Computer Architecture 614 Term Project

Student: Wei-Che Hsu
UIN: 826001725

Abstract:

A policy of cache replacement is one main factor of running time performance. Lower cache miss rate will speed up the speed of system under other fixed conditions. In this term project, I choose one policy, Probabilistic Replacement Policy (PRP), to study. I implement it and test it under different conditions. One pattern mcf achieves 25% running time reduction compared to LRU when LLC is 4MB. In addition, with more instructions regarding LLC access and more experienced line usage information, the system with this policy would performs faster.

Introduction:

I select the paper “Reuse Distance-Based Probabilistic Cache Replacement” [1] as my term project. This paper proposes the policy to choose the victim when cache replacement happens, which is based on memory usage history. This policy, which is called Probabilistic Replacement Policy (PRP), is able to distinguish long-term reused blocks and medium-term reused blocks as different cases. For example, if in one set all cache are filled, this policy will select the longest-term reuse block as victim of cache replacement. In this selection, the victim choice is nearly the same as the choice of the Optimal cache replacement algorithm (OPT) [2], which knows the order of what block will be accessed in the future. That is, PRP

would like to choose the block which is not used for the longest period in the future.

On the other hand, PRP would like to “kick-out” the block which lives in the cache for too much longer time than expected time. For example, if history shows the reused distance of one block frequently is 50. If now the timestamp of this block shows it has already in the cache over 100. Then, this block is likely to be replaced when cache replacement happens.

However, in the general application, it is hard to obtain the order of what block will be accessed in the future for long enough period. Thus, this paper proposes its model to simulate the feature of OTP. This model, which is based on the record of each memory block access, can apply on choosing victim of cache replacement the OPT-like cases and “kick-out” cases. In the next section, I will describe more detail of this model in PRP.

Paper Idea and Its Model Description:

According to Paper observation, Least Recent Used (LRU) policy is a good policy for level 1 cache and level2 cache for short-term reused blocks. But, for long-term reused blocks and medium-term reused blocks in the Last Level Cache (LLC), LRU is not good choice, because it can’t distinguish them, and can’t leave the suitable blocks from them in LLC. Here, I would define reused distance: the number of accesses to the set containing a cache block between consecutive accesses to that cache block. Between consecutive accesses, this cache block has the same memory location.

To distinguish long-term reused blocks and medium-term reused blocks, the paper takes

the following model:

F is a policy, and some references to LLC is miss in F but hit in OPT. The number of these references is Δ_F. x_t^e is the evicted cache line of policy F at time t. $I_{x_t^e}$ is a random variable, which is 1 if the evicted cache line is hit in the future under OPT, and 0 otherwise. Then, apply Expectation on eq1 to obtain eq2. $P_{x_t^e}(hit)$ is the probability which x_t^e is hit in the future under OPT. To obtain better policy, Δ_F should be as lower as possible. That is, the policy should find the victim in cache replacement with the lowest $P_{x_t^e}(hit)$.

$$\Delta_F = \sum_t I_{x_t^e} \quad \text{...(eq1)}$$

$$\mathrm{E}[\Delta_F] = \sum_t P_{x_t^e}(hit) \quad \text{...(eq2)}$$

Next, paper describe how to estimate the probability of a hit. The candidate cache line L, which would have probability P_L^{hit} to receive a hit under OPT from the current time to the future. In eq3, T_L is the age of the line L.

The line distribution $P_L(\mathrm{t})$: the probability that reused distance for line L is t.

The cache distribution $P^{hit}(\mathrm{t})$: the probability that reused distance which equals t for any line would receive the hit under OPT.

$$P_L^{hit} = \frac{\sum_{t>T_L} P_L(\mathrm{t}) P^{hit}(\mathrm{t})}{\sum_{t>T_L} P_L(\mathrm{t})} \quad \text{...(eq3)}$$

To choose the candidates, it prefers the victim with the lowest P_L^{hit}.

Then, this paper explains how to obtain $P_L(\mathrm{t})$ and $P^{hit}(\mathrm{t})$. For $P_L(\mathrm{t})$, Use eq4. This paper separate reused period of this line L into discrete multiple bins, each bin records $N_L(\mathrm{i})$

the frequency (probability) to the corresponding reused bin i. For finding $P_L(\mathrm{t})$ with reused period t, which belongs to one bin, from this line L can described as eq4.

$$P_L(\mathrm{t}) = \frac{N_L(\mathrm{t})}{\sum_i N_L(\mathrm{i})} \quad \text{...(eq4)}$$

For $P^{hit}(\mathrm{t})$, the paper provides average hit rate of each reuse distance t (6 bins) under OPT over the SPEC-CPU2006 suite as Table I shown, so it seems as the known information in my experiment. Then, take eq4 into eq3, and eliminate $\sum_i N_L(\mathrm{i})$ in numerator and denominator. Finally, it derives to eq5, which is mainly used for PRP implement.

$$P_L^{hit} = \frac{\sum_{t>T_L} N_L(\mathrm{t}) P^{hit}(\mathrm{t})}{\sum_{t>T_L} N_L(\mathrm{t})} \text{...(eq5)}$$

Now, I will describe how to separate the bins in the paper. The first bin, Bin0, records reuse distances in the interval [1, W), where W is the way size of the cache. If there are H bins, Bin index i= 1,2,3…,H-2 is with reuse distances in the interval $[\mathrm{W}\alpha^{i-1}, \mathrm{W}\alpha^{i})$. The last bin records reuse distances in the interval [Wα^{H-2}, infinity). In the experiments of this paper and my experiments is mainly on 16-way associated cache, the size of cache is 4 MB, and take H = 6.

Bin (t)	0-15	15-31	32-63	64-127	128-255	256-infinity
$2^4 P^{hit}(\mathrm{t})$	15	14	12	10	9	1

Table1. Cache distribution $P^{hit}(\mathrm{t})$ from paper's Table1[1]

For hardware cost, it should avoid using float point, so it takes 4-bit counter to each bin, when this cache is access at reused distance t, the counter in $N_L(\mathrm{t})$ add 1. if the number is larger than 15, it halves the counter value of all bins. Therefore, you can see every value of counter is between 0 and 15. Halve operation just need offset 1 bit.

