Master's Thesis

Pseudo Re-Reference Interval Prediction for Last-Level Cache Replacement

Taeho Lim

Department of Electrical Engineering

Graduate School of UNIST

2019

Pseudo Re-Reference Interval Prediction for Last-Level Cache Replacement

Taeho Lim

Department of Electrical Engineering

Graduate School of UNIST

Pseudo Re-Reference Interval Prediction for Last-Level Cache Replacement

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Taeho Lim

Month/Day/Year of submission

Approved by

____

Advisor

Seong-Jin Kim

Pseudo Re-Reference Interval Prediction for Last-Level Cache Replacement

Taeho Lim

This certifies that the thesis of Taeho Lim is approved.

12 / 11 / 2018

signature

Advisor: Seong-Jin Kim

signature

Seokhyeong Kang: Thesis Committee Member #1

signature

Woongki Baek: Thesis Committee Member #2

Abstract

For the last decade, many modern replacement policies for last-level cache (LLC) adopted Static Re-reference Interval Prediction (SRRIP) as their base algorithm. In the LLC, SRRIP outperforms other traditional replacement policies like Least-Recently Used (LRU). SRRIP works with a few bits of counter, called Re-Reference Prediction Value (RRPV), but we find that RRPV can be implemented with a binary tree.

In this thesis, we propose a new cache replacement policy, Pseudo Re-Reference Interval Prediction (PRRIP). Our proposed PRRIP mimics SRRIP, so PRRIP outperforms other replacement policies such as LRU. Moreover, we find that PRRIP becomes more resistant to non-temporal data access pattern than SRRIP by using binary tree. In terms of overhead, we halve the hardware cost to implement PRRIP compared to SRRIP. Our experimental results show that PRRIP achieves 1.26% speedup over LRU while SRRIP gets 0.53% speedup over LRU for single-core workloads. For multi-core workloads, our experimental results show that the performance difference between PRRIP and SRRIP is less than 0.3%.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank all of the people who helped make this dissertation possible.

Especially, I would like to express my deep appreciation to my academic advisor Dr. Seokhyeong Kang. Not only for advising me, but also being my role model. All of the work that I accomplished during M.S. course have not been possible without his guidance. Also, he showed me a wider vision of this area.

I would like to thank my colleagues in CAD & SoC Design Lab, Yesung Kang, Seungwon Kim, Jaemin Lee, Sunmean Kim, Jaewoo Kim, Sanggi Do, Mingyu Woo, Yoonho Park, Daeyeon Kim, Sunghoon Kim and Sungyun Lee. The knowledge they shared to me inspired me and the time shared with them support me up when I got tough times.

I would also like to express my appreciation to thesis committee members, Prof. Seong-Jin Kim and Prof. Woongki Baek for taking their time out to review and evaluate my research work.

Lastly, I have no words to express my unlimited appreciation to my family, Changsook Park and Taeyong Lim for all the support they have provided me. I dedicate this thesis to them.

Vita

1992	Born, Busan, South Korea
2017	B.S., Electronics Engineering, Pusan National University, Busan, South Korea

- **Taeho Lim**, Yoonho Park, Woongki Baek and Seokhyeong Kang, “PRRIP : Pseudo Re-Reference Interval Prediction for Last-Level Cache Replacement”, *Proc. IEEE/ACM Design Automation Conference*, 2019, (submitted).

- Yesung Kang, Yoonho Park, Sunghoon Kim, **Taeho Lim** and Seokhyeong Kang, “Analysis and Solution of CNN Accuracy Reduction over Channel Loop Tiling”, *Proc. IEEE/ACM Design Automation Conference*, 2019, (submitted).

- Sunmean Kim, **Taeho Lim** and Seokhyeong Kang, “An Optimal Gate Design for the Synthesis of Ternary Logic Circuits”, *Proc. IEEE/ACM Asia and South Pacific Design Automation Conference*, 2018, pp. 476-481.

Chapter I

Introduction

Limitations in memory speed impose limitations on the evolution of computing systems. Computing capability is increasing faster than memory speed, so this gap causes the memory wall problem [1], [2]. Especially, the last-level cache (LLC) is the last on-chip memory buffer before off-chip memory is accessed, so large disparity of speed exists between LLC and DRAM. Access to off-chip memory is costly in both time and energy cost. Thus, reducing the number of off-chip memory accesses increases computing speed and reduces its energy cost. To achieve this goal, LLC replacement policy is responsible. The replacement policy is about determining which data to keep and which to evict. Appropriate selection of data that should stay reduces the number of off-chip memory accesses, and appropriately released data frees storage space for other data. Currently, LLC occupies up to 50% of entire chip area [5], so data of LLC must be managed efficiently.

LLC differs from upper-level cache in several points [3], [4]. First of all, LLC has a large miss penalty. Second, LLC is in a polluted environment: the ratio of access counts to memory size is much smaller than in upper-level cache, so LLC has less opportunity than upper-level caches to replace useless data. LLC keeps many data that are not re-referenced anymore [6]. Third, LLC has high associativity. A cache that has higher associativity has more options when it evicts data. At the same time, it requires more information to decide which option to select compared with a cache with low associativity; e.g., the commonly-used Least Recently Used (LRU) policy stores the access order of blocks. Therefore, as associativity increases, the overhead to save access order increases rapidly. Consequently, high associativity complicates the task of designing a replacement policy for LLC. Finally, LLC is shared by many processing units, so threads on different tasks may interfere with it.

Re-Reference Interval Prediction (RRIP) [7] is a practical replacement policy to compensate for the characteristics of LLC. Many replacement policies use RRIP as their base algorithm [8], [9], [10], [11]. However, recent trends of computing such as multi-core, aggressive prefetching and dynamic replacement policy increase the overhead of the replacement policy for LLC. In this thesis, we propose a new replacement policy, Pseudo Re-Reference Interval Prediction (PRRIP) for LLC replacement. Our proposed PRRIP mimics the behavior of RRIP with a binary tree. PRRIP consumes less than half of the overheads of RRIP while maintaining the performance of RRIP.

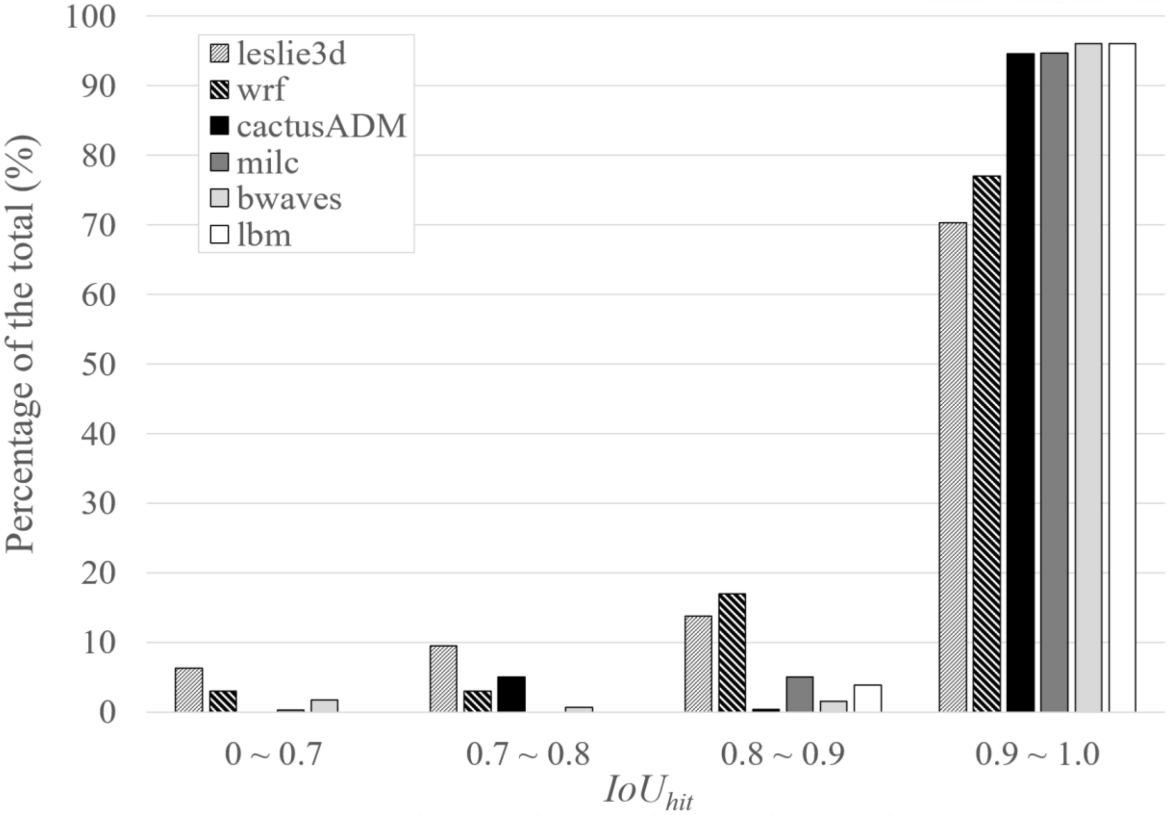


Figure 1. The behavioral similarity between PRRIP and RRIP. On average, calculated IoU_{hit} for six benchmarks is 0.97.

