Adaptive cache insertion policies

Patrycja Balik

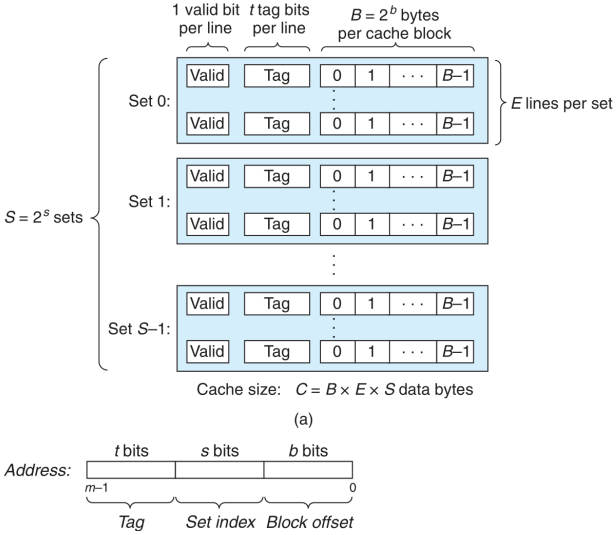
December 3, 2019

Cache policies

During this seminar so far, we talked about **cache organization**, or how things are structured within the cache, e.g. the number of sets and lines, block size.

Today, we'll focus on **cache policies** which describe how a cache should work, e.g. write-back vs write-through; write-allocate vs no-write-allocate; inclusive, exclusive or neither; line replacement, etc.

A quick look at cache organization again



Cache eviction

The need for eviction is obvious: the cache is smaller than the storage below it in the memory hierarchy.

Requested data isn't in cache and the set is full $\rightarrow$ a line needs to be evicted to make room for the new data.

Cache replacement policy

Which line to pick for eviction is a matter of **replacement policy**. The strategy used can have a profound impact on performance.

The choice is a matter of balancing out metadata size overhead, implementation complexity and resulting performance.

Cache replacement policy

- (Pseudo-)**Random**
- **Round-Robin**
- **FIFO** (first in first out)
- **LRU** (least recently used)
- **Pseudo-LRU**
- ...

Random

The victim line is choosen randomly, for a good enough definition of random.

Random

The victim line is choosen randomly, for a good enough definition of random.

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```

Figure: Not a good PRNG

Round-Robin

A counter global for the entire cache determines which line is going to be evicted Metadata overhead is minimal, and complexity is low, though behavior might not be optimal for code with good locality.

FIFO

Lines are evicted on a per-set basis. Additional metadata in every set, simple implementation.

LRU

Pick the least recently used line for eviction. Good idea, but metadata overhead is high. (Store line age in each line? Encode permutation in set?).

Pseudo-LRU

LRU is good, but complicated—we can try to approximate it.
Some approaches, for an n-way cache:

- A variation of random: just store the half in which the LRU is (1 bit)
- Bit-PLRU (n bits)
- Tree-PLRU (n-1 bits)
- ...

Bit-PLRU

For each line, an additional bit is stored. At the beginning, they're all 0. When a line is accessed, its bit gets set to 1. If this were to give us 1s in every line in the set, the other lines are reset to 0.

A victim is choosen among the lines with this bit set to 0.

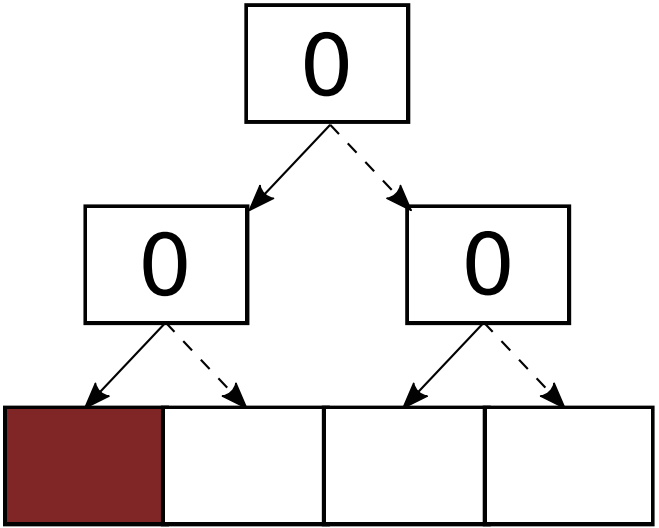
Tree-PLRU

For each set store a sequence representation of a path in a binary search tree, where 0 means "go left" and 1 means "go right".

