Enhanced LRU Policy for Selecting Victim Cache Line

Aishwarya Dhandapani and Lalitha Geddapu
Department of Electrical and Computer Engineering
University of Florida
aishwarya.d@ufl.edu, lgeddapu@ufl.edu

Abstract

Miss penalty of the second level cache continues to increase as the main memory size increases. As the second level caches are moving towards full associativity, implementation of an effective replacement algorithm is becoming more important than reducing conflict misses. In this paper we present an idea that combines Least Recently Used (LRU) Replacement Policy with the concept of temporal locality to find the block that has to be replaced on a second level cache miss. This can be achieved by using a small locality table which indicates the temporal locality of each cache line.

1. Introduction

With the increasing gap between processor speed and memory latency, techniques to tolerate latency have been developed. Cache is one such technique to reduce the memory latency. Several optimizations have been described to improve cache performance with the most important one being the implementation of multi-level caches. Introduction of multi-level caches into the cache organization has reduced the miss penalty. The foremost difference between L1 and L2 cache is that the speed of L1 cache affects the clock rate of the processor while the speed of the L2 cache only affects the miss penalty of the L1 cache. For L2 caches there are fewer hits than in L1 and hence the emphasis shifts to fewer misses. The cache performance formula assures that improvements in miss penalty can be as beneficial as improvements in miss rate [2].

Average memory access time = Hit time + (Miss rate x Miss penalty)

To improve the performance we can reduce the miss rate, the miss penalty or the hit time. The solutions to reduce the miss rate are increasing associativity or increasing the cache size. We can reduce the miss rate in L2 by using an effective replacement policy on a cache miss. There are 3 general strategies to determine which block has to be replaced on a cache miss – random, LRU and FIFO. LRU is the most commonly used replacement technique. In LRU policy the cache line that was accessed least recently would be the victim on a cache miss. However this has some room for improvement in terms of selecting the victim for replacement by considering factors like locality. The proposed enhancement to LRU takes into account the concept of temporal locality while choosing its victim in addition to its normal operation.

1.2 Motivation

The trend in cache design is moving towards increasing the associativity and capacity of the L2 cache. This has made the access of L2 cache less time critical than the miss penalty of L2 cache. It is therefore important to concentrate on improving the miss rate of the L2 cache with effective replacement algorithms. The random policy chooses the victim cache line randomly among the available cache lines while the FIFO chooses its victim cache line using first-in first-out strategy. LRU is the most popular replacement which replaces the least recently used block on a cache miss. To demonstrate that LRU performs better than other existing replacement policies, we initially tested random, LRU and FIFO replacement policies for test-math benchmark and SPEC2000 benchmarks. The results have been tabulated as below.

The cache configuration is as follows:

-cache:il1 il1:64:32:4:(r/l/f)

-cache:dl1 dl1:64:32:4:(r/l/f)

-cache:il2 dl2

-cache:dl2 ul2:1024:32:8:(r/l/f)

Replacement Policy	**Average CPI**	**Miss Rate**
		ul2
Random	1.2343	0.1007
LRU	1.1944	0.1035
FIFO	1.2181	0.0989

Table 1. The miss rates of L2 cache for random, FIFO and LRU policies. L2 cache is configured as unified cache. Benchmark used is test-math.

Among the three replacement policies, FIFO performs better than LRU and random policies for the test-math benchmark. The cycles per instruction are also less than random and LRU. Further, the simulations have been performed with the above configuration for the provided SPEC2000 benchmarks. The maximum number of instructions is limited to 50000000.

The L1 cache configuration has separate instruction cache and data cache. The cache has 32 byte blocks, 4-way associativity and is configured to either random policy or LRU policy. The L1 instruction cache has 64 sets and data cache has 64 sets. The L2 cache configuration specifies a unified cache with 1024 sets (256KB) , 32 byte blocks, 8-way associativity which is configured to either random or LRU or FIFO replacement policies.

SPEC2000 Benchmarks	**Miss Rate**		
	Random	LRU	FIFO
equake	0.0048	0.0035	**0.0032**
crafty	0.0071	**0.0044**	0.0057
gzip	0.0711	**0.0488**	0.0620
lucas	**0.3562**	0.4965	0.3632
swim	**0.4433**	0.4970	0.4493
vortex	0.0430	**0.0394**	0.0414

Table 2. The miss rates of L2 cache for random, LRU and FIFO policies for SPEC2000 benchmarks. The bold values are the lowest miss rates for that particular benchmark.

It is clear from figure 1 that LRU performs better for most of the cases. Random replacement outperforms LRU for lucas and swim benchmarks.The IPC values have also been obtained to compare the performance of the three policies. LRU clearly has an upper hand over the other two policies as is evident from the statistics.

SPEC2000 Benchmarks	**IPC**		
	Random	LRU	FIFO
equake	**1.7066**	1.5730	1.5161
crafty	**1.0936**	1.0338	1.0237
gzip	1.8191	**1.8596**	1.8309
lucas	1.8790	**1.8810**	1.8792
swim	1.7621	**1.7692**	1.7629
vortex	1.3417	**1.3933**	1.3365

Table 3: The IPC values for random, FIFO and LRU policies. The bold values are the highest IPC of that particular benchmark.