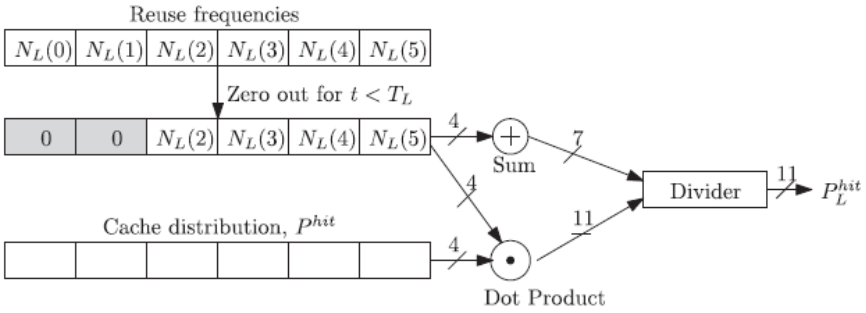


Figure1. Probability calculator unit from paper's Figure4[1]

Figure1. shows the Probability calculator unit, which is the main calculation for each candidate in one set when cache replacement happens. In this figure, t is between 32 and 63.

Apply eq5: $P_L^{hit} = \frac{N_L(2)P^{hit}(2)+N_L(3)P^{hit}(3)+N_L(4)P^{hit}(4)+N_L(5)P^{hit}(5)}{N_L(2)+N_L(3)+N_L(4)+N_L(5)}$

The above computation takes Dot product in the numerator and sum in denominator. Then, divide numerator over denominator.

Figure2. shows why PRP works well. Assume S is references with long-term used distance, and A is references with median-term used distance. P_S and P_A are N_L with 6 bins respectively. Apply eq5, PRP prefer to choose cache block with S reference as the victim because it has lower P_L^{hit}. Here, $P^{hit}(\mathrm{t})$ is float point, and N_L is known fixed fact, so not update the frequency. Just for the example of victim choice in cache replacement. In the real case, PRP will update N_L during cache access and use 4-bit counter to represent $P^{hit}(\mathrm{t})$.

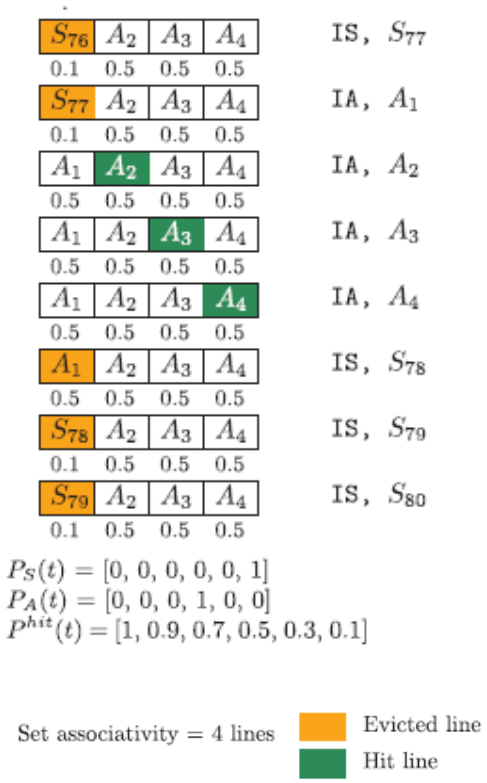


Figure2. PRP Example from paper's Figure5[1]

The above paragraph explains OPT-like behavior in PRP. Last, I explain "kick-out" cases, which is mentioned in the end of the previous section. For example, Given 6 bins of N_L = [15,11,0,0,0,0]. For this line, when reused time of this line is 65. eq5 will gives the result of P_L^{hit} = 0. Therefore, this line is easily to choose as victim during cache replacement happens. That is, the out-date cache line is likely to "kick-out".

In this section, I describe the main idea of PRP, and how it decides victim of cache replacement. Although it is a sound policy, it has high recording cost on memory and time-cost to read recording data from DRAM. To consider cost, the paper proposes tag-sampling method to reduce memory cost significantly. I only implement main idea of this paper without tag-sampling, and I will estimate the hardware cost of original PRP in the

discussion section.

My Work:

My implementation is based on Figure 3 without TLB part.

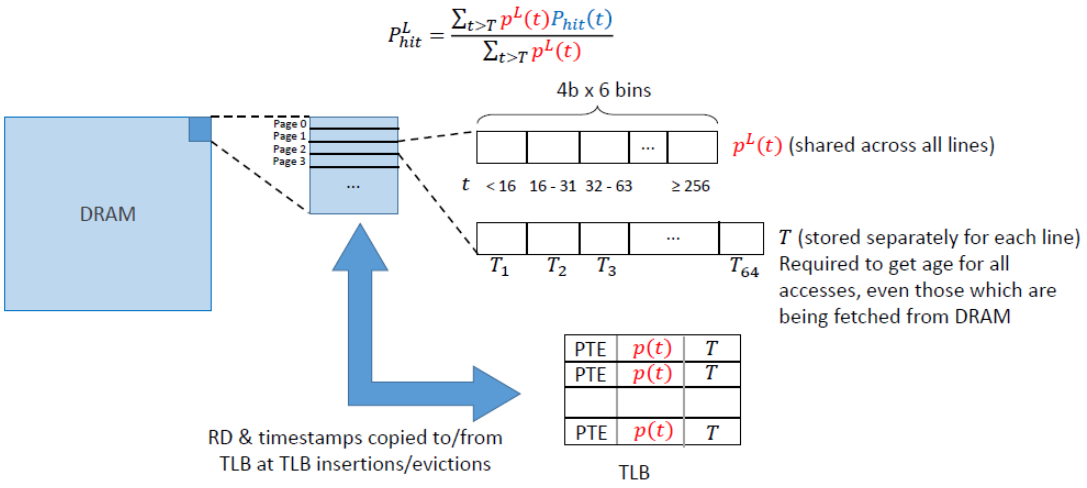


Figure3. Reused Distance**(RD)** and Timestamps storage diagram from paper's PowerPoint [1].

Because paper declares that cache lines has locality, one page can share one N_L, but time stamp should store each value for each stamp. According to paper's setting, page size is 4 KB. I put storage directly in the heap space of zsim. Although Virtual Address (VA) is 64 bits in zsim, I only take last 32 bits to assign each VA to the storage of the corresponding page and the corresponding cache index, because total heap space is limited.

Below is virtual address calculation in my code, **prp_repl.h**.

Zsim is 64-bit address, MemReq* req->lineAddr is line Address, which is VA[63:6]

Take VA[63:0] align to line unit and truncate to 32-bit:

reqAddr = ((req->lineAddr)<<6)&0xFFFFFFFF, which is VA[31:0].

Take **page Index**: (reqAddr>>12), which is VA[31:12].

