Improving Cache Performance by Exploiting Read-Write Disparity

Abstract

The difference in the criticality between read and writes for computer architecture is very crucial in the design of cache and processor, because read misses will cause processor stall while writes missed can be handled by buffers and other types of cache before committing to memory. This observation for the disparity between reads and writes have been taken good care of for processor pipeline. However, for traditional cache design mechanisms, this observation of "disparity between read and write criticality" has not been paid much attention to [1][2].

As a result, the key objective for this project is to exploit the read-write disparity in cache so that crucial load requests can be favored over less crucial store requests. We achieve this objective by applying Read-Write Partitioning (RWP) policy on the cache management strategy, which is designed to separate all the cache lines into two groups, a read partition and a write partition [1][2]. For our experiment, the partition size (allocated for dirty lines) is adapted in the range from 0 to the maximum associativity, to exploit the best strategy for the underlying benchmark which could attain the best performance improvement compared to baseline method. In this manner, RWP is supposed to get rid of write-only lines and preserve read lines, and thus improve the overall system performance.

We evaluate our RWP on simple-scalar on dl2. The benchmarks we have chosen are from Spec2000. We exploited 4-way, 8-way and 16-way cache for all the benchmarks in Spec2000 and for each associativity, we traverse the range of

allocated dirty lines to exploit the best partition size by evaluating the performance exposed by RWP. Our result shows that for most benchmarks in Spec2000, the IPC decreases with best partition size increasing (no lines allocated for dirty lines has the best performance), which indicates that for most Spec2000 benchmarks, read-from-clean lines are critically more than read-from-dirty lines.

Introduction

Cache efficiency has been focused these years in the literature, aimed at filling the performance gap between the memory and processor, which is a critical bottleneck in the microprocessor design nowadays. In traditional cache management, treatment of reads and writes is not very distinguished as well as in processor pipeline. However, load and store instructions have totally distinctive characteristics. The latency of critical load instructions is often more non-trivial than the latency of non-critical store instructions. The processor would stall if cache read misses occur, while most cache write misses can be directed to other types of buffers and caches that are off the critical path, so that write misses can be handled without hurting the performance very much. As a result, for most cases, cache blocks which deal with read instructions are more important than those that deal with only write instructions in the means of performance. With this observation, Samira Khan, et al [1][2] proposed a method to focus on this point, and according to their method, we propose to design a cache management policy that would favor cache blocks which deal with read instructions than those that deal with only write instructions. This cache management scheme is called Read-Write Partitioning (RWP) [1][2].

RWP makes important improvement on cache performance by preferring more crucial load requests over less crucial store requests. With this methodology, caches lines with critical load requests would be much more likely to be hit while some less critical store requests would be sacrificed to miss as a cost. Thus, it is also important to ensure the proportion of load and store in the program so that the RWP could attain the highest performance improvement. [1][2]