Figure 1 shows how many cache hits are matched between PRRIP and RRIP on six benchmarks from SPEC CPU 2006. To evaluate the similarity between PRRIP and RRIP, we use intersection over union of the hit address (IoU_{hit}). IoU_{hit} is calculated as follows.

$$IoU_{hit} = \frac{\#\ of\ common\ hits\ in\ PRRIP\ and\ RRIP}{total\ \#\ of\ hits}$$

Increase in IoU_{hit} indicates increase in agreement of the two policies. If two policies have exactly the same hit, IoU_{hit} becomes 1.0. We measured IoU_{hit} every 50,000 accesses on LLC. Then, we show the distribution of IoU_{hit} as a histogram. On average, 88% of entire simulation points yield IoU_{hit} more than 0.9 for each benchmark and calculated IoU_{hit} equals 0.97. Thus, it is reasonable to say that PRRIP acts much like RRIP. Moreover, we apply our PRRIP algorithm to a dynamic replacement policy in Chapter V. Dynamic policies which are based on RRIP [10], [11] also can be implemented with PRRIP.

Our main contributions are summarized as follows.

- We introduce a PRRIP cache replacement policy that mimics RRIP algorithm with less than half the overhead that is required to implement RRIP.

- We identify considerations such as reward of re-reference and the initial order of newly-inserted line, when PRRIP is implemented.

- We evaluate PRRIP with SPEC CPU 2006 benchmarks. Our proposed replacement policy achieved more instructions per cycle (IPC) than other replacement policies.

- We extend PRRIP to dynamic replacement policies which are based on RRIP algorithm. Then, we compare the performance of ours with the original works.

The remainder of this thesis is organized as follows. Chapter II provides an overview of related works in the literature. Chapter III describes our proposed replacement policy. Chapter IV presents a description of our experimental setup and results. Chapter V extends PRRIP to a dynamic replacement policy that uses RRIP, then compare the performance between them. Chapter VI concludes our work.

Chapter II

Background

We build this work on previous research of the cache replacement policy. To give proper background, we review some closely related works.

A. Re-reference Interval Prediction (RRIP)

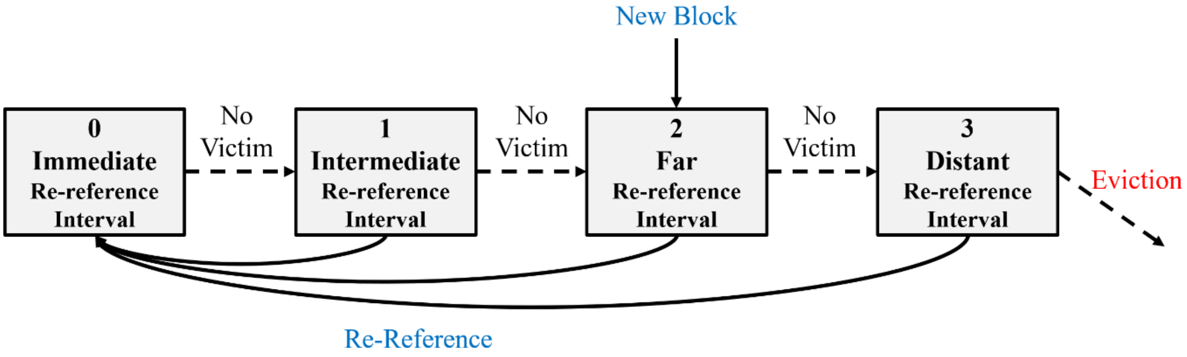


Figure 2. The state diagram of RRPV

RRIP predicts the usage interval of blocks with re-reference prediction value (RRPV). A few bits of counter represent the RRPV for each block. Figure 2 shows a state diagram of 2-bit RRPV. As RRPV of a block approaches its maximum value, that block is predicted to have distant usage interval. Conversely, if the RRPV of a block approaches to zero, the block is predicted to have an immediate re-reference interval. RRIP works with two key ideas. First, RRIP moves a re-referenced line to the safest location. This makes reused block safe and the other blocks to age. Second, when a new block arrives, RRIP inserts the block near the victim. These ideas enable RRIP to resist bursts of references to non-temporal data and to adapt to upcoming working set. The overhead of SRRIP equals $2n$ (in 2-bit RRPV) or $3n$ bits (in 3-bit RRPV) per cache set. RRIP is called Static RRIP (SRRIP) and Dynamic RRIP (DRRIP) depending on whether Set Dueling [12] is used.

B. Binary Tree-Based Replacement Policies

(a) Tree-Based PseudoLRU (PLRU)

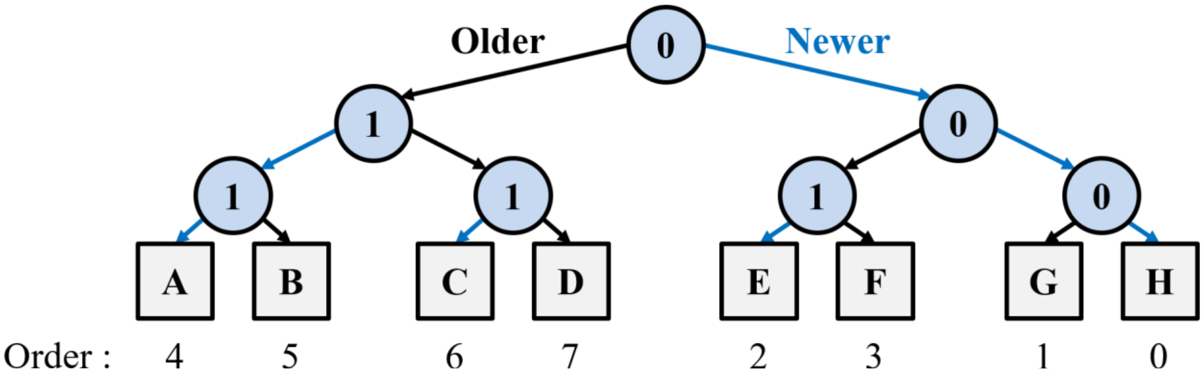


Figure 3. An example of PLRU tree. Associativity is 8.

Tree-based PLRU [13] is a cost-efficient alternative to LRU. PLRU uses a binary tree in which leaf nodes are blocks in a set. Two adjacent nodes are connected to an upper node. It contains a single bit indicator to record which child node is the older one. The oldest data become the victim along the binary tree. As a node is closer to the root node, a block, which is pointed by that node receives lower order. Figure 3 illustrates an example of 8-way associative PLRU tree. If the next access misses in the cache, 'D' becomes a victim. For an n-way associative cache, PLRU consumes $n-1$ bits per cache set, whereas while LRU consumes $n\log_2 n$ bits per cache set.

(b) Minimal Disturbance Placement and Promotion (MDPP)

MDPP [14] is the improved version of PLRU based on principles of minimal disturbance placement and promotion. 'Placement' (also known as insertion) is an update of the replacement state so that the newly-inserted block has a specific order in a set. 'Promotion' is an update of the replacement state so that the re-referenced block has a specific order in a set. When a promotion or a placement occurs, PLRU modifies the values of all nodes related to the re-referenced block. This process also changes the order of blocks near the referenced line; this reordering is called 'disturbance'. MDPP minimizes this disturbance, when the promotion or the placement occur.

C. Dynamic Replacement Policies

Dynamic replacement policy is a recently popular approach to achieve real-time change in the behaviors of a policy according to the characteristics of a workload [6], [8], [9], [15], [16], [17], [18], [19], [20]. For example, DIP [12] and DRRIP [7] change the placement position of the new block. SHiP [8] and SDBP [6] use Program Counter (PC) information to predict whether the line is reused.

To combine multiple features of a workload, MPPPB [18], Perceptron [19] use perceptron learning. Hawkeye [15] learns from past decisions of Belady’s algorithm [21]. PACMan [9] and Harmony [22] are prefetch-aware dynamic replacement policies. However, to learn the features of workloads during run-time, these policies require more overhead to save information such as PC, which core requested a line and prefetch status.

