Cache Replacement Policy Using Frequency Information

Yang Wang (321004893)
{wyang1989}@tamu.edu

ABSTRACT

The widely used LRU replacement policy has the following problems. First, LRU does not use frequency information of cache access. This makes LRU easy to be polluted by infrequently used blocks, which evict frequently used blocks from LRU stack. Second, LRU may work poorly when the there is a lot of cycle pattern cache access and cache capacity is less than the working set.

However efficient last-level cache utilization is crucial to avoid long latency cache misses to main memory. Always predicting a near-immediate re-reference interval on all cache insertions limits cache performance on the other hand always predicting a distant re-reference interval significantly decreases cache performance for access pattern that predominately have a near-immediate re-reference interval.

For this matter, a cache replacement based on Re-reference Interval Prediction (RRIP) may be used to solve the problems. This cache replacement policy is called Re-Reference Interval Prediction replacement policy. And in this project, I apply this idea to L2 cache, which is the last level cache of my project. The performance and sensitivities of this replacement policy is analyzed.

1. INTRODUCTION

An ideal replacement policy should make its replacement decision using the pattern of each block and replace the block that will be re-referenced furthest in the future. However there is no perfect replacement policy. Current replacement policies make decision on certain kind of assumption of which block will be re-referenced furthest in the future. Generally when misses happened replacement policies predicts the re-referenced possibility of this block in future.

The commonly used LRU (Least Recently Used) replacement policy record the pattern of recent referenced blocks. When a block needs to be replaced, LRU will evict the block at the tail of LRU chain and put new block at the head of that chain. The block at the head of the LRU chain is predicted to be near-immediate re-reference, which means it is

assumed to be re-referenced again in soonest and the block at the tail implies that block might be re-reference in the distant future, which means it might be re-referenced again in longest future. So when a new block is inserted into LRU chain, LRU predicts it will be re-referenced in immediate future and put it at the head of the chain. LRU works well on workloads with high locality, but it cannot make right prediction when re-references mostly occurs in distant future. Benchmark usually performs poorly under LRU when its reference occurs in the distant future. The limit of LRU is cause by following two aspects.

First, LRU only records the present of blocks without frequency information. LRU assumes that the most recently used blocks will have the highest probability to be used again in the near future. This makes it is easy to be polluted by some infrequently used blocks. Some blocks might be used only few times, but they will replace the blocks with distant references, which might be used a lot. Considering the following access pattern: A, B, B, A, C, D, E, F, A, B will be put in same set with 4 blocks. Even though 'A' and 'B' appears 2 times, but they will be evicted by less referenced block 'E' and 'F'.

Second, LRU may suffer from cache thrashing for cyclic access patterns when the working set is greater than the available cache size because this kind of pattern will traverse LRU chain. For this matter, this project explores a different kind of replacement policy, which record the frequency information of each block as well. When a block needs to be replaced, this policy will choose the block occurs least up to now and replace it. This replacement policy is based on Re-reference Interval Prediction (RRIP).

RRIP prevents cache blocks with distant re-reference interval from evicting blocks that have near-immediate re-reference interval. This is implemented by using M-bit

information per block to store its Re-reference Prediction Value (RRPV). This project use RRIP method to make comparison between RRIP based replacement policy and LRU policy and also explore the sensitivity of RRIP to different parameters.

2. MOTIVIATION

Reducing the miss rate of last-level cache is important since it can avoid long latency cache misses to main memory. However commonly used LRU replacement policy does not perform well on workloads with many distant reference blocks. To better understand when LRU performs poorly, we should understand the feature of different kind of pattern.

(a) $(a_1, a_2, \ldots, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N$ LRU-friendly Access Pattern

(b) $(a_1, a_2, \ldots, a_k)^N$ Thrashing Access Pattern

(c) $(a_1, a_2 \ldots a_k \ldots)$ Infinite Access Pattern

Figure 1: Cache Access Pattern

Fig1(a) shows a LRU friendly access pattern. This access has a near-immediate re-reference interval. For any value of k, the access pattern can be well predicted by LRU policy. Fig1(b) shows a thrashing access pattern. It is a cyclic access and when N is large enough LRU will make 0 cache hits due to cache thrashing. For example like the following access way: "A, B, C, D, E, F, A, B, C, D, E, F, A, B, C, D, E, F" There is a cyclic access "A, B, C, D, E, F". If cache set has 3 blocks, then the access will like this:

Next Ref	**Cache Set**			
A				Miss
B	A			Miss
C	B	A		Miss
D	C	B	A	Miss

E	D	C	B	Miss
F	E	D	C	Miss
A	F	E	D	Miss
B	A	F	E	Miss
...	...	...	...	Miss...

Fig1(c) presents a streaming access pattern, when k is unlimited large, the access pattern has no locality. This access pattern will receive no cache hits.

So we can regards any workload as a mix access pattern of above 3 kind of access way. Then if the 2nd access pattern is dominant, the LRU will perform poorly. But if the LRU friendly access pattern is dominating, LRU should work well. However, RRIP will works well for both 1st and 2nd access patter, so there is improvement space for LRU.

3. Re-Reference Interval Prediction (RRIP)

Both recent and frequency information are important to cache replacement, while LRU only used the recent information. When benchmark is a mix access pattern, LRU cannot distinguish between blocks that have a distant re-reference interval from blocks that have a near-immediate re-reference interval.

Always predicting a near-immediate re-reference interval on all cache insertions limits the cache performance. Since blocks will occupy the cache space without receiving any hits. On the other hand, always predicting a distant-immediate re-reference interval also significantly decreases the cache performance.

RRIP uses M-bits per block information to record 2^M possibility Re-reference Prediction Values (RRPV). RRIP will dynamically learn re-reference information for each block in the cache access pattern. When RRPV=0, it means this block will be re-reference in the

near-immediate future. While when RRPV equals $2^M - 1$ means this block will be re-reference in the furthest future. So all the value between 0 to $2^M - 1$ indicated the different re-reference possibility from near-immediate to furthest re-reference. For instance, with 2 bits information (RRPV range from 0 to 3), 0 predicts the cache block will be reference in the soonest future and 3 predicts the cache block will be reference in longest future.

Then higher M value means more bits to represent the pattern of each block. And it can discriminate more different patterns. However, the performance of this replacement policy does not always go up with higher M value. I will explore the sensitivity to M value in later.

On initial all blocks is set to highest RRPV value. When a new block is inserted into cache, RRIP reduce the value of this block by 1. And when this block is re-referenced, RRPV is set directly to 0 or reduced by 1 again. These two different choices will be analyzed in later. In this way, RRIP learns the block's re-reference pattern. When a block needs to be replaced, RRIP always chose the first block with highest prediction value. If there is no highest value, then increasing all the value of each block by 1 and repeat till one block reaches the highest value.

The following is an example for the decision procedure with 2-bit RRPV information. So when the value is 0, it means the block will be re-referenced in near-immediate and value 3 represents that block might be re-referenced in long time later.

The decision process (With 2 bit RRPV information):
(1) Cache Hit:
Set RRPV of block to '0'
(2) Cache Miss:
1. Search for first block with '3' from left
2. If '3' found go to step 5.
3. Increase all RRPV by 1.
4. Go to step 1.

5. Replace block and decrease RRPV by 1.

One thing should be mentation is how we set the RRPV when there is a hit. Generally there are two different ways. One is set RRPV to 0 once there is a hit; the other way is reducing RRPV by 1 till 0 on each hit.

Another thing is how we set the RRPV when there is a hit. There are 3 choices. One is set it to 0 for the new block brought into, which means we predict this block will be re-referenced in nearest future. Or set it to $2^M - 1$, which means we predict this block will be re-referenced in longest future. Or set it to $2^M - 2$, which means we predicts this block might be re-referenced in long future not nearest nor longest. Also I will explore this sensitivity in later chapter.

4. Methodology

4.1 Simulator

I use SimpleScalar for this project to do the simulation. I use 2 level memory hierarchy. Level 1 cache is a 4-way 32KB with 64B block size. L1 cache sizes are kept constant in this study.

4.2 Benchmarks

I use 2 float point benchmark equake and art and 2 integer benchmark gcc and gzip.

4.3 Project Content

The following chapter will explore different static RRIP sensitivity. First we will explore the sensitivity to RRPV on insertion and decide which strategy we should use to set the RRPV when we bring in new block. Second we will decide which strategy we should use to set the RRPV when there is a hit. Third exploring the sensitivity to M

value. Forth exploring the sensitivity to L2 cache configuration. And finally probe the sensitivity to associativity.

4.4 Implementation

This replacement policy is implemented by revising the cache.c, cache.h and simoutorder.c file. The main idea is to add re-reference prediction value (RRPV) to each block, that is the cache_blk_t structure in cache.h. Then it is initialized to $2^M - 1$ at first. And in cache.c, the code will handle the miss and hit cases.

