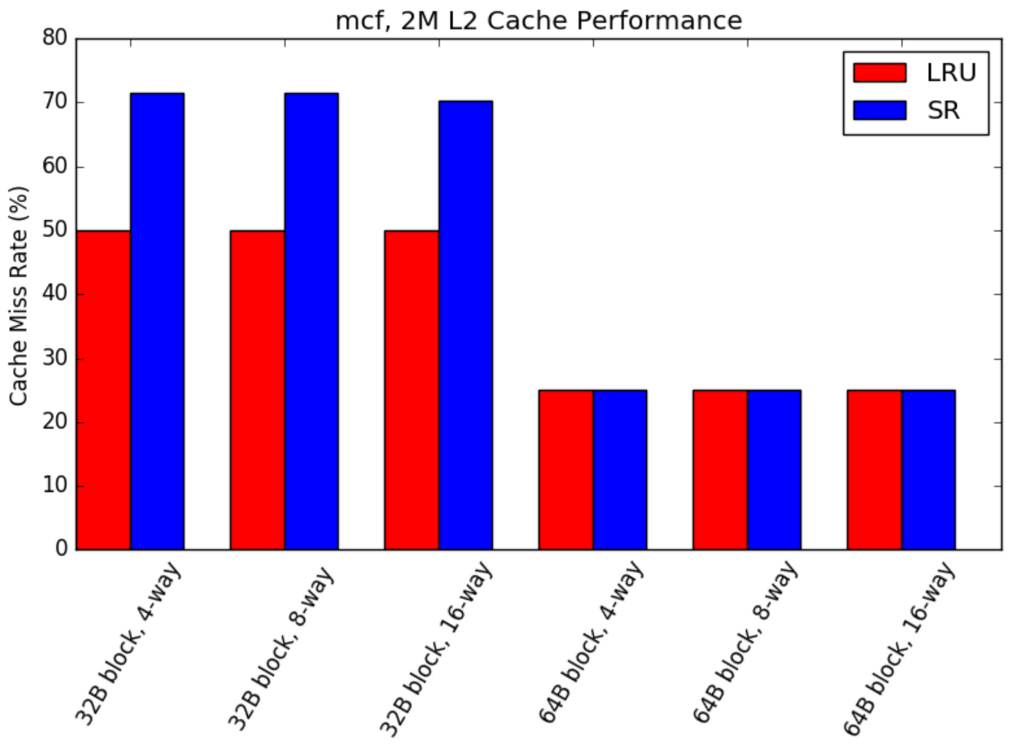


Figure 9: mcf Results

This benchmark reveals a large performance gap at a block size of 32. Interestingly, this performance gab disappears with an increased block size and SR actually very slightly outperforms LRU. This probably occurs due to more regular and predictable access patterns in the mcf benchmark on some struct that is larger than a single block of size 32. Since mcf uses a network simplex algorithm, these structs are probably nodes in a graph.