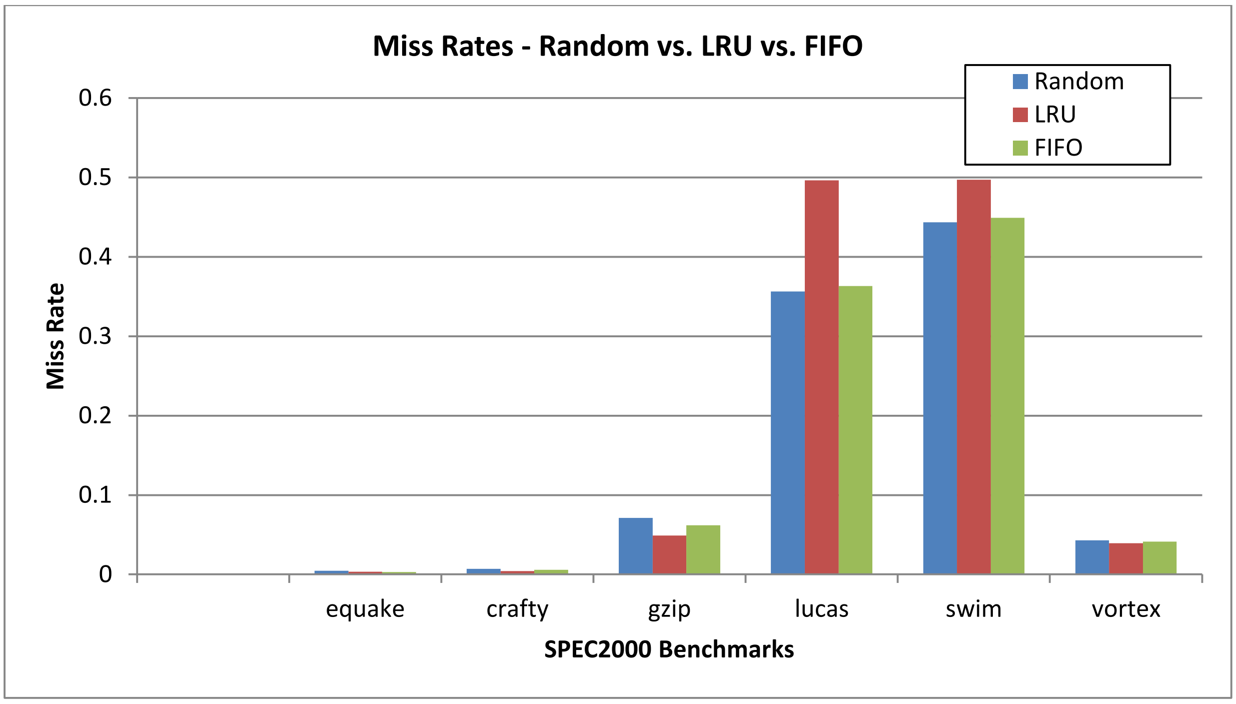

Figure 1. Miss rates for random, LRU and FIFO polices for various SPEC2000 Benchmarks.

However, with increase in associativity and capacity of the L2 cache the probability that the LRU victim line is not the optimal one increases. Therefore there is scope for improvement in the LRU policy by giving it more flexibility for better performance when there are more victim lines to choose from.

1.3 Related work

Temporal locality states that if an item is referenced, it will tend to be referenced again soon [3]. Using this temporal locality, a locality bit is added to every cache line. Whenever LRU looks for a line to replace, in addition to checking for the least recently used line it also checks for the locality bit so that a line that has high probability to be used again(temporal locality) is not replaced. This project is proposed along the lines of the research paper 'Modified LRU Policies for Improving Second Level Cache Behavior' by Wayne A. Wong and Jean-Loup Baer of University of Washington, Seattle presented in the Sixth International Symposium on High Performance Computer Architecture (HPCA-6), February 2000.

2. Overview

2.1 Simple scalar

Simplesclar is an execution driven simualator, i.e., a simulator that runs the program rather than reading a trace generated by a previous execution, to generate the statistics. The toolset includes compilers, assembler, linker, libraries and simulator. The toolset also includes all the source code which makes it possible to alter them and test the performance with the modified parameters. We made changes to the cache structure to implement the modified LRU policy. The simulation suite consists of 4 functional simulators and one performance simulator. Sim-fast, sim-safe, sim-profile and sim-cache are the functional simulators. Sim-outorder is the performance simulator. It generates more detailed statistics than the other simulators.