5. Results and Analysis

5.1 RRIP sensitivity to RRPV on insertion

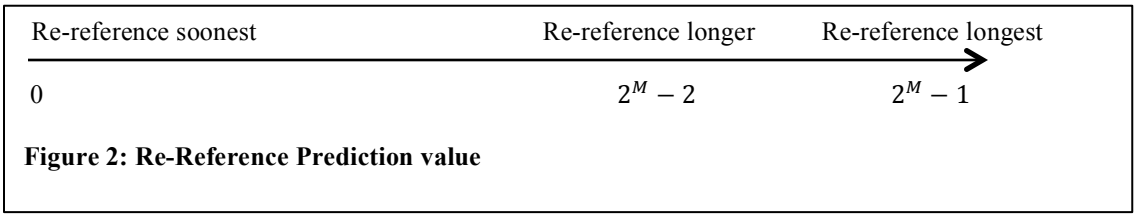

Figure 2: Re-Reference Prediction value

In this section, I first investigate the performance of this replacement policy by changing the re-reference prediction value, which is the value when we first insert a block into cache set. As mentioned above, there are 3 different choices. Set 0 (Predicting immediate), Set $2^M - 1$ (Predicting distant re-reference) and Set $2^M - 2$ (Predicting long re-reference). We can see from Fig 2, if this value is set to 0 that means predicting this newly inserted block will be re-referenced immediately, which is what LRU does. If re-reference prediction value is set to $2^M - 1$ that means predicting this newly inserted block will not be re-referenced in near future. And if this value is set to $2^M - 2$ that means predicting the re-reference possibility of this newly inserted is neither the soonest nor the longest.

The configuration for L2 cache will be 16-way associativity and 64B block size for the rest if this project without explicitly mention. The cache size changes among 1MB, 2MB and 4MB. And M=2, that means there are 4 bits to record frequency information. Value 0 stands for immediate re-reference, 2 means long re-reference and 3 means distant (longest) re-reference.

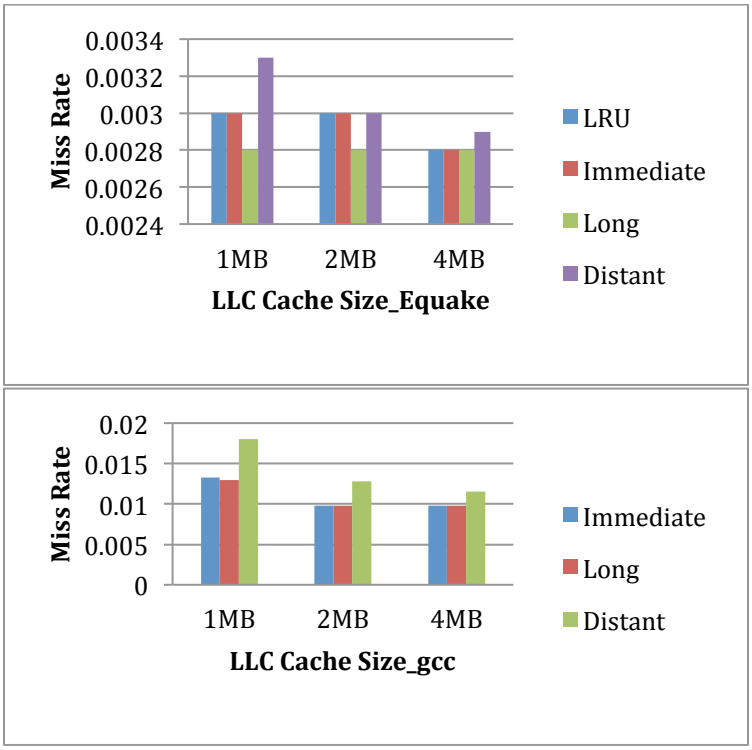

Figure 3: Sensitivity to re-reference prediction value.

As we can see from Fig 3, always predicting long re-reference, that is, setting RRPV to $2^M - 2$ (in this experiment it is 2) constantly yield the best performance. And always predicting the distant re-reference gives the worst performance, that is, setting PPPV to $2^M - 1$ when we insert new block into cache. This feature can be explained by the replacing strategy. RRIP will replace a block with value $2^M - 1$ (in current experiment it is 3) when a block must be replaced. So if we set the new

block value to $2^M - 1$ when we insert it to cache. There will not be enough time to reduce the RRPV. And it might be replaced immediately by the RRIP strategy. And predicting the block to be long re-referenced will give RRIP enough time to learn the block pattern and do not pollute cache.

So in this project we will always use long RRIP insertion configuration for rest analysis.

5.2 Sensitivity to Hit Priority And Frequency Priority

Next Reference	RRPV of certain block	Action
a1	2	Miss
a1	0	Hit
a1	0	Hit
...	...	...

Hit Priority: (Using M=2, so there are 4 different RRPV value. A newly inserted block RRPV is always set to 2 $(2^M - 2)$. When hit happen, RRPV is set to 0 directly)

Next Reference	RRPV of certain block	Action
a1	2	Miss
a1	1	Hit
a1	0	Hit
...	...	...

Frequency Priority: (Using M=2, so there are 4 different RRPV value. A newly inserted block RRPV is always set to 2 $(2^M - 2)$ When hit happen, RRPV is reduced by one each time.)

In last section, what choice should be made when new block is inserted into the cache set is decided. So in this section, the action of setting RRPV is discussed when

a hit happened. There are 2 different way to handle the hit situation. As shown in above table, one is to set re-referenced predicting value directly to 0 when there is a hit called hit priority. Another is to reduce that value by one till zero when there is a hit called frequency priority. So the frequency priority will record more information about the frequency pattern of each block than the hit priority. In summary hit priority predicts a hit block will be re-referenced in near-immediate re-reference and frequency priority predicts not all hit block have a near-immediate re-reference but only the frequently hit block have near-immediate re-reference.

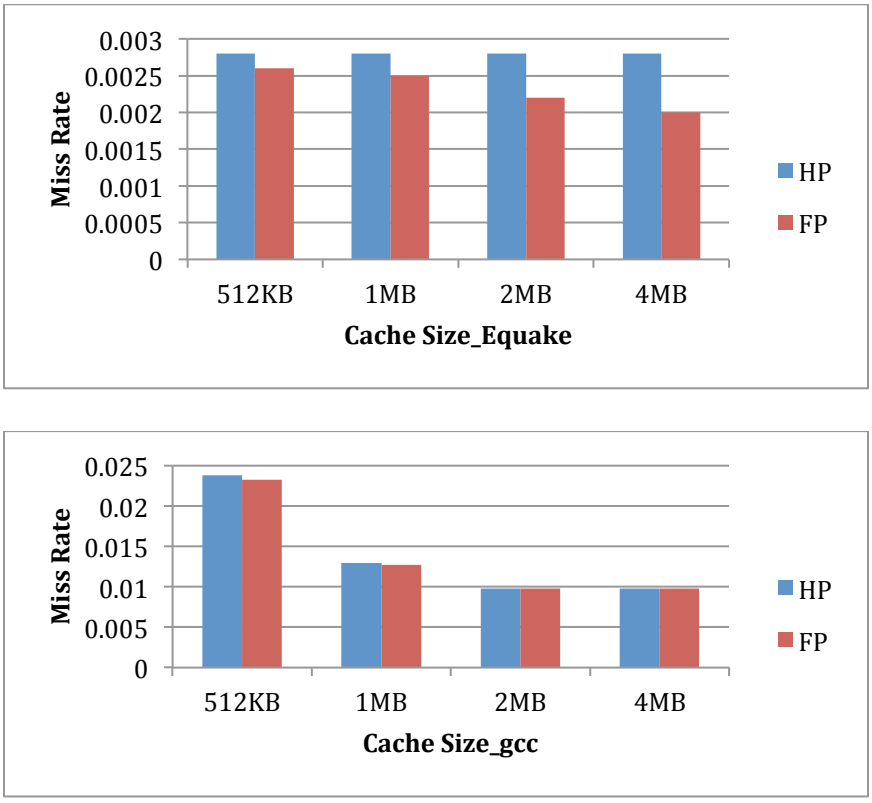

Figure 4: Sensitivity to hit priority and frequent priority

Fig 4 shows the results of RRIP performance on HP and FP under different L2 cache size. Both float point and integer benchmark works better on frequency

priority. The reason for this kind of pattern shows the benefits of this algorithm is from recording the frequently referenced block. Allowing more time to learn whether a block is really near-immediate re-reference is important. So for the following project, the FP will always be used for simulation.

5.3 Sensitivity to M value

In each block we use $2^M\ bits$ value to record the reference frequency, which predicts the re-reference possibility of each block. Value 0 means the highest possibility of immediate re-reference and value $2^M - 1$ means the lowest so that block with value $2^M - 1$ will always be chosen to replace by RRIP because it is predicted not to be reference again in near future. For this matter, with M value increase, there will be more bits to record the frequency information, making a frequently referenced block less likely to be polluted.