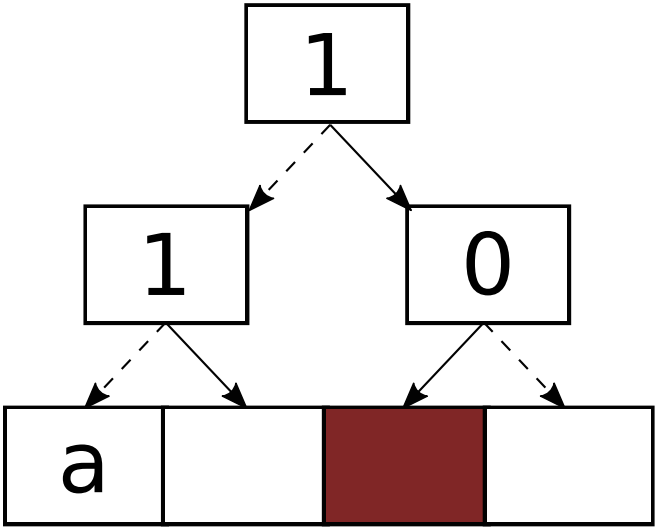
Leaves represent cache lines, and the path indentifies the victim.

On access to line n, flip bits on the path to n in the sequence.

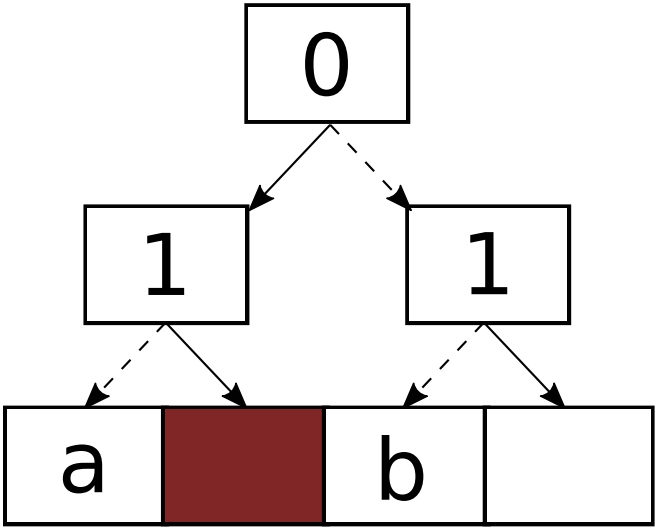
Tree-PLRU



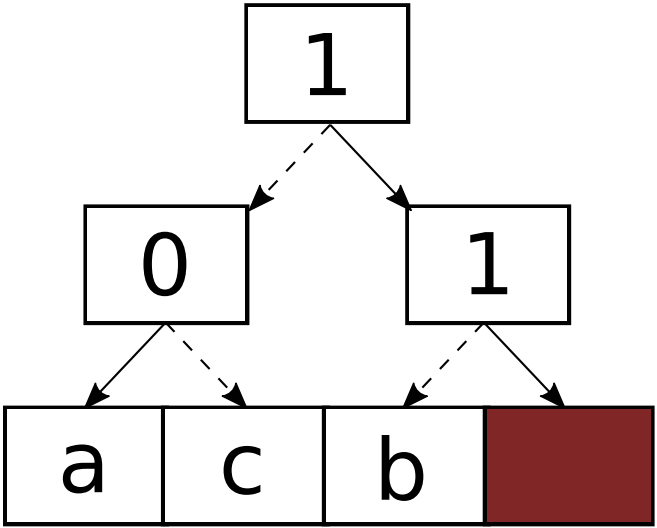
Tree-PLRU



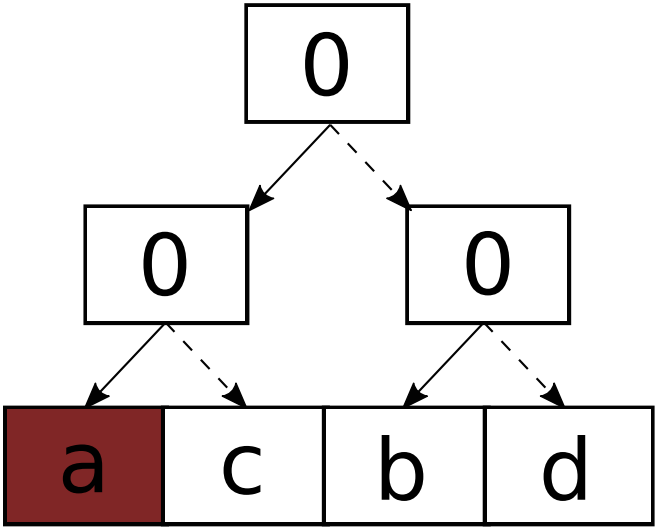
Tree-PLRU



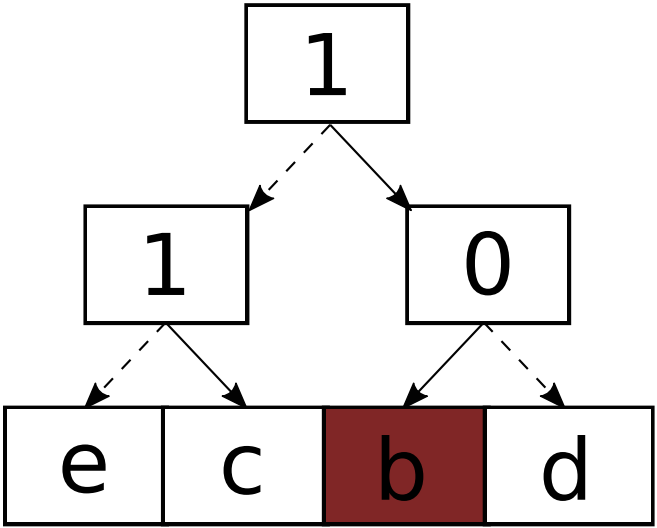
Tree-PLRU



Tree-PLRU

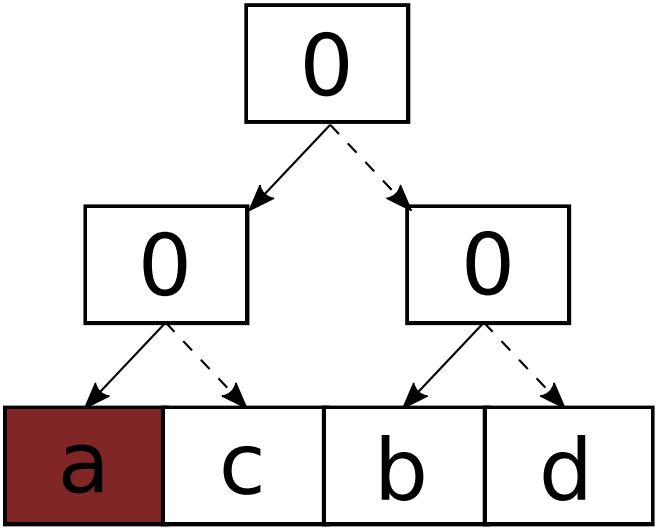


Tree-PLRU

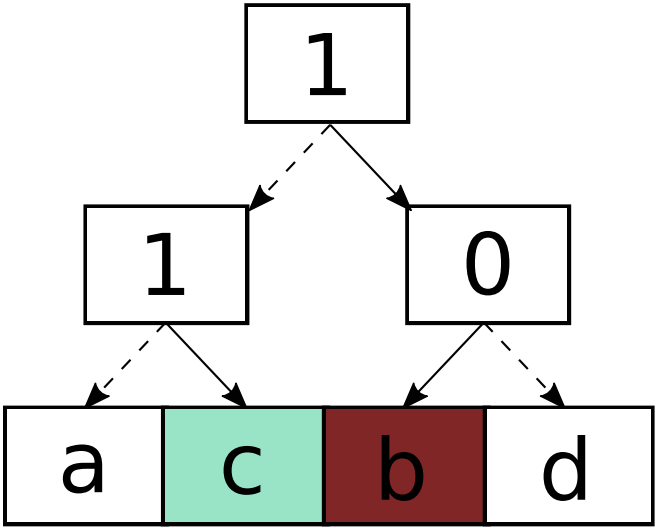


Fun as that was, it's easy to show that this is not "true" LRU.

Tree-PLRU



Tree-PLRU



Thrashing

When the working set is too large for the cache, the cache will **thrash** by constantly attempting to catch up with the working set, only to need the previously evicted data brought in again.

Adaptive Insertion Policies for High Performance Caching

Moinuddin K. Qureshi† Aamer Jaleel§ Yale N. Patt† Simon C. Steely Jr.§ Joel Emer§

†ECE Department
The University of Texas at Austin
{moin, patt}@hps.utexas.edu

§Intel Corporation, VSSAD
Hudson, MA
{aamer.jaleel, simon.c.steely.jr, joel.emer}@intel.com

Simulated cache configuration

Cache	Size	Block size	Ways
L1-I	16kB	64B	2-way
L1-D	16kB	64B	2-way
L2 (LLC)	1MB	64B	16-way
- Non-inclusive, non-exclusive.
- Unless said otherwise, traditional LRU replacement is assumed.
- We'll analyze access patterns to the L2 cache.