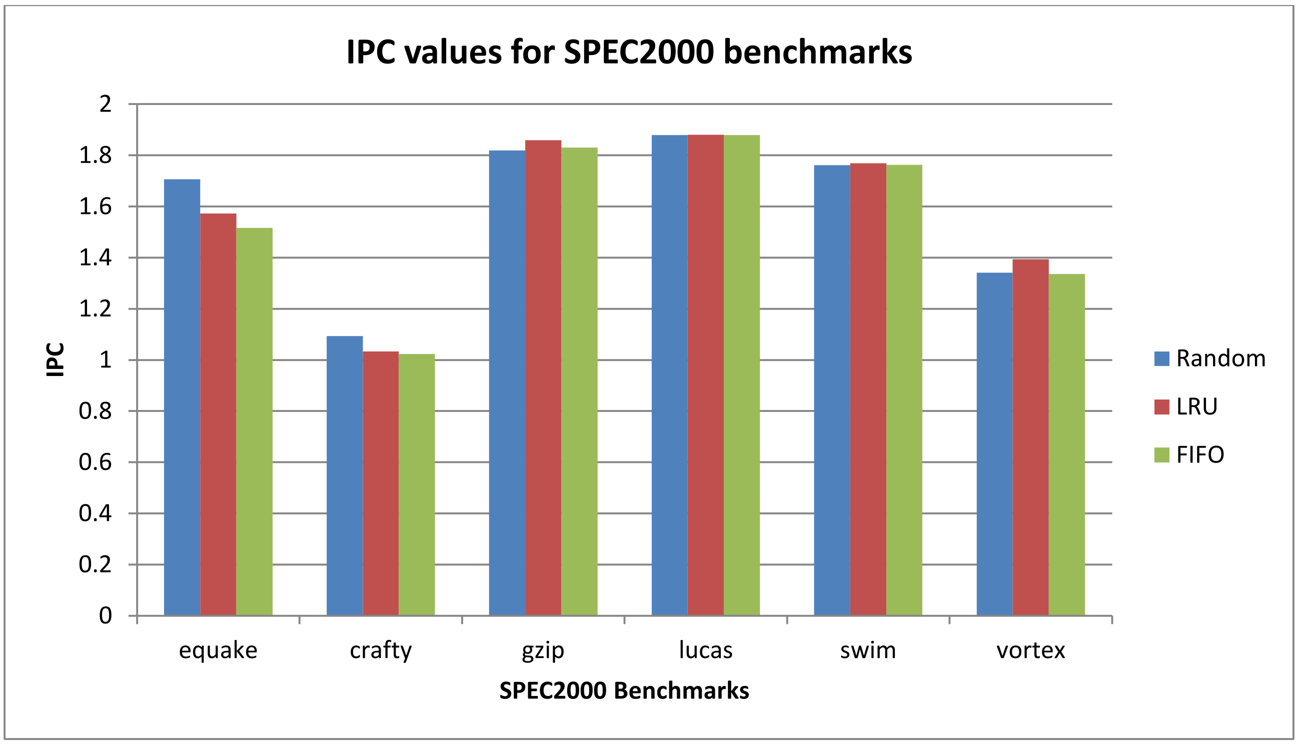

Figure 2: The IPC values for random, FIFO and LRU policies.

2.2 Online reference locality

To make the victim line selection different from LRU, the Online Reference Locality Algorithm adds a temporal bit and a PC_tag to every cache line and uses a locality table to keep track of the temporal bits. The locality table consists of a Program Counter field and a locality bit field. Every time a new instruction accesses the cache, the PC of that instruction is added to the locality table and the locality bit of that instruction is set to zero. The PC_tag field in the cache keeps track of the instructions that access the cache line and the temporal bit keeps track of the temporal locality of each cache line.

The following happens on a L2 reference from an instruction:

- If the reference is to the MRU line. No change in either the cache or the locality table.
- The reference is a hit to a non-MRU line. The locality bit entry in the locality table corresponding to the previous program counter that has accessed the cache line is set to one. The cache line becomes MRU and its temporal bit is set to the value of the locality bit in the current PC entry of the locality table. The *PC_tag* field for the cache line is set to the value of the current PC.
- The reference is a miss. The victim line is chosen to be the LRU line which has its temporal bit set to zero. The missing line is brought in, becomes MRU, and its temporal bit is set to the value of the locality bit in the current PC entry of the locality table. The *PC_tag* field for that line is set to current PC. The locality table entry with the victim's *PC_tag* is reset to zero.

3. Implementation in Simplescalar

The ORL algorithm was implemented both with and without the locality table. The algorithm is applied to all levels of cache for simplicity. The implementations are briefly described below. The code snippets can be found in the appendix.

- **Without Locality Table**

  A temporal bit was added to the cache line and every time there is a cache miss the LRU line with the temporal bit set to zero is chosen as the victim line. So if the cache line at the bottom of the stack does not have its temporal bit as zero then the line which is closest to the bottom of the stack with temporal bit as zero is chosen as the victim for replacement. In such cases the temporal bit of the LRU line is set to zero.

  The following changes were made in Simplescalar to implement this method:

  Changes to cache.h

  i. Added a value *'ORL'* to *enum cache_policy*
  ii. Added a variable *bool_t temporal_bit* to the structure *struct cache_blk_t*

  Changes to cache.c

  i. Added a new case to accommodate the ORL policy in the function *cache_char2policy*
  ii. Added a case to implement ORL in the function *cache_access*

- **With Locality Table**

  In this implementation a locality table is also added to the and the ORL algorithm is implemented as described in section 2.2.
  The following changes were made in Simplescalar to implement this method:

  Changes to cache.h

  The following variables and function declarations were added:

  i. A value *'ORL'* to *enum cache_policy*
  ii. A variable *bool_t temporal_bit* and *md_addr_t pc_tag* to the structure *struct cache_blk_t*
  iii. A structure for the locality table, *struct locality_table*
  iv. A function to set/reset the temporal bit
  v. A function to set/reset the locality bit
  vi. A function to search the locality table
  vii. A function for reordering the cache stack on a miss ie all temporal blocks are taken to the top and all non temporal blocks are brought to the bottom
  viii. A function to update the locality table on each cache access.

  Changes to cache.c

  The following were added:

  i. A new case to accommodate the ORL policy in the function *cache_char2policy*
  ii. A case to implement ORL in the function *cache_access*
  iii. Definitions for all the functions listed above.

- **Hurdles Faced**

  The main reason for the two different implementations is that the method with locality table needed the program counter value to be executed successful. However, even after getting the value from the current PC function, *regs.regs_PC*, the value of the current PC could not be retrieved and hence the ORL algorithm always worked like the LRU algorithm as there was only one entry in the locality table with PC value zero. Therefore another method with only the temporal bit was implemented and is found to show improvement in the second level cache miss rates.

3.1 Simulation Results

Initially the tests have been done with simple benchmarks such as test-math, test-fmath, test-printf, test-llong, anagram and go. The miss rates for three replacement policies – random_lh, LRU and ORL have been compared.

The cache configuration is as follows:

-cache:il1 il1:64:16:4:(r/l/o)

-cache:dl1 dl1:64:16:4:(r/l/o)

-cache:il2 dl2

-cache:dl2 ul2:128:16:8:(r/l/o)

Table 4 presents the miss rates for the three replacement policies for L2 unified cache. The L1 instruction cache has 64 sets and data cache has 64 sets. The L2 cache configuration specifies a unified cache with 128 sets, 16 byte blocks, 8-way associativity which is configured to either random_lh or LRU or ORL replacement policies.