Chapter III

Pseudo Re-Reference Interval Prediction (PRRIP)

A. Binary Tree Based Re-Reference Prediction

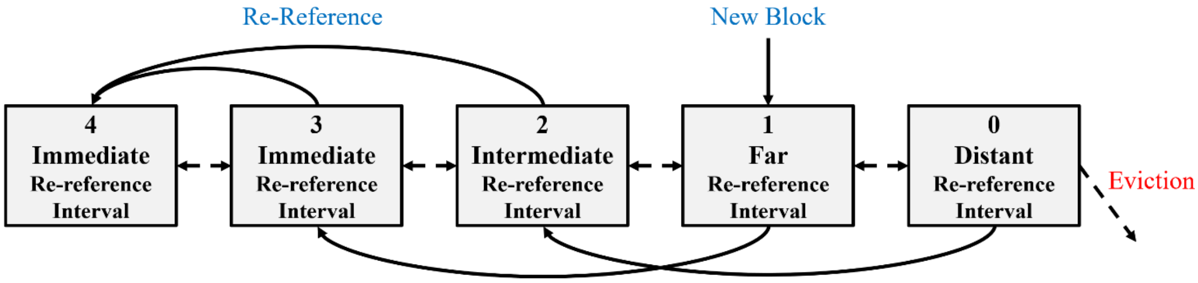


Figure 4. State diagram according to the number of shells.

We suggest a new binary tree-based replacement policy that mimics SRRIP. Teran et al. formalize the protection behavior of PLRU tree [14]. Each node of the binary tree protects the opposite sides of the lower nodes. When a block is protected by a node, we state that the block got a *shell*. The block can be protected by multiple shells. Figure 5 shows the number of shells protecting each block and where they are located. A total number of shells applied to a block represents the degree of safety, as RRPV does. With appropriate promotion and insertion, RRPV can be represented using shells (Figure 4). Empirically, we found that applying two more shells from the bottom is appropriate for reward of promotion when the associativity of a cache equals 16 (most common in LLC). In case of placement, it is best to give newly-inserted block one shell. Details about promotion and insertion of PRRIP will be provided in Chapter III-B and the Chapter III-C.

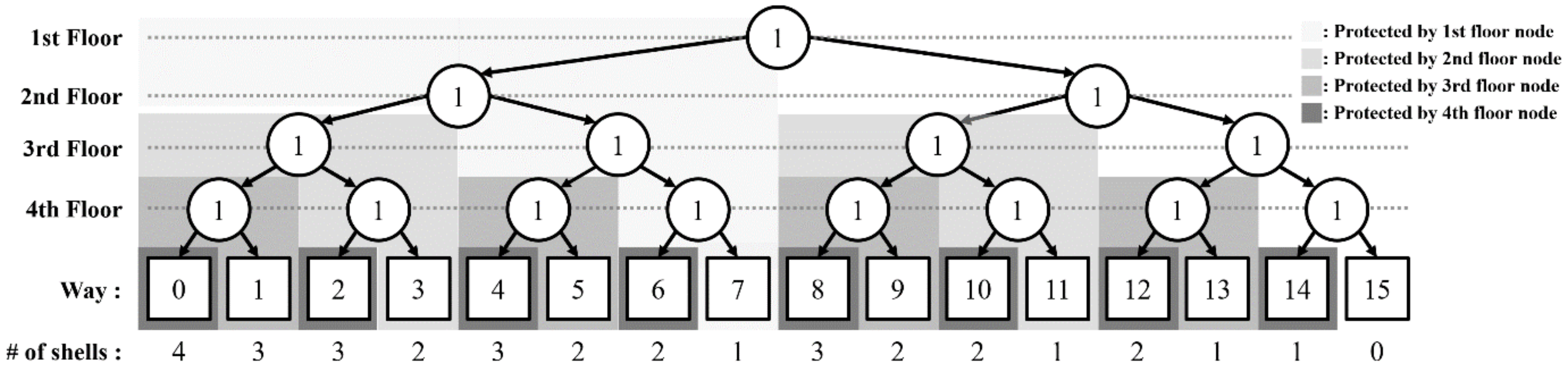
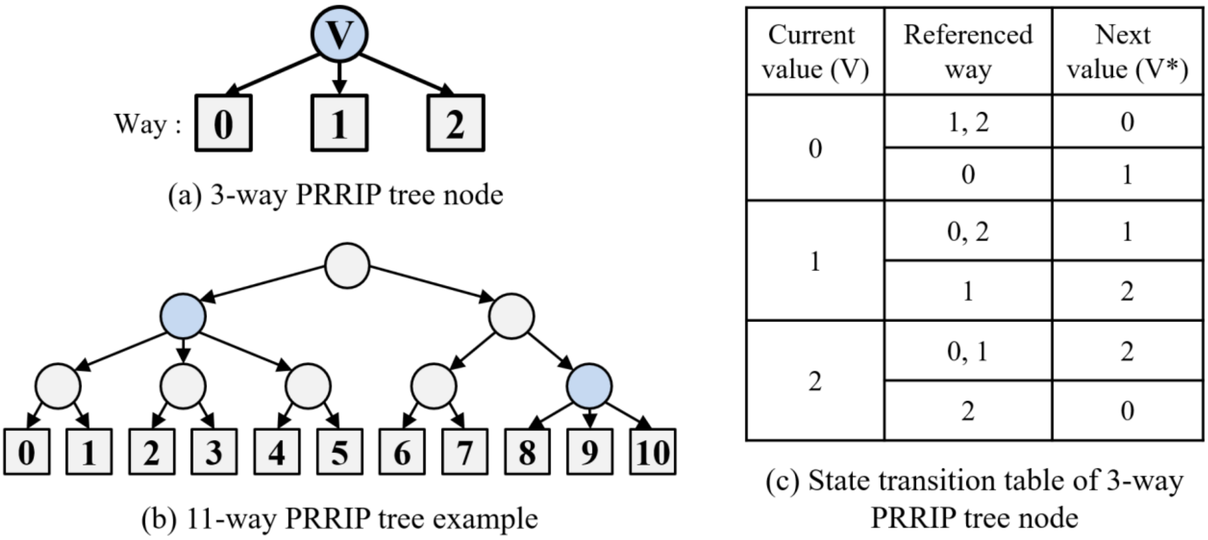

1st Floor
2nd Floor
3rd Floor
4th Floor
Way :
# of shells :
1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
4
3
3
2
3
2
2
1
3
2
2
1
2
1
1
0
: Protected by 1st floor node
: Protected by 2nd floor node
: Protected by 3rd floor node
: Protected by 4th floor node

Figure 5. Example PLRU tree. Associativity is 16. For each block, the number of shells is represented.

If associativity of a cache is not power of two, we can use a 3-way PRRIP tree node (3-way node) to construct the binary tree (Figure 6(a)). In this case, we can construct the tree by connecting the pair of two as usual, if a line is left, we can connect it with 3-way node. Then repeat this until the only root node remains. When constructing the tree with 3-way node, 3-way node must be spread evenly over the entire tree. With this method, the maximum number of shells for each line can be maintained.