Problem Description & Motivation

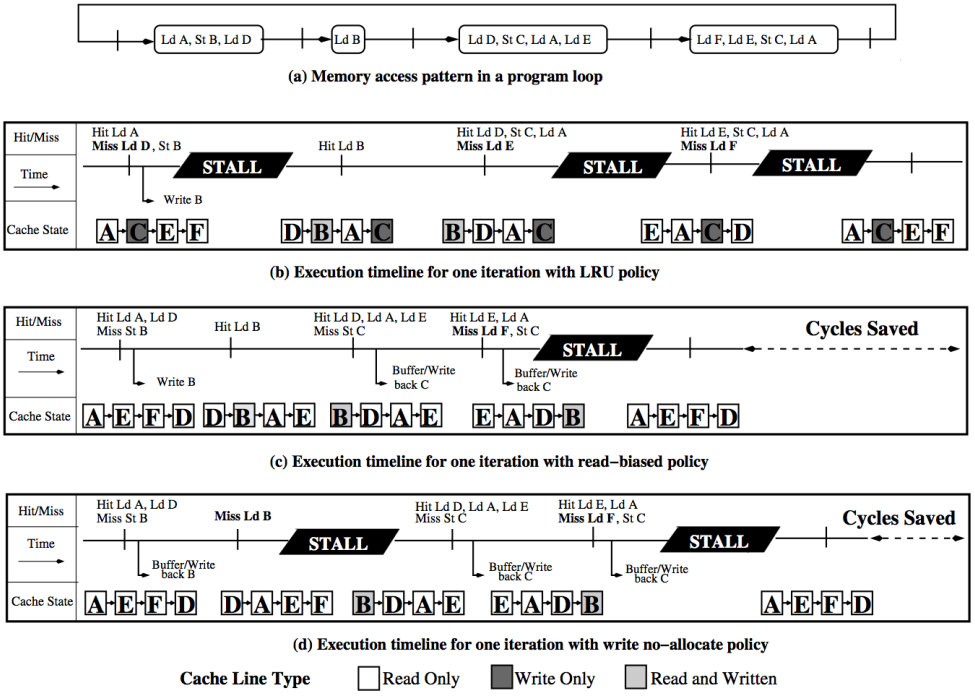

Figure 1. Different strategies in cache management. (a) Memory accesses in a loop; (b) LRU replacement; (c) Read-biased LRU replacement; (d) Write no-allocate policy [1][2].

As Figure 1 [1][2] shows, read-biased LRU replacement method has the fewest number of stalls compared to write no-allocate policy and LRU policy. This intuitively demonstrates that favoring read requests in the cache is able to improve performance. Moreover, it is crucial to differentiate between loads and stores;

otherwise we may get poor performance if we take for granted of the program structure before exploiting the actual proportion of read and write requests for a program, for e.g., Figure 1(d) [1][2] does not have good performance. As a real world example shown in Figure 2 [1][2], first benchmark performs best when all the cache blocks are assigned to the clean partition while the second benchmark performs better when more lines are allocated for dirty lines, because for the second benchmark the load requests performed from dirty lines are much more than from clean lines. This observation motivates us to investigate deeper into the best partition size for clean or dirty partition, to exploit the best proportion assigned to dirty and clean lines would be appropriate to extensively elevate the performance of the cache.

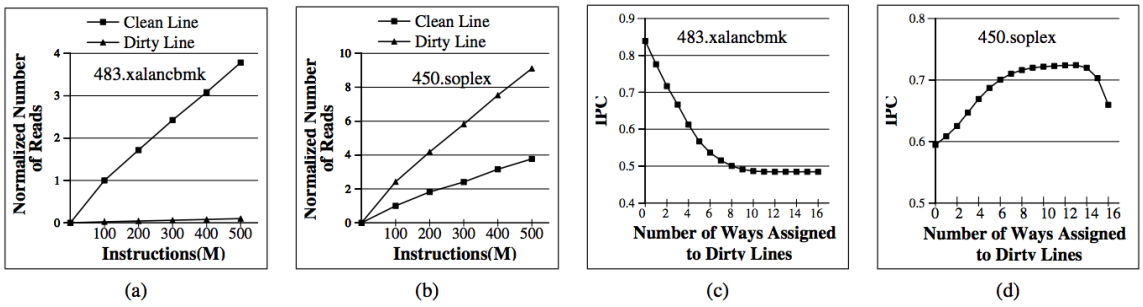

Figure 2 (a, b) number of reads in clean and dirty lines; (c, d) IPC when the number of ways allocated to dirty lines changes for a 16-way 4MB LLC [1][2].

Read-Write Partitioning

RWP is designed to separate all the cache lines into two groups, a read partition and a write partition [1][2]. For our experiment, the partition size (allocated for dirty lines) is adapted in the range from 0 to the maximum associativity (4, 8 or 16), to exploit the best strategy for the underlying benchmark which could attain the best performance improvement compared to baseline method, for e.g., LRP or RRIP [4].

In our implementation, RWP is only constrained to the Last Level Cache. This makes implementation easier as it does not need other information from other cache levels or the processor. Based on the above specifications and the structure of RWP shown in Figure 3 [1][2], we modify the general cache, cache set and block structure and also change the cache management policy by adding lazy partition enforcement to make sure we could preserve more read lines and consequently eliminate the write-only lines.