Benchmarks		**Miss Rate**	
	Random_lh	LRU	ORL
test-math	0.5299	0.5279	**0.4832**
test-fmath	**0.6670**	0.6775	0.6832
test-printf	0.2557	0.2875	**0.2104**
test-llong	0.7038	**0.6705**	0.6827
anagram	**0.8761**	0.8996	0.8913
go	**0.5808**	0.8996	0.7190

Table 4. The miss rates of L2 cache for random_lh, ORL and LRU policies. L2 cache is configured as unified cache. The bold values are the lowest miss rates of that particular benchmark.

LRU was out-performed by both ORL and random_lh policies. This clearly indicates that the scope for improving LRU, which was our motivation for the project, existed. Among ORL and random_lh, the latter performed better for three benchmarks – test-fmath, anagram and go. These benchmarks might not be having more number of instructions that have temporal locality, hence the difference.

The IPC values have also been obtained to observe the efficiency of each policy. ORL policy has better instructions per cycle value for three of the six benchmarks tested. Table 5 presents the IPC values obtained for the three replacement policies.

Benchmarks	**IPC**		
	Random_lh	LRU	ORL
test-math	0.3819	0.3646	**0.3999**
test-fmath	**0.4143**	0.3882	0.3822
test-printf	0.6023	0.5502	**0.6578**
test-llong	**0.3937**	0.3829	0.3822
anagram	0.2708	**0.2731**	0.2730
go	1.6273	**1.6371**	1.6351

Table 5: The IPC values for random_lh, ORL and LRU policies. The bold values are the highest IPC of that particular benchmark.

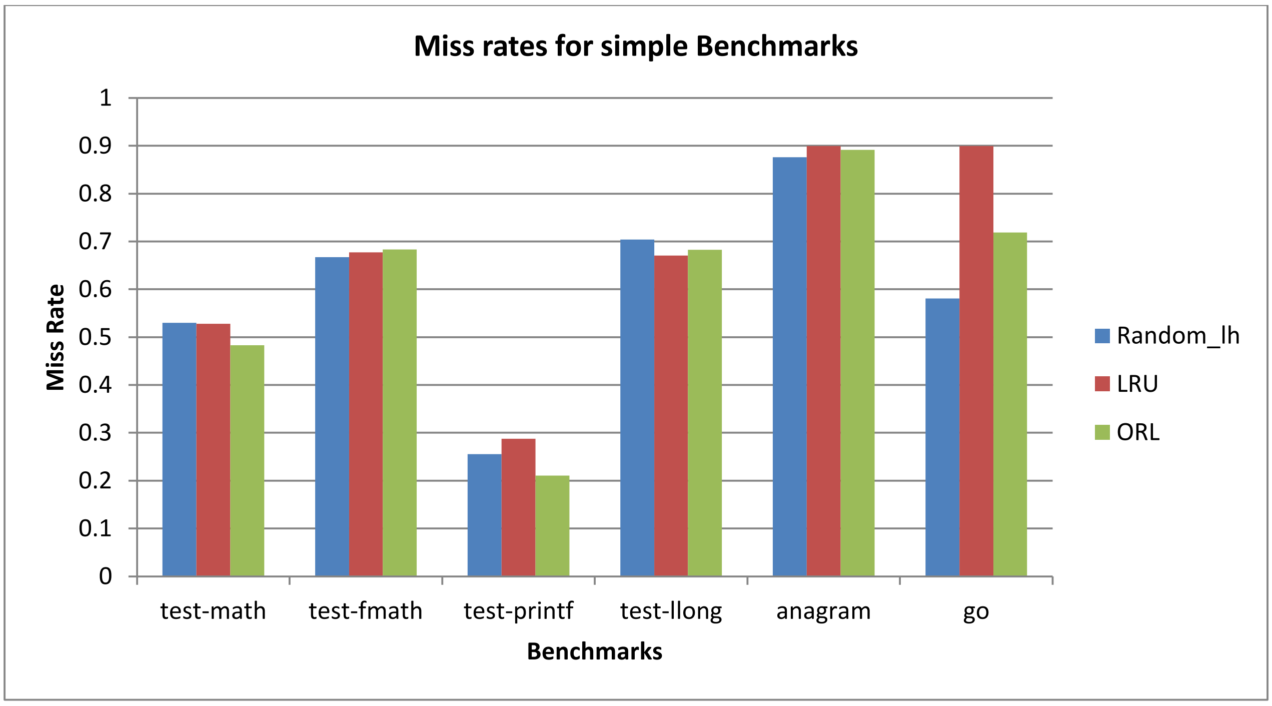

Figure 3. Miss rates for random_lh, LRU and ORL polices for various benchmarks.