(a) 3-way PRRIP tree node

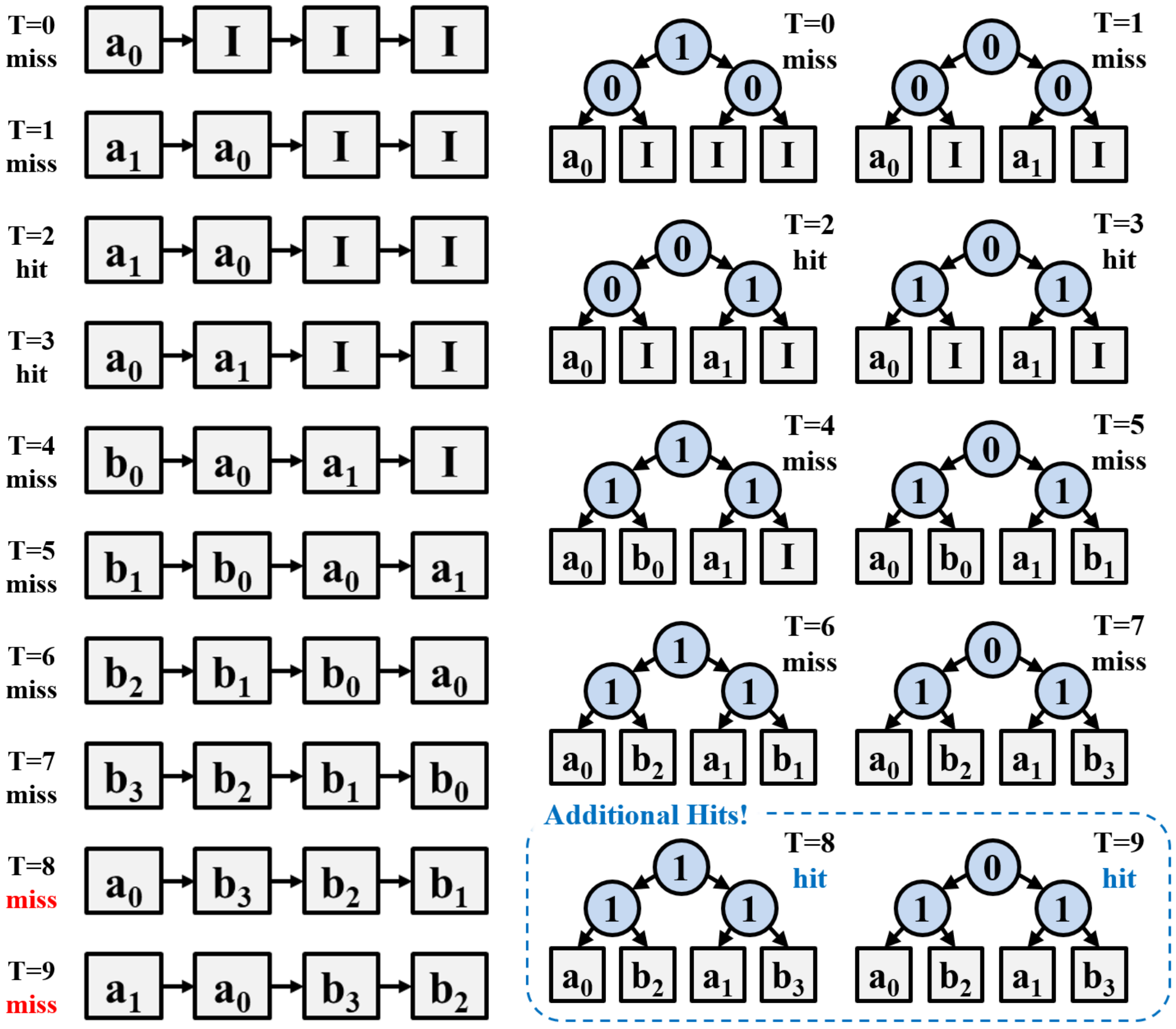
(b) 11-way PRRIP tree example

Current value (V)	Referenced way	Next value (V*)
0	1, 2	0
	0	1
1	0, 2	1
	1	2
2	0, 1	2
	2	0

(c) State transition table of 3-way PRRIP tree node

Figure 6. PRRIP tree when associativity is not a power of two.

Figure 7 shows the behavior of LRU and PRRIP for an access pattern with a scan. In the example, PRRIP gives one shell (root node) to a new block. If a block is promoted, PRRIP gives an additional shell from the bottom. While executing a scan pattern [b_0, b_1, b_2, b_3], LRU evicts useful blocks. However, PRRIP predicts a frequently reused block (a_0, a_1) which has near-immediate re-reference interval. Then, PRRIP resists the scan pattern. Consequently, PRRIP results in two additional hits at T=8 and 9.



(a) LRU

(b) PRRIP

Figure 7. Behaviors of LRU (a) and PRRIP (b) for example access pattern. The access pattern is [a_0, a_1, a_1, a_0, b_0, b_1, b_2, b_3, a_0, a_1]. (T=k: time immediately after the k-th access; 'I' means invalid data.)

B. Sensitivity to Promotion Step

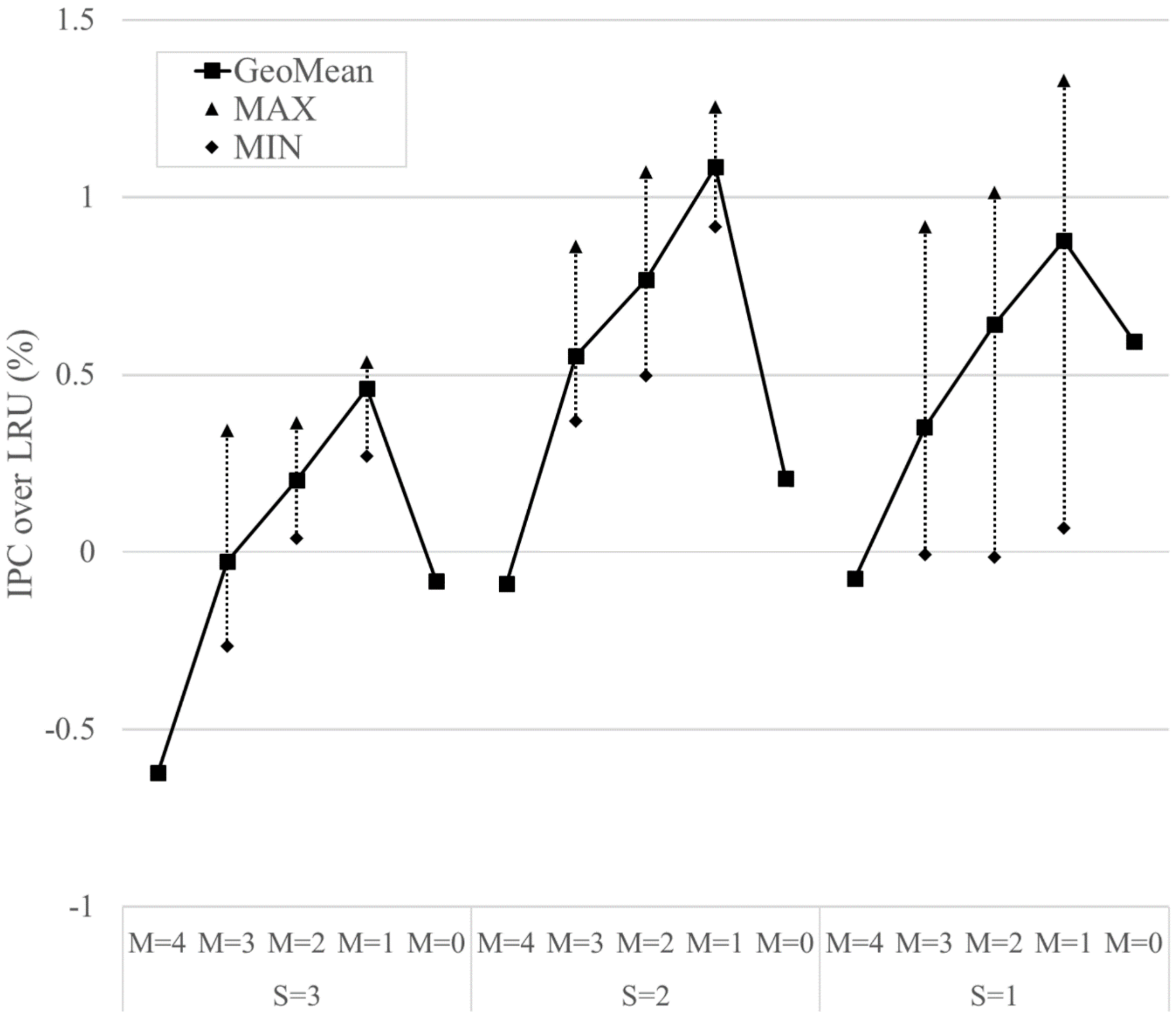
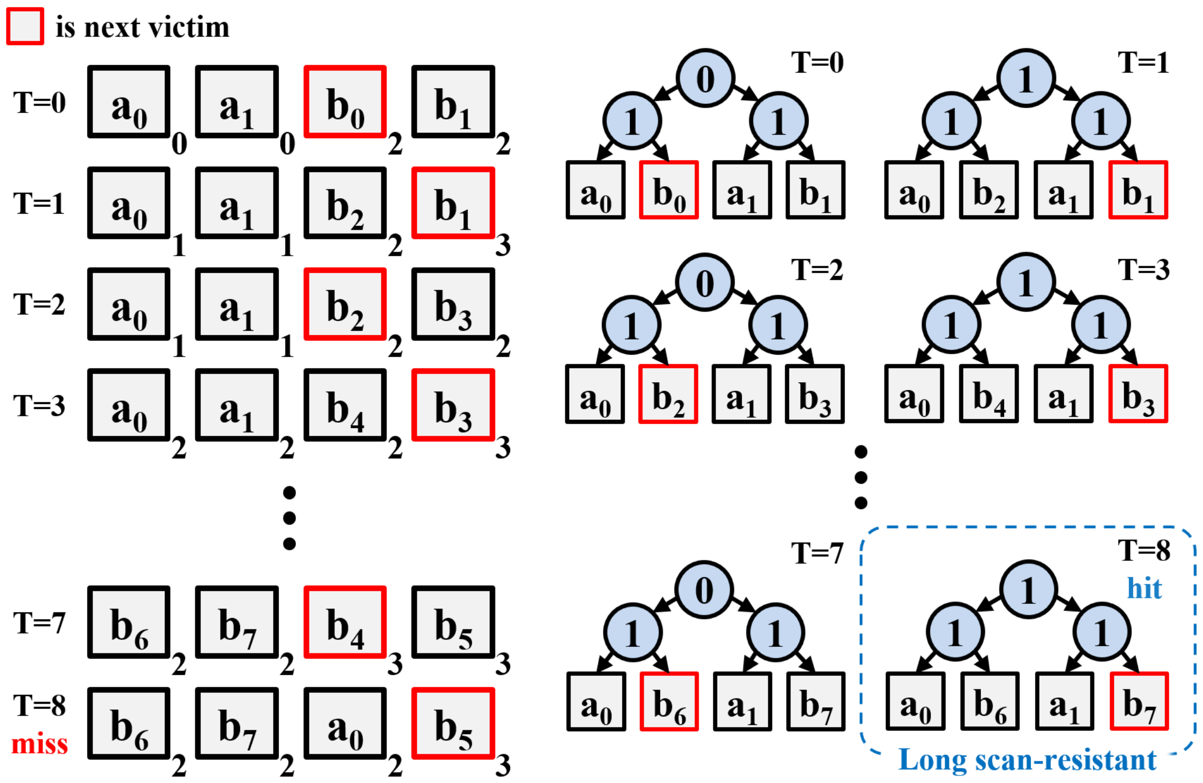


Figure 8. IPC over LRU (%) according to the promotion step (S) and number of shells applied during placement (M).