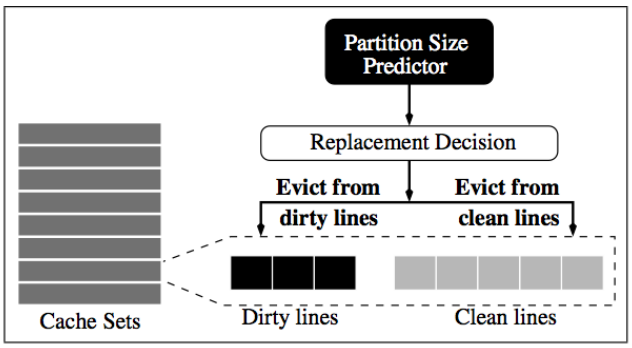

Figure 3 Structure of Read Write Partitioning (RWP)[1][2]

Namely, the cache structure is changed to support the RWP policy [1][2]. Cache set is modified to holds the best partition size, based on which the cache set will be logically divided into two partitions for clean and dirty lines. The cache block is attached with a dirty status bit to indicate if the line belongs to the read (clean) partition or the write (dirty) partition. The cache set also holds current dirty lines number, which will be referred when replacement happens under Dynamic Insertion Policy which we explain below.

We employ lazy partition enforcement to enforce partition size, which means the partition sizes are only enforced when block replacement happens, that is when a

new block is inserted [1][2]. Then upon the insertion, we employ Dynamic Insertion Policy (DIP), which will choose the correct block to replace by adopting LRU strategy combined with different cases. To locate the replacement victim, the current number of dirty lines is compared with the best dirty partition size we have set, which are maintained and updated in the enclosing cache set. The detailed policy algorithm [1][2], including conditions and actions are as below, which is very straight method:

if (current_dirty_line_num > best_dirty_line_ number):

//more dirty lines than it should have

replace LRU from dirty partition

else if (current_dirty_line_num < best_dirty_line_ number):

replace LRU from dirty partition

else: // current_dirty_line_num == best_dirty_line_ number

replace LRU depends on cache access type (r/w)

Method Implementation

We implement the prototype on simple-scalar 3.0. Major modifications can be divided into structure modification, policy modification and simulation modification. The modified source code is attached in the appendices.

Structure modifications are mainly made on cache structure (cache.h and cache.c).

This part includes changing the structure of cache_set_t and cache_block_t, which could incorporate critical information for later use in RWP.

Policy modifications are the most critical part of the code implementation, which are mainly made on cache methods (cache.c). This part includes initialization of LLC cache in cache_create() method and updated miss and hit handlers for LLC cache based on RWP policy in cache_access() method. The detailed logic for cache_access and case handler is shown in Figure 4, where we add four separate actions for Write-Hit, Write-Miss, Read-Hit and Read-Miss.

The detailed explanation for the protocol is as follows:

For Write-Hit, if the hit cache line is clean we need to set it dirty bit and meanwhile increase the current dirty lines number in the enclosing cache set. However, following the observations in the paper, we do not enforce the partition size at this time, which is the meaning of lazy enforcement.

For Read-Hit, if the hit cache line is dirty, then this line will become dirty-read, which by RWP should be prioritized to preserve. Thus we decrease the dirty lines number (increase the clean lines number), so that the possibility of replacing this dirty line will be lowered.

When it comes to the case of cache miss, this is where we need to enforce partition size by DIP, which has been explained above.

For Write-Miss, if fewer dirty lines are in the set than its best partition size, the LRU in the clean partition will be replaced and meanwhile this clean block will become

dirty. Otherwise, we need to replace a dirty block by another dirty block. This does not involve any block status change or dirty lines number change. Thus we take no actions.

For Read-Miss, the same rubric can be inferred. If we have more dirty lines in the set, we need to set the dirty bit t clean and decrease the dirty line number in the cache set. No other actions needed for other two cases.