Dissecting cache replacement

A replacement policy contains two distinct parts: an eviction (victim selection) policy and an insertion policy.

The eviction policy is responsible for choosing a victim when space needs to be freed up in a set, e.g. picking LRU in the LRU replacement policy.

The insertion policy is responsible for where a new line is ordered among the rest at the point it's added, e.g. at the most recently used (MRU) position in traditional LRU replacement.

MRU insertion policy

In principle, MRU insertion seems like a good idea; it gives blocks a chance to be used before they get evicted from the cache.

However, MRU insertion contributes to the effect of thrashing: a block can be inserted, pass through all the positions and get evicted, without ever being accessed.

For a working set exceeding the cache size, most blocks will just pass through the cache in this manner, leading to a cache miss ratio of 100%.

MRU insertion policy: Zero Reuse Lines

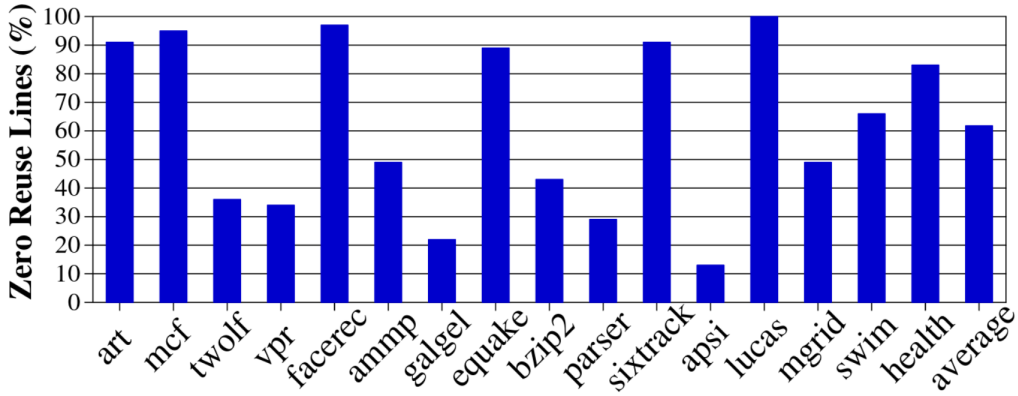


Figure: Zero Reuse Lines for 1MB 16-way L2 cache

MRU insertion policy: Cache misses I

Name	FFWD	MPKI	Compulsory Misses
art	18.25B	38.7	0.5%
mcf	14.75B	136	1.8%
twolf	30.75B	3.48	2.9%
vpr	60B	2.16	4.3%
facerec	111.75B	3.66	4.8%
ammp	4.75B	2.83	5.0%
galgel	14B	5.34	5.9%
equake	26.25B	18.4	14.2%

Figure: FFWD - fast-forward interval, MPKI - misses/kiloinstruction, B - billion

MRU insertion policy: Cache misses II

Name	FFWD	MPKI	Compulsory Misses
bzip2	2.25B	2.4	14.8%
parser	66.25B	1.57	20.0%
sixtrack	8.5B	0.42	20.7%
apsi	3.25B	0.32	21.4%
lucas	2.5B	16.2	41.6%
mgrid	3.5B	7.73	46.6%
swim	3.5B	23.0	50.4%
health	0B	61.7	0.73%

Figure: FFWD - fast-forward interval, MPKI - misses/kiloinstruction, B - billion

LRU insertion policy (LIP)

Idea: modify the insertion policy so that some blocks may be retained and reused in the 100% MR scenario from earlier.

Attempt #1: insert in the LRU position instead of the MRU.

Sounds too simple? Well, it actually helps! Somewhat.

LRU insertion policy (LIP)

Attempt #1: insert in the LRU position instead of the MRU.

The implementation is almost identical to LRU, or any PLRU variant—the only change is skipping the recency update on insertion, no additional overhead required.

LRU insertion policy (LIP)

Problem: LIP doesn't respond well to changes in working set; old blocks will linger in cache despite not being needed anymore, because there's no block "aging".

Attempt #2: modify LIP so that occasionally, a block is inserted in MRU position.

Bimodal insertion policy (BIP)

Attempt #2: modify LIP so that occasionally, a block is inserted in MRU position.

