IMPLEMENTATION AND EVALUATION OF SELECTIVE VICTIM CACHING ON A DIRECT-MAPPED CACHE

Arunkumar Prabhashankar(UFID 39939249) and Sidhartha Behura(UFID: 09708313)
Department of Electrical and Computer Engineering
University of Florida

ABSTRACT

Victim cache is a small fully associative cache which aims to augment a direct mapped cache by storing the blocks that are evicted from the main cache due to replacements. To further improve the performance of normal victim cache a slight variation of it called the selective victim cache was introduced. It differed from the normal victim cache in the way in which it selected evicted blocks into the victim cache. In the selective victim caching scheme incoming blocks into the first level cache are selectively placed in the main cache or the victim cache using a prediction scheme based on their past history of use. This prediction algorithm is also used to interchange blocks between the main cache and the selective victim cache. Our aim in this paper is to evaluate the performance of a selective victim cache as compared to a normal victim cache and a cache without a victim cache in terms of miss rate and the simulation time.

I Motivation

The past few years have seen a tremendous increase in CPU processing power and clock speeds. This, compounded by the fact that the proportional improvement in memory access times have been miniscule, have led to an increasing gap between the processor speed and average memory access time. In this light, there has been a rapid increase in the cost that miss penalty incurs in terms of clock cycles. So the role of cache memory in decreasing the average clock cycles per instruction becomes even more pronounced. The direct mapped cache, although simple and offers access time and cost advantages, suffers from a high amount of conflict misses, which necessitates the implementation of more complex and expensive cache setups like set-associative and fully-associative. The concept of Victim cache was proposed by Jouppi[2] as a method to enhance the miss rate and miss penalty of direct-mapped cache. The direct mapped cache is enhanced with a small fully-associative cache that stores cache blocks evicted from the direct-mapped cache. This low-cost improvisation potentially addresses the above shortcoming. Additionally an improvement in the placement of newly fetched blocks and interchange of blocks between the L1 and the victim cache by making use of a prediction mechanism, as proposed in IEEE paper Stiliadis, D., and Varma, A. "Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches.", further enhances cache performance. We aimed to implement and evaluate this improvisation by tweaking simplescalar simulator and gain hands-on experience on Instruction Set Architecture enhancement.

II Introduction

The purpose of cache memory is to provide expedited access to instructions and data for the processor. Among the cache implementations, the direct mapped cache enables the fastest access times and is often the only implementation that provides single cycle access. However, the direct mapped cache is subject to a substantial amount of conflict misses. In the event of a cache miss, the replacement policy normally followed would thrash or evict an existing line when a line that conflicts with it is fetched from lower hierarchy. If the thrashed block is accessed again in the future, it would lead to a miss penalty thereby increasing average memory access time and consequently average cycles per instruction.

The victim cache, which is a small fully associative cache, acts as a catcher of the blocks that are being thrashed out of the L1 cache. When the block is referenced again, it would not be necessary to go to the next memory level as it will searched in parallel to the L1 cache. It is evident that avoiding a fetch from the next level would by itself do wonders for the average memory access time and miss rate. Once the block in the victim cache is accessed, it is brought back to the L1 cache and is replaced by the evicted block from the L1 cache. In case of a miss in both caches, the evicted block from the L1 cache replaces the Least Recently Used block in the victim cache. The reduction of miss rate would be even more substantial for smaller caches as they would experience more conflict misses. In larger caches, the time to interchange data between the L1 and the victim cache would take a toll on the gained speed.

Stiliadis, D., and Varma, A[1] proposed an improvisation on the normal replacement policy followed in victim caches. In selective victim caching, a selection algorithm is implemented which determines if incoming blocks should be placed either in the main cache or the victim cache using a prediction scheme based on their past history of use. Blocks that are more likely to be accessed frequently are placed in the L1 cache while the rest are placed in the victim cache. Likewise, when there is a victim cache hit, the prediction scheme is used to determine whether the blocks in the two caches have to be swapped. This would substantially reduce overhead time lost due to this interchange and lead to further performance improvement even in lager caches. The prediction scheme makes use of state bits that are associated with cache blocks. These denote the access history of a block the last time it was placed in the L1 cache.

III Algorithm

Selective Victim Caching involves the use 2 state bits, hit and sticky, to maintain and utilize the history of accesses of a certain block. These state bits are stored in the L1, victim and L2 caches. That means that every time a block has to be read from the L2 cache, its state bits also have to be read and each time a block is written back to the L2 cache, its state bits have to be written as well. Initially the state bits in L2 cache are given a value of zero and updated as the program runs.

Here is a brief description of the algorithm:

Let ‘β’ be the desired block and ‘α’ be the conflicting block

Case 1: L1 cache hit

- Access to line β, hit in main cache;
- Update hit and sticky bits as follows:
  - Hit[β] -> 1, Sticky[β] -> 1

Case 2: Victim cache hit

If sticky[α] = 0
- interchange α and β;
- sticky[β] -> 1, hit[β] -> 1

Else
if hit[β] = 0 then
- sticky[α] -> 0

else
- interchange α and β;
- sticky[β] -> 1, hit[β] -> 0

Case 3 : Miss both caches

If sticky[α] = 0
- Move α to victim cache.
- Move β to main cache
- sticky[β] -> 1, hit[β] -> 1

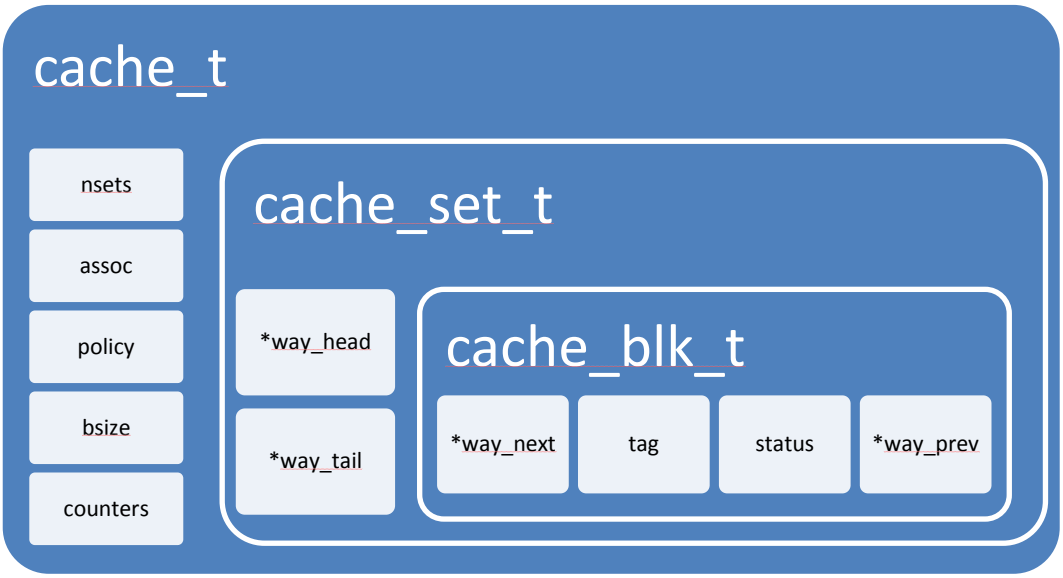
Else

if hit[β] = 0 then
- transfer β to victim cache
- sticky[α] -> 0

else
- Move α to victim cache.
- Move β to main cache
- sticky[β] -> 1, hit[β] -> 0