	Write	Read

Hit	• clean:
	  • clean → dirty
	  • dirty_lines ++
	• dirty:
	  • nothing
		• clean:
		  • nothing
		• dirty:
		  • dirty_lines --

Miss	• dirty_lines < best size
	  • LRU in clean → dirty
	  • dirty_lines ++
	• dirty_lines > best size
	• dirty_lines = best size
	  • nothing
		• dirty_lines > best size
		  • LRU in dirty → clean
		  • dirty_lines --
		• dirty_lines < best size
		• dirty_lines = best size
		  • nothing

Figure 4 Cache access handler

Last modification is on simulation process, which are made on (sim-outorder.c and sim-cache.c). This part includes adding options for best partition size and modified initialization for LLC where we need to pass the best partition size into for RWP policy.

Evaluation & Result Anlaysis:

We evaluated our implementations on simple-scalar on dl2. The benchmarks we have

chosen are from SPEC CPU 2000.

ammp art equake gap lucas mgrid twolf applu bzip2 fma3d gcc mcf parser vortex apsi crafty galgel gzip mesa swim vpr.

In order to achieve better comparison, we configured the dl2 cache with different associativity, say 4-way, 8-way and 16-way cache for all the benchmarks in Spec2000. Also for evaluating the impact of partition size, we traversed different dirty line threshold. In this way, we could get the best partition size for each benchmark. We exhaustively experimented on 4-way, 8-way and 16-way cache by traversing the range of partition size allocated for dirty lines. Namely, for 4-way cache, we change the lines allocated for dirty partition from 0 to 4, with 0 meaning no lines for dirty partition, 4 meaning all lines for dirty partition; for 8-way cache, we change the range from 0 to 8; for 16-way cache, we traverse the range in the array (0, 1, 4, 8, 12, 16). All the shell scripts and result files can be found in the attached resource.

The corresponding shell scripts are:

for LRU: (:$ASSOCAITIVITY is the associativity for dl2)

```
./RUNammp ../../simplesim-3.0-RWP/sim-outorder ammp00.peak.ev6 -max:inst 50000000 -fastfwd 20000000 -redir:sim ammp_lru.txt -cache:dl2 ul2:1024:64:$ASSOCAITIVITY:l
```

for RWP: ($n is the number for partition size for dirty partition, ‘p’ is for RWP)

./RUNammp ../../simplesim-3.0-RWP/sim-outorder ammp00.peak.ev6 -max:inst 50000000 -fastfwd 20000000 -redir:sim ammp_rwp_${n}.txt -cache:dl2 ul2:1024:64:ASSOCAITIVITY:p -rwp:size $n

To analyze the performance, the metric we selected are *IPC, ul2_miss_rate and ul2_replacement_rate*. The baseline methods are LRU and SRRIP/DRRIP.

Due to the huge amount of results we get, we do not paste all the result here. Please refer to the files in the resource attached. Below are the sampled data results we got for 16-way cache.

ammp_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	0.9082	0.0087	0.0037
SRRIP	0.9094	0.0070	0.0019
DRRIP	0.9094	0.0067	0.0017
RWP-0	0.8583	0.1193	0.1185
RWP-1	0.8583	0.1194	0.1183
RWP-4	0.8469	0.1533	0.1513
RWP-8	0.8278	0.2113	0.2082
RWP-12	0.8134	0.2528	0.2487
RWP-16	0.7863	0.3433	0.3386

art_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	0.5957	0.6016	0.5998

art_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
SRRIP	0.6540	0.2948	0.2930
DRRIP	0.6845	0.2444	0.2426
RWP-0	0.6900	0.3607	0.3605
RWP-1	0.6900	0.3607	0.3604
RWP-4	0.6900	0.3607	0.3601
RWP-8	0.6900	0.3607	0.3596
RWP-12	0.6905	0.3598	0.3586
RWP-16	0.6906	0.3597	0.3584