Come bimodal insertion policy (BIP), which has a probability ϵ of using MRU insertion, or LIP otherwise, for a small value of ϵ. For the purpose of data displayed later in this presentation $\epsilon = 1/32 = 0.03125$.

This combines the best of both worlds; adapting to working set changes and thrashing protection.

Case studies: thrashing workloads

First, we'll analyze benchmarks where LIP and BIP provide a significant improvement over traditional LRU.

These benchmarks are:

- mcf (single-depot vehicle scheduling in public mass transportation)
- art (neural network recognizing objects in a thermal image)
- health (health care system simulation—working set increases with time)

Case study: mcf


```
while (arcin) {
  tail = arcin->tail;

  if (tail->time + arcin->org_cost > latest) {
    arcin = (arc_t *)tail->mark;
    continue;
  }
  ...
  arcin = (arc_t *)tail->mark;
}
```

Listing 1: **Bold** instructions cause 84% of all cache misses in mcf.

mcf's behavior can be approximated by a linked list traversal of about 3.5MB of data.

Case study: mcf

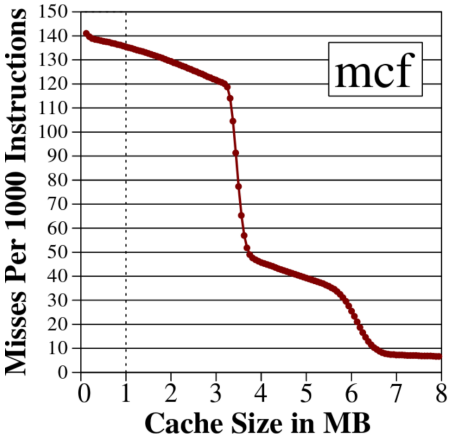


Figure: MPKI with the LRU replacement policy for mcf for various L2 sizes.

Case study: mcf

Policy	MPKI	Reduction over LRU
LRU	136	-
LIP/BIP	115	17%
Optimal	101	26%

Figure: Results for the 1MB L2 cache. LIP/BIP retain around 1MB of the working set.

Case study: art


```
numf1s = lwidth*lheight; // = 100*100 for ref input set
numf2s = numObjects + 1; // = 10 + 1 for ref input set
...
for (tj = spot; tj < numf2s; ++tj) {
  Y[tj].y = 0;

  if (!Y[tj].reset) {
    for (ti = 0; ti < numf1s; ++ti) {
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
    }
  }
}
```

Listing 2: **Bold** instructions cause 39% and 41% of L2 misses, respectively.

Case study: art

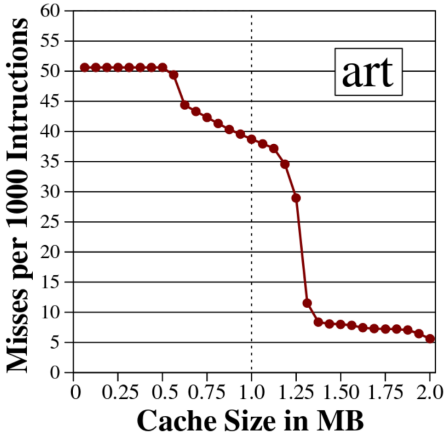


Figure: MPKI with the LRU replacement policy for art for various L2 sizes.

Case study: art

Policy	MPKI	Reduction over LRU
LRU	38.7	-
LIP	23.6	39%
BIP	18.0	54%
Optimal	12.8	67%

Figure: Results for the 1MB L2 cache. Note the differences for LIP and BIP.

Case study: health


```
while (list != NULL) {
  ...
  p = list->patient;
  ...
  list = list->forward;
}
```

Listing 3: **Bold** is responsible for 71% of all L2 misses.

The health benchmark performs a linked list traversal with frequent insertions and deletions. The size of the working set increases with time.

Case study: health

To display these changes, the program can be split into a few phases, around 50M instructions each.