Take **cache Index in one page**: (req->lineAddr)&0x3f, which is VA[11:6]

Next, Introduce the storage of Reused distance (RD) and Timestamps in the heap space.

dramBinArray: It records RD for 2^{20} pages, and one N_L has 6 bins in one page.

dramTsArray: It records Timestamps for 2^{20} pages, and there are 64 lines in one page.

They can seem as dramBinArray[2^{20}][6] and dramTsArray[2^{20}][64].

Third, I implement the feature to store the Reused distance of all dramBinArray elements to one file when cache access times is more than defined value, or load the reused-distance from one file when initial situation. This is used for PRP to have enough N_L information. At line 23 and 24 in **prp_repl.h**, user can define whether store/load file. Between line is defined cache access times for each test pattern.

There are 3 main functions in **prp_repl.h** for cache replacement:

void **update**(uint32_t id, const MemReq* req): When update function is called, the timestamp of the accessed set increases 1. Next, according to req->lineaddr to find corresponding page index and cache line Index in this page. Then, find timestamp t_line in this set and dramTsArray[page index][cache line Index], and calculate time difference between t_line and timestamp of this set as reused time period t. Update reused distance information to the corresponding N_L(t) of dramBinArray[page_index][reused time interval]. Here, I tried to update reused distance information only in cache hit cases, but the performance is not good, I will show the result of gcc in the next section. Therefore, the

above update progress happens in both cache-hit and cache-miss cases. Even if this line is new-coming cache line because cache miss, it still updates N_L(t) for this page with this new-coming cache line, Also, it assigns the timestamp of this set to this new-coming line.

void **replaced**(uint32_t id): The only task is that recording the requested line-address ,VA[63:6], of new-coming cache block. This line-address is saved in an array, **idArray**, with index = id. This line-address information is essential for finding victim when cache replacement happens.

uint32_t **rank**(const MemReq* req, C cands): Give a set, and request victim for cache replacement. Here, for all cache lines find its corresponding page index and cache line index through **idArray**. Then, find time information from dramTsArray[page index][cache line Index] to calculate age of every cache line in this set. Next, Find RD information from dramBinArray[page_index][ref_time interval]. Do the P_L^{hit} calculation which is followed eq5. Finally, choose the candidate with lowest P_L^{hit}.

Overall, my work for this term project includes reading this paper, coding for its main idea, adding store/load mode for N_L, collecting as well as analyzing data, and writing this report.

Experimental Results and Discussion:

This paper focus on testing SPEC 2006 patterns and also provides $P^{hit}(\mathrm{t})$ as shown in Table1. I had 6 patterns which are the same as they used: gcc, leslie3D, mcf, soplex,

xalancbmk, and cactusADM. I used these 6 patterns for testing. The data of LRU, LFU, SRRIP(2 bit) is obtained from my HW4 simulation in this course.

Core	1 Core, Out-of-order execution, 2.4GHz
Level1 Cache	I-Cache: 32KB, 4way, LRU D-Cache: 32KB, 8way, LRU
Level2 Cache	256KB, 8way, LRU
Level3 Cache	2MB, 16way. 4MB, 16way for Figure 8, Figure9, and Figure10.
Cache-setting	Write-Back, inclusive, cache line size = 64 Bytes.

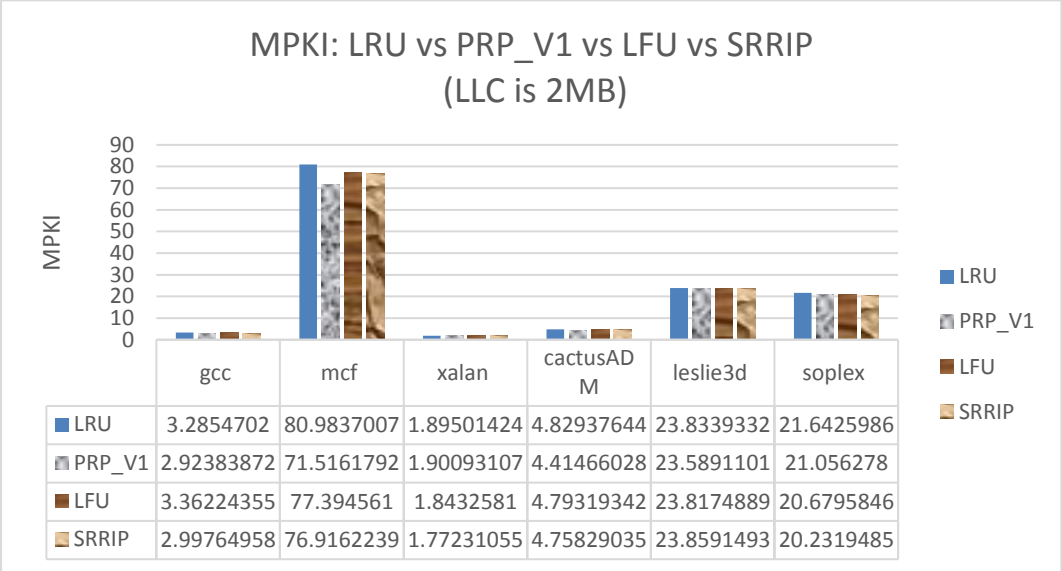
Table2. Environment setting of zsim

The original test result is good, the performance of mcf patterns has significant improvement about 9% running time reduction p, which is faster than LFU as well as SRRIP, and the performance of other patterns looks mostly faster, the running time reduction of geometric average is 2.4%. For further testing PRP, I decide to collect reused-distances information from **PRP_V1,** where its reused-distances N_L(i) starts from all 0s. On the other hand, **PRP_V2** loads this reused-distances information before it starts the first cache access. Compared to PRP_V1, 4 of 6 patterns have smaller misses per kilo-instructions(MKPI) in PRP_V2 version, and soplex pattern even gets 4% running time reduction. After loading reused-distances information previously, PRP_V2 runs soplex pattern faster than LRU, LFU, and SRRIP(2bit).