Promotion step indicates how many additional shells are to be applied when a block is hit. During a promotion, if the additionally applied number of shells is small, the difference in the number of shells between reused blocks and non-reused blocks is small. In this case, only frequently-accessed blocks will have many shells and move to a safe location in the set; i.e., PRRIP gives priority to a block that has been frequently accessed in a short time. In contrast, if many shells are given in promotion, the frequently reused block may become difficult to identify. Instead of being insensitive to frequency, PRRIP with large promotion step can better distinguish Fig. 8. IPC over LRU (%) according to the promotion step (S) and number of shells applied during placement (M). blocks that have never been reused. The scan-resistance property is a result of evicting never reused block, not of distinguishing frequently referenced line accurately [7].

Figure 8 reports the IPC over LRU of PRRIP according to promotion step (S) and number of shells received initially (M). This result is obtained in the same experimental setup as described in Chapter IV-A. When there are multiple cases of having shells, we have reported maximum value, minimum value and the geometric means of all cases. To minimize the effect of displacement caused by promotion, we apply additional shells from the bottom node when a block is promoted. Compared to one-step promotion, PRRIP with two-step promotion achieved higher IPC and showed lower difference between maximum IPC and minimum IPC. Also, its deviation between the maximum value and minimum values is smaller compared with one step promotion. PRRIP with two-step promotion also achieved higher IPC compared to PRRIP with three-step promotion. This is because disturbance has a stronger influence in three-step promotion than in the others. Thus, we conclude that applying two additional shells achieves a good trade-off point between scan-resistance and minimization of disturbance.

C. Sensitivity to Order of Insertion



(a) SRRIP

(b) PRRIP

Figure 9. Behaviors of SRRIP (a) and PRRIP (b) for example access pattern. The access pattern is [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7 a_0]. (T=k: time immediately after the k-th access; Before the time T<0, a_0, a_1 are protected in the set as a reused block.)

When continuous misses occur in the set, the placement position determines the number of new lines that are kept. For example, if only one shell is applied to a newly-inserted line, then if a miss occurs in the set, a block that lacks shell is replaced and changes value of a node to receive a shell. Then the next miss changes the value of the node again and the block that came in the last access becomes the next victim. Then the victim is evicted in two ways alternately until a hit occurs (Figure 9(b)). When scan access pattern (b_0 to b_7) comes, the way-1 and the way-3 alternately evict the blocks. If M shells are applied during placement, the 2^M lines alternate. As M decreases, PRRIP becomes increasingly resistive to scan pattern but the working set becomes increasingly difficult to replace. In other words, PRRIP can cope with thrashing access pattern whose length is less than $2^M + 1$. This is another characteristic that distinguishes PRRIP from SRRIP. In SRRIP, if a sufficiently long scan pattern is executed, all data in the set is replaced (Figure 9(a)).

As shown in Figure 8, the smaller the number of shells (M) received at the placement, the better the performance in terms of the geometric mean. The improvement was greatest when the number of shells received at the placement is one. If the number of shells received initially is small, blocks that will not be reused less pollute the cache set. When the number of shells received during the placement equals to zero, the new block has difficulty remaining in the cache until it is reused, so the IPC drops significantly. In other words, once a block is moved to a safe position, it becomes difficult to replace that block. Consequently, when a new block is inserted, one is the optimal number of shells to assign to it. This conclusion is consistent with [7]. It is known that the highest IPC of SRRIP can be obtained when insertion RRPV is $RRPV_{max} - 1$ ($RRPV_{max}$ is maximum value of RRPV). In the PRRIP algorithm, $RRPV_{max} - 1$ corresponds to one shell. One further decision is to identify the node at which the value must be changed during the placement. The shell of a node that is located near leaf nodes can only be obtained by the promotion process; therefore, that kind of shells enables identification of whether a line is reused at least once. Thus, the best approach is to apply a shell to the upper-side node.

Chapter IV

Experimental Setup and Result

A. Experimental Setup

To compare the performance of replacement policies, we use ChampSim [23], a trace-based memory-system simulator that was used in the 2nd Cache Replacement Championship (CRC-2) [24] held at ISCA 2017. ChampSim was also used in state-of-the-art research on replacement policies [20], [22]. ChampSim models a six-wide pipelined out-of-order processor with a 256-entry instruction window. The memory hierarchy of ChampSim consists of three level on-chip cache memory and the main memory. We summarize some details about memory hierarchy in TABLE I. ChampSim also models behaviors of branch predictor; we use gshare branch predictor [25] in this work.

We used 20 benchmarks from SPEC CPU 2006 [26] that were used in CRC-2. To put stress on LLC, its miss per kilo-instructions (MPKI) of selected benchmarks is greater than one. For each benchmark, we executed 200 million instructions as a warm-up. Then we ran 1 billion instructions to measure the IPC. On single-core experiments, we normalized results to the IPC of LRU to eliminate the effect of IPC differences on workload. For four-core experiments, we randomly pick four out of the 20 benchmarks to create 100 of four-core workloads. A thread finishes its workload faster than others will continue executing to keep competing for shared resources. We measured weighted IPC normalized by LRU for each four-core workload. Weighted IPC over LRU is commonly-used metric to evaluate performance of shared caches [14], [15], [18], [19], [22]. Weighted IPC over LRU for policy P can be estimated follows. First of all, for each thread i sharing the 8 MB LLC, we measured $IPC_{multi.P.i}$. Then we computed $IPC_{single.LRU.i}$ as executing the same workload in isolation with 8 MB LLC with LRU policy. Then weighted IPC for P equals to $\sum IPC_{multi.P.i} / IPC_{single.LRU.i}$. Finally, weighted IPC for P is normalized by weighted IPC for LRU to calculate the weighted IPC over LRU for P.

Components	Specifics
L1 Instr. Cache	Private 32 KB, LRU, 4-cycle latency, 8-way, 64 sets, 64 Bytes lines
L1 Data Cache	Private 32 KB, LRU, 4-cycle latency, 8-way, 64 sets, 64 Bytes lines
L2 Cache	Private 256 KB, LRU, 8-cycle latency, 8-way, 512 sets, 64 Bytes lines
LLC (Single Core)	Private 2 MB, 20-cycle latency, 16-way, 2,048 sets, 64 Bytes lines
LLC (Four Core)	Shared 8 MB, 20-cycle latency, 16-way, 8,192 sets, 64 Bytes lines
DRAM	13.75 ns latency for row buffer hits, 41.25 ns latency for row buffer misses, 12.8 GB/s throughput

Table I. Experimental Configuration

No.	Name	Programming Language	Job Description
1	GemsFDTD	Fortran	Computational Electromagnetics
2	astar	C++	Path-finding Algorithms
3	bwaves	Fortran	Fluid Dynamics
4	bzip2	C	Compression
5	cactusADM	C/Fortran	Physics/General Relativity
6	gcc	C	C Compiler
7	gobmk	C	Artificial Intelligence: go
8	gromacs	C/Fortran	Biochemistry/Molecular Dynamics
9	lbm	C	Fluid Dynamics
10	leslie3d	Fortran	Fluid Dynamics
11	libquantum	C	Physics: Quantum Computing
12	mcf	C	Combinatorial Optimization
13	milc	C	Physics: Quantum Chromodynamics
14	omnetpp	C++	Discrete Event Simulation
15	perlbench	C	PERL Programming Language
16	soplex	C++	Linear Programming, Optimization
17	sphinx3	C	Speech recognition
18	wrf	C/Fortran	Weather Prediction
19	xalancbmk	C++	XML Processing
20	zeusmp	Fortran	Physics/CFD