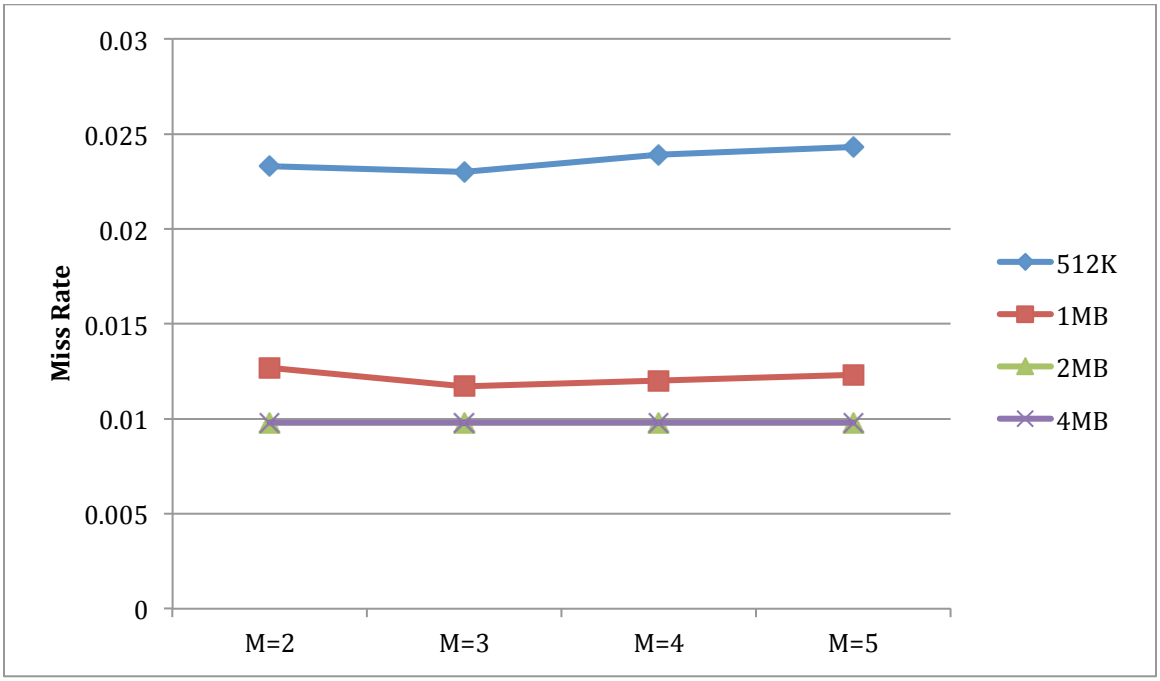

Figure 5: M-value sensitivity study

Above figure indicates that this replacement policy does not perform better with higher M value. That means even though there is more bits to record the frequency information, it does not help reducing miss rate. And in this project, it shows that when M=3, the RRIP performance best. When the cache size is more than 2MB, miss rate does not change with different M value. This feature might caused by the compulsory miss rate, which limits the performance of RRIP. Then M value should be decided for different cache configuration and different benchmark for best performance. So for the rest of this project, I will choose M=3 for RRIP configuration.

5.4 Sensitivity to Cache Configuration

Given the results and anaysis of above three section, I have probe the sensitivity of this replacement policy to its own configuration. Now it is time to analyze the performance by varying cache size.

In this section, I explore the sensitivity of RRIP on different cache size: 256K, 512K, 1MB, 2MB. All the cahce is 16-way associativity with 64B block size. And the performance imporvement is compared to LRU. Two integer benchmark is used. And one float point benchmark is used. The configuration for RRIP is 3 bits RRPV information (M=3) and frequency priority with long re-reference predicting on each insertion.

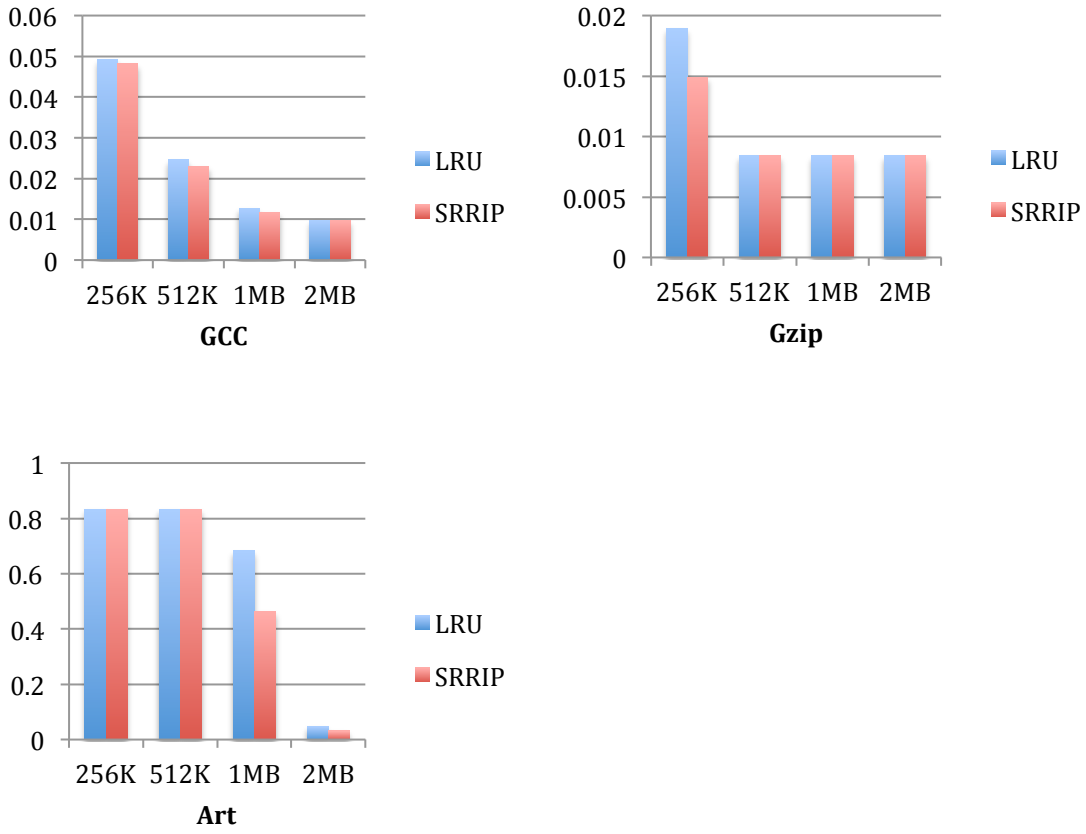

Figure 6: Performance analysis by varying L2 cache size.

The above graph shows the performance difference between LRU and RRIP among different L2 cache size. First we can see that RRIP replacement policy outperform LRU in some cases. And for other cases, it is at least as well as LRU. Also it shows that for integer benchmark RRIP performance better for small cache size and for float point benchmark RRIP performance better for larger cache size. So the performance of RRIP is also dependent on different benchmark. If all the data in that benchmark is LRU friendly, then the advantage of RRIP cannot be revealed.

In this study, RRIP works constantly better on all benchmark and work best on benchmark art with 1MB L2 cache. This can be explained by there might be a lot scan-

pattern on benchmark art. Also this study shows the advantage of RRIP might not be seen for small cache size.

5.5 Sensitivity to Associativity

At last section, I use following parameters for study the sensitivity of RRIP to associativity. L2 cache is 1MB with 64B block size and the associativity is: 4way, 8way, 16way and 32way.