Overall, mcf has obviously better performance under PRP policy compared to LRU. After previous reused-distances information, PRP can get better performance. It implies that calculate P_L^{hit} scheme works fine if it has enough reused-distance information N_L. However, as Figure 8 shown, my improvement is not so significant as that of paper, where mcf gains

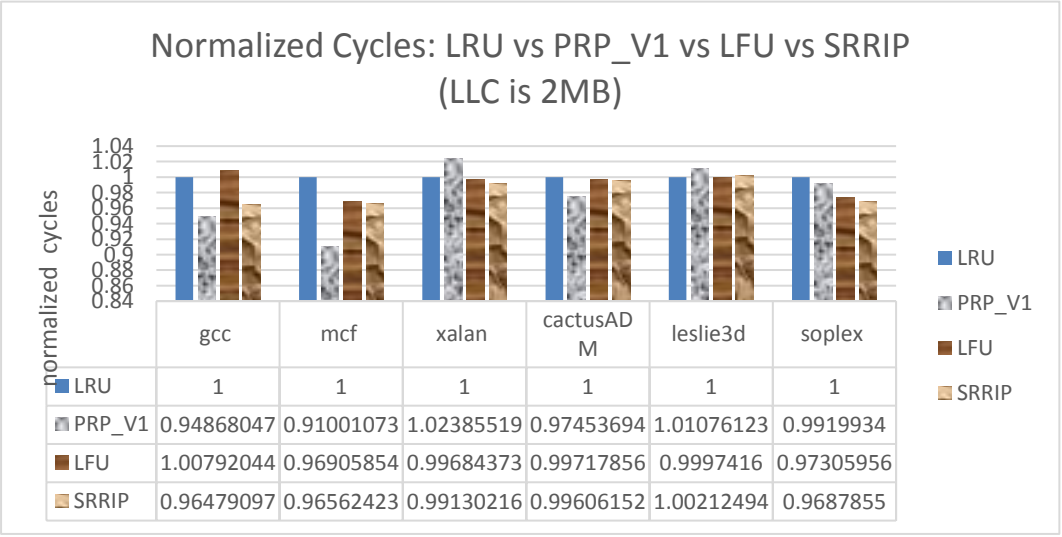
about 40% running time reduction. Thus, I think my PRP can't work as well as paper, the problem may happen at N_L collection and update.

In my experiment, N_L may not reflect reused distance well for cache replacement. The one reason may be too big page size for index. The index with smaller page size can reflect well to each line. For example, dramBinArray[page_index] can change from 2^{20} to 2^{24}, where a page index is only mapping to 16 cache lines. Another reason is that I truncate 64-bit VA into 32-bit VA, may overlapped some addresses with data table and instruction.



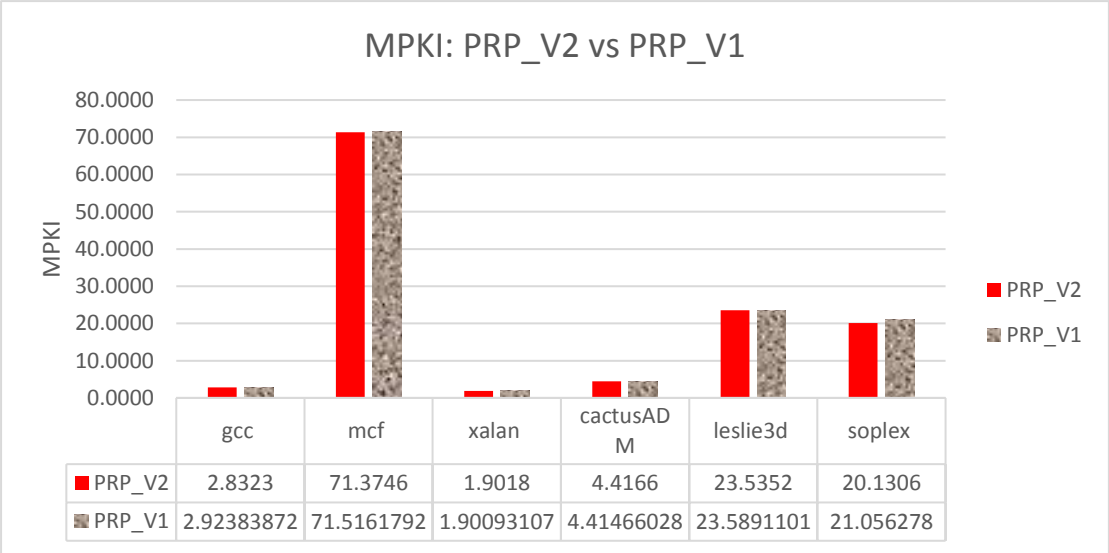
	gcc	mcf	xalan	cactusADM	leslie3d	soplex
LRU	3.2854702	80.9837007	1.89501424	4.82937644	23.8339332	21.6425986
PRP_V1	2.92383872	71.5161792	1.90093107	4.41466028	23.5891101	21.056278
LFU	3.36224355	77.394561	1.8432581	4.79319342	23.8174889	20.6795846
SRRIP	2.99764958	76.9162239	1.77231055	4.75829035	23.8591493	20.2319485

Figure4. MPKI: LRU vs PRP_V1 vs LFU vs SRRIP, LLC is 2MB



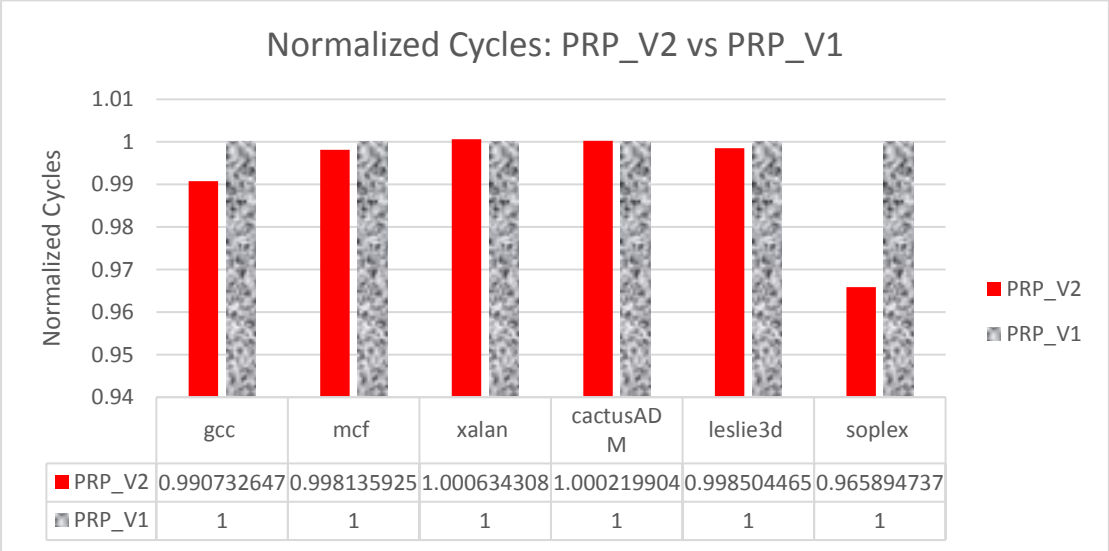
	gcc	mcf	xalan	cactusADM	leslie3d	soplex
LRU	1	1	1	1	1	1
PRP_V1	0.94868047	0.91001073	1.02385519	0.97453694	1.01076123	0.9919934
LFU	1.00792044	0.96905854	0.99684373	0.99717856	0.9997416	0.97305956
SRRIP	0.96479097	0.96562423	0.99130216	0.99606152	1.00212494	0.9687855