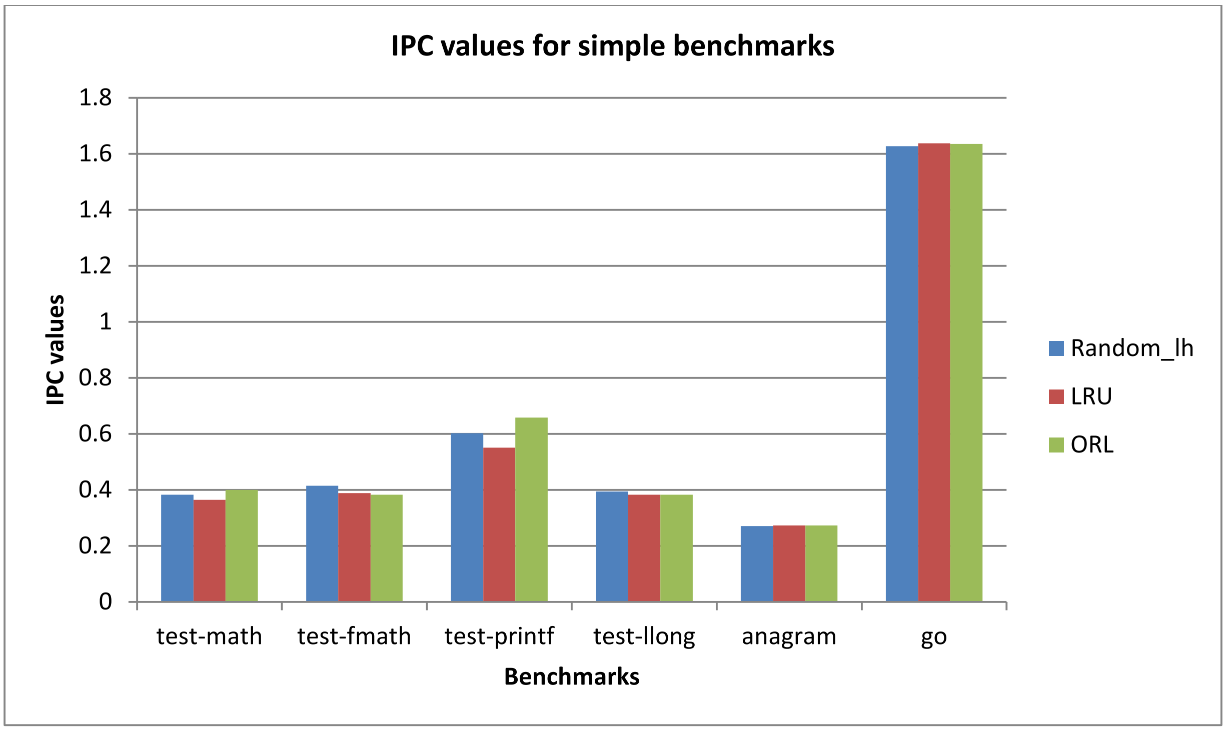

Figure 4. IPC values for random_lh, LRU and ORL polices for various benchmarks.

The simulations have been again performed with SPEC2000 benchmarks. The maximum number of instructions were restricted to 50000000. The cache configuration is as follows:

-cache:il1 il1:64:16:4:(r/l/o)

-cache:dl1 dl1:128:16:4:(r/l/o)

-cache:il2 dl2

-cache:dl2 ul2:256:16:8:(r/l/o)

Table 6 presents the miss rates for the three replacement policies for SPEC2000 benchmarks.

The L1 instruction cache has 64 sets and data cache has 128 sets. The L2 cache configuration specifies a unified cache with 256 sets, 16 byte blocks, 8-way associativity which is configured to either random_lh or LRU or ORL replacement policies.

SPEC2000 Benchmarks	**Miss Rate**		
	Random_lh	LRU	ORL
equake	**0.0370**	0.0418	0.0371
crafty	0.3436	0.3072	**0.2679**
gzip	**0.6587**	0.6692	0.7123
lucas	**0.5003**	0.5603	0.5202
swim	0.5198	**0.5060**	0.6706
vortex	0.4976	0.5418	**0.4730**

Table 6: The miss rates of L2 cache for random_lh, LRU and ORL policies for SPEC2000 benchmarks. The bold values are the lowest miss rates for that particular benchmark.

It can be observed from the tabulated results that the performance of LRU is poor when compared to random_lh and ORL replacement policies.

The miss rate for LRU policy was less only in the case of *swim* benchmark. ORL performed well for *crafty* and *vortex* benchmarks while random_lh performance is good for the rest of the benchmarks.

The IPC values have been obtained and presented in table 7. ORL's performance is the best among the three replacement policies in IPC values. As the IPC is more, the cycles per instruction would be less as CPI = 1/IPC. CPI directly influences the performance and thus, ORL proved to be the best when performance is the criterion.

SPEC2000 Benchmarks	**IPC**		
	Random_lh	LRU	ORL
equake	0.8663	0.8590	**0.9223**
crafty	0.4322	0.4308	**0.4601**
gzip	1.4511	1.4652	**1.4657**
lucas	1.8543	1.8512	**1.8664**
swim	1.7411	**1.7673**	1.7672
vortex	0.5171	0.5418	**0.5648**

Table 7: The IPC values for random_lh, LRU and ORL policies for SPEC2000 benchmarks. The bold values are the highest IPC for that particular benchmark.

The simulation times of all the three replacement policies have been tabulated in table 8. All the values are in seconds.

SPEC2000 Benchmarks	**Simulation time (in seconds)**		
	Random_lh	LRU	ORL
equake	73	77	**72**
crafty	88	79	**78**
gzip	72	**66**	**66**
lucas	53	54	**52**
swim	**55**	56	56
vortex	85	**77**	**77**

Table 8: The simulation time for random_lh, LRU and ORL policies for SPEC2000 benchmarks. The bold values are the lowest times for that particular benchmark.

ORL policy has the least simulation time for most of the benchmarks. It equaled LRU simulation time only for *gzip* and *vortex* benchmarks. Again, the performance of ORL proved to be the best among the tested policies. The cases where ORL time equaled LRU might be the cases where the ORL behaves as LRU itself when the temporal bits of all cache lines are set. ORL is a modification of LRU, so that would be reasonable.