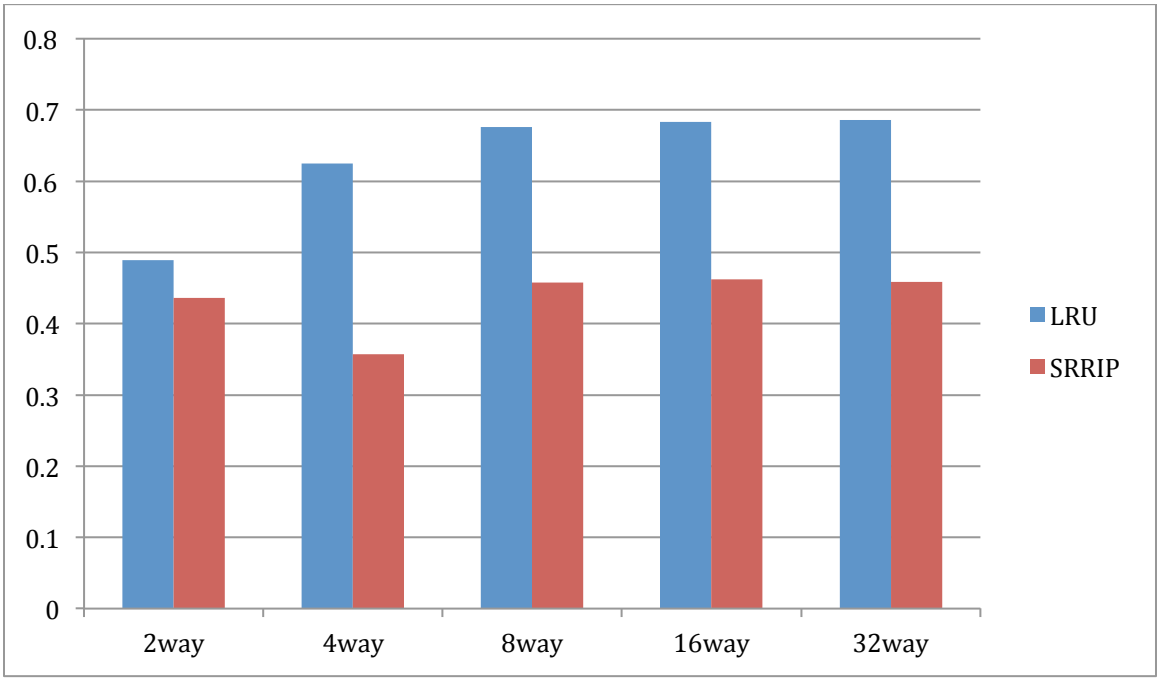

Figure 7: Performance analysis under different associativity.

As it indicates in the figure, RRIP outperform LRU about 30-40%. The best performance happens on 4-way associativity for 1MB L2 cache. For 2-way associativity, the advantage of RRIP cannot be fully revealed since there are only 2 blocks in each set. So in any way one of the two blocks will be replaced when a new block must be inserted into current set. However the reason for high associativity also does not yield the best performance has nothing to do with RRIP.

6. SUMMARY

Cache replacement policies try to replace the block with least possibility of immediate re-reference. The commonly used LRU replacement policy always predicts a newly inserted block with immediate re-reference, that means always predicting a block with the highest possibility of re-reference soonest in the future. However, in fact it is not always true. Some frequently accessed block may be eliminated by block with distant re-reference interval. So in this project, I conduct a experiment by adding frequency information into each block. When a block should be replaced, this re-reference interval prediction replacement policy will always choose the block with lowest possibility of sooner re-reference again in the future. In this way, the frequently reference blocks are prevent from polluting by less frequently reference blocks. The sensitivity of this replacement policy to different configuration is explored. The performance is also compared to LRU and shows this policy can outperform in different configuration.

7. ACKNOWLEDGEMENTS

The core idea and the decision procedure of this Re-Reference Interval Prediction replacement policy is from Aamer Jaleel's paper.

9. REFERENCES

[1] Aamer Jaleel. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *ISCA'10*, June 19-23, 2010.

Appendix A: Simulation Code

cache.h

```
/* cache.h - cache module interfaces */

#ifndef CACHE_H
#define CACHE_H

#include <stdio.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "memory.h"
#include "stats.h"


/* highly associative caches are implemented using a hash table lookup to
   speed block access, this macro decides if a cache is "highly associative" */
#define CACHE_HIGHLY_ASSOC(cp)       ((cp)->assoc > 4)

/* cache replacement policy */
enum cache_policy {
  LRU,                          /* replace least recently used block (perfect LRU) */
  Random,       /* replace a random block */
  FIFO,                         /* replace the oldest block in the set */
  /*******************************************************/
  SRRIP    /*Re-Reference Interval Prediction*/

  /*******************************************************/
};

/* block status values */
#define CACHE_BLK_VALID                                 0x00000001    /* block in valid, in use */
#define CACHE_BLK_DIRTY                                 0x00000002    /* dirty block */

/* cache block (or line) definition */
struct cache_blk_t
{
  //*************************
  // Add RRPV to cache block
  //*************************
  int rrpv;

  //****************************

  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 */
};

/* cache set definition (one or more blocks sharing the same set index) */
struct cache_set_t
{
  struct cache_blk_t **hash;    /* hash table: for fast access w/assoc, NULL
                                                   for low-assoc caches */
  struct cache_blk_t *way_head;                 /* head of way list */
  struct cache_blk_t *way_tail;                 /* tail pf way list */
  struct cache_blk_t *blks;     /* cache blocks, allocated sequentially, so
                                                   this pointer can also be used for random
                                                   access to cache blocks */
};

/* cache definition */
struct cache_t
{
  /* parameters */
  char *name;                                   /* cache name */
  int nsets;                                    /* number of sets */
  int bsize;                                    /* block size in bytes */
  int balloc;                                   /* maintain cache contents? */
  int usize;                                    /* user allocated data size */
  int assoc;                                    /* cache associativity */
  enum cache_policy policy;   /* cache replacement policy */
```

```
  unsigned int hit_latency;	/* cache hit latency */

  /* miss/replacement handler, read/write BSIZE bytes starting at BADDR
     from/into cache block BLK, returns the latency of the operation
     if initiated at NOW, returned latencies indicate how long it takes
     for the cache access to continue (e.g., fill a write buffer), the
     miss/repl functions are required to track how this operation will
     effect the latency of later operations (e.g., write buffer fills),
     if !BALLOC, then just return the latency; BLK_ACCESS_FN is also
     responsible for generating any user data and incorporating the latency
     of that operation */
  unsigned int                                          /* latency of block access */
    (*blk_access_fn)(enum mem_cmd cmd,                  /* block access command */
                     md_addr_t baddr,                   /* program address to access */
                     int bsize,                         /* size of the cache block */
                     struct cache_blk_t *blk,           /* ptr to cache block struct */
                     tick_t now);                       /* when fetch was initiated */

  /* derived data, for fast decoding */
  int hsize;                                            /* cache set hash table size */
  md_addr_t blk_mask;
  int set_shift;
  md_addr_t set_mask;                                   /* use *after* shift */
  int tag_shift;
  md_addr_t tag_mask;                                   /* use *after* shift */
  md_addr_t tagset_mask;	/* used for fast hit detection */

  /* bus resource */
  tick_t bus_free;                                      /* time when bus to next level of cache is
                                                           free, NOTE: the bus model assumes only a
                                                           single, fully-pipelined port to the next
                                                           level of memory that requires the bus only
                                                           one cycle for cache line transfer (the
                                                           latency of the access to the lower level
                                                           may be more than one cycle, as specified
                                                           by the miss handler */

  /* per-cache stats */
  counter_t hits;                                       /* total number of hits */
  counter_t misses;                                     /* total number of misses */
  counter_t replacements;	/* total number of replacements at misses */
  counter_t writebacks;                                 /* total number of writebacks at misses */
  counter_t invalidations;	/* total number of external invalidations */

  /* last block to hit, used to optimize cache hit processing */
  md_addr_t last_tagset;	/* tag of last line accessed */
  struct cache_blk_t *last_blk;/* cache block last accessed */

  /* data blocks */
  byte_t *data;                                         /* pointer to data blocks allocation */

  /* NOTE: this is a variable-size tail array, this must be the LAST field
     defined in this structure! */
  struct cache_set_t sets[1];	/* each entry is a set */
};

/* create and initialize a general cache structure */
struct cache_t *                                        /* pointer to cache created */
cache_create(char *name,                                /* name of the cache */
             int nsets,                                 /* total number of sets in cache */
             int bsize,                                 /* block (line) size of cache */
             int balloc,                                /* allocate data space for blocks? */
             int usize,                                 /* size of user data to alloc w/blks */
             int assoc,                                 /* associativity of cache */
             enum cache_policy policy,/* replacement policy w/in sets */
             /* block access function, see description w/in struct cache def */
             unsigned int (*blk_access_fn)(enum mem_cmd cmd,
                                           md_addr_t baddr, int bsize,
                                           struct cache_blk_t *blk,
                                           tick_t now),
             unsigned int hit_latency);/* latency in cycles for a hit */

/* parse policy */
enum cache_policy                                       /* replacement policy enum */
cache_char2policy(char c);                              /* replacement policy as a char */

/* print cache configuration */
void
cache_config(struct cache_t *cp,                        /* cache instance */
             FILE *stream);                             /* output stream */

/* register cache stats */
void
cache_reg_stats(struct cache_t *cp,                     /* cache instance */
                struct stat_sdb_t *sdb);/* stats database */

/* print cache stats */
void
cache_stats(struct cache_t *cp,                         /* cache instance */
            FILE *stream);                              /* output stream */

/* print cache stats */
void cache_stats(struct cache_t *cp, FILE *stream);
```

```
/* access a cache, perform a CMD operation on cache CP at address ADDR,
   places NBYTES of data at *P, returns latency of operation if initiated
   at NOW, places pointer to block user data in *UDATA, *P is untouched if
   cache blocks are not allocated (!CP->BALLOC), UDATA should be NULL if no
   user data is attached to blocks */
unsigned int                                    /* latency of access in cycles */
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 */

/* cache access functions, these are safe, they check alignment and
   permissions */
#define cache_double(cp, cmd, addr, p, now, udata)      \
  cache_access(cp, cmd, addr, p, sizeof(double), now, udata)
#define cache_float(cp, cmd, addr, p, now, udata)       \
  cache_access(cp, cmd, addr, p, sizeof(float), now, udata)
#define cache_dword(cp, cmd, addr, p, now, udata)       \
  cache_access(cp, cmd, addr, p, sizeof(long long), now, udata)
#define cache_word(cp, cmd, addr, p, now, udata)        \
  cache_access(cp, cmd, addr, p, sizeof(int), now, udata)
#define cache_half(cp, cmd, addr, p, now, udata)        \
  cache_access(cp, cmd, addr, p, sizeof(short), now, udata)
#define cache_byte(cp, cmd, addr, p, now, udata)        \
  cache_access(cp, cmd, addr, p, sizeof(char), now, udata)

/* return non-zero if block containing address ADDR is contained in cache
   CP, this interface is used primarily for debugging and asserting cache
   invariants */
int                                             /* non-zero if access would hit */
cache_probe(struct cache_t *cp,                 /* cache instance to probe */
            md_addr_t addr);                    /* address of block to probe */

/* flush the entire cache, returns latency of the operation */
unsigned int                                    /* latency of the flush operation */
cache_flush(struct cache_t *cp,                 /* cache instance to flush */
            tick_t now);                        /* time of cache flush */

/* flush the block containing ADDR from the cache CP, returns the latency of
   the block flush operation */
unsigned int                                    /* latency of flush operation */
cache_flush_addr(struct cache_t *cp,            /* cache instance to flush */
                 md_addr_t addr,                /* address of block to flush */
                 tick_t now);                   /* time of cache flush */

#endif /* CACHE_H */
```