Figure5. Normalized Cycles: LRU vs PRP_V1 vs LFU vs SRRIP (Normalize to LRU), LLC is 2MB



	gcc	mcf	xalan	cactusADM	leslie3d	soplex
PRP_V2	2.8323	71.3746	1.9018	4.4166	23.5352	20.1306
PRP_V1	2.92383872	71.5161792	1.90093107	4.41466028	23.5891101	21.056278

Figure6. MKPI: PRP_V2 vs PRP_V1, LLC is 2MB



	gcc	mcf	xalan	cactusADM	leslie3d	soplex
PRP_V2	0.990732647	0.998135925	1.000634308	1.000219904	0.998504465	0.965894737
PRP_V1	1	1	1	1	1	1

Figure7. Normalized Cycles: PRP_V2 vs PRP_V1 (Normalize to PRP_V1), LLC is 2MB

By the way, although the page size of zsim is 8KB, I think it is fine that I choose 4KB as the page index like the paper setting. After all, taking 4KB continuous memory as one-page index just means that 64 lines sharing one N_L information.

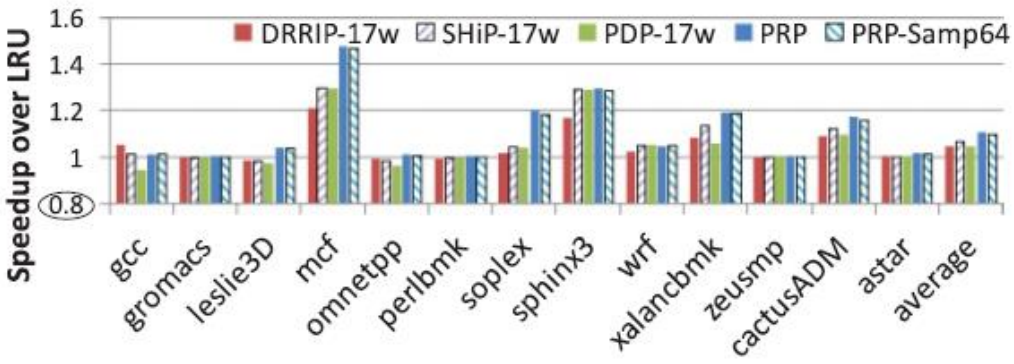
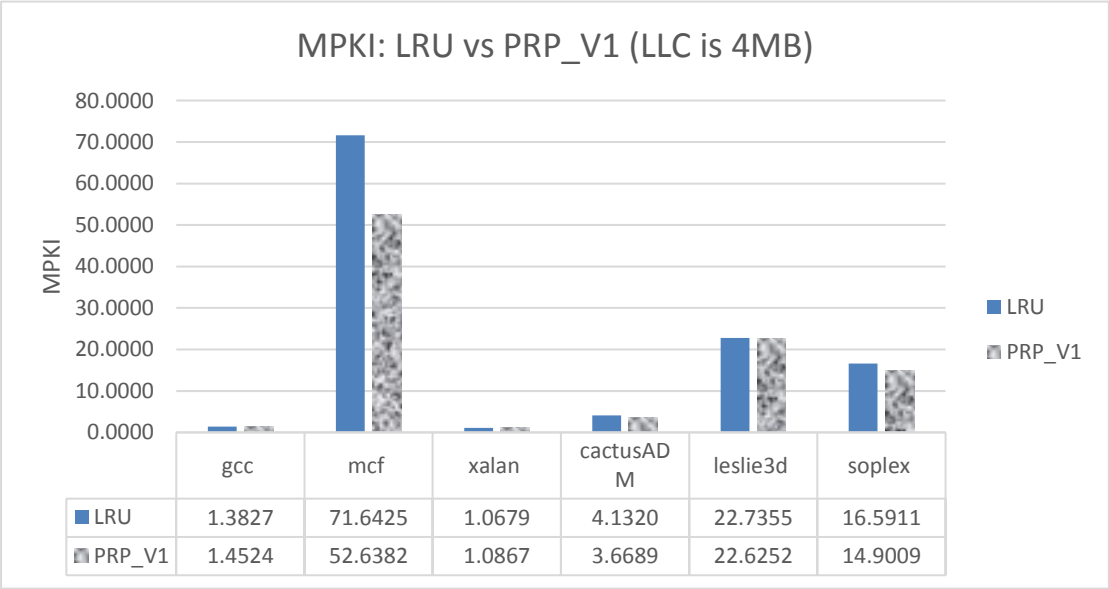


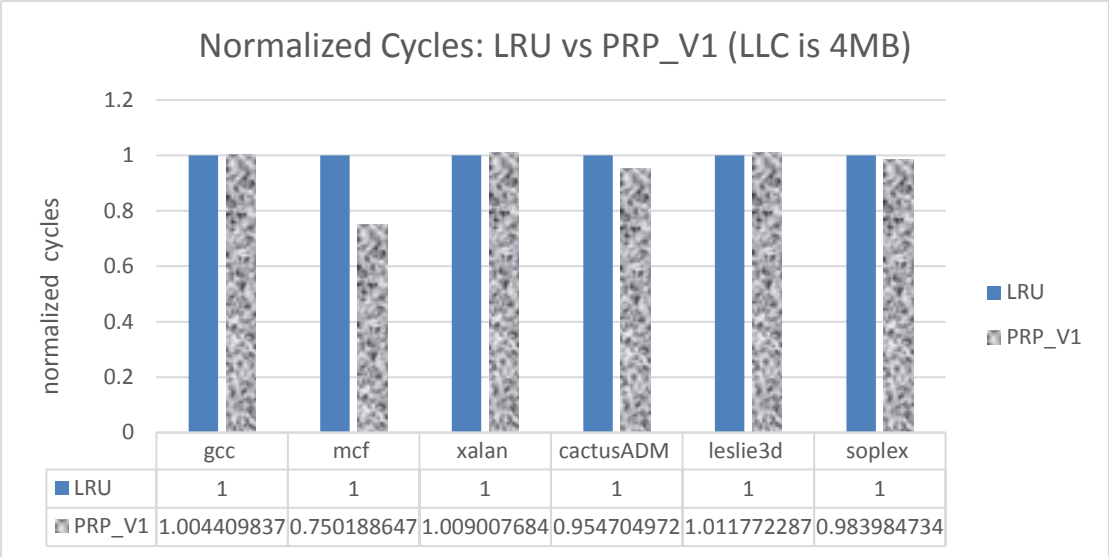
Figure8. Performance of various replacement policies from paper's figure8 [1]