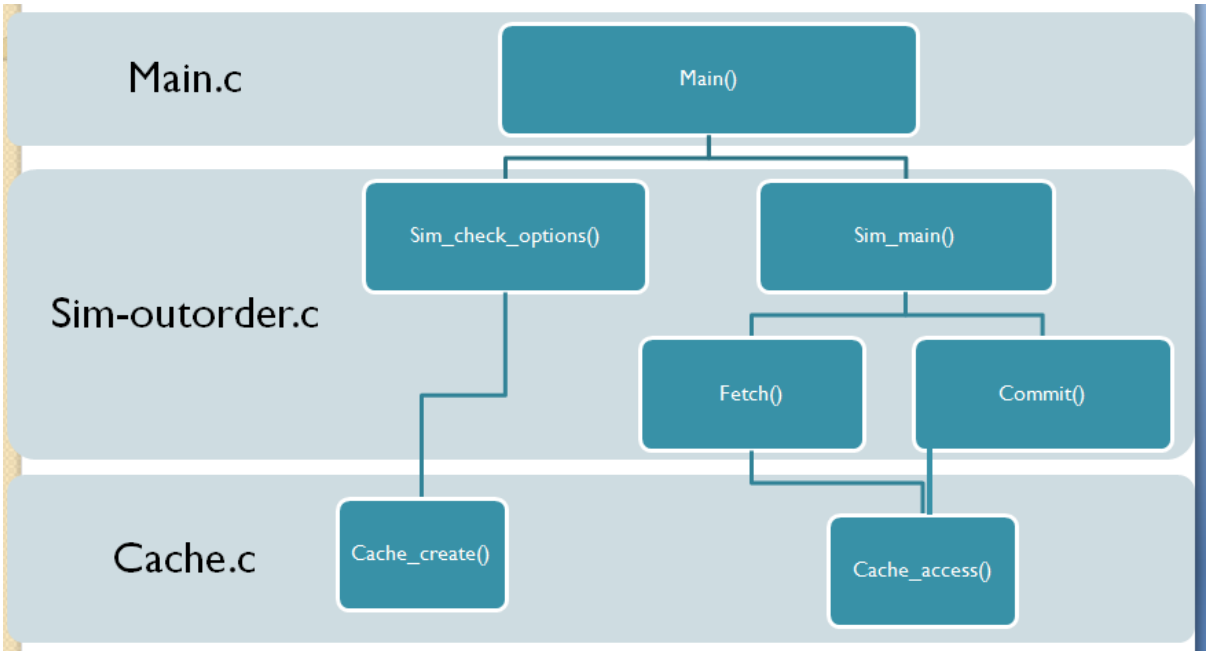
IV Cache implementation in Simplescalar Overview

Simplescalar is a simulator that approximates the execution of a dynamically scheduled processor. It is capable of simulating alpha and pisa configurations in each of its 4 functional simulators,sim-safe,sim-fast,sim-bpred and sim-cache, and one performance simulator sim-outorder. It implements cache structure using a combination of structures and linked lists.



The cache is defined by the structure cache_t, which comprises various parameters like number of sets, associativity, block size etc and implements the sets within it as an array of structures of type cache_set_t. Each set comprises of a linked list of blocks of type cache_blk_t. The blocks, in addition to data, hold variables like tag and status. The cache is primarily dealt with by two functions, cache_create() and cache_access(). Cache create deals with cache creation and instantiation while cache_access() controls the cache accesses.

Normal Program Flow:



V Implementation in Simplescalar:

Our implementation of the project in simplescalar involved two stages. The first was to implement a simple victim cache and next to implement the selective victim cache algorithm. The code may be referenced from the appendix. We only give a brief overview here.

1) **Simple Victim Cache**

Sim-outorder() was modified to create a victim cache of size 8 lines and line size 32 bytes along with the L1 cache. We also implemented the following other changes:

Cache.c

A new L1 cache access function Cache_access_victim() was implemented, which in addition to the usual cache_access() arguments, accepts the victim and its miss handler function as arguments . This function searches the victim cache in parallel with the L1 cache. On a victim cache hit, it would swap the block with the L1 cache block. In case of a miss in both L1 and victim caches, L1 cache miss handler would be invoked, and the LRU block in the victim cache would be evicted to make way for the evicted block from the L1 cache.

Sim-outorder.c

A miss handler for the victim cache victim_cache_access_fn() was added. This function returns a predefined value of latency on a victim cache miss.

2) Selective Victim Cache

Once a simple victim cache was implemented, we went on to implement the selective victim caching algorithm. Initially state bits of all blocks in the L2 cache are set as '0' and then modified as the block is written back. This implementation involved several changes to pre-existing constructs and functions we created to implement the victim cache.

Cache.h

1) The following constants were added to set and reset state bits of a block.

```
CACHE_BLK_VALID         /* block in valid, in use */
CACHE_BLK_DIRTY         /* dirty block */
HIT_BIT                 /*hit bit*/
STICKY_BIT              /*sticky bit*/
NOT_HIT_BIT             /*remove hit bit*/
NOT_STICKY_BIT          /*remove sticky bit*/
```

2) The cache_blk_t structure was modified to include an extra byte called 'state', which holds the state bits.

Cache.c

1) Cache_access_victim() function was modified to implement the selective victim caching algorithm. On an L1 cache miss this function checks the state bits of the incoming block and decides where to place it

2) Victim_cache_access() function checks for a victim cache hit and invokes the get_state() function on a victim cache miss.

3) Get_state() function handles fetching the state bits from the next level of the memory hierarchy on a L1 and victim cache miss.

4) Write_state() function handles writing the state bits to the next level of the memory hierarchy on a writeback.