cache.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "host.h"
#include "misc.h"
#include "machine.h"
#include "cache.h"

/* cache access macros */
#define CACHE_TAG(cp, addr)     ((addr) >> (cp)->tag_shift)
#define CACHE_SET(cp, addr)     (((addr) >> (cp)->set_shift) & (cp)->set_mask)
#define CACHE_BLK(cp, addr)     ((addr) & (cp)->blk_mask)
#define CACHE_TAGSET(cp, addr)          ((addr) & (cp)->tagset_mask)

/* extract/reconstruct a block address */
#define CACHE_BADDR(cp, addr)           ((addr) & ~(cp)->blk_mask)
#define CACHE_MK_BADDR(cp, tag, set)                                    \
  (((tag) << (cp)->tag_shift)|((set) << (cp)->set_shift))

/* index an array of cache blocks, non-trivial due to variable length blocks */
#define CACHE_BINDEX(cp, blks, i)                                       \
  ((struct cache_blk_t *)(((char *)(blks)) +                            \
                          (i)*(sizeof(struct cache_blk_t) +             \
                               ((cp)->balloc                            \
                                ? (cp)->bsize*sizeof(byte_t) : 0))))

/* cache data block accessor, type parameterized */
#define __CACHE_ACCESS(type, data, bofs)                                \
  (*((type *)(((char *)data) + (bofs))))

/* cache data block accessors, by type */
#define CACHE_DOUBLE(data, bofs)  __CACHE_ACCESS(double, data, bofs)
#define CACHE_FLOAT(data, bofs)           __CACHE_ACCESS(float, data, bofs)
#define CACHE_WORD(data, bofs)            __CACHE_ACCESS(unsigned int, data, bofs)
#define CACHE_HALF(data, bofs)   __CACHE_ACCESS(unsigned short, data, bofs)
#define CACHE_BYTE(data, bofs)   __CACHE_ACCESS(unsigned char, data, bofs)

/* cache block hashing macros, this macro is used to index into a cache
   set hash table (to find the correct block on N in an N-way cache), the
   cache set index function is CACHE_SET, defined above */
#define CACHE_HASH(cp, key)                                             \
  (((key >> 24) ^ (key >> 16) ^ (key >> 8) ^ key) & ((cp)->hsize-1))
```

```
/* copy data out of a cache block to buffer indicated by argument pointer p */
#define CACHE_BCOPY(cmd, blk, bofs, p, nbytes)  \
  if (cmd == Read)                                                                                                  \
    {                                                                                                               \
      switch (nbytes) {                                                                                             \
      case 1:                                                                                                       \
                *((byte_t *)p) = CACHE_BYTE(&blk->data[0], bofs); break;            \
      case 2:                                                                                                       \
                *((half_t *)p) = CACHE_HALF(&blk->data[0], bofs); break;            \
      case 4:                                                                                                       \
                *((word_t *)p) = CACHE_WORD(&blk->data[0], bofs); break;            \
      default:                                                                                                      \
                { /* >= 8, power of two, fits in block */                                           \
                  int words = nbytes >> 2;                                                                  \
                  while (words-- > 0)                                                                       \
                    {                                                                                       \
                      *((word_t *)p) = CACHE_WORD(&blk->data[0], bofs);              \
                      p += 4; bofs += 4;                                                                    \
                    }\
                }\
      }\
    }\
  else /* cmd == Write */                                                                                           \
    {                                                                                                               \
      switch (nbytes) {                                                                                             \
      case 1:                                                                                                       \
                CACHE_BYTE(&blk->data[0], bofs) = *((byte_t *)p); break;            \
      case 2:                                                                                                       \
        CACHE_HALF(&blk->data[0], bofs) = *((half_t *)p); break;        \
      case 4:                                                                                                       \
                CACHE_WORD(&blk->data[0], bofs) = *((word_t *)p); break;            \
      default:                                                                                                      \
                { /* >= 8, power of two, fits in block */                                           \
                  int words = nbytes >> 2;                                                                  \
                  while (words-- > 0)                                                                       \
                    {                                                                                       \
                      CACHE_WORD(&blk->data[0], bofs) = *((word_t *)p);                 \
                      p += 4; bofs += 4;                                                                    \
                    }\
                }\
    }\
  }

/* bound sqword_t/dfloat_t to positive int */
#define BOUND_POS(N)                    ((int)(MIN(MAX(0, (N)), 2147483647)))

/* unlink BLK from the hash table bucket chain in SET */
static void
unlink_htab_ent(struct cache_t *cp,             /* cache to update */
                struct cache_set_t *set,        /* set containing bkt chain */
                struct cache_blk_t *blk)        /* block to unlink */
{
  struct cache_blk_t *prev, *ent;
  int index = CACHE_HASH(cp, blk->tag);

  /* locate the block in the hash table bucket chain */
  for (prev=NULL,ent=set->hash[index];
       ent;
       prev=ent,ent=ent->hash_next)
    {
      if (ent == blk)
                break;
    }
  assert(ent);

  /* unlink the block from the hash table bucket chain */
  if (!prev)
    {
      /* head of hash bucket list */
      set->hash[index] = ent->hash_next;
    }
  else
    {
      /* middle or end of hash bucket list */
      prev->hash_next = ent->hash_next;
    }
  ent->hash_next = NULL;
}

/* insert BLK onto the head of the hash table bucket chain in SET */
static void
link_htab_ent(struct cache_t *cp,               /* cache to update */
              struct cache_set_t *set,          /* set containing bkt chain */
              struct cache_blk_t *blk)          /* block to insert */
{
  int index = CACHE_HASH(cp, blk->tag);

  /* insert block onto the head of the bucket chain */
  blk->hash_next = set->hash[index];
  set->hash[index] = blk;
}

/* where to insert a block onto the ordered way chain */
enum list_loc_t { Head, Tail };

/* insert BLK into the order way chain in SET at location WHERE */
static void
update_way_list(struct cache_set_t *set,        /* set contained way chain */
                struct cache_blk_t *blk,        /* block to insert */
                enum list_loc_t where)                  /* insert location */
{
  /* unlink entry from the way list */
  if (!blk->way_prev && !blk->way_next)
    {
      /* only one entry in list (direct-mapped), no action */
      assert(set->way_head == blk && set->way_tail == blk);
      /* Head/Tail order already */
      return;
    }
  /* else, more than one element in the list */
  else if (!blk->way_prev)
    {
```

```
      assert(set->way_head == blk && set->way_tail != blk);
      if (where == Head)
                        {
                          /* already there */
                          return;
                        }
      /* else, move to tail */
      set->way_head = blk->way_next;
      blk->way_next->way_prev = NULL;
    }
  else if (!blk->way_next)
    {
      /* end of list (and not front of list) */
      assert(set->way_head != blk && set->way_tail == blk);
      if (where == Tail)
                        {
                          /* already there */
                          return;
                        }
      set->way_tail = blk->way_prev;
      blk->way_prev->way_next = NULL;
    }
  else
    {
      /* middle of list (and not front or end of list) */
      assert(set->way_head != blk && set->way_tail != blk);
      blk->way_prev->way_next = blk->way_next;
      blk->way_next->way_prev = blk->way_prev;
    }