Except for the 2 reasons mentioned above, I think the performance difference between my PRP and paper's PRP would be caused from parameters. LLC is 4MB in the paper. Thus, I tried to run PRP_V1 under 4MB LLC compared to LRU. The performance comparison looks better when I use 4MB LLC. The pattern mcf gains 25% running time reduction, and average 5.3% running time reduction. The reason would be there are more spaces for short-term reused blocks compared to 2 MB LLC. In PRP, mixed reused blocks, which is combined short-term and midterm-term, have higher P_L^{hit} than that of pure short-term reused blocks at some distribution records.



	gcc	mcf	xalan	cactusADM	leslie3d	soplex
LRU	1.3827	71.6425	1.0679	4.1320	22.7355	16.5911
PRP_V1	1.4524	52.6382	1.0867	3.6689	22.6252	14.9009

Figure9. MPKI: LRU vs PRP_V1, LLC is 4MB



	gcc	mcf	xalan	cactusADM	leslie3d	soplex
LRU	1	1	1	1	1	1
PRP_V1	1.004409837	0.750188647	1.009007684	0.954704972	1.011772287	0.983984734

Figure10. Normalized Cycles: LRU vs PRP_V1 (Normalize to LRU), LLC is 4MB

Besides, I would like to try another way to update N_L. I modified one version PRP_V3, which updates N_L only when cache hit case, but it works little worse, even use collected N_L data for it to execute patterns again. LLC has higher misses in PRP_V3 because N_L can't reflect higher frequency accessed cache-block but with high miss rate. If update N_L of this type block as PRP_V1 or V2, it can help this block has higher priority to stay in cache later.

PRP_V3	**Pattern: gcc**	**PRP_V2**
132034262	**Total cycles**	128569682
100003983	**instrs**	100010799
237491	**LLC mGETS**	214152
60305	**LLC mGETXIM**	69104

Table3. Simple Comparison between PRP_V3 and PRP_V2, LLC is 2MB

There is one interesting observation that the patterns with more LLC accessed times would bring more improvement on PRP. For example, mcf pattern has the largest times to access LLC compared to other 5 patterns. Large LLC accessed record for N_L may bring better

estimate for P_L^{hit}. Table4 provides the information of LLC accesses times in PRP_V1.

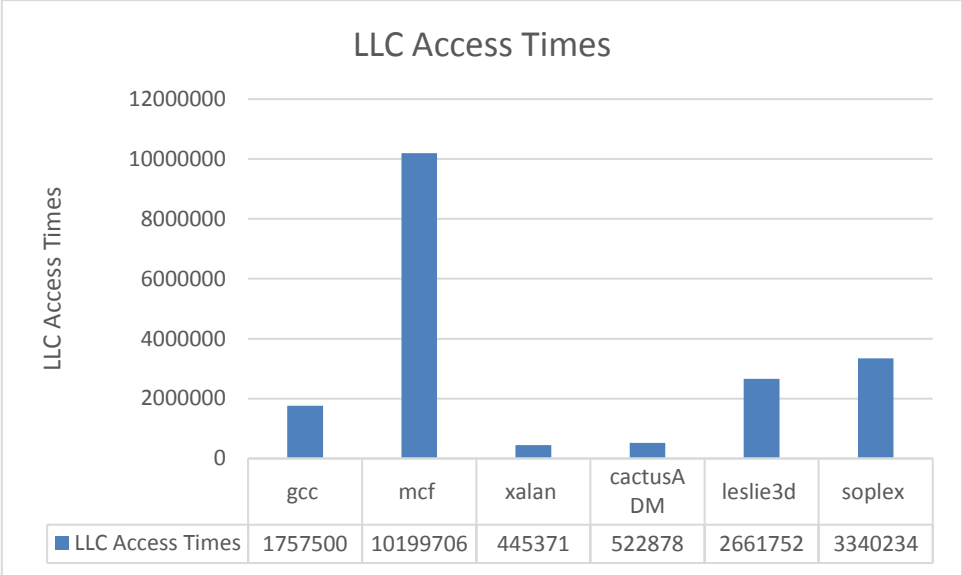


Figure11. Access Times of LLC in PRP_V1, LLC is 2MB

Last, the cost of original PRP is significant. For memory cost, there are 2^{20} pages, each page has 6 bins and 64 timestamps. One bin has 4-bit counter, and one timestamp is 10bit. For reused distance and timestamp storage, it cost 2^{20}(4*6+10*64 bit) is about 2^{27} Bytes, and the size for timestamps of all cache set is (4*2^{20}/16)*10bit is about 2^{19} Bytes. Totally, the space cost of PRP is 2^{27} Bytes, 128MB Bytes. For the time cost, although look up array is O(1) to access the storage data and calculate P_L^{hit} of each line, but each array store/load will cost one DRAM communication time. Consequently, Time and space cost is the main issue of PRP.

Conclusion:

In this term project, I read and implement the original version of PRP, measure its

performance under zsim. Under 4MB LLC, I get 25% running time reduction in mcf pattern compared to LRU policy and average 5.3% running time reduction, and I found that more times to access LLC and more reused distance information can enhance the performance of PRP. To pursue the better speedup performance like the performance of the paper, how to increase the accuracy of N_L is the goal for further improvement.

Reference:

[1] Subhasis Das, Tor M. Aamodt, and William J. Dally. 2015. Reuse distance-based probabilistic cache replacement. ACM Trans. Archit. Code Optim. 12, 4, Article 33 (October 2015), 22 pages. DOI: http://dx.doi.org/10.1145/2818374

[2] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. 5, 2 (1966), 78–101.