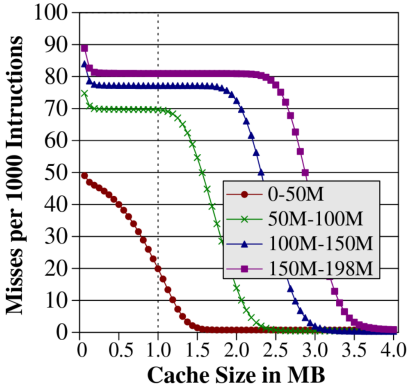


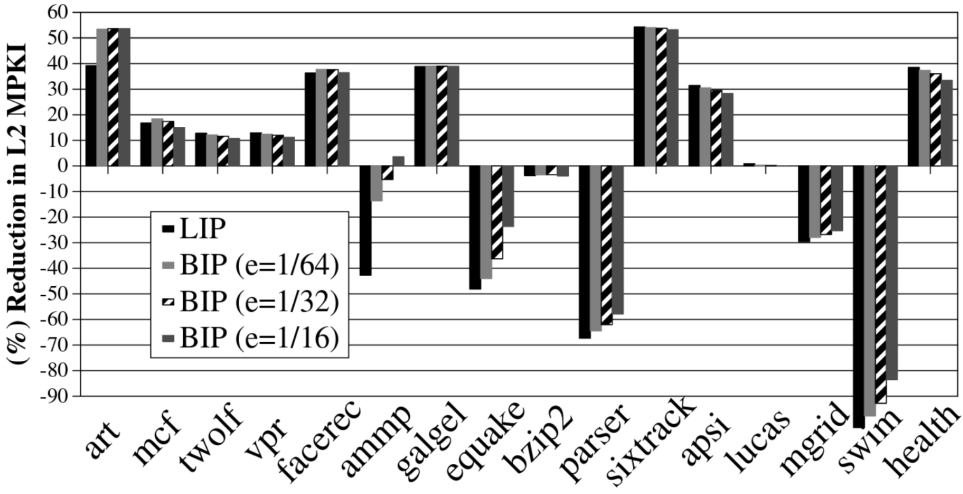
Figure: MPKI with the LRU replacement policy for health with various L2 sizes.

Case study: health

Policy	MPKI	Reduction over LRU
LRU	61.7	-
LIP	38.0	38.5%
BIP	39.5	36.0%
Optimal	34.0	45.0%

Figure: Results for the 1MB L2 cache.

LIP and BIP vs LRU



Case study: swim

swim - performs weather prediction. Heavy on matrix multiplication (but it's in Fortran, so I haven't checked).

Case study: swim

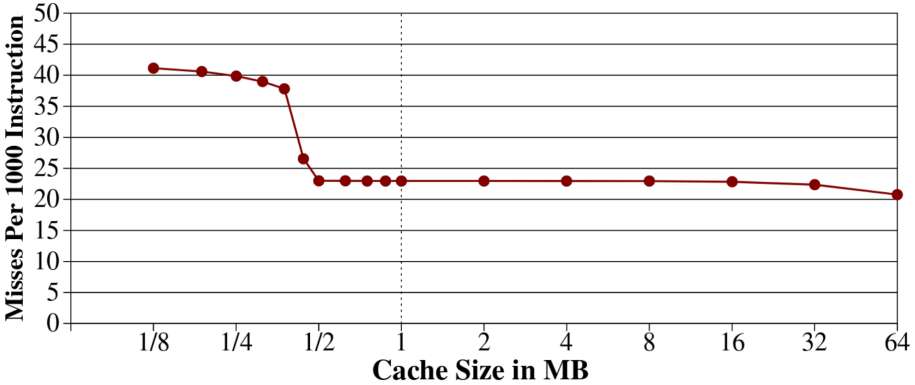


Figure: MPKI for swim with LRU. (log scale size axis)

Case study: swim

Policy	MPKI	Reduction over LRU
LRU	23.0	-
LIP	46.5	-102.0%
BIP	44.3	-92.5%
Optimal	22.8	0.9%

Figure: Results for the 1MB L2 cache.

Dynamic insertion policy (DIP)

Problem: BIP performs very poorly for workloads which benefit from MRU insertion.

Attempt #3: a hybrid solution which picks between traditional LRU, with MRU insertion, and BIP, depending on which causes fewer misses.

Problem: how do we determine which is better at runtime?

Attempt #3.1: For every set analyze misses caused by either policy, modifying a counter global for all sets, and pick a single best policy to use for all sets.

DIP-Global

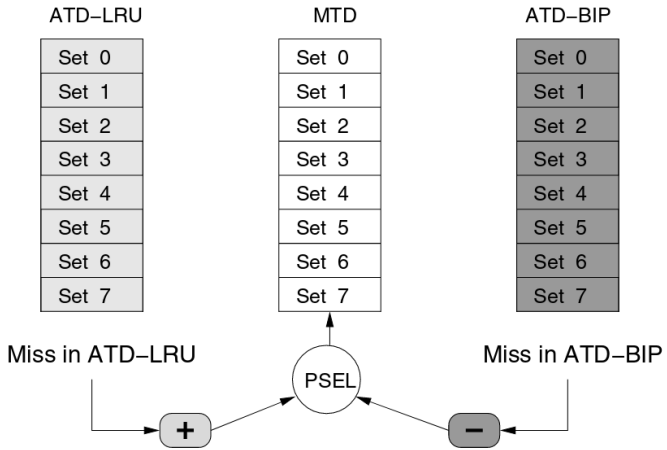


Figure: DIP-Global with 8 sets
MTD - Main Tag Directory
ATD - Auxiliary Tag Directory
PSEL - Policy Selector (saturating counter)

DIP-Global

Problem: the overhead of two extra tag directories isn't acceptable.