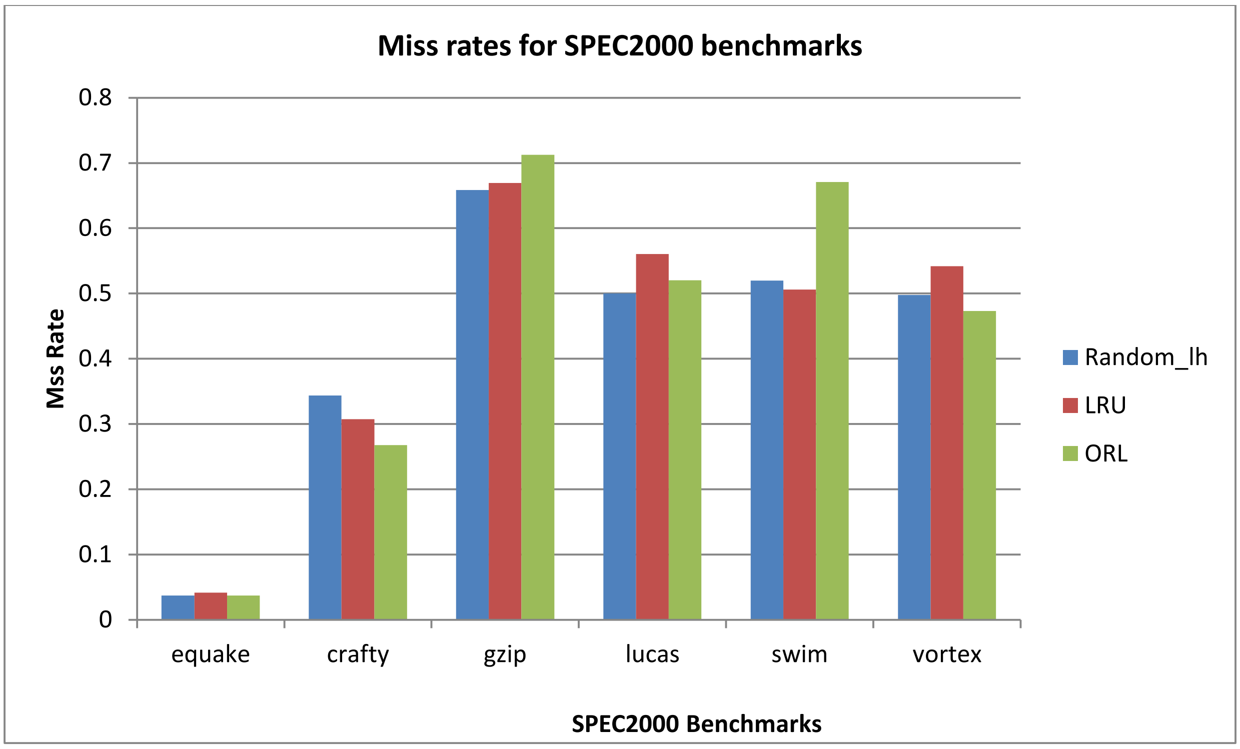

Figure 5. Miss rates for random_lh, LRU and ORL polices for various SPEC2000 benchmarks.

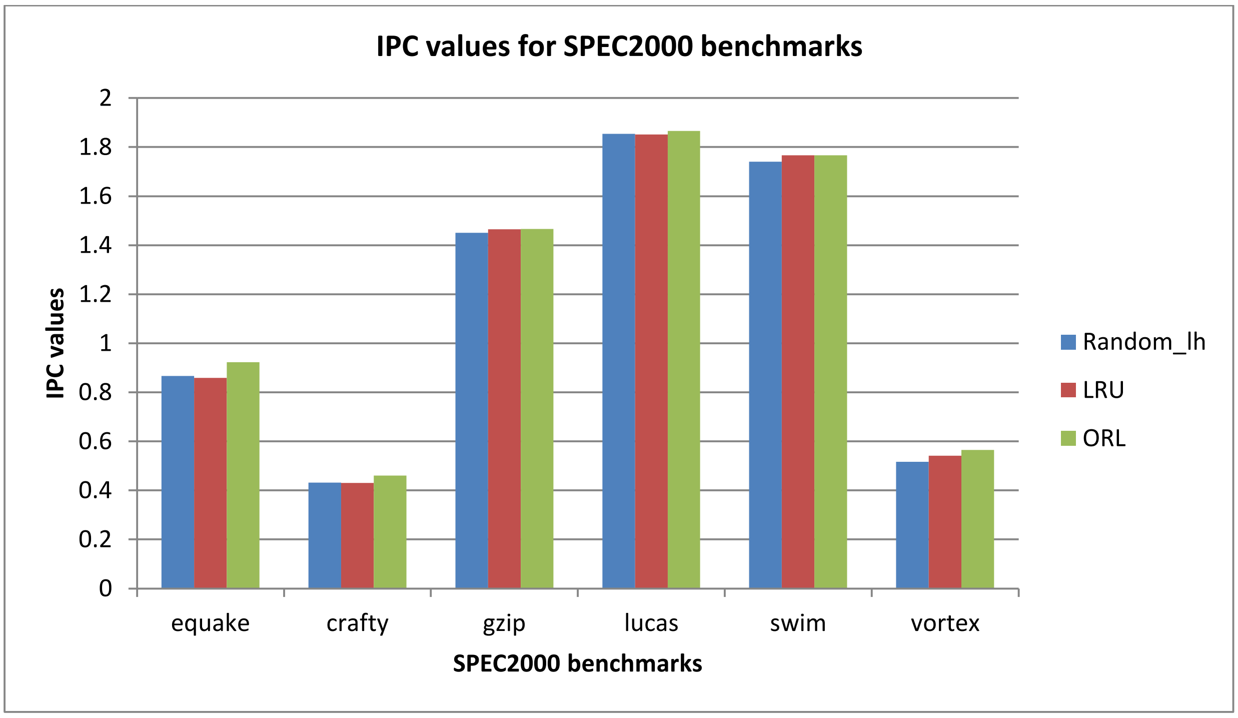

Figure 6. IPC values for random_lh, LRU and ORL polices for various SPEC2000 benchmarks.