Table II. List of Single-Threaded Workloads

No.	CPU 0	CPU 1	CPU 2	CPU 3
1	gobmk	libquantum	perlbench	xalancbmk
2	astar	bwaves	lbm	zeusmp
3	cactusADM	lbm	milc	perlbench
4	bwaves	lbm	sphinx3	wrf
5	astar	cactusADM	GemsFDTD	perlbench
6	cactusADM	GemsFDTD	gobmk	soplex
7	astar	cactusADM	leslie3d	sphinx3
8	bwaves	libquantum	perlbench	sphinx3
9	cactusADM	gobmk	milc	soplex
10	bzip2	gobmk	lbm	perlbench
11	astar	gobmk	milc	soplex
12	gobmk	leslie3d	libquantum	perlbench
13	bwaves	bzip2	gobmk	wrf
14	gobmk	lbm	leslie3d	milc
15	cactusADM	gobmk	milc	perlbench
16	bwaves	bzip2	gobmk	leslie3d
17	astar	bzip2	leslie3d	xalancbmk
18	gobmk	libquantum	wrf	xalancbmk
19	gobmk	lbm	milc	zeusmp
20	milc	perlbench	wrf	zeusmp
21	perlbench	soplex	xalancbmk	zeusmp
22	milc	sphinx3	xalancbmk	zeusmp
23	bzip2	GemsFDTD	gobmk	soplex
24	astar	bwaves	perlbench	wrf
25	bwaves	bzip2	cactusADM	sphinx3
26	bwaves	cactusADM	lbm	wrf
27	astar	leslie3d	soplex	sphinx3
28	cactusADM	leslie3d	libquantum	perlbench
29	bwaves	cactusADM	milc	xalancbmk
30	GemsFDTD	libquantum	soplex	xalancbmk
31	astar	bzip2	soplex	xalancbmk
32	bzip2	libquantum	perlbench	xalancbmk
33	cactusADM	perlbench	wrf	xalancbmk
34	leslie3d	libquantum	sphinx3	xalancbmk
35	bwaves	gobmk	soplex	zeusmp
36	bzip2	milc	soplex	zeusmp
37	GemsFDTD	perlbench	soplex	zeusmp
38	bwaves	gromacs	leslie3d	zeusmp
39	astar	gromacs	sphinx3	zeusmp
40	GemsFDTD	libquantum	milc	zeusmp
41	cactusADM	soplex	wrf	zeusmp
42	GemsFDTD	gobmk	gromacs	perlbench
43	gromacs	libquantum	perlbench	wrf
44	bzip2	gromacs	libquantum	perlbench
45	astar	gromacs	lbm	wrf
46	cactusADM	gobmk	gromacs	perlbench
47	gromacs	lbm	sphinx3	wrf
48	cactusADM	soplex	sphinx3	xalancbmk
49	astar	gromacs	libquantum	xalancbmk
50	gcc	omnetpp	soplex	xalancbmk
51	gcc	gobmk	perlbench	zeusmp
52	lbm	soplex	sphinx3	zeusmp
53	cactusADM	GemsFDTD	omnetpp	perlbench
54	gcc	libquantum	milc	sphinx3

55	astar	gcc	omnetpp	wrf
56	cactusADM	gromacs	libquantum	wrf
57	GemsFDTD	lbm	leslie3d	libquantum
58	gcc	libquantum	milc	xalancbmk
59	cactusADM	lbm	libquantum	xalancbmk
60	gromacs	lbm	leslie3d	xalancbmk
61	gobmk	omnetpp	sphinx3	wrf
62	astar	gcc	lbm	omnetpp
63	gromacs	libquantum	omnetpp	soplex
64	milc	omnetpp	sphinx3	wrf
65	gromacs	leslie3d	milc	soplex
66	gcc	omnetpp	sphinx3	xalancbmk
67	bzip2	gcc	milc	zeusmp
68	cactusADM	gromacs	libquantum	omnetpp
69	GemsFDTD	gcc	leslie3d	sphinx3
70	astar	bwaves	cactusADM	omnetpp
71	astar	omnetpp	sphinx3	wrf
72	bwaves	bzip2	gcc	libquantum
73	omnetpp	perlbench	sphinx3	xalancbmk
74	bwaves	gcc	libquantum	zeusmp
75	leslie3d	omnetpp	wrf	zeusmp
76	bzip2	GemsFDTD	gcc	zeusmp
77	bwaves	cactusADM	gcc	zeusmp
78	gobmk	gromacs	leslie3d	omnetpp
79	bwaves	GemsFDTD	gcc	lbm
80	gromacs	lbm	soplex	xalancbmk
81	cactusADM	gcc	sphinx3	xalancbmk
82	gcc	gobmk	gromacs	sphinx3
83	gobmk	mcf	milc	omnetpp
84	gromacs	lbm	leslie3d	omnetpp
85	bzip2	gcc	gromacs	wrf
86	GemsFDTD	gobmk	mcf	soplex
87	astar	bwaves	gcc	mcf
88	GemsFDTD	gobmk	mcf	wrf
89	gcc	lbm	mcf	sphinx3
90	gcc	mcf	wrf	xalancbmk
91	GemsFDTD	mcf	omnetpp	wrf
92	bwaves	leslie3d	mcf	soplex
93	mcf	soplex	wrf	xalancbmk
94	bwaves	libquantum	mcf	wrf
95	cactusADM	lbm	mcf	sphinx3
96	mcf	milc	sphinx3	wrf
97	bzip2	mcf	milc	zeusmp
98	gromacs	mcf	milc	omnetpp
99	gromacs	libquantum	mcf	xalancbmk
100	astar	bwaves	gromacs	mcf

Table III. List of 4-Core Workloads

B. Result for Single-Core Workloads

To evaluate the performance of PRRIP, we additionally experimented on four replacement policies (LRU, PLRU, MDPP, SRRIP (2-bit)). Figure 10 shows the results of the single-core experiment. PRRIP achieved the highest geometric mean of IPC over LRU compared to other four replacement policies (PLRU -0.03%, MDPP 0.45%, SRRIP 0.53%, PRRIP 1.26%). Non-LRU algorithms achieved low IPC on *zeusmp* and *GemsFDTD* but achieved higher IPC than LRU-based algorithms on *sphinx3*.

Overall, PRRIP and SRRIP achieve similar IPC but differed in *zeusmp, omnetpp, leslie3d, gcc, libquantum, milc, cactusADM, lbm, mcf* and *sphinx3*. The reason caused this difference can be found by performance changes of PRRIP according to the M (Figure 11). PRRIP regards thrashing access pattern longer than 2^M as scan pattern (Chapter III-C). In *zeusmp, omnetpp, gcc* and *leslie3d*, as M increases, IPC of PRRIP also increases. In these benchmarks, PRRIP with small M evicts potentially hittable lines, which are part of the thrashing access pattern. Conversely, in *cactusADM*, *lbm*, *libquantum*, *mcf* and *milc* as M decreases, IPC of PRRIP increases. This is because, PRRIP with large M, suffers from more pollution due to scan pattern compared to PRRIP with small M. Additionally, *libquantum, lbm, milc, omnetpp* and *mcf* have large total number of accesses to LLC. Average access counts of these five benchmarks are more than three times compared with average access counts of the others. These differences in access counts affected the difference in IPC over LRU between SRRIP and PRRIP. However, although the number of accesses is not large, the difference in IPC over LRU between PRRIP and SRRIP is large in *sphinx3*; this difference occurs because LRU performs poorly with *sphinx3*. In *sphinx3*, IPC of PRRIP and SRRIP are divided by the relatively low IPC of LRU, that results in a relatively big difference.

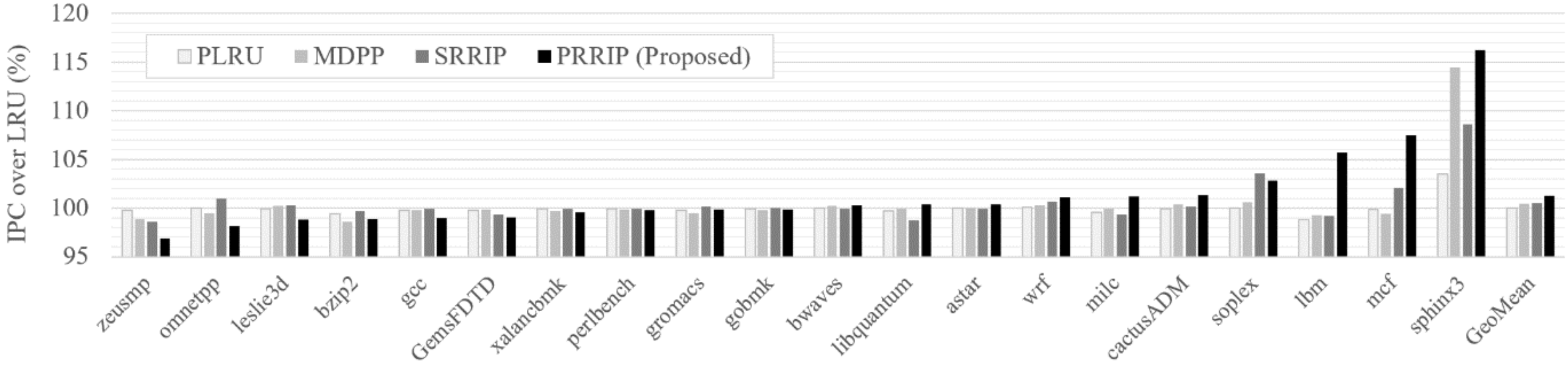


Figure 10. IPC over LRU for four replacement policies on single-threaded workloads. The order of benchmark is sorted by ascending order of PRRIP.