Appendix (Simulation code of PRP_V2):

```
#ifndef PRP_REPL_H_
#define PRP_REPL_H_

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

#include "repl_policies.h"
// the number of asscociated ways
#define PRP_WAY_VALUE 16
//P_hit(t): 6bins value [bin0 bin1...bin5]
#define PRP_PTable0 15
#define PRP_PTable1 14
#define PRP_PTable2 12
#define PRP_PTable3 10
#define PRP_PTable4 9
#define PRP_PTable5 1

// OUTPUT_LOG = 1, dump N_status to dump_test at the result time
// INPUT_LOG = 1, load dump_test at initial time
// Only one of them is 1 at one time, default is OUTPUT_LOG = 0, INPUT_LOG = 0
#define OUTPUT_LOG      0
#define INPUT_LOG       1

//the result access time for each pattern dump "dramBinArray"
//for b_zip
```

```
#define SAVE_COUNT 1950000

//for gcc
//#define SAVE_COUNT 1740000

//for xalan
//#define SAVE_COUNT 430000

//for cactus
//#define SAVE_COUNT 510000

//for leslie3d
//#define SAVE_COUNT 2630000

//for mcf
//#define SAVE_COUNT 10190000

//for soplex
//#define SAVE_COUNT 3310000


// PRP
class PRPReplPolicy : public ReplPolicy {
    protected:
        // add class member variables here
            uint32_t*     set_TS;       //one timestamp for each set, unit:0-1023, M in paper
            uint32_t      PTable[6]; //15-14-12-10-9-1, P_hit(t) in paper
            uint64_t*     idArray;      //each line to which mem, to record Virtual line Address(lineAddr), VA[63:6]
            // for dramBinArray and dramTsArray, first index: VA[31:12] which is page index.
            //assume page size is 4KB = VA[11:0].
            uint32_t**    dramBinArray;    //each element with 6 bins for 6 intervals, each elemnet is N_L(i) in paper.
            uint32_t**    dramTsArray;      //each element with 64 timeStamps for 64 cache blocks,
            // 4KB Page includes 64 cache blocks(one block is 64 Byte)
            uint32_t      numLines;           //the number of cache lines in LLC
            uint64_t      new_insert_lineAddr;      // to temp record the line address of new inserted cache block
            uint32_t      log_counter;      // to global record how many times is cache accessed. For dump/load data counter.

            //function for normalizing overflow 4-bit counter, which is applied on 6 bins of one dramBinArray[pageAddr].
            void half_value_arrayN(uint32_t pageAddr){
                    (dramBinArray[pageAddr][0])/=2;
                    (dramBinArray[pageAddr][1])/=2;
                    (dramBinArray[pageAddr][2])/=2;
                    (dramBinArray[pageAddr][3])/=2;
                    (dramBinArray[pageAddr][4])/=2;
                    (dramBinArray[pageAddr][5])/=2;
            }

    public:
        // add member methods here, refer to repl_policies.h
            explicit PRPReplPolicy(uint32_t _numLines) : numLines(_numLines) {
                    dramBinArray = gm_calloc<uint32_t*>(1024*1024);    // VA[31:12]=20bits, there is 2^20 pages.
                    dramTsArray = gm_calloc<uint32_t*>(1024*1024);     // VA[63:32] is truncated.
                    for(uint32_t i = 0; i < (1024*1024); i++){
                            dramBinArray[i] = gm_calloc<uint32_t>(6);       // 6 bins for each page
                            dramTsArray[i] = gm_calloc<uint32_t>(64);       // each page includes 64 cache blocks
                            for(uint32_t j = 0; j < 6; j++)
                                    dramBinArray[i][j] = 0;
                            for(uint32_t j = 0; j < 64; j++)
                                    dramTsArray[i][j] = 0;
                    }
                    set_TS =    gm_calloc<uint32_t>(numLines/PRP_WAY_VALUE);
                    // the number of sets is lines/16 way-associated
                    idArray = gm_calloc<uint64_t>(numLines);
                    for(uint32_t i = 0; i < numLines; i++){
                            idArray[i] = 0;
                    }
                    for(uint32_t i = 0; i < numLines/PRP_WAY_VALUE; i++){
                            set_TS[i] = 0;
                    }
                    PTable[0] = PRP_PTable0;
                    PTable[1] = PRP_PTable1;
```

```
        PTable[2] = PRP_PTable2;
        PTable[3] = PRP_PTable3;
        PTable[4] = PRP_PTable4;
        PTable[5] = PRP_PTable5;
        new_insert_lineAddr = 0;
        log_counter = 0;
    }
    ~PRPReplPolicy() {
        for(uint32_t i = 0; i < (1024*1024); i++){
            gm_free(dramBinArray[i]);
            gm_free(dramTsArray[i]);
        }
        gm_free(dramBinArray);
        gm_free(dramTsArray);
        gm_free(idArray);
        gm_free(set_TS);
    }

    void update(uint32_t id, const MemReq* req) {
        uint64_t reqAddr = ((req->lineAddr)<<6)&0x00000000FFFFFFFF;
        //req->lineAdd = VA[63:5], reqAddr = VA[31:0] align to line
        uint32_t pageAddr = (reqAddr>>12);    //pageAddr = VA[31:12]
        uint32_t line_index = ((uint32_t)(req->lineAddr))&0x0000003F;
        //line_index = VA[11:6], one page with 64 cache blocks
#if INPUT_LOG
        if(log_counter==0){    //load N_L(i) into dramBinArray from the file called dump_test
            string fileName = "dump_test";
            fstream fp(fileName, ios::in);
            if (!fp) {
                cout << "Fail to open file: " << fileName << endl;
            }
            string tmpLine;
            uint32_t log_page_num = 0;
            uint32_t N_0, N_1, N_2, N_3, N_4, N_5;
            while (getline(fp, tmpLine)) {
                if(log_page_num < 1024*1024){    //protect for the unexpected last endl or illegal input
                    stringstream ssin(tmpLine);
                    ssin >> N_0 >> N_1 >> N_2 >> N_3 >> N_4 >> N_5;
                    dramBinArray[log_page_num][0] = N_0;
                    dramBinArray[log_page_num][1] = N_1;
                    dramBinArray[log_page_num][2] = N_2;
                    dramBinArray[log_page_num][3] = N_3;
                    dramBinArray[log_page_num][4] = N_4;
                    dramBinArray[log_page_num][5] = N_5;
                }
                log_page_num++;
            }
            cout << "load source done" << endl;
            fp.close();
        }
#endif
        uint32_t set_index = id/PRP_WAY_VALUE;
        set_TS[set_index]+=1;         //update the specific set's TimeStamp
        if(set_TS[set_index] >= 1024){
           set_TS[set_index] = 0; //0-1023, 10 bit accuracy
        }
        uint32_t set_t =set_TS[set_index];
        uint32_t line_t = dramTsArray[pageAddr][line_index];
        uint32_t diff_t = 0;
        //diff_t <-- calc re-ref time period this time = set_TS - line_TS:
        if(line_t > set_t){
            diff_t = (1024 + set_t) - line_t;    //wrap around case, becasue set TS is overflow back to 0.
        }
        else{
            diff_t = set_t - line_t;
        }

        //update NL
        if(diff_t < 16){
            dramBinArray[pageAddr][0]+=1;        //re-ref time: 1-15
            if(dramBinArray[pageAddr][0] == 16)
```

```
                        half_value_arrayN(pageAddr);