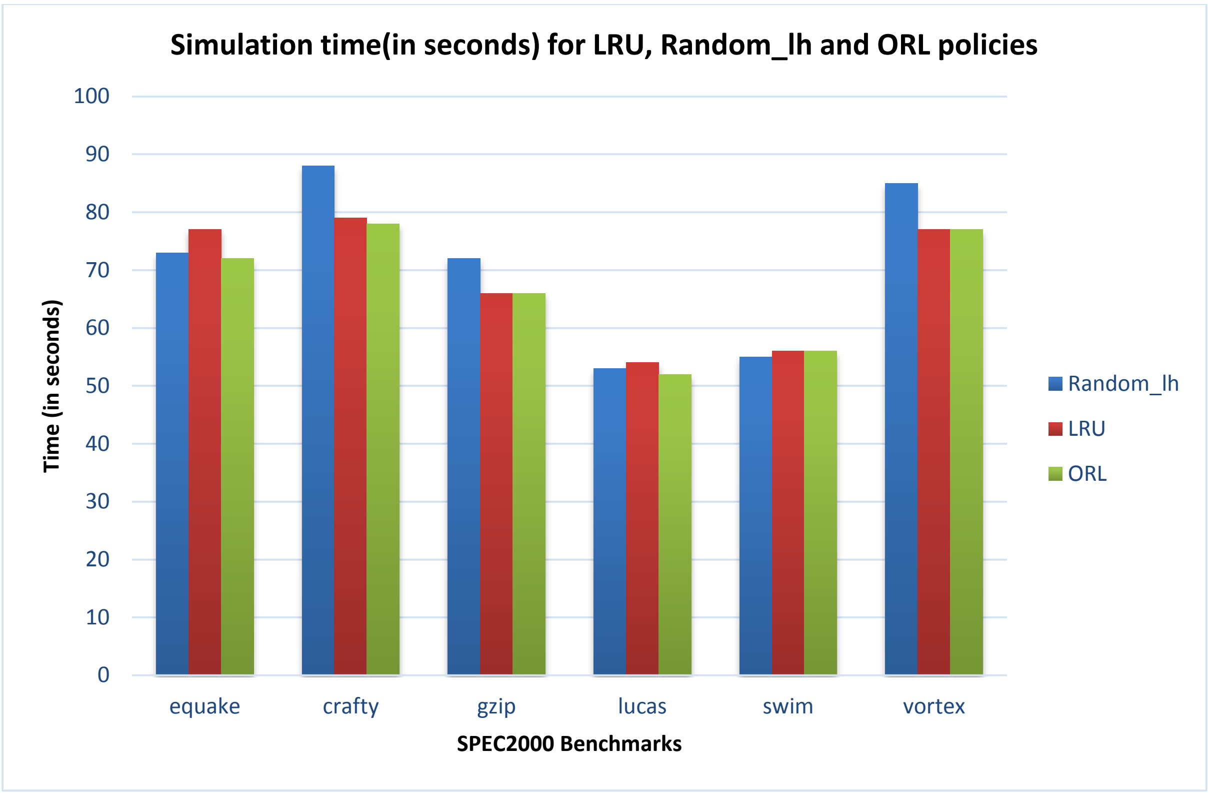

Figure 7. Simulation times of the three replacement policies for SPEC2000 benchmarks.

4. Conclusions

For all the tests performed, ORL's performance has been the best when IPC values and simulation time is considered. Random_lh is atleast as good as ORL or in some cases better, the reason being the instructions of that particular benchmark might not be exhibiting temporal locality as good as other benchmarks. LRU is the worst performer in almost all the cases baring a few benchmarks. The initial assumption that the scope for improvement in LRU existed has been proved and presented through the implementation of both ORL and random_lh policies. Online reference locality has the overall best performance among the three policies compared.

APPENDIX

This appendix includes the code snippets that have been added to Simplescalar. Only those parts that have been added are shown here along with the function or the block into which they were added. The code that was already there in the simulator is not shown.

CACHE.H

1. ‘ORL’was added to cache_policy

```
      enum cache_policy
{
 ORL              /*aishd: online reference locality*/
};
```

2. Variables for temporal bit and PC tag were declared in structure cache_blk_t

```
struct cache_blk_t
{ bool_t temporal_bit; /*aishd: temporal bit*/
  md_addr_t pc_tag; /*aishd: PC tag */
}
```

3. Structure for Locality Table

```
struct locality_table
{
        bool_t locality_bit; /*aishd: locality bit*/
        md_addr_t pc;  /*aishd: PC */
        struct locality_table *next; /* aishd: pointer to the next link in the list */
};
```

4. Function to set/reset the temporal bit
   void set_temporal_bit(int flag, struct cache_blk_t *blk, struct cache_set_t *set, struct locality_table *t);

5. Function to set/reset the locality bit
   void set_locality_bit(int flag, struct cache_blk_t *blk, struct cache_set_t *set, struct locality_table *t);

6. Function to search the locality table
   struct locality_table* search_loc_table(md_addr_t pc, struct locality_table *t);

7. Function for reordering the cache stack on a miss ie all temporal blocks are taken to the top and all non temporal blocks are brought to the bottom
   void reorder_list(struct cache_blk_t *blk, struct cache_set_t *set);

8. Function to update the locality table on each cache access.
   void update_locality_table(struct locality_table **t);

CACHE.C

1. A case is added to char2policy to accommodate ORL

```
cache_char2policy(char c)                    /* replacement policy as a char */
{
 case 'o': return ORL; /*aishd: returns ORL replacement policy*/
   }
```