3.2.5 milc

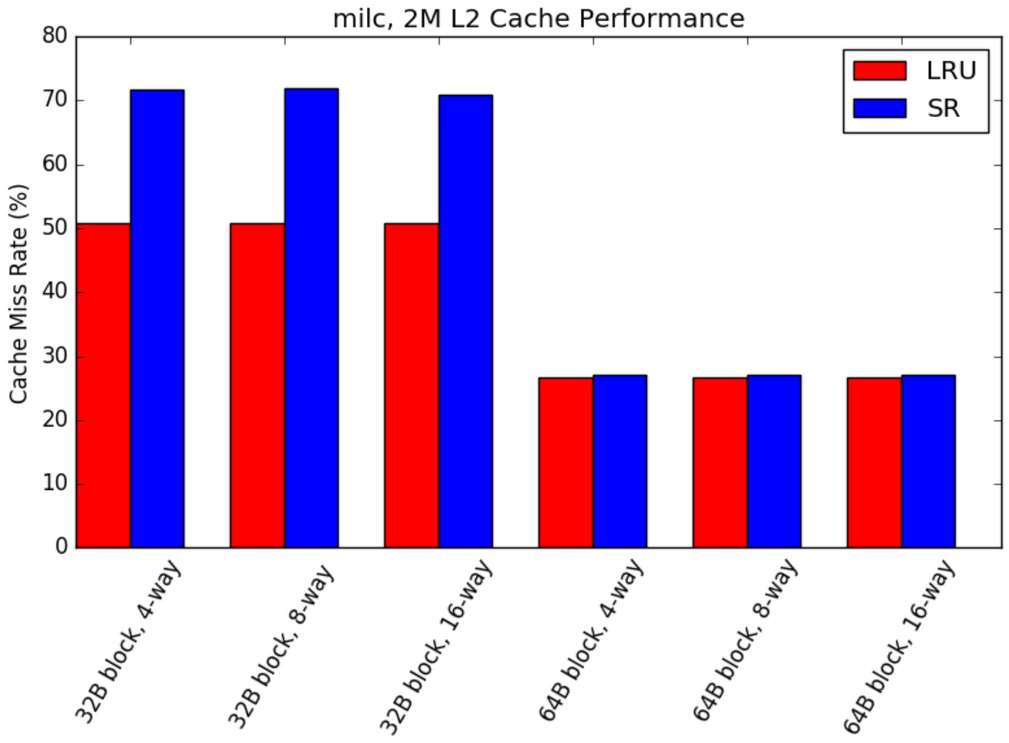


Figure 10: milc Results

Here, we see similar results to the mcf benchmark - worse performance by SR at block size 32 but very comparable performance at 64 - again probably caused by some higher level language construct. Neither algorithm seems to benefit from increased associativity in this case.

3.3 Discussion

These results do not show the increased performance that Zhang et al. claim by using using their SR cache. However, on closer inspection of their results and methodology there are a few clear reasons why.

First, the performance they show for a pure SR cache (like the one implemented here) actually indicate that it performs about 20% worse than a

LRU in their testing (1.2 MPKI vs 1.0 MPKI for the programs they tested). Indeed, this is in line with the results shown in this paper. It is not until after they implement what they call a Dynamic SR cache - a cache that implements both SR and LRU in parallel and selects the better performing one - that they are able to show performance on par with and slightly improved over LRU.

Second, the SR cache performs well in situations where access patterns are highly regular and the same data are used frequently. In the context of a higher level language, this would mean the SR cache performs well when the same small set of variables is referenced often in a tight loop. Many programs, however, operate on much larger data structures than the SR cache can maintain virtual blocks to keep track of given it's cache size limitations.

Finally, implementing an SR cache in a simulator requires liberties to be taken that are not so easily achieved in hardware. Updating the idle count for each block, real and virtual, stored in the cache on every access would consume a considerable amount of power. Spontaneously reloading data would increase traffic between the CPU and memory considerably and draw substantially more power with the possibility of a memory access on every cache access instead of just on cache misses.

4 Conclusion

Overcoming relatively slow storage technologies with effective caching schemes is a constant battle for computer architects. While novel solutions are frequently researched and proposed, few are feasible enough and provide enough performance improvement to justify their implementation in a real processor design. This paper has shown that, while Zhang et al.'s SR cache provides a slight performance improvement in select cases, it does not outperform LRU in general and would pose some real drawbacks if implemented in hardware.

References

[1] Austin, T., E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling." *Computer 35.2* (2002): 59-67

[2] Chaudhuri, Mainak. "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches." *IEEE/ACM International Symposium on Microarchitecture* (2009)

[3] Hazelwood, Kim, James E. Smith. "Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems." *International Symposium on Code Generation and Optimization* (2004)

[4] Henning, John. "SPEC CPU 2006 Documentation." *Standard Performance Evaluation Corporation*. Web.

[5] Jiang, Song, Xiaodong Zhang. "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance." *ACM SIGMETRICS* (2002)

[6] McKee, Sally A. "Reflections on the Memory Wall." *1st Conference on Computing Frontiers* (2004)

[7] Parihar, Raj. "SPEC CPU2006 Compilation for SimpleScalar/Alpha-OSF." *University of Rochester, College of Electrical and Computer Engineering*. Web.

[8] Zhang, Lunkai, Mingzhe Zhang, Lingjun Fan, Da Wang, and Paolo Ienne. "Spontaneous Reload Cache: Mimicking a Larger Cache with Minimal Hardware Requirement." *2013 IEEE Eighth International Conference on Networking, Architecture and Storage* (2013)

[9] https://www.spec.org/cpu2000/

[10] https://www.spec.org/cpu2006/