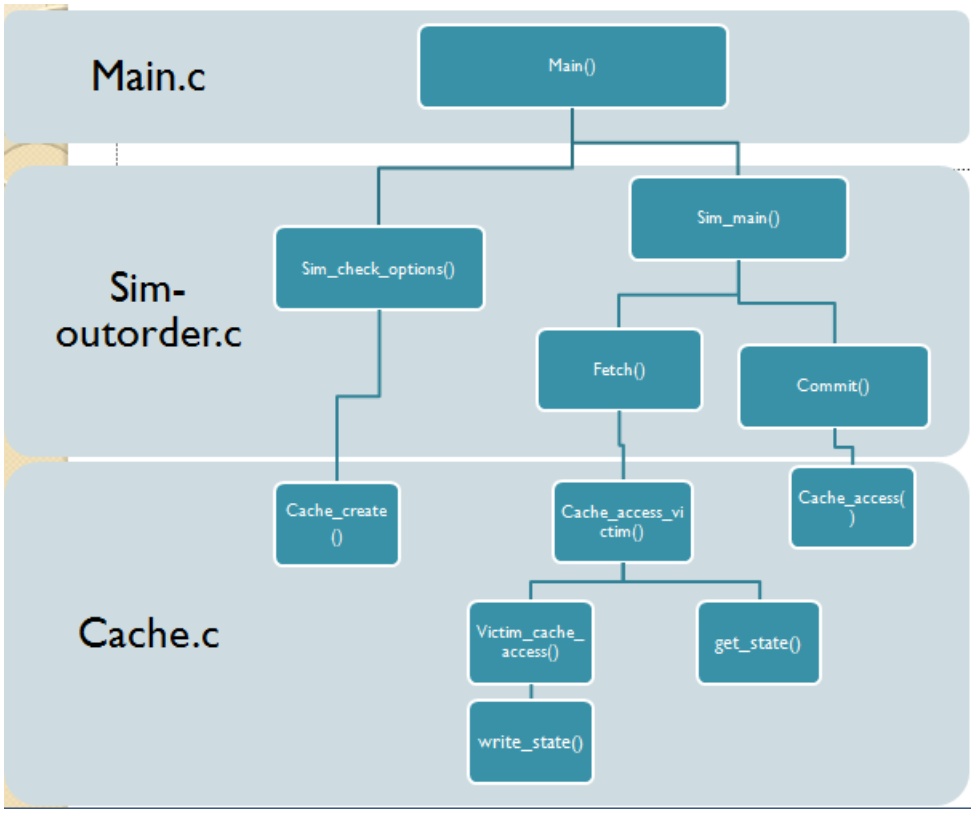
Sim-outorder.c

1) victim_cache_access_fn() was modified to invoke il1_write_state() function on a writeback.

2) il1_write_state() function checks for the presence of an il2 cache and writes back the state.

3) Il1_fetch_state() function checks for the presence of an il2 cache and fetches the state bits.

Modified program flow:



VI Simulation

All the simulations have been done with a victim cache of size 8 lines where each line is of 32 bytes. The L2 cache configuration is –cache il2 il2:4096:128:1.All the simulations have been done in alpha configuration. The L1 cache has been given a fixed line size of 32 bytes and its size has been varied from 2 KB to 4KB for evaluating with various benchmarks.

CPI:

Initially the tests have been done with simple benchmarks such as test-math, test-fmath, test-llong,test-printf,test-args. The L1 instruction cache has 64 sets,32 byte blocks and is direct mapped.

	Normal cache	Victim cache	Selective victim cache
test-math	2.7227	2.7062	2.6928
test-fmath	2.9837	2.9707	2.9483
test-llong	3.311	3.2892	3.2665
test-printf	2.4431	2.4147	2.4084
test-args	3.812	3.8023	3.7575

Table 1: CPI vs simple benchmarks for a 4 KB L1 cache

The table above shows CPI for different simulations for direct mapped L1 cache without a victim cache, with a victim cache and with a selective victim cache. Clearly a cache with a victim cache has better performance as compared to a cache without victim cache. This is as expected as many of the

blocks evicted from the direct mapped L1 cache will now be fetched from the victim cache rather than the lower level cache.

Including the selective victim cache shows an improvement over a normal victim cache. This is because we are selectively placing the blocks in the main cache or the victim cache based on prediction of which block is likely to be accessed again.

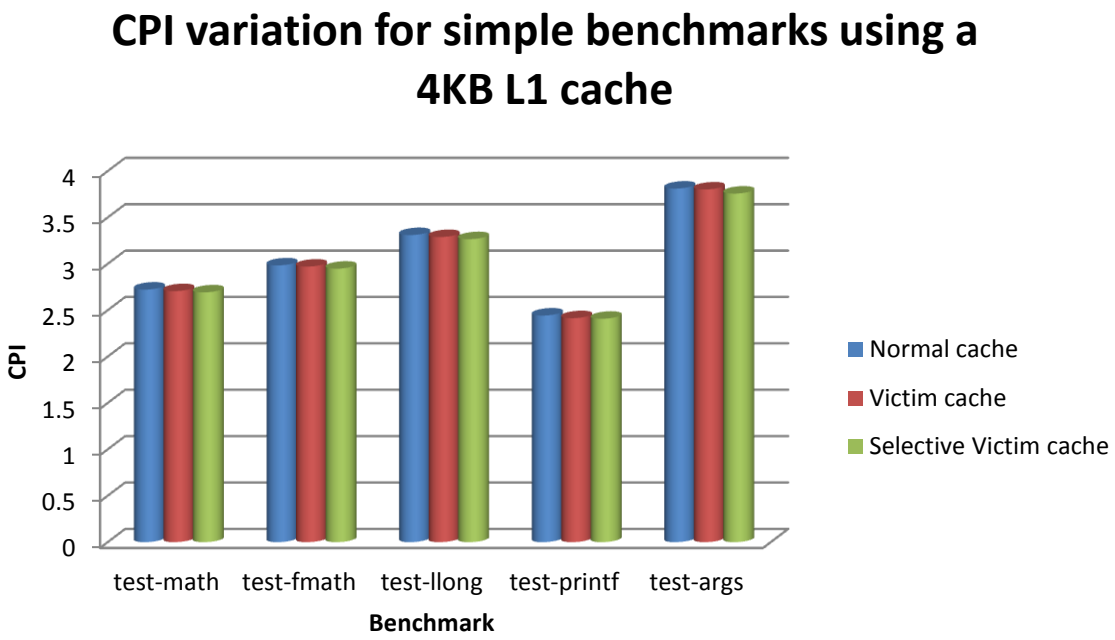


Figure 1 : CPI vs simple benchmarks for 4KB L1 cache

The table below presents the cycles per instruction (CPI) for three benchmarks (equake,crafty and gzip).The L1 instruction cache has 64 sets,32 byte blocks and is direct mapped.

	Normal cache	Victim cache	Selective Victim cache
equake	2.3771	2.3488	2.3456
crafty	2.5212	2.5125	2.5101
gzip	1.8623	1.7918	1.7918

Table 2: CPI of L1 cache using different simulations.

Introduction of the victim cache has a big impact on the CPI when we compared to an L1 cache without a victim cache Selective victim cache clearly outperforms the normal victim cache..We do not see any improvement while using selective victim caching as compared to normal victim caching in gzip benchmark. The reason for this is apparent from the nature of programs used. Programs that used static allocation of data and regular data structures showed significant improvements as they can be easily taken care of by the prediction algorithm.