2. Function to set/reset the temporal bit on a hit/miss

```
void set_temporal_bit(int flag, struct cache_blk_t *blk, struct cache_set_t *set, struct
locality_table *t)
{

  if(blk!=NULL)
  {        if(flag==0)
           {
                    md_addr_t curr_pc;
                    struct locality_table *ptr;
                    curr_pc=blk->pc_tag;
/*aishd: search the locality table for this PC and set the temporal bit of this block to the
locality bit of the PC from the table*/
                    ptr=search_loc_table(curr_pc,t);
                    if(ptr!=NULL)
                             blk->temporal_bit=ptr->locality_bit;
           }
           if(flag==1)
           {
                    blk->temporal_bit=1;
           }
}
}
```

3. Function to set/reset the locality bit on a hit/miss

```
void set_locality_bit(int flag, struct cache_blk_t *blk, struct cache_set_t *set, struct
locality_table *t)
{
         struct locality_table *ptr;
         md_addr_t prev_pc;
         prev_pc=blk->pc_tag;
         ptr=search_loc_table(prev_pc,t);
         if(ptr !=NULL)
{
         if(flag==1)
         {
                  ptr->locality_bit=1;
         }
         if(flag==0)
         {
                  ptr->locality_bit=0;
         }
}

}
```

4. Function to update the locality table everytime the cache is accessed

```
void update_locality_table(struct locality_table **t)
{
md_addr_t curr_pc;
struct locality_table *flag,*new;
curr_pc=regs.regs_PC;//curr_pc_function(regs.regs_PC);
if(t==NULL) //aishd: if the list is empty
        {
                new=(struct locality_table *)malloc(sizeof(struct locality_table));
                new->pc=curr_pc;
                new->locality_bit=0;
                new->next=NULL;
                *t=new;
        }
        else
{

          flag=search_loc_table(curr_pc,*t);
          if(flag==NULL)/*aishd:if the curr_pc is not found in the list*/
          {
            /*aishd: using malloc create a new node in the list and assign curr_pc to the pc
field and locality_bit as zero*/
             new=(struct locality_table *)malloc(sizeof(struct locality_table));
                new->locality_bit=0;
                new->pc=curr_pc;
                new->next=*t;
                *t=new;
          }
          else if(flag!=NULL)/*aishd:if the PC is already in the list*/
                {
                  flag->locality_bit=1;
                }
        }
```

5. Function to search the locality table for the given PC*/

```
struct locality_table* search_loc_table(md_addr_t curr_pc, struct locality_table *t)
{
        if(t != NULL)
{
        while(1)
        {
          if(t==NULL || t->next==NULL)
                break;

          else if(t->pc==curr_pc)
          {
                return t;
          }
          else
          {
                t=t->next;/*aishd: this pointer may need modification*/
          }
        }
}

        return NULL;
}
```

6. Function to reorder the cache stack when there is a miss*/

```
void reorder_list(struct cache_blk_t *blk, struct cache_set_t *set)
{
        int count=0;
        struct cache_blk_t *blk_copy, *blk_prev;
        blk=blk->way_prev;
        blk_prev=blk;
        if(blk!=NULL)
        {
        while(1)
        {
                if(blk->temporal_bit == 0 && count==0)
        {
                  blk_copy=blk;

                  blk->way_prev=set->way_tail;
                  blk->way_next=NULL;
                  set->way_tail->way_next=blk;
                  set->way_tail=blk;

                  blk=blk_prev;
                  blk=blk->way_next;
                  count=count+1;
                }
                else if(blk->temporal_bit==0 && count!=0)
                {
                  if(blk == blk_copy)
                    break;
                  else
                    blk_prev=blk;

                    blk->way_prev=set->way_tail;
                    blk->way_next=NULL;
                    set->way_tail->way_next=blk;
                    set->way_tail=blk;

                    blk=blk_prev;
                    blk=blk->way_next;
                }

                blk=blk->way_next;
        }
        }
}
```

7. Changes to cache_access()

```
Cache_access()
{
```

```
                if(t==NULL)/*aishd:initializing memory to the locality table in the first access of the
cache */
                {
                  t=malloc(sizeof(struct locality_table));
                  t->next = NULL;
            }

                struct cache_blk_t *blk, *repl, *new; /*aishd: a pointer 'new' to the cache block*/
```

//FOR A CACHE MISS:

```
  switch (cp->policy)
{
/* aishd: when the replacement policy is ORL, the following actions take place: */
  case ORL:
        repl=cp->sets[set].way_head;
        reorder_list(repl,&cp->sets[set]);
        repl=cp->sets[set].way_tail;
        set_locality_bit(0,repl,&cp->sets[set],t);
        update_way_list(&cp->sets[set],repl,Head);
        new=cp->sets[set].way_head;
        new->pc_tag=curr_pc_function(regs.regs_PC);
        set_temporal_bit(0,new,&cp->sets[set],t);
        break;
}
```

//FOR A CACHE HIT:

```
/*aishd: if ORL replacement policy and the hit is on a non-MRU line, make that block the head of the
list and set its temporal bit to 1 */
if(blk->way_prev && cp->policy == ORL)
{
        set_temporal_bit(1, blk, &cp->sets[set], NULL);
        set_locality_bit(1, blk, &cp->sets[set], NULL);
        update_way_list(&cp->sets[set],blk,Head);
}
```

//FOR RANDOM_LH

```
case Random:
    {
      int bindex = myrand() & (cp->assoc/2 - 1);
    int i;
     // if(bindex == 0)
    repl = cp->sets[set].way_tail;

      for(i = bindex; i>0; i--)
    {
        repl = repl->way_prev;
    }
      update_way_list(&cp->sets[set], repl, Head);
      //repl = CACHE_BINDEX(cp, cp->sets[set].blks, bindex);
    }
    break;
```