crafty_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	1.2622	0.0036	0.0000
SRRIP	1.2638	0.0036	0.0000
DRRIP	1.2635	0.0039	0.0004
RWP-0	1.1597	0.0626	0.0617
RWP-1	1.1585	0.0634	0.0621
RWP-4	1.1371	0.0772	0.0752
RWP-8	1.1111	0.0951	0.0924
RWP-12	1.0961	0.1060	0.1031
RWP-16	1.0894	0.1113	0.1083

equake_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	1.7856	0.0029	0.0000
SRRIP	1.2638	0.0036	0.0000
DRRIP	1.2635	0.0039	0.0004
RWP-0	1.7403	0.0282	0.0273
RWP-1	1.7398	0.0285	0.0274
RWP-4	1.6501	0.0812	0.0788
RWP-8	1.6499	0.0813	0.0788
RWP-12	1.6499	0.0813	0.0788
RWP-16	1.6499	0.0813	0.0788

lucas_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	1.8810	0.2511	0.1264
SRRIP	1.8810	0.2511	0.1264
DRRIP	1.8810	0.2511	0.1264
RWP-0	1.8810	0.2511	0.2433
RWP-1	1.8810	0.2511	0.2433
RWP-4	1.8810	0.2511	0.2277
RWP-8	1.8810	0.2511	0.2121
RWP-12	1.8810	0.2511	0.1965

lucas_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
RWP-16	1.8810	0.2511	0.1809

mgrid_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	1.8062	0.1928	0.1839
SRRIP	1.8072	0.1921	0.1838
DRRIP	1.8042	0.1911	0.1839
RWP-0	1.7933	0.2017	0.2011
RWP-1	0.2051	0.2045	0.2045
RWP-4	1.7765	0.2084	0.2067
RWP-8	1.7601	0.2151	0.2123
RWP-12	1.7440	0.2217	0.2178
RWP-16	1.7282	0.2285	0.2235

parser_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
LRU	1.6769	0.1450	0.0588
SRRIP	1.6769	0.1450	0.0588
DRRIP	1.6769	0.1450	0.0588
RWP-0	1.6769	0.1456	0.1397
RWP-1	1.6768	0.1461	0.1360
RWP-4	1.6767	0.1468	0.1244

parser_result_16	sim_IPC	ul2_miss_rate	ul2_replacement_rate
RWP-8	1.6765	0.1484	0.1147
RWP-12	1.6761	0.1503	0.1056
RWP-16	1.6758	0.1515	0.0959

From the above results, we could find that, the difference of different cache replacement method (LRU,RRIP,DRRIP,RWP) is not very evident. After analyzing the benchmark feature, we find the reason may lies in that SPEC 2000 is not suitable for evaluating LLC cache replacement method. In the future, we would experiment on SPEC 2006 CPU benchmarks, since the general feedback from our classmates and the paper[1][2], this set of benchmark would generate more evident differences of results.

However, we do get some similar observation in the p aper [1][2]. Our result shows that for most benchmarks in Spec2000, the IPC decreases, ul2_miss_rate increases and ul2_replacement_rate decreases with the allocated dirty partition size increasing, which indicates that for these benchmarks in Spec2000, read-from-clean lines are critically more than read-from-dirty lines, That is why when there is 0 lines allocated for dirty partition we get the best performance. Another reason maybe some of the benchmarks are not very memory-intensive. Below is the IPC, ul2_missrat and ul2_replacement_rate graph these group: (ammp benchmark is selected as an example)