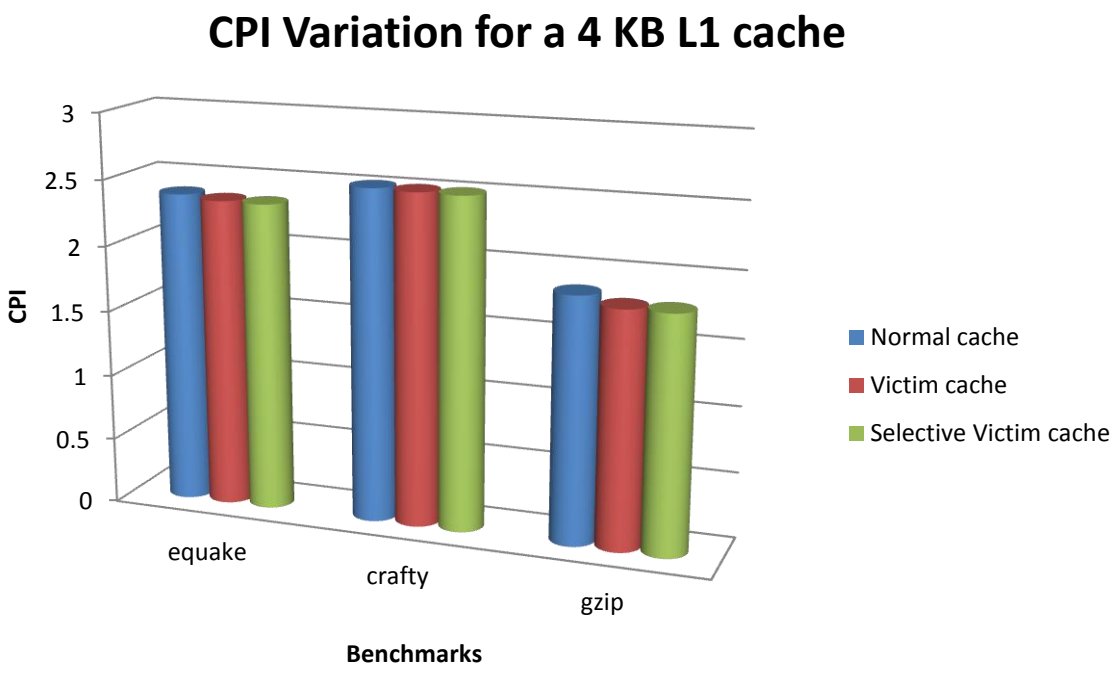


Figure 2: CPI vs SPEC2000 benchmarks for 4KB L1 cache

Miss Rate:

The simulations have again been performed to evaluate the performance of 4KB L1 cache to see the improvement in miss rates while using victim caching and speculative victim caching. All the simple benchmark simulations show an expected improvement while using the selective victim cache over the victim cache which shows an improvement over a cache without a victim cache. The only abnormality is while using the test-args simulation which may be due to scattered memory access patterns i.e. without any repetition.

	Normal cache	Victim cache	Selective Victim cache
test-math	0.0968	0.0946	0.0941
test-fmath	0.0905	0.0886	0.0879
test-llong	0.0803	0.0774	0.0771
test-printf	0.0941	0.0905	0.09
test-args	0.081	0.0797	0.0804

Table 3: Miss rate of L1 cache using different simulations.

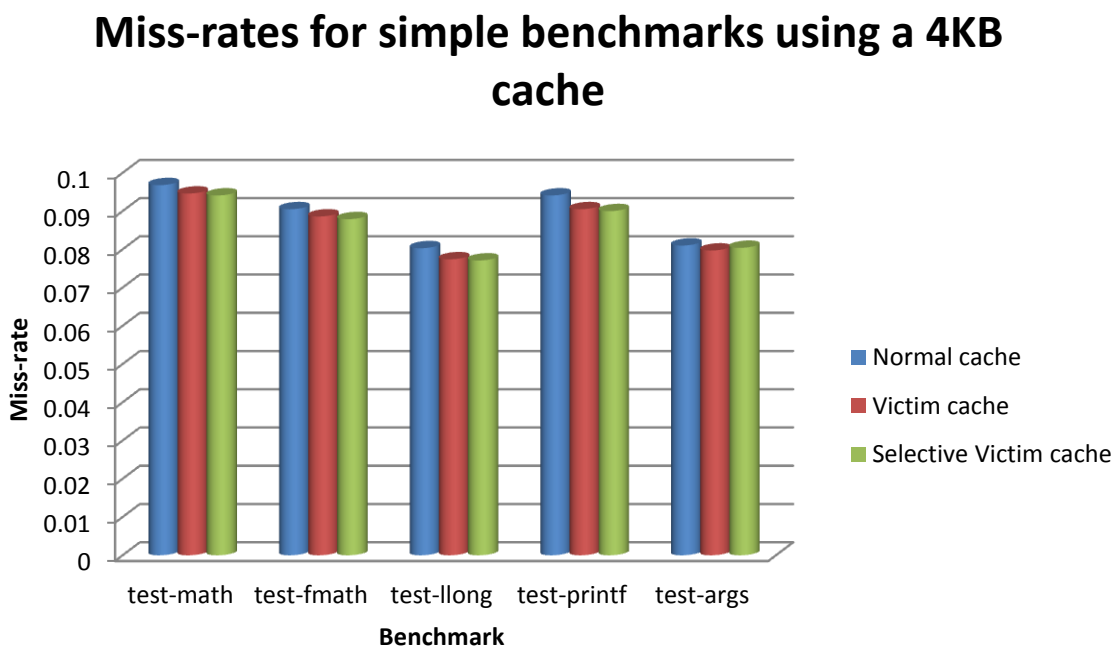


Figure 3: Miss rate of L1 cache using different simulations.

For SPEC2000 benchmarks, we see a drastic improvement of 99% in case of gzip with the introduction of victim cache. Further improvement with the introduction of a selective victim cache is almost negligible as the miss rates are already very low.

	Normal cache	Victim cache	Selective Victim cache
equake	0.0731	0.0696	0.0693
crafty	0.0988	0.0976	0.0973
gzip	0.0108	0.00002	0.00002

Table 4: Miss rate of L1 cache vs SPEC2000 benchmarks for a 4KB L1 direct mapped cache.

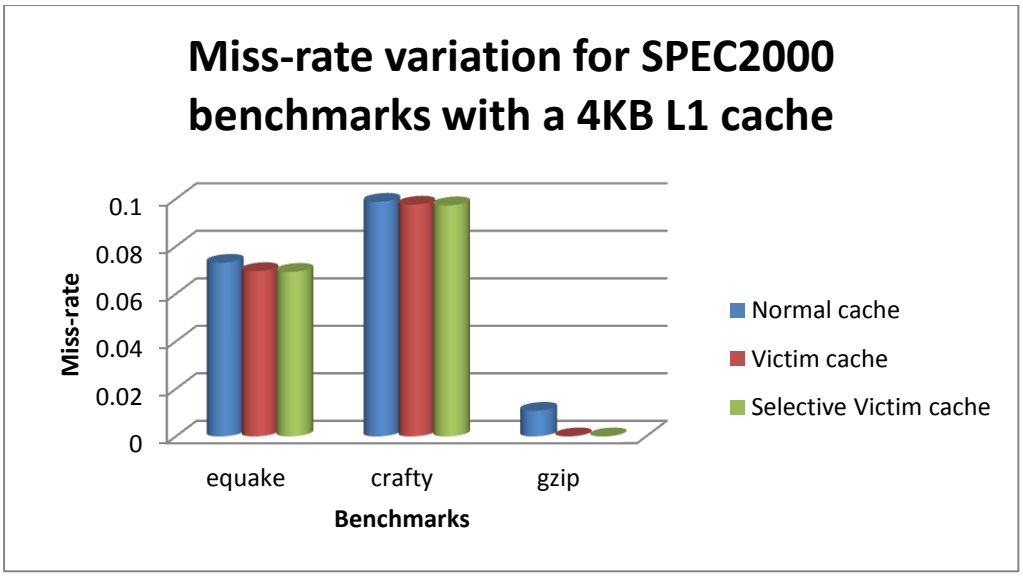


Figure 4: Miss rate vs SPEC2000 benchmarks for 4KB L1 cache

Performance while varying L1 caches

The table and graph below show the variation of miss rate using equake simulation for the various schemes. We can see that as the L1 cache size increases from 2KB to 8KB the performance of selective victim cache increases. As the cache size increases the amount of conflict misses in the L1 cache decreases, which justifies the improvement.