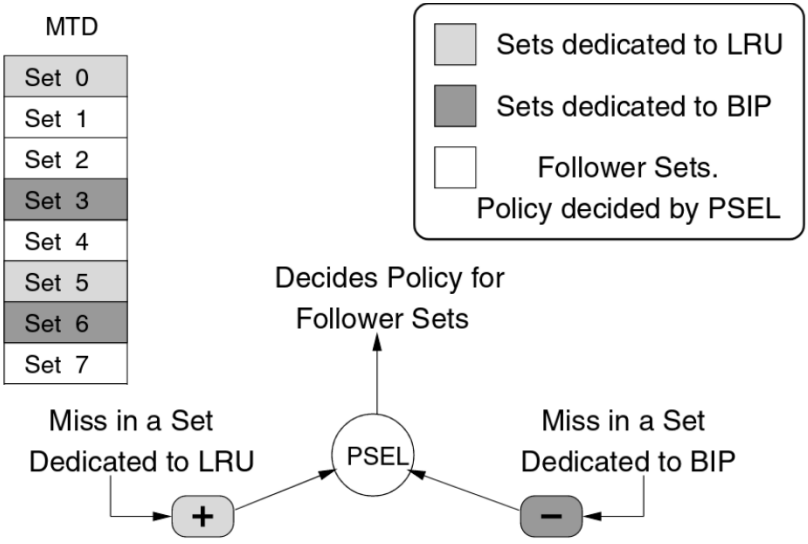
Attempt #3.2: Sample just a select few sets to reduce the size of ATDs.

Set Dueling

Problem: while the storage overhead is smaller, major changes still need to be made to the cache organization. Can we avoid this?

Attempt #3.3: for each of the two policies select a few sets that will use them, and use the best policy for the remaining sets. We'll call this **Set Dueling**.

Set Dueling



Set Dueling

Selecting the dedicated sets for DIP-SD...

- ...statically at design time?
- ...dynamically at runtime?

Set Dueling

Authors propose the following scheme:

- N - total number of sets.
- K - number of sets dedicated to each policy.
- The cache is logically divided into K *constituencies*, each containing N/K sets.
- From each constituency, dedicate one set to each of the competing policies.
- Generally assume that K is a power of 2 for the purpose of the study.

Complement-select policy

- Out of $log_2(N)$ set index bits, let the most significant $log_2(K)$ bits identify the constituency and the remaining $log_2(N/K)$ the offset from the first set in the constituency.
- (Yes, at this point "constituency" stops looking like a real word.)

Complement-select policy

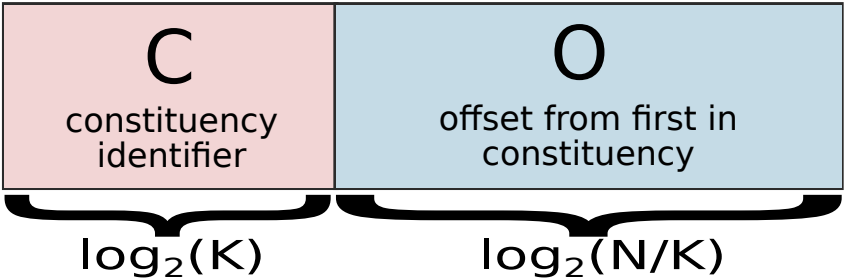


Figure: I made this atrocity in Inkscape myself, so now you have to look at it.

Complement-select policy

The paper authors only provided examples where $log_2(K) = log_2(N/K)$, but the method works for mismatched lengths of C and O if we treat the comparison appropriately.

- $C = O \rightarrow$ dedicate the set to LRU.
- $C = \overline{O} \rightarrow$ dedicate the set to BIP.
- The remaining sets are follower sets.