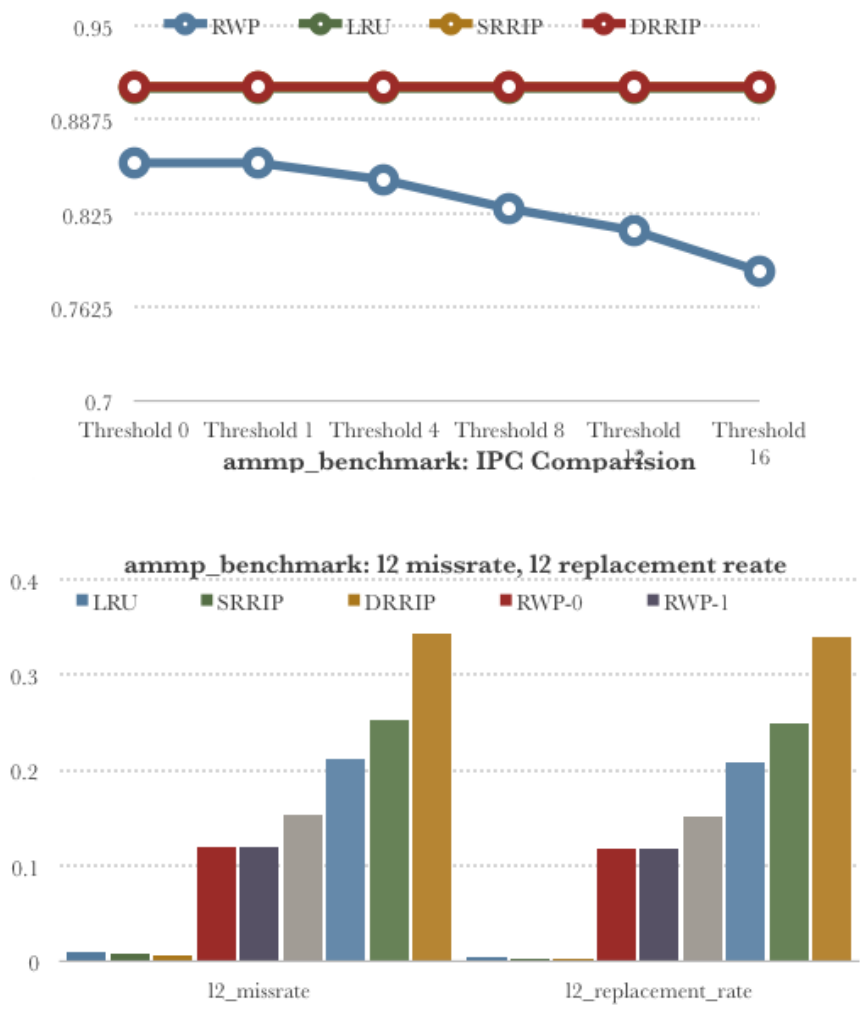

Figure 5 Result of ammp benchmark

The second similar observation could be find in the result of benchmark “art”. This benchmark could reflect the promising result using RWP method: first, it gets higher IPC compared to LRU (17%) and RRIP (6%); second, its performance gets better when more dirty lines are allocated. This indicates that this benchmark has more critical read from dirty lines than from clean lines. So we may have to favor more dirty lines to get better performance. Below is the plot of result for this benchmark:

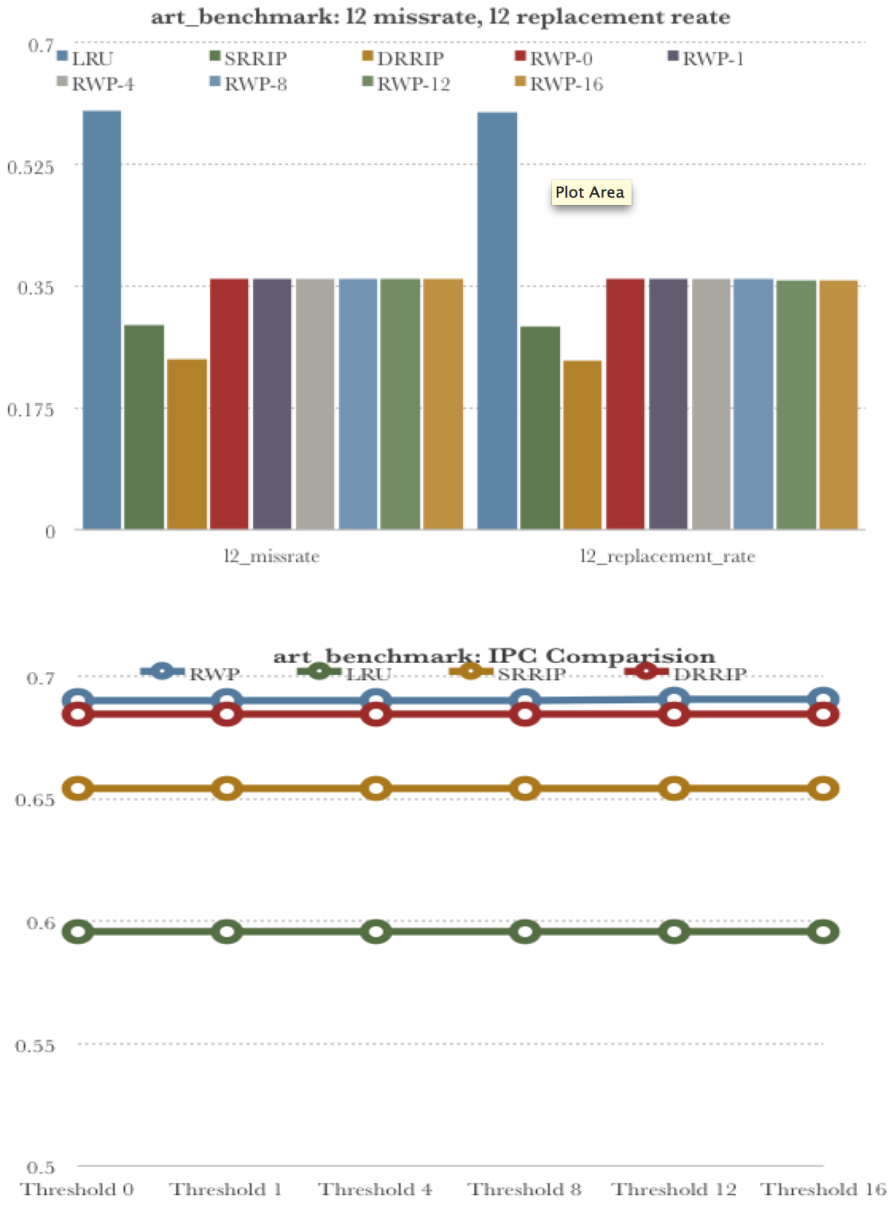

Figure 6 Result of art benchmark

Conclusions and Future work:

In this project, we learnt how to realizing RWP LLC replace method in simplescalar. Knowledge about simplescalar cache organization are consolidated through implementation. Also we learnt how to evaluate and analysis of other research work by setting up proper experiment method and choosing proper rubrics.

Because of the time limitation, we chose the SPEC 2000 as the basic benchmark. This choice somehow render us in an “embarrassing” situation that, the LLC replacement scheme didn’t affect two much on the result. Thus, in the future, we would like to run experiment on SPEC 2006. Also when considering RWP method, we would like to be able to predict partition size, this could be achieved by set sampling and using shadow directories. Besides, the read reference predictor could be introduced for its ability to directly identify write-only lines.

Besides the limitation of time and benchmark, we get some similar observation with the original paper.

References:

[1] Samira Khan, Alaa R.Alameldeen, Chris Wilkerson “Improving Cache Performance Using Read-Write Disparity” HPCA 2014.

[2] Samira Khan, Alaa R.Alameldeen, Chris Wilkerson “Improving Cache Performance Using Read-Write Partitioning” Internal Report 2014.

[3] Aamer Jaleel Kevin B, Theobald Simon C, Stelly Jr, Joel Emer “Improving Cache Performance Using Read-Write Partitioning” ACM SIGARCH ,2010.

[4] Jaleel, Aamer, et al. "High performance cache replacement using re-reference interval prediction (RRIP)." ACM SIGARCH Computer Architecture News. Vol. 38. No. 3. ACM, 2010.

Appendix

Simulation code: all modified code is commented with “RWP”. Please refer to the source code in the resource attached with the report.

Sample runs of your experiments/simulations: all related shell scripts and files are attached with the report.

Division of work:

Tian Lan: implementation of RWP

Yukun Gao: implementation of RRIP