For all the L1 caches (i.e 2KB,4KB and 8KB) we have a constant victim cache size of 8 lines,32 byte blocks and fully associative cache. The L2 instruction cache is of 4096 lines,128 bytes and direct mapped.

For a 2KB cache the parameters are
Instruction level 1 cache- cache il1 il1:64:32:1
For a 4KB cache the parameters are
Instruction level 1 cache- cache il1 il1:64:32:1
For a 2KB cache the parameters are
Instruction level 1 cache- cache il1 il1:64:32:1

	Normal cache	Victim cache	Selective Victim cache
2 KB	0.0863	0.0831	0.0831
4 KB	0.0731	0.0696	0.0693
8 KB	0.0472	0.0438	0.0431

Table 5: Miss rate of L1 cache for different L1 cache sizes.

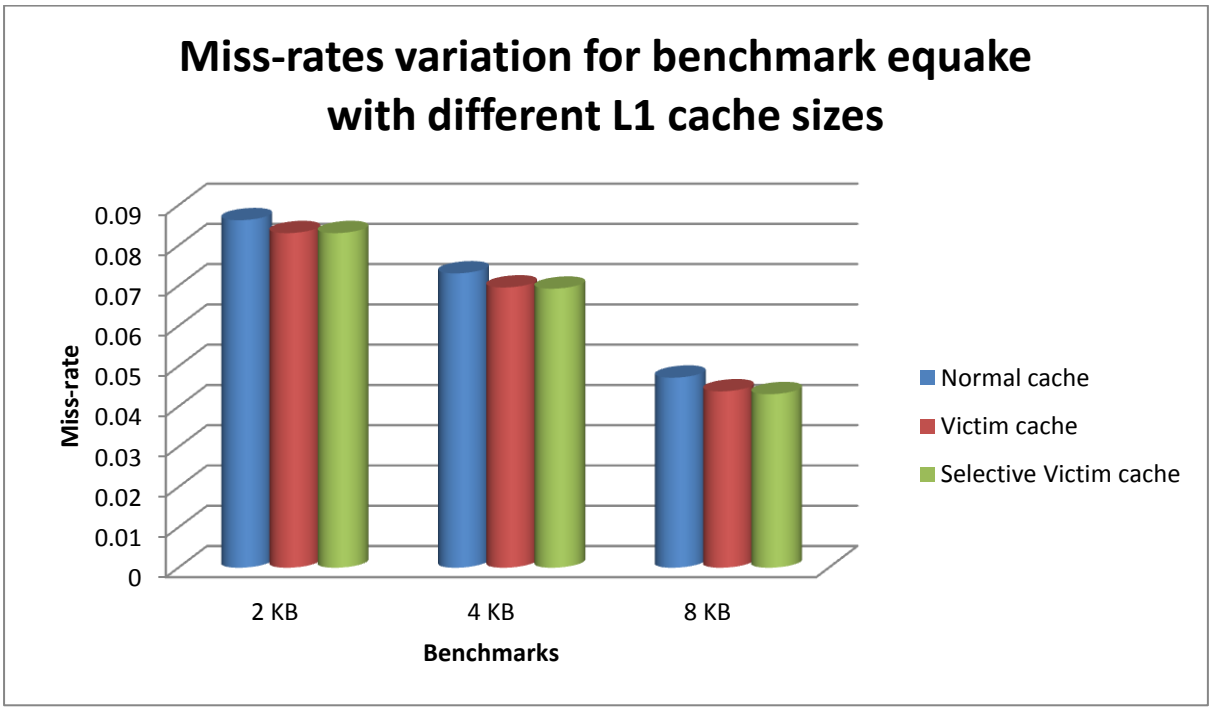


Figure 5: Miss rate vs cache size for benchmark equake

Total Simulation Time

Another performance indicator the simulation time has been used to evaluate the improvement that selective victim cache provides. Results obtained from simulating lucas benchmark show an improvement of 4% and a 3% improvement while using the gzip benchmark. This shows how effective the use of selective victim cache could be.

	Normal cache	Victim cache	Selective Victim cache
equake	188	194	191
crafty	184	191	184
gzip	143	154	148
lucas	106	121	116