Complement-select policy: example I

$$N = 2^6,\ K = 8 \rightarrow log_2(K) = log_2(N/K) = 3.$$

LRU sets

C	O
000	000
001	001
010	010
...	...
111	111

BIP sets

C	O
000	111
001	110
010	101
...	...
111	000

Complement-select policy: example II

$$N = 2^5,\ K = 4 \rightarrow log_2(K) = 2,\ log_2(N/K) = 3.$$

LRU sets

C	O
00	000
01	001
10	010
11	011

BIP sets

C	O
00	011
01	010
10	001
11	000

Alternatively the MSB in O could be changed in either of those, as long as there is one per constituency.

Complement-select policy: example III

$N = 2^5,\ K = 8 \rightarrow log_2(K) = 3,\ log_2(N/K) = 2.$

LRU sets

C	O
000	00
001	01
010	10
011	11
100	00
101	01
110	10
111	11

BIP sets

C	O
000	11
001	10
010	01
011	00
100	11
101	10
110	01
111	00

DIP-SD

We'll consider two DIP-SD variants for the 1MB L1 cache from earlier.

- $N = 1024$
- $K = 32$
- 10-bit PSEL

- $N = 1024$
- $K = 64$
- 11-bit PSEL

Where not specified, DIP is assumed to mean DIP-SD with $K = 32$.

DIP vs LRU

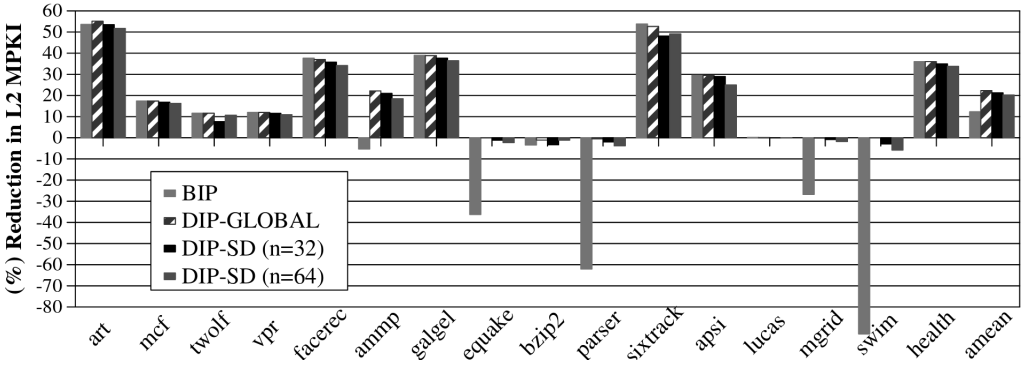


Figure: amean: reduction in arithmetic mean of MPKI of all the benchmarks.
n: K from previous slides about SD.
DIP-Global: initial concept with complete ATDs for every set.

Analyzing PSEL changes

For 10-bit PSEL:

- PSEL $\geq 512 \rightarrow$ BIP
- PSEL $< 512 \rightarrow$ LRU

DIP can adapt to changes during a program's runtime. We can observe this effect by tracking the PSEL value.

Analyzing PSEL changes: mcf

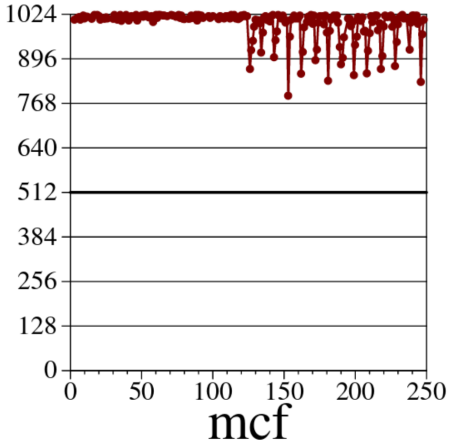


Figure: Vertical: PSEL value, horizontal: instructions in millions

Analyzing PSEL changes: health

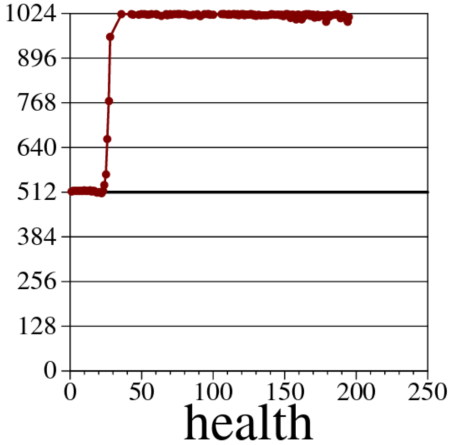


Figure: Vertical: PSEL value, horizontal: instructions in millions

Analyzing PSEL changes: swim