  /* link BLK back into the list */
  if (where == Head)
    {
      /* link to the head of the way list */
      blk->way_next = set->way_head;
      blk->way_prev = NULL;
      set->way_head->way_prev = blk;
      set->way_head = blk;
    }
  else if (where == Tail)
    {
      /* link to the tail of the way list */
      blk->way_prev = set->way_tail;
      blk->way_next = NULL;
      set->way_tail->way_next = blk;
      set->way_tail = blk;
    }
  else
    panic("bogus WHERE designator");
}

/* create and initialize a general cache structure */
struct cache_t *                                        /* pointer to cache created */
cache_create(char *name,                                /* name of the cache */
                          int nsets,                                    /* total number of sets in cache */
                          int bsize,                                    /* block (line) size of cache */
                          int balloc,                   /* allocate data space for blocks? */
                          int usize,                                    /* size of user data to alloc w/blks */
                          int assoc,                                    /* associativity of cache */
                          enum cache_policy policy,     /* replacement policy w/in sets */
                          /* block access function, see description w/in struct cache def */
                          unsigned int (*blk_access_fn)(enum mem_cmd cmd,
                                                                                                md_addr_t baddr, int bsize,
                                                                                                struct cache_blk_t *blk,
                                                                                                tick_t now),
                          unsigned int hit_latency)     /* latency in cycles for a hit */
{
  struct cache_t *cp;
  struct cache_blk_t *blk;
  int i, j, bindex;

  /* check all cache parameters */
  if (nsets <= 0)
    fatal("cache size (in sets) `%d' must be non-zero", nsets);
  if ((nsets & (nsets-1)) != 0)
    fatal("cache size (in sets) `%d' is not a power of two", nsets);
  /* blocks must be at least one datum large, i.e., 8 bytes for SS */
  if (bsize < 8)
    fatal("cache block size (in bytes) `%d' must be 8 or greater", bsize);
  if ((bsize & (bsize-1)) != 0)
    fatal("cache block size (in bytes) `%d' must be a power of two", bsize);
  if (usize < 0)
    fatal("user data size (in bytes) `%d' must be a positive value", usize);
  if (assoc <= 0)
    fatal("cache associativity `%d' must be non-zero and positive", assoc);
  if ((assoc & (assoc-1)) != 0)
    fatal("cache associativity `%d' must be a power of two", assoc);
  if (!blk_access_fn)
    fatal("must specify miss/replacement functions");

  /* allocate the cache structure */
  cp = (struct cache_t *)
    calloc(1, sizeof(struct cache_t) + (nsets-1)*sizeof(struct cache_set_t));
  if (!cp)
    fatal("out of virtual memory");

  /* initialize user parameters */
  cp->name = mystrdup(name);
  cp->nsets = nsets;
  cp->bsize = bsize;
  cp->balloc = balloc;
  cp->usize = usize;
  cp->assoc = assoc;
  cp->policy = policy;
  cp->hit_latency = hit_latency;

  /* miss/replacement functions */
  cp->blk_access_fn = blk_access_fn;

  /* compute derived parameters */
  cp->hsize = CACHE_HIGHLY_ASSOC(cp) ? (assoc >> 2) : 0;
  cp->blk_mask = bsize-1;
```

```
  cp->set_shift = log_base2(bsize);
  cp->set_mask = nsets-1;
  cp->tag_shift = cp->set_shift + log_base2(nsets);
  cp->tag_mask = (1 << (32 - cp->tag_shift))-1;
  cp->tagset_mask = ~cp->blk_mask;
  cp->bus_free = 0;

  /* print derived parameters during debug */
  debug("%s: cp->hsize     = %d", cp->name, cp->hsize);
  debug("%s: cp->blk_mask  = 0x%08x", cp->name, cp->blk_mask);
  debug("%s: cp->set_shift = %d", cp->name, cp->set_shift);
  debug("%s: cp->set_mask  = 0x%08x", cp->name, cp->set_mask);
  debug("%s: cp->tag_shift = %d", cp->name, cp->tag_shift);
  debug("%s: cp->tag_mask  = 0x%08x", cp->name, cp->tag_mask);

  /* initialize cache stats */
  cp->hits = 0;
  cp->misses = 0;
  cp->replacements = 0;
  cp->writebacks = 0;
  cp->invalidations = 0;

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

  /* allocate data blocks */
  cp->data = (byte_t *)calloc(nsets * assoc,
                                                        sizeof(struct cache_blk_t) +
                                                        (cp->balloc ? (bsize*sizeof(byte_t)) : 0));
  if (!cp->data)
    fatal("out of virtual memory");

  /* slice up the data blocks */
  for (bindex=0,i=0; i<nsets; i++)
    {
      cp->sets[i].way_head = NULL;
      cp->sets[i].way_tail = NULL;
      /* get a hash table, if needed */
      if (cp->hsize)
                        {
                          cp->sets[i].hash =
                            (struct cache_blk_t **)calloc(cp->hsize,
                                                                                                sizeof(struct cache_blk_t *));
                          if (!cp->sets[i].hash)
                            fatal("out of virtual memory");
                        }
      /* NOTE: all the blocks in a set *must* be allocated contiguously,
                        otherwise, block accesses through SET->BLKS will fail (used
                        during random replacement selection) */
      cp->sets[i].blks = CACHE_BINDEX(cp, cp->data, bindex);

      /* link the data blocks into ordered way chain and hash table bucket
        chains, if hash table exists */
      for (j=0; j<assoc; j++)
                        {
                          /* locate next cache block */
                          blk = CACHE_BINDEX(cp, cp->data, bindex);
                          bindex++;


      //*****************************
      //**                         **
      //**     Set RRPV Here       **
      //*****************************
      //blk->rrpv=3; //M=2
      blk->rrpv=7; //M=3
        //blk->rrpv=15; //M=4
       // blk->rrpv=31;//M=5
                          /* invalidate new cache block */
                          blk->status = 0;
                          blk->tag = 0;
                          blk->ready = 0;
                          blk->user_data = (usize != 0
                                                                ? (byte_t *)calloc(usize, sizeof(byte_t)) : NULL);

                          /* insert cache block into set hash table */
                          if (cp->hsize)
                            link_htab_ent(cp, &cp->sets[i], blk);

                          /* insert into head of way list, order is arbitrary at this point */
                          blk->way_next = cp->sets[i].way_head;
                          blk->way_prev = NULL;
                          if (cp->sets[i].way_head)
                            cp->sets[i].way_head->way_prev = blk;
                          cp->sets[i].way_head = blk;
                          if (!cp->sets[i].way_tail)
                            cp->sets[i].way_tail = blk;
                        }
    }
  return cp;
}

/* parse policy */
enum cache_policy                                                       /* replacement policy enum */
cache_char2policy(char c)                                       /* replacement policy as a char */
{
  switch (c) {
  case 'l': return LRU;
  case 'r': return Random;
  case 'f': return FIFO;
  case 's': return SRRIP;   //*********  Choose THE SRRIP replacement policy
  default: fatal("bogus replacement policy, `%c'", c);
  }
}

/* print cache configuration */
void
cache_config(struct cache_t *cp,     /* cache instance */
                         FILE *stream)                          /* output stream */
{
  fprintf(stream,
```

```
                        "cache: %s: %d sets, %d byte blocks, %d bytes user data/block\n",
                        cp->name, cp->nsets, cp->bsize, cp->usize);
  fprintf(stream,
                        "cache: %s: %d-way, `%s' replacement policy, write-back\n",
                        cp->name, cp->assoc,
                        cp->policy == LRU ? "LRU"
                        : cp->policy == Random ? "Random"
                        : cp->policy == FIFO ? "FIFO"
      :cp->policy==SRRIP? "SRRIP"
                        : (abort(), ""));
}

/* register cache stats */
void
cache_reg_stats(struct cache_t *cp, /* cache instance */
                                        struct stat_sdb_t *sdb)                 /* stats database */
{
  char buf[512], buf1[512], *name;