Table 6: Simulation time vs SPEC2000 benchmarks for 4KB L1 cache

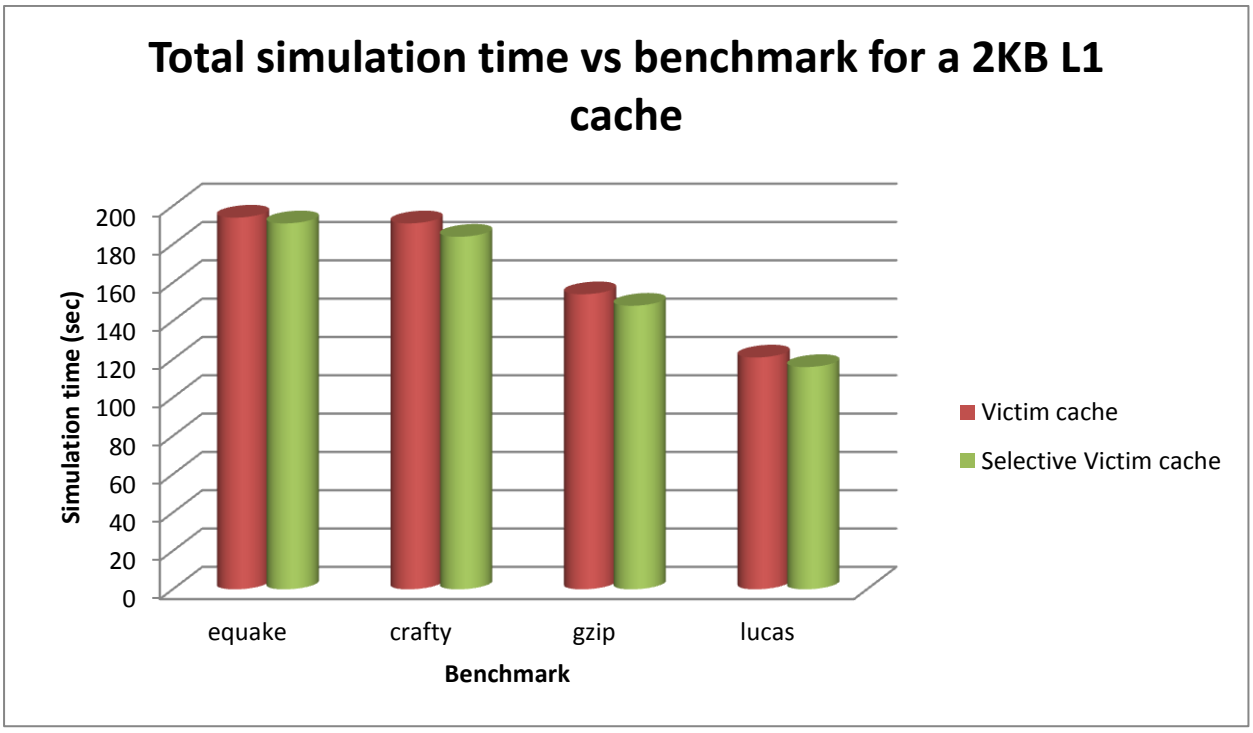


Figure 6: Total Simulation time vs SPEC2000 benchmarks for a 2KB direct mapped L1 cache

VII Results

- The miss rate and CPI have decreased as with the introduction of victim cache due to lesser conflict misses.
- With selective victim cache this value decreased further and shows a maximum decrease of 3.21% and 99% respectively wrt the normal cache.
- But as the L1 cache size keeps growing the improvement reduces. This happens because as the L1 cache size increases the chance of a conflict miss decreases.
- Also not all benchmarks show the same improvement due to the difference in the type and focus of instructions in each benchmark.

VIII Conclusion

In this paper we evaluated the performance of a victim cache and compared it to a cache without a victim cache .Results from simulations show a significant improvement when the victim cache is used with a first level instruction cache. Results obtained in a gzip simulation show a 99% improvement in CPI when we include a victim cache as compared to a cache without a victim cache. We also evaluated a method (selective victim caching) for improving the performance of a victim cache. The method selectively places instruction blocks in the main cache or the victim cache based on a prediction algorithm which determines how likely the block is to accessed in the future. Results from simulations of nine benchmarks of which four are SPEC 2000 benchmarks show improvements when applied to small level 1 instruction caches of size 2KB to 8KB. Both miss rate and number of interchanges between the main cache and the victim cache affect the average access time reducing either one could potentially improve the memory access time. Future work includes evaluating selective victim caching scheme for data caches and also improving the effectiveness of the prediction algorithm used to further improve the memory access time.

IX Downside of Selective Victim Caching

- The prediction algorithm requires the maintenance of state bits in the L1 and L2 caches, which may take a toll on the available memory for small caches.
- There is a marginal increase in complexity when compared to a pure victim cache to accommodate for the prediction algorithm.

X Shortcomings/ Further Improvements

- We were not able to produce improvements as high as was mentioned in our reference paper on selective victim caching.
- Implementing the same with the L1 data cache may show a higher improvement in miss rate/CPI.
- A common state bit array may be employed at the L1 cache as described by the paper to prevent memory reservation at the L2 cache.

XI Major Roadblocks

- We initially set about debugging the simplescalar source code simply using text editors which turned out to be very time consuming. We then tried out the same using the eclipse IDE, whose tools made it easier for us to understand the program flow.

- Figuring out the code was difficult because it was spread across many different files.

- Rather than making a lot of changes in the structure we thought using the available constructs would be easier. But this was also a challenge because we had to make changes in the structure without affecting other parts of the program that were using the same construct.

XII References

1. Dimitrios Stiliadis,Anujan Varma. Selective Victim Caching: A method to improve the performance of Direct Mapped caches. IEEE Transactions on Computers, Vol. 46, no. 5, may 1997.

2. Jouppi, N. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." Proceedings, 17th Annual International Symposium on Computer Architecture, May 1990.

XIII Appendix

This section includes code that we have written to implement selective victim caching in simplescalar.

Cache.h:

```
/* block status values */
#define CACHE_BLK_VALID          0x00000001     /* block in valid, in use */
#define CACHE_BLK_DIRTY          0x00000002     /* dirty block */
#define HIT_BIT                  0x01           /*hit bit*/
#define STICKY_BIT   0x02                       /*sticky*/
#define NOT_HIT_BIT   0xFE                      /*remove hit bit*/
#define NOT_STICKY_BIT 0xFD                     /*remove sticky bit*/

/* 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 */
  /* 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 */
  byte_t state;             // Stores state bits
};
```

Cache.c:

// Function to access an L1 cache with a victim cache

```
unsigned int                            /* latency of access in cycles */
cache_access_victim(struct cache_t *cp,         /* cache to access */
            enum mem_cmd cmd,                   /* access type, Read or Write */
            md_addr_t addr,             /* address of access */
            void *vp,                   /* ptr to buffer for input/output */
            int nbytes,             /* number of bytes to access */
            tick_t now,             /* time of access */
            byte_t **udata,             /* for return of user data ptr */
            md_addr_t *repl_addr,       /* for address of replaced block */
            struct cache_t *vc,
            byte_t (*state_access_fn) (enum mem_cmd cmd,
                                       md_addr_t baddr, int bsize,
                                       struct cache_blk_t *blk,
```

```
                                        tick_t now))    /* for address of replaced block */

{
  byte_t *p = vp;
  md_addr_t tag = CACHE_TAG(cp, addr);
  md_addr_t set = CACHE_SET(cp, addr);
  md_addr_t bofs = CACHE_BLK(cp, addr);
  md_addr_t addr_repl;
  md_addr_t tag_repl_vc;
  md_addr_t set_repl_vc;
  byte_t state;

  int lat_victim;
  struct cache_blk_t *blk, *repl;
  int lat = 0;

  /* check for a fast hit: access to same block */
  if (CACHE_TAGSET(cp, addr) == cp->last_tagset)
    {
      /* hit in the same block */
      blk = cp->last_blk;
      cp->last_blk->state = 0x03;
      goto cache_fast_hit;
    }

      /* low-associativity cache, linear search the way list */
      for (blk=cp->sets[set].way_head;
               blk;
               blk=blk->way_next)
            {
             if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
               {
                 blk->state = 0x03;
                        goto cache_hit;
               }
            }


  lat_victim = victim_cache_access(vc, Read, addr,
                                   NULL, nbytes, now,
                                   NULL, NULL);



  switch (cp->policy) {
    case LRU:
    case FIFO:
      repl = cp->sets[set].way_tail;
      update_way_list(&cp->sets[set], repl, Head);
      break;
    case Random:
      {
        int bindex = myrand() & (cp->assoc - 1);
        repl = CACHE_BINDEX(cp, cp->sets[set].blks, bindex);
      }
```

```
    break;
  default:
    panic("bogus replacement policy");
  }

  addr_repl = CACHE_MK_BADDR(cp, repl->tag, set);

  tag_repl_vc = CACHE_TAG(vc, addr_repl);
  set_repl_vc = CACHE_SET(vc, addr_repl);

  //state=state_access_fn(Read,CACHE_BADDR(cp, addr),cp->bsize,repl, now+lat);

if (lat_victim == 1)
{
        //vc->last_blk->tag = tag_repl_vc;
        //vc->last_blk->status = repl->status;
        cp->hits++;
        lat+=cp->hit_latency;
        if (!(repl->state & STICKY_BIT))
        {
                if (cp->hsize)
           unlink_htab_ent(cp, &cp->sets[set], repl);
                /d* blow away the last block to hit */
                cp->last_tagset = 0;
                cp->last_blk = NULL;

                vc->last_blk->tag =  tag_repl_vc;
                vc->last_blk->status = repl->status;
                vc->last_blk->state = repl->state;
                repl->state = 0x03;
                //repl->state = 0x02;
                repl->tag = tag;
                repl->status = CACHE_BLK_VALID;
               /* remove this block from the hash bucket chain, if hash exists */
        }
        else
        {
                if ( !(vc->last_blk->state & HIT_BIT))
                        repl->state &= NOT_STICKY_BIT;
                else
                {
                        vc->last_blk->tag =  tag_repl_vc;
                        vc->last_blk->status = repl->status;
                        vc->last_blk->state = repl->state;
                        repl->state = 0x02;
                        repl->tag = tag;
                        repl->status = CACHE_BLK_VALID;
                }
        }
}
else
{
        if ( !(repl->state & STICKY_BIT))
        {
                vc->sets[set_repl_vc].way_head->tag = tag_repl_vc;
```

```
                vc->sets[set_repl_vc].way_head->status = repl->status;
                vc->sets[set_repl_vc].way_head->state = repl->state;
                if (repl->status & CACHE_BLK_VALID)
                     cp->replacements++;
                repl->tag = tag;
                repl->status = CACHE_BLK_VALID;
                repl->state = 0x03;
                //repl->state = 0x02;
        }
        else
        {
                state=state_access_fn(Read,CACHE_BADDR(cp, addr),cp->bsize,repl, now+lat);
                if (!(state & HIT_BIT))
                {
                        addr_repl = CACHE_MK_BADDR(cp,tag,set);
                        tag_repl_vc = CACHE_TAG(vc, addr_repl);
                        set_repl_vc = CACHE_SET(vc, addr_repl);
                        if (vc->sets[set_repl_vc].way_head->status & CACHE_BLK_VALID)
                             cp->replacements++;
                           vc->sets[set_repl_vc].way_head->tag = tag_repl_vc;
                           vc->sets[set_repl_vc].way_head->status = CACHE_BLK_VALID;
                           repl->state &= NOT_STICKY_BIT;
                }
                else
                {
                        vc->sets[set_repl_vc].way_head->tag = tag_repl_vc;
                        vc->sets[set_repl_vc].way_head->status = repl->status;
                        vc->sets[set_repl_vc].way_head->state = repl->state;
                        if (repl->status & CACHE_BLK_VALID)
                                        cp->replacements++;
                        repl->tag = tag;
                        repl->status = CACHE_BLK_VALID;
                        repl->state = 0x02;
                }
        }
        /* cache block not found */

/* **MISS** */
cp->misses++;


/* read data block */
lat += cp->blk_access_fn(Read, CACHE_BADDR(cp, addr), cp->bsize,
                         repl, now+lat);

}

/* update block tags */
//repl->tag = tag;
//repl->status = CACHE_BLK_VALID;           /* dirty bit set on update */

/* update dirty status */
if (cmd == Write)
  repl->status |= CACHE_BLK_DIRTY;
```

```
 /* get user block data, if requested and it exists */
 if (udata)
   *udata = repl->user_data;

 /* update block status */
 repl->ready = now+lat;

 /* link this entry back into the hash table */
 if (cp->hsize)
   link_htab_ent(cp, &cp->sets[set], repl);

 /* return latency of the operation */
 return lat;


cache_hit: /* slow hit handler */

 /* **HIT** */
 cp->hits++;

 /* update dirty status */
 if (cmd == Write)
   blk->status |= CACHE_BLK_DIRTY;

 /* if LRU replacement and this is not the first element of list, reorder */
 if (blk->way_prev && cp->policy == LRU)
   {
     /* move this block to head of the way (MRU) list */
     update_way_list(&cp->sets[set], blk, Head);
   }

 /* tag is unchanged, so hash links (if they exist) are still valid */

 /* record the last block to hit */
 cp->last_tagset = CACHE_TAGSET(cp, addr);
 cp->last_blk = blk;

 /* return first cycle data is available to access */
 return (int) MAX(cp->hit_latency, (blk->ready - now));

cache_fast_hit: /* fast hit handler */

 /* **FAST HIT** */
 cp->hits++;

 /* copy data out of cache block, if block exists */
 if (cp->balloc)
   {
     CACHE_BCOPY(cmd, blk, bofs, p, nbytes);
   }

 /* update dirty status */
 if (cmd == Write)
   blk->status |= CACHE_BLK_DIRTY;
```

```
  /* this block hit last, no change in the way list */

  /* tag is unchanged, so hash links (if they exist) are still valid */

  /* get user block data, if requested and it exists */
  if (udata)
    *udata = blk->user_data;

  /* record the last block to hit */
  cp->last_tagset = CACHE_TAGSET(cp, addr);
  cp->last_blk = blk;

  /* return first cycle data is available to access */
  return (int) MAX(cp->hit_latency, (blk->ready - now));
}
```

// cache access for victim cache

```
unsigned int                            /* latency of access in cycles */
victim_cache_access(struct cache_t *cp,         /* cache to access */
                enum mem_cmd cmd,               /* access type, Read or Write */
                md_addr_t addr,                 /* address of access */
                void *vp,                       /* ptr to buffer for input/output */
                int nbytes,             /* number of bytes to access */
                tick_t now,             /* time of access */
                byte_t **udata,                 /* for return of user data ptr */
                md_addr_t *repl_addr )          /* for address of replaced block */


{
  byte_t *p = vp;
  md_addr_t tag = CACHE_TAG(cp, addr);
  md_addr_t set = CACHE_SET(cp, addr);
  md_addr_t bofs = CACHE_BLK(cp, addr);
  struct cache_blk_t *blk, *repl;
  int lat = 1;
```

// Skipping code for checking for cache hit

```
  /* cache block not found */

  /* **MISS** */
  cp->misses++;

  lat+=1;

  /* select the appropriate block to replace, and re-link this entry to
     the appropriate place in the way list */
  switch (cp->policy) {
  case LRU:
  case FIFO:
    repl = cp->sets[set].way_tail;
```

```
  update_way_list(&cp->sets[set], repl, Head);
  break;
 case Random:
  {
   int bindex = myrand() & (cp->assoc - 1);
   repl = CACHE_BINDEX(cp, cp->sets[set].blks, bindex);
  }
  break;
 default:
  panic("bogus replacement policy");
 }

 /* remove this block from the hash bucket chain, if hash exists */
 if (cp->hsize)
  unlink_htab_ent(cp, &cp->sets[set], repl);

 /* blow away the last block to hit */
 cp->last_tagset = 0;
 cp->last_blk = NULL;

 /* write back replaced block data */
 if (repl->status & CACHE_BLK_VALID)
  {
   cp->replacements++;

   if (repl_addr)
        *repl_addr = CACHE_MK_BADDR(cp, repl->tag, set);

   /* track bus resource usage */
   cp->bus_free = MAX(cp->bus_free, (now + lat)) + 1;

          lat += cp->blk_access_fn(Write,
                                   CACHE_MK_BADDR(cp, repl->tag, set),
                                   cp->bsize, repl, now+lat);
        }
```