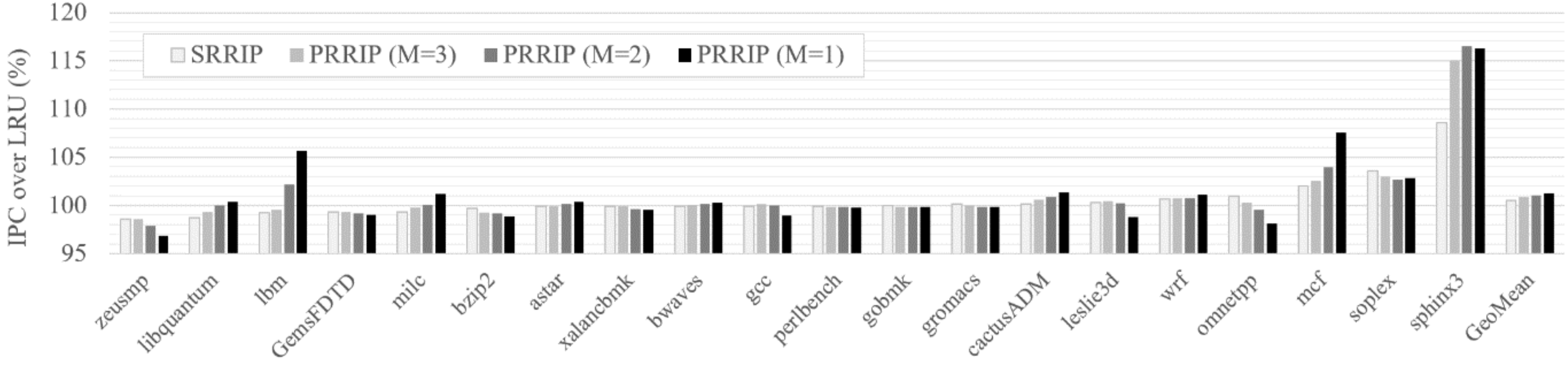


Figure 11. IPC over LRU of PRRIP on single-threaded workloads according to number of shells applied during placement (M). The order of benchmark is sorted by ascending order of SRRIP.

C. Result for Multi-Core Workloads

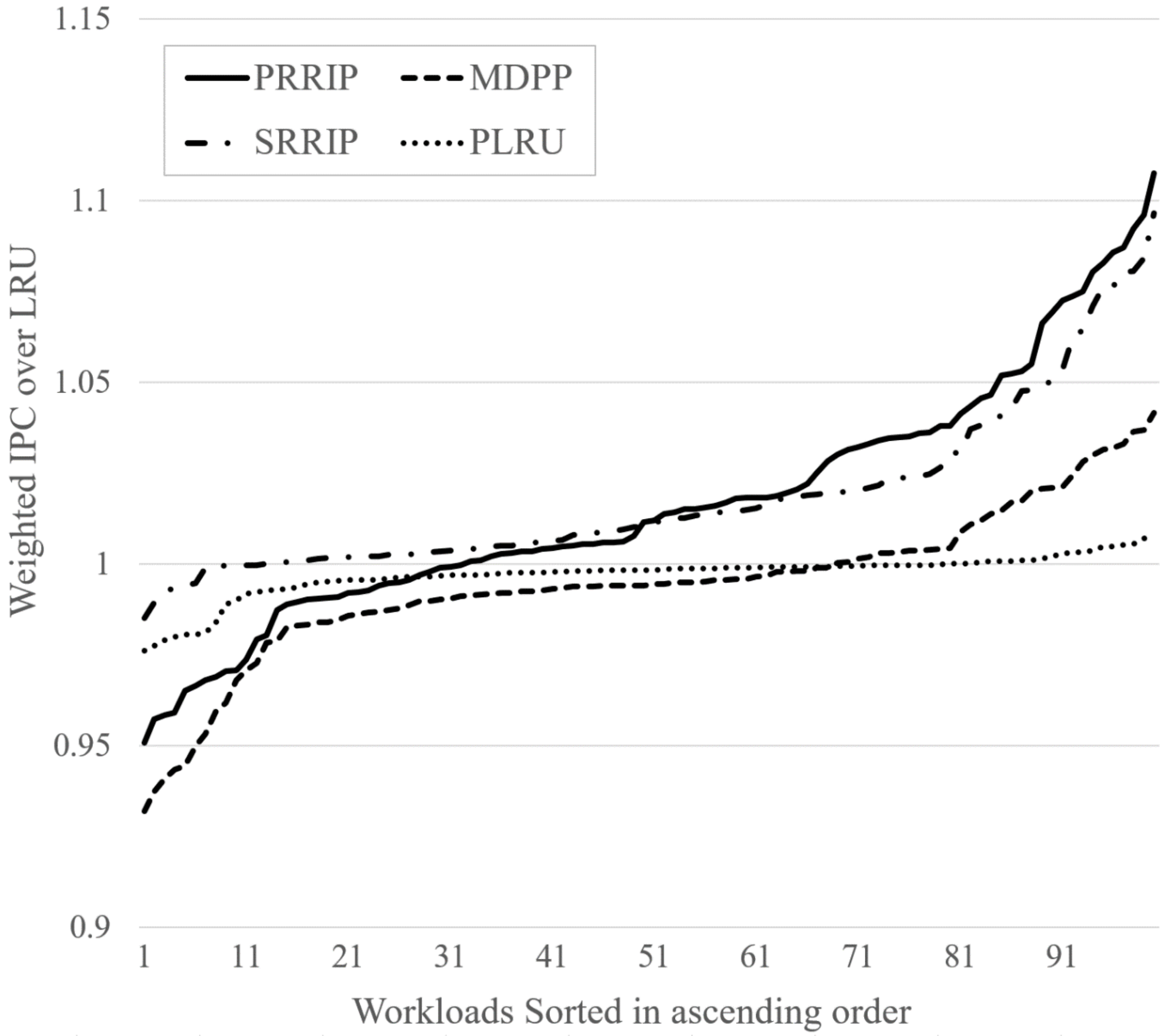


Figure 12. Normalized Weighted IPC over LRU for 4-core workloads. For each policy, workloads are sorted in ascending order to get S-curves.

As in the single-core experiment, we experimented on each replacement policy (LRU, PLRU, MDPP, SRRIP (2-bit), PRRIP) for multi-core workloads. Figure 12 reports weighted IPC over LRU for five replacement policies (LRU, PLRU, MDPP, SRRIP, PRRIP). On multi-core workloads, SRRIP and PRRIP achieved the highest and second-highest weighted IPC over LRU respectively compared to other replacement policies (PLRU -0.30%, MDPP 0.55%, SRRIP 1.82%, PRRIP 1.6%). The largest difference of speedup PRRIP over SRRIP was 6.04%, whereas the best speedup of SRRIP over PRRIP was 4.88%. Although SRRIP achieved higher geometric mean than PRRIP, it is a reasonable trade-off to consider overhead (will be discussed in Chapter IV-D).

D. Overhead

TABLE IV summarizes the hardware overhead for each replacement policies. PLRU, MDPP and PRRIP are based on a binary tree and require only one bit per block to save a replacement state. We ignore the additional logic overheads that are used to control the promotion and the placement, because they can be implemented with existing PLRU-based cache and a simple additional lookup table. Our proposed PRRIP consumes less than half of overhead of SRRIP but maintains similar IPC of SRRIP. Moreover, the higher associativity increases the difference in overhead between PRRIP and SRRIP. Considering that LLC typically has high associativity, PRRIP is an appropriate replacement policy for LLC.

In our experimental setup (TABLE I), the difference in overhead between PRRIP and SRRIP is 4.25 KB (2-bit RRPV) or 8.25 KB (3-bit RRPV) per core. This difference is not much compared to the total capacity of LLC, but it can be used to improve the performance of dynamic replacement policy. For example, SDBP predictor [6], Hawkeye predictor [15], Signature History Counter Table (SHCT) of SHiP [8] and prefetching status for PACMan approach [9] can be implemented within 4 KB of storage. Furthermore, storage for replacement state is usually implemented using 8T SRAM cells; the size of an 8 KB 8T SRAM is roughly the same as a moderate dual-ported branch direction predictor or branch target buffer [27].