                }
                else if(diff_t < 32){
                        dramBinArray[pageAddr][1]+=1;       //re-ref time: 16-31
                        if(dramBinArray[pageAddr][1] == 16)
                                half_value_arrayN(pageAddr);
                }
                else if(diff_t < 64){
                        dramBinArray[pageAddr][2]+=1;       //re-ref time: 32-63
                        if(dramBinArray[pageAddr][2] == 16)
                                half_value_arrayN(pageAddr);
                }
                else if(diff_t < 128){
                        dramBinArray[pageAddr][3]+=1;       //re-ref time: 64-127
                        if(dramBinArray[pageAddr][3] == 16)
                                half_value_arrayN(pageAddr);
                }
                else if(diff_t < 256){
                        dramBinArray[pageAddr][4]+=1;       //re-ref time: 128-255
                        if(dramBinArray[pageAddr][4] == 16)
                                half_value_arrayN(pageAddr);
                }
                else{
                        dramBinArray[pageAddr][5]+=1;       //re-ref time: 256-inf
                        if(dramBinArray[pageAddr][5] == 16)
                                half_value_arrayN(pageAddr);
                }
                dramTsArray[pageAddr][line_index] = set_TS[set_index];
                //update line TS <-- set_TS
                log_counter++;
#if OUTPUT_LOG
                if(log_counter == SAVE_COUNT){    //dump N_L(i) in dramBinArray to the file called dump_test
                        string fileName = "dump_test";
                      fstream fp;
                        fp.open(fileName, ios::out);
                        if (!fp) {
                        cout << "Fail to open file: " << fileName << endl;
                        }
                        for(uint32_t t_i = 0; t_i < (1024*1024); t_i++){
                                for(uint32_t t_j = 0; t_j < 6; t_j++){
                                        fp << dramBinArray[t_i][t_j];
                                        if(t_j < 5)
                                                fp << " ";
                                        else
                                                fp << endl;    //next line for next page
                                }
                        }
                        cout << "log done" << endl;
                        fp.close();
                }
#endif
        }

        void replaced(uint32_t id) {
                idArray[id] = new_insert_lineAddr;    //record access line-addr to idArray[id]
                return;
        }

        template <typename C> inline uint32_t rank(const MemReq* req, C cands) {
        uint32_t bestCand = 0;    //return cache id
                uint32_t test_array[PRP_WAY_VALUE];    //to save calculated P_L^hit of these candidates in one set
                auto si = cands.begin();    //the start line id of this set
                uint32_t set_index = (*si)/PRP_WAY_VALUE;
                uint32_t start_line_id = (*si);
                uint32_t line_id = 0;

                //calc every candidate's P_L^hit value, in each line in this set
                for (auto ci = cands.begin(); ci != cands.end(); ci.inc()) {
                        line_id = *ci;
                        uint32_t set_t =    set_TS[set_index];    //Set_TS
                        uint64_t savedLineAddr = idArray[line_id];    // VA[63:6]
```

```
                        uint64_t reqAddr = ((savedLineAddr)<<6)&0x00000000FFFFFFFF;    //VA:[31:0] align line
                        uint32_t pageAddr = (reqAddr>>12);    //VA[31:12]
                        uint32_t line_index = ((uint32_t)(savedLineAddr))&0x0000003F; //VA[11:6], 64 lines in 1 page

                        uint32_t line_t = dramTsArray[pageAddr][line_index];    //Line_TS
                        uint32_t diff_t = 0;
                        //calc re-ref time period this time:
                        if(line_t > set_t){
                                diff_t = (1024 + set_t) - line_t;
                        }
                        else{
                                diff_t = set_t - line_t;
                        }

                        //calc P_L^hit in 6 accumulated terms
                        //calc. progress: with smaller re-ref time t, the more terms need to add.
                        uint32_t sum_NL_P = 0;    // numerator of Probability calculator unit...eq5 in paper
                        uint32_t sum_NL =0;    // denominator of Probability calculator unit...eq5 in paper
                        sum_NL_P += (dramBinArray[pageAddr][5]*PTable[5]);
                        sum_NL += dramBinArray[pageAddr][5];
                        if(diff_t < 256){
                                sum_NL_P += (dramBinArray[pageAddr][4]*PTable[4]);
                              sum_NL += dramBinArray[pageAddr][4];
                        }
                        if(diff_t < 128){
                                sum_NL_P += (dramBinArray[pageAddr][3]*PTable[3]);
                              sum_NL += dramBinArray[pageAddr][3];
                        }
                        if(diff_t < 64){
                                sum_NL_P += (dramBinArray[pageAddr][2]*PTable[2]);
                              sum_NL += dramBinArray[pageAddr][2];
                        }
                        if(diff_t < 32){
                                sum_NL_P += (dramBinArray[pageAddr][1]*PTable[1]);
                              sum_NL += dramBinArray[pageAddr][1];
                        }
                        if(diff_t < 16){
                                sum_NL_P += ( dramBinArray[pageAddr][0]*PTable[0]);
                              sum_NL +=    dramBinArray[pageAddr][0];
                        }
                        if(sum_NL != 0)    //protect denominator == 0
                              test_array[line_id - start_line_id] = sum_NL_P/sum_NL;
                        else
                                test_array[line_id - start_line_id] = 0;
                  }

                  // choose the smallest P_L^hit value in test_array, has lowest hit P in the future
                  uint32_t min_value = 1536; // 16*16*6/1 = 1536, P_L^hit value can't larger than this value
                  bestCand = 0;    //if all candidates have the same value, return the left one
                  for(uint32_t i = 0; i < PRP_WAY_VALUE; i++){
                        if(test_array[i] < min_value){
                              bestCand = i;    //0-15, offset index in this set
                                min_value = test_array[i];
                        }
                  }
                  bestCand += start_line_id; //return to line_addr
                  new_insert_lineAddr = req->lineAddr;    //record access line-addr to cache idArray[id]
                  return bestCand;
            }
            DECL_RANK_BINDINGS;
};
#endif // PRP_REPL_H_
```