// Skipping code for hit handling..same as normal cache

```
  /* return latency of the operation */
  return lat;


}

// Function to fetch state bits of incoming block

byte_t                          /* latency of access in cycles */
get_state(struct cache_t *cp,   /* cache to access */
          enum mem_cmd cmd,                 /* access type, Read or Write */
          md_addr_t addr,               /* address of access */
          void *vp,                     /* ptr to buffer for input/output */
          int nbytes,           /* number of bytes to access */
          tick_t now,           /* time of access */
          byte_t **udata,               /* for return of user data ptr */
```

```
                md_addr_t *repl_addr)          /* for address of replaced block */

{
  byte_t state = 0x00;
  byte_t *p = vp;
  md_addr_t tag = CACHE_TAG(cp, addr);
  md_addr_t set = CACHE_SET(cp, addr);
  md_addr_t bofs = CACHE_BLK(cp, addr);
  struct cache_blk_t *blk, *repl;
  //int lat = 0;

  /* default replacement address */
  if (repl_addr)
    *repl_addr = 0;

  /* check alignments */
  if ((nbytes & (nbytes-1)) != 0 || (addr & (nbytes-1)) != 0)
    fatal("cache: access error: bad size or alignment, addr 0x%08x", addr);

  /* access must fit in cache block */
  /* FIXME:
     ((addr + (nbytes - 1)) > ((addr & ~cp->blk_mask) + (cp->bsize - 1))) */
  if ((addr + nbytes) > ((addr & ~cp->blk_mask) + cp->bsize))
    fatal("cache: access error: access spans block, addr 0x%08x", addr);

  /* permissions are checked on cache misses */

  /* check for a fast hit: access to same block */
  if (CACHE_TAGSET(cp, addr) == cp->last_tagset)
    {
      /* hit in the same block */
      blk = cp->last_blk;
      state = cp->last_blk->state;
      return state;
    }

  if (cp->hsize)
    {
      /* higly-associativity cache, access through the per-set hash tables */
      int hindex = CACHE_HASH(cp, tag);

      for (blk=cp->sets[set].hash[hindex];
               blk;
               blk=blk->hash_next)
            {
              if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                        {
                                  state = blk->state;
                                  return state;
                        }
            }
    }
  else
    {
      /* low-associativity cache, linear search the way list */
```

```
    for (blk=cp->sets[set].way_head;
         blk;
         blk=blk->way_next)
        {
         if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                {
                        state = blk->state;
                        return state;
                }
        }
   }
  return state;

}

// Function to writeback state bits on eviction

void                                    /* latency of access in cycles */
write_state(struct cache_t *cp,  /* cache to access */
            enum mem_cmd cmd,                   /* access type, Read or Write */
            md_addr_t addr,                 /* address of access */
            int nbytes,             /* number of bytes to access */
            byte_t state)  /* for address of replaced block */

{
  md_addr_t tag = CACHE_TAG(cp, addr);
  md_addr_t set = CACHE_SET(cp, addr);
  md_addr_t bofs = CACHE_BLK(cp, addr);
  struct cache_blk_t *blk, *repl;


  /* check alignments */
  if ((nbytes & (nbytes-1)) != 0 || (addr & (nbytes-1)) != 0)
    fatal("cache: access error: bad size or alignment, addr 0x%08x", addr);

  /* access must fit in cache block */
  /* FIXME:
     ((addr + (nbytes - 1)) > ((addr & ~cp->blk_mask) + (cp->bsize - 1))) */
  if ((addr + nbytes) > ((addr & ~cp->blk_mask) + cp->bsize))
    fatal("cache: access error: access spans block, addr 0x%08x", addr);

  /* permissions are checked on cache misses */

  /* check for a fast hit: access to same block */
  if (CACHE_TAGSET(cp, addr) == cp->last_tagset)
    {
      /* hit in the same block */
      cp->last_blk->state = state;
      goto end;
    }

  if (cp->hsize)
    {
      /* higly-associativity cache, access through the per-set hash tables */
      int hindex = CACHE_HASH(cp, tag);
```

```
    for (blk=cp->sets[set].hash[hindex];
            blk;
            blk=blk->hash_next)
          {
           if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                  {
                        blk->state = state;
                        break;
                  }
                  //goto cache_hit;
          }
   }
  else
   {
    /* low-associativity cache, linear search the way list */
    for (blk=cp->sets[set].way_head;
            blk;
            blk=blk->way_next)
          {
           if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                  {
                        blk->state = state;
                        break;
                  }
          }
   }
  end:;
}
```

Sim-outorder.c:

```
// fn to get state on il1 miss

byte_t
il1_fetch_state(enum mem_cmd cmd,               /* access cmd, Read or Write */
                md_addr_t baddr,                /* block address to access */
                int bsize,                      /* size of block to access */
                struct cache_blk_t *blk,        /* ptr to block in upper level */
                tick_t now)                     /* time of access */
{
  byte_t state;


if (cache_il2)
    {
     /* access next level of inst cache hierarchy */
     state = get_state(cache_il2, cmd, baddr, NULL, bsize,
                               /* now */now, /* pudata */NULL, /* repl addr */NULL);
     if (cmd == Read)
           return state;
     else
           panic("use il1_write_state fn");
    }
```

```
  else
    {
            state = 0x00;
     /* access main memory */
     if (cmd == Read)
          return state;
     else
          panic("use il1_write_state fn");
    }
}

// fn to write back il1 block state bits

void
il1_write_state(enum mem_cmd cmd,               /* access cmd, Read or Write */
                md_addr_t baddr,                /* block address to access */
                int bsize,                /* size of block to access */
                struct cache_blk_t *blk,        /* ptr to block in upper level */
                byte_t state)             /* time of access */
{

if (cache_il2)
   {
           if (cmd == Read)
           panic("use il1_fetch_state fn");

          /* access next level of inst cache hierarchy */
     write_state(cache_il2, cmd, baddr, bsize,state);

          }
}


static unsigned int                             /* latency of block access */
victim_cache_access_fn(enum mem_cmd cmd,              /* access cmd, Read or Write */
                md_addr_t baddr,                /* block address to access */
                int bsize,                /* size of block to access */
                struct cache_blk_t *blk,        /* ptr to block in upper level */
                tick_t now)             /* time of access */
{
  unsigned int lat;
  lat = 2;
  if (cmd == Write)
  il1_write_state(Write,baddr,
                                        bsize,blk,blk->state);
 // if (cmd == Read)
          return lat;
 // else
  //        panic("writes to instruction memory not supported");
}
```