Policies	Overhead for n-way (bits)	Overhead for 2 MB LLC (KB)	Overhead for 8 MB LLC (KB)
PLRU	$n-1$	3.75	15
MDPP	$n-1$	3.75	15
PRRIP	$n-1$	3.75	15
SRRIP	$2n$ (2-bit RRPV)	8	32
	$3n$ (3-bit RRPV)	12	48
LRU	$n\log_2 n$	16	64

Table IV. Overhead Requirements for Various Replacement Policies

Chapter V

Extend to Dynamic Replacement Policy

The PRRIP is one of the static replacement policy that does not adjust its behavior according to real-time change of workload. Although the static policies are still widely-used, research on dynamic replacement policy has been active in recent years. Thus, we extend PRRIP to dynamic replacement policy. SHiP++ [10] and ReD [11] are state-of-the-art replacement policies that ranked second and third in CRC-2. Additionally, SHiP++ and ReD policies are based on SRRIP so we apply PRRIP to SHiP++ (P-SHiP++) and ReD (P-ReD). SHiP++ is a modified version of SHiP [8], that use Signature History Counter Table (SHCT) to adjust placement position. To replace SRRIP used in SHiP++ to PRRIP, we apply all shells to a line that is requested by positively trained PC and add no shell to a line requested by negatively trained PC. ReD is an aggressive block selection algorithm, that bypasses all blocks predicted not to be reused. Since ReD bypasses the first requested blocks, ReD is robust to scan access pattern. To reflect this characteristic, PRRIP in ReD apply three shells to give more opportunity to the newly-inserted line.

Policies	IPC over LRU (1-core)	Weighted IPC over LRU (4-core)	Overhead for 2 MB LLC (KB)
SHiP++	4.98%	8.18%	16
P-SHiP++	3.33%	6.59%	11.75
ReD	3.00%	9.56%	31.875
P-ReD	2.59%	7.98%	27.625

Table V. Experimental Results of Dynamic Replace Policies

We evaluate P-SHiP++ and P-ReD through experiments described in Chapter IV. Then, we compare the speedup over LRU with original SHiP++ and ReD. TABLE V shows the summary of experimental results for each dynamic policy. With PRRIP, we can reduce the hardware overhead from 16 KB to 11.75 KB and from 31.875 KB to 27.625 KB in SHiP++ and ReD, respectively. P-SHiP++ achieves lower speedup over LRU than original SHiP++. PRRIP with one shell placement allows the promotion of the newly-inserted line within two accesses in a set. This induces more misses than SRRIP in some benchmarks (*zeusmp, leslie3d, omnetpp*), disturbs training SHCT of P-SHiP++. Consequently, miss-trained SHCT of P-SHiP++ degrades the performance. On single-core experiments, PReD achieves comparable IPC over LRU with that of ReD. However, ReD achieves

1.58% higher speedup than P-ReD. As we discussed in Chapter III-C, PRRIP is more resistant to long scan pattern than SRRIP. Since ReD is robust to scan pattern, PRRIP relatively becomes disadvantageous with ReD.

Chapter VI

Conclusion

Practical cache replacement policy for LLC should consider both performance and overhead at the same time. In this thesis, we proposed PRRIP; it reduced the overhead of SRRIP by less than half, but achieved similar IPC to SRRIP. Our experimental results confirmed that PRRIP mimics SRRIP. Moreover, our experimental results show that PRRIP achieves 1.26% speedup over LRU while SRRIP gets 0.53% speedup over LRU for single-core workloads. For multi-core workloads, our experimental results show that the performance difference between PRRIP and SRRIP is less than 0.3%. We also study details to consider when implementing PRRIP such as promotion step and position of the newly-inserted line. By replacing SRRIP to PRRIP, we extend PRRIP to dynamic replacement policy. Our ongoing work seeks to develop a dynamic replacement algorithm which is suitable for PRRIP.

REFERENCES

[1] W. A. Wulf and S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious”, ACM SIGARCH Computer Architecture News 23(1) (1995), pp. 20-24.

[2] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang and Y. Solihin, “Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling”, ACM SIGARCH Computer Architecture News 37(3) (2009), pp. 371-382.

[3] G. Keramidas, P. Petoumenos and S. Kaxiras, “Where Replacement Algorithms Fail: a Thorough Analysis”, Proc. ACM International Conference on CF, 2010, pp. 141-150.

[4] N. Beckmann and D. Sanchez, “Modeling Cache Performance Beyond LRU”, Proc. IEEE International Symposium on HPCA, 2016, pp. 225-236.

[5] N. A. Kurd, S. Bhamidipati, C. Mozak, J. L. Miller, T. M. Wilson, M. Nemani and M. Chowdhury, “Westmere: A Family of 32nm IA Processors”, Proc. IEEE International Conference on ISSCC, 2010, pp. 96-97.

[6] S. Khan, Y. Tian and D. A. Jim´enez, “Sampling Dead Block Prediction for Last-Level Caches”, Proc. IEEE/ACM International Symposium on MICRO, 2010, pp. 175-186.

[7] A. Jaleel, K. B. Theobald, S. C. Steely Jr. and J. Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)”, Proc. IEEE International Symposium on ISCA, 2010, pp. 60-71.

[8] C. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr. and J. Emer, “PACMan: Prefetch-Aware Cache Management for High Performance Caching”, Proc. IEEE/ACM International Symposium on MICRO, 2011, pp. 442-453.

[9] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr. and J. Emer, “SHiP: Signature-based Hit Predictor for High Performance Caching”, Proc. IEEE/ACM International Symposium on MICRO, 2011, pp. 430-441.

[10] V. Young, C. Chou, A. Jaleel and M. Qureshi, “SHiP++: Enhancing Signature-based Hit Predictor for Improved Cache Performance”, 2017. [Online]. Available: http://crc2.ece.tamu.edu/

[11] P. Faldu and B. Grot, “Reuse-aware Management for Last-Level Caches”, 2017. [Online]. Available: http://crc2.ece.tamu.edu/

[12] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely and J. Emer, “Adaptive Insertion Policies for High Performance Caching”, Proc. IEEE International Symposium on ISCA, 2007, pp. 381-391.

[13] J. Handy, The Cache Memory Book, Morgan Kaufmann, 1993.

[14] E. Teran, Y. Tian, Z. Wang and D. A. Jim´enez, “Minimal Disturbance Placement and Promotion”, Proc. IEEE International Symposium on HPCA, 2016, pp. 201-211.

[15] A. Jain and C. Lin, “Back to the Future: Leveraging Beladys Algorithm for Improved Cache Replacement”, Proc. IEEE International Symposium on ISCA, 2016, pp. 78-89.

[16] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero and A. V. Veidenbaum, “Improving Cache Management Policies Using Dynamic Reuse Distances”, Proc. IEEE/ACM International Symposium on MICRO, 2012, pp. 389-400.

[17] N. Beckmann and D. Sanchez, “Maximizing Cache Performance Under Uncertainty”, Proc. IEEE International Symposium on HPCA, 2017, pp. 109-120.

[18] D. A. Jim´enez and E. Teran, “Multiperspective Reuse Prediction”, Proc. IEEE/ACM International Symposium on MICRO, 2017, pp. 436-448.

[19] E. Teran, Z. Wang and D. A. Jim´enez, “Perceptron Learning for Reuse Prediction”, Proc. IEEE/ACM International Symposium on MICRO, 2016, pp. 1-12.

[20] J. Kim, E. Teran, P. V. Gratz, D. A. Jim´enez, S. H. Pugsley and C. Wilerson, “Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy”, Proc. ASPLOS, 2017, pp. 737-749.

[21] L. A. Belady, “A Study of Replacement Algorithms for a Virtual-storage Computer”, IBM Systems Journal 5(2) (1996), pp. 78-101.

[22] A. Jain and C. Lin, “Rethinking Belady’s Algorithm to Accommodate Prefetching”, Proc. IEEE International Symposium on ISCA, 2018, pp. 110-123.

[23] “The ChampSim Simulator”, [Online]. Available: https://github.com/ChampSim/ChampSim/

[24] “2nd Cache Replacement Championship”, 2017. [Online]. Available: http://crc2.ece.tamu.edu/

[25] S. McFarling, “Combining Branch Predictors”, Digital Western Research Laboratory, Tech. Rep., 1993.

[26] "SPEC CPU 2006", 2006. [Online]. Available: http://www.spec.org/cpu2006.

[27] E. Teran, "Principled Approaches to Last-Level Cache Management", Doctoral Dissertation of Texas A&M University, Aug. 2017.

[28] G. Keramidas, P. Petoumenos and S. Kaxiras, "Cache Replacement Based on Reuse-Distance Prediction", Proc. IEEE International Conference on ICCD, 2007, pp. 245-250.

[29] D. A. Jim´enez, "Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches", Proc. IEEE/ACM International Symposium on MICRO, 2013, pp. 284-296.

[30] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance", Proc. IEEE International Symposium on HPCA, 2015, pp. 64-75.

[31] A. Pablo, P. Prieto, V. Puente, and J. Gregorio, "Improving Last Level Shared Cache Performance through Mobile Insertion Policies (MIP)", Parallel Computing 49 (2015), pp. 13-17.