  /* get a name for this cache */
  if (!cp->name || !cp->name[0])
    name = "<unknown>";
  else
    name = cp->name;

  sprintf(buf, "%s.accesses", name);
  sprintf(buf1, "%s.hits + %s.misses", name, name);
  stat_reg_formula(sdb, buf, "total number of accesses", buf1, "%12.0f");
  sprintf(buf, "%s.hits", name);
  stat_reg_counter(sdb, buf, "total number of hits", &cp->hits, 0, NULL);
  sprintf(buf, "%s.misses", name);
  stat_reg_counter(sdb, buf, "total number of misses", &cp->misses, 0, NULL);
  sprintf(buf, "%s.replacements", name);
  stat_reg_counter(sdb, buf, "total number of replacements",
                                        &cp->replacements, 0, NULL);
  sprintf(buf, "%s.writebacks", name);
  stat_reg_counter(sdb, buf, "total number of writebacks",
                                        &cp->writebacks, 0, NULL);
  sprintf(buf, "%s.invalidations", name);
  stat_reg_counter(sdb, buf, "total number of invalidations",
                                        &cp->invalidations, 0, NULL);
  sprintf(buf, "%s.miss_rate", name);
  sprintf(buf1, "%s.misses / %s.accesses", name, name);
  stat_reg_formula(sdb, buf, "miss rate (i.e., misses/ref)", buf1, NULL);
  sprintf(buf, "%s.repl_rate", name);
  sprintf(buf1, "%s.replacements / %s.accesses", name, name);
  stat_reg_formula(sdb, buf, "replacement rate (i.e., repls/ref)", buf1, NULL);
  sprintf(buf, "%s.wb_rate", name);
  sprintf(buf1, "%s.writebacks / %s.accesses", name, name);
  stat_reg_formula(sdb, buf, "writeback rate (i.e., wrbks/ref)", buf1, NULL);
  sprintf(buf, "%s.inv_rate", name);
  sprintf(buf1, "%s.invalidations / %s.accesses", name, name);
  stat_reg_formula(sdb, buf, "invalidation rate (i.e., invs/ref)", buf1, NULL);
}

/* print cache stats */
void
cache_stats(struct cache_t *cp,                         /* cache instance */
                        FILE *stream)                   /* output stream */
{
  double sum = (double)(cp->hits + cp->misses);

  fprintf(stream,
                        "cache: %s: %.0f hits %.0f misses %.0f repls %.0f invalidations\n",
                        cp->name, (double)cp->hits, (double)cp->misses,
                        (double)cp->replacements, (double)cp->invalidations);
  fprintf(stream,
                        "cache: %s: miss rate=%f  repl rate=%f  invalidation rate=%f\n",
                        cp->name,
                        (double)cp->misses/sum, (double)(double)cp->replacements/sum,
                        (double)cp->invalidations/sum);
}

/* access a cache, perform a CMD operation on cache CP at address ADDR,
   places NBYTES of data at *P, returns latency of operation if initiated
   at NOW, places pointer to block user data in *UDATA, *P is untouched if
   cache blocks are not allocated (!CP->BALLOC), UDATA should be NULL if no
   user data is attached to blocks */
unsigned int                                                    /* latency of access in cycles */
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 = NULL;
  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;
    goto cache_fast_hit;
  }

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))
                            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))
                            goto cache_hit;
                        }
  }

/* cache block not found */

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

/* select the appropriate block to replace, and re-link this entry to
   the appropriate place in the way list */
int flag=0;
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;
case SRRIP:

        while (flag==0) {
            for (blk=cp->sets[set].way_head; blk; blk=blk->way_next) {
                if (blk->rrpv==7) {
                    repl=blk;
                    flag=1;
                    break;
                }
            }
            if (flag==0) {
                for (blk=cp->sets[set].way_head; blk; blk=blk->way_next){
                    if (blk->rrpv!=7) {
                        blk->rrpv++;
                    }
                }
            }
        }


    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);

    /* don't replace the block until outstanding misses are satisfied */
    lat += BOUND_POS(repl->ready - now);

    /* stall until the bus to next level of memory is available */
    lat += BOUND_POS(cp->bus_free - (now + lat));

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

    if (repl->status & CACHE_BLK_DIRTY)
                        {
                          /* write back the cache block */
                          cp->writebacks++;
                          lat += cp->blk_access_fn(Write,
                                                                                CACHE_MK_BADDR(cp, repl->tag, set),
                                                                                cp->bsize, repl, now+lat);

                        }
  }
```

```
  /* update block tags */
  repl->tag = tag;
  repl->status = CACHE_BLK_VALID;                   /* dirty bit set on update */
    //************************************************
  // Update the RRPV value of replaced block    **
  //************************************************

    repl->rrpv=6; // 0 means near-immediate/ 2^M means distant RRIP/ 2^M-2 means long RRIP

    //************************************************

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

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

  /* 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++;

  /* 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;

  /* 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);
    }
  //************************************************************
  //       Update the RRPV if replacement policy is SRRIP **
  //************************************************************
    if (blk->rrpv!=0 && cp->policy==SRRIP) {
        /* set RRPV of block to 0 */
        //blk->rrpv=0;  //HP
        blk->rrpv=blk->rrpv-1; //FP
    }

  //************************************************************

  /* 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;

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

  /* 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 */

  /* this block hit last, no change in  the RRPV value */

    if (blk->rrpv!=0 && cp->policy==SRRIP) {
        /* set RRPV of block to 0 */
        //blk->rrpv=0;  //HP
        blk->rrpv=blk->rrpv-1; //FP
    }


  /* 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));
}
```

Appendix B: Simulation Sample

This is just a sample of simulation results.

```
sim-outorder: SimpleScalar/Alpha Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use.  No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

warning: section `.comment' ignored...
sim: command line: ../../simplesim-3.0/sim-outorder -max:inst 50000000 -fastfwd 20000000 -redir:sim 2W_s.txt -bpred 2lev -bpred:2lev 1 256 4 0 -bpred:ras 8 -bpred:btb 64 2 -
cache:dl1 dl1:128:64:4:l -cache:dl2 dl2:8192:64:2:s ../../spec2000binaries/art00.peak.ev6 -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -
endx 160 -endy 240 -objects 10

sim: simulation started @ Sun Dec  1 16:51:35 2013, options follow:

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config                   # load configuration from a file
# -dumpconfig               # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt                <null> # restore EIO trace execution from <fname>
# -redir:sim             2W_s.txt # redirect simulator output to file (non-interactive only)
# -redir:prog           <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst            50000000 # maximum number of inst's to execute
-fastfwd             20000000 # number of insts skipped before timing starts
# -ptrace               <null> # generate pipetrace, i.e., <fname|stdout|stderr> <range>
-fetch:ifqsize              4 # instruction fetch queue size (in insts)
-fetch:mplat                3 # extra branch mis-prediction latency
-fetch:speed                1 # speed of front-end of machine relative to execution core
-bpred                   2lev # branch predictor type {nottaken|taken|perfect|bimod|2lev|comb|tournament}
-bpred:bimod     2048 # bimodal predictor config (<table size>)
-bpred:2lev     1 256 4 0 # 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:comb      1024 # combining predictor config (<meta_table_size>)
-bpred:tournament 4906 12 1024 1024 # tournament predictor config (<sel_size> <global_regsize> <local_htb_size> <local_hrsize>)
-bpred:ras                  8 # return address stack size (0 for no return stack)
-bpred:btb       64 2 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update       <null> # speculative predictors update in {ID|WB} (default non-spec)
-decode:width               4 # instruction decode B/W (insts/cycle)
-issue:width                4 # instruction issue B/W (insts/cycle)
-issue:inorder          false # run pipeline with in-order issue
-issue:wrongpath         true # issue instructions down wrong execution paths
-commit:width               4 # instruction commit B/W (insts/cycle)
-ruu:size                 16 # register update unit (RUU) size
-lsq:size                  8 # load/store queue (LSQ) size
-cache:dl1       dl1:128:64:4:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat               1 # l1 data cache hit latency (in cycles)
-cache:dl2       dl2:8192:64:2:s # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat               6 # l2 data cache hit latency (in cycles)
-cache:il1       il1:512:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat               1 # l1 instruction cache hit latency (in cycles)
-cache:il2               dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2lat               6 # l2 instruction cache hit latency (in cycles)
-cache:flush            false # flush caches on system calls
-cache:icompress         false # convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat         18 2 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width                  8 # memory access bus width (in bytes)
-tlb:itlb        itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat                  30 # inst/data TLB miss latency (in cycles)
-res:ialu                 4 # total number of integer ALU's available
-res:imult                1 # total number of integer multiplier/dividers available
-res:memport              2 # total number of memory system ports available (to CPU)
-res:fpalu                4 # total number of floating point ALU's available
-res:fpmult               1 # total number of floating point multiplier/dividers available
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat             false # operate in backward-compatible bugs mode (for testing only)

 Pipetrace range arguments are formatted as follows:
```

{{@|#}<start>}:{{@|#|+}<end>}

Both ends of the range are optional, if neither are specified, the entire
execution is traced. Ranges that start with a `@' designate an address
range to be traced, those that start with an `#' designate a cycle count
range. All other range values represent an instruction count range. The
second argument, if specified with a `+', indicates a value relative
to the first argument, e.g., 1000:+100 == 1000:1100. Program symbols may
be used in all contexts.

Examples: -ptrace FOO.trc #0:#1000
-ptrace BAR.trc @2000:
-ptrace BLAH.trc :1500
-ptrace UXXE.trc :
-ptrace FOOBAR.trc @main:+278

Branch predictor configuration examples for 2-level predictor:
Configurations: N, M, W, X
N # entries in first level (# of shift register(s))
W width of shift register(s)
M # entries in 2nd level (# of counters, or other FSM)
X (yes-1/no-0) xor history and address for 2nd level index
Sample predictors:
GAg : 1, W, 2^W, 0
GAp : 1, W, M (M > 2^W), 0
PAg : N, W, 2^W, 0
PAp : N, W, M (M == 2^(N+W)), 0
gshare : 1, W, 2^W, 1
Predictor `comb' combines a bimodal and a 2-level predictor.

The cache config parameter <config> has the following format:

<name>:<nsets>:<bsize>:<assoc>:<repl>

<name> - name of the cache being defined
<nsets> - number of sets in the cache
<bsize> - block size of the cache
<assoc> - associativity of the cache
<repl> - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random, 's'-SSRIP

Examples: -cache:dl1 dl1:4096:32:1:l
-dtlb dtlb:128:4096:32:r

Cache levels can be unified by pointing a level of the instruction cache
hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
configuration arguments. Most sensible combinations are supported, e.g.,

A unified l2 cache (il2 is pointed at dl2):
-cache:il1 il1:128:64:1:l -cache:il2 dl2
-cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

Or, a fully unified cache hierarchy (il1 pointed at dl1):
-cache:il1 dl1
-cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

sim: ** fast forwarding 20000000 insts **
sim: ** starting performance simulation **

sim: ** simulation statistics **
sim_num_insn 50000000 # total number of instructions committed
sim_num_refs 18898516 # total number of loads and stores committed
sim_num_loads 15101474 # total number of loads committed
sim_num_stores 3797042.0000 # total number of stores committed
sim_num_branches 5652369 # total number of branches committed
sim_elapsed_time 70 # total simulation time in seconds
sim_inst_rate 714285.7143 # simulation speed (in insts/sec)
sim_total_insn 52060555 # total number of instructions executed
sim_total_refs 18909716 # total number of loads and stores executed
sim_total_loads 15111343 # total number of loads executed
sim_total_stores 3798373.0000 # total number of stores executed
sim_total_branches 6400085 # total number of branches executed
sim_cycle 76081542 # total simulation time in cycles
sim_IPC 0.6572 # instructions per cycle
sim_CPI 1.5216 # cycles per instruction
sim_exec_BW 0.6843 # total instructions (mis-spec + committed) per cycle
sim_IPB 8.8458 # instruction per branch
IFQ_count 294367876 # cumulative IFQ occupancy
IFQ_fcount 72000579 # cumulative IFQ full count
ifq_occupancy 3.8691 # avg IFQ occupancy (insn's)
ifq_rate 0.6843 # avg IFQ dispatch rate (insn/cycle)
ifq_latency 5.6543 # avg IFQ occupant latency (cycle's)
ifq_full 0.9464 # fraction of time (cycle's) IFQ was full
RUU_count 998779473 # cumulative RUU occupancy
RUU_fcount 21943415 # cumulative RUU full count
ruu_occupancy 13.1278 # avg RUU occupancy (insn's)
ruu_rate 0.6843 # avg RUU dispatch rate (insn/cycle)
ruu_latency 19.1850 # avg RUU occupant latency (cycle's)
ruu_full 0.2884 # fraction of time (cycle's) RUU was full
LSQ_count 475655555 # cumulative LSQ occupancy
LSQ_fcount 48851906 # cumulative LSQ full count

```
lsq_occupancy               6.2519 # avg LSQ occupancy (insn's)
lsq_rate                    0.6843 # avg LSQ dispatch rate (insn/cycle)
lsq_latency                 9.1366 # avg LSQ occupant latency (cycle's)
lsq_full                    0.6421 # fraction of time (cycle's) LSQ was full
sim_slip                 1538829233 # total number of slip cycles
avg_sim_slip               30.7766 # the average slip between issue and retirement
bpred_2lev.lookups          6686562 # total number of bpred lookups
bpred_2lev.updates          5652364 # total number of updates
bpred_2lev.addr_hits        5365280 # total number of address-predicted hits
bpred_2lev.dir_hits         5365314 # total number of direction-predicted hits (includes addr-hits)
bpred_2lev.misses            287050 # total number of misses
bpred_2lev.jr_hits            40007 # total number of address-predicted hits for JR's
bpred_2lev.jr_seen            40008 # total number of JR's seen
bpred_2lev.jr_non_ras_hits.PP           0 # total number of address-predicted hits for non-RAS JR's
bpred_2lev.jr_non_ras_seen.PP           0 # total number of non-RAS JR's seen
bpred_2lev.bpred_addr_rate    0.9492 # branch address-prediction rate (i.e., addr-hits/updates)
bpred_2lev.bpred_dir_rate     0.9492 # branch direction-prediction rate (i.e., all-hits/updates)
bpred_2lev.bpred_jr_rate      1.0000 # JR address-prediction rate (i.e., JR addr-hits/JRs seen)
bpred_2lev.bpred_jr_non_ras_rate.PP <error: divide by zero> # non-RAS JR addr-pred rate (ie, non-RAS JR hits/JRs seen)
bpred_2lev.retstack_pushes        42175 # total number of address pushed onto ret-addr stack
bpred_2lev.retstack_pops          40024 # total number of address popped off of ret-addr stack
bpred_2lev.used_ras.PP            40008 # total number of RAS predictions used
bpred_2lev.ras_hits.PP            40007 # total number of RAS hits
bpred_2lev.ras_rate.PP        1.0000 # RAS prediction rate (i.e., RAS hits/used RAS)
il1.accesses                53208276 # total number of accesses
il1.hits                    53208031 # total number of hits
il1.misses                    245 # total number of misses
il1.replacements                4 # total number of replacements
il1.writebacks                  0 # total number of writebacks
il1.invalidations               0 # total number of invalidations
il1.miss_rate              0.0000 # miss rate (i.e., misses/ref)
il1.repl_rate              0.0000 # replacement rate (i.e., repls/ref)
il1.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses               18643249 # total number of accesses
dl1.hits                   11979587 # total number of hits
dl1.misses                  6663662 # total number of misses
dl1.replacements            6663150 # total number of replacements
dl1.writebacks              1334027 # total number of writebacks
dl1.invalidations                0 # total number of invalidations
dl1.miss_rate              0.3574 # miss rate (i.e., misses/ref)
dl1.repl_rate              0.3574 # replacement rate (i.e., repls/ref)
dl1.wb_rate                0.0716 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
dl2.accesses                7997934 # total number of accesses
dl2.hits                    4506649 # total number of hits
dl2.misses                  3491285 # total number of misses
dl2.replacements            3474901 # total number of replacements
dl2.writebacks               277408 # total number of writebacks
dl2.invalidations                0 # total number of invalidations
dl2.miss_rate              0.4365 # miss rate (i.e., misses/ref)
dl2.repl_rate              0.4345 # replacement rate (i.e., repls/ref)
dl2.wb_rate                0.0347 # writeback rate (i.e., wrbks/ref)
dl2.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses              53208276 # total number of accesses
itlb.hits                  53208270 # total number of hits
itlb.misses                    6 # total number of misses
itlb.replacements                0 # total number of replacements
itlb.writebacks                  0 # total number of writebacks
itlb.invalidations               0 # total number of invalidations
itlb.miss_rate             0.0000 # miss rate (i.e., misses/ref)
itlb.repl_rate             0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate               0.0000 # writeback rate (i.e., wrbks/ref)
itlb.inv_rate              0.0000 # invalidation rate (i.e., invs/ref)
dtlb.accesses              18907132 # total number of accesses
dtlb.hits                  18723506 # total number of hits
dtlb.misses                183626 # total number of misses
dtlb.replacements          183498 # total number of replacements
dtlb.writebacks                  0 # total number of writebacks
dtlb.invalidations               0 # total number of invalidations
dtlb.miss_rate             0.0097 # miss rate (i.e., misses/ref)
dtlb.repl_rate             0.0097 # replacement rate (i.e., repls/ref)
dtlb.wb_rate               0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate              0.0000 # invalidation rate (i.e., invs/ref)
sim_invalid_addrs                0 # total non-speculative bogus addresses seen (debug var)
ld_text_base          0x0120000000 # program text (code) segment base
ld_text_size               237568 # program text (code) size in bytes
ld_data_base          0x0140000000 # program initialized data segment base
ld_data_size                76672 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base         0x011ff9b000 # program stack segment base (highest address in stack)
ld_stack_size               16384 # program initial stack size
ld_prog_entry         0x012000b410 # program entry point (initial PC)
ld_environ_base       0x011ff97000 # program environment base address address
ld_target_big_endian            0 # target executable endian-ness, non-zero if big endian
mem.page_count                496 # total number of pages allocated
mem.page_mem                  3968k # total size of memory pages allocated
mem.ptab_misses             261432 # total first level page table misses
mem.ptab_accesses        529442078 # total page table accesses
mem.ptab_miss_rate          0.0005 # first level page table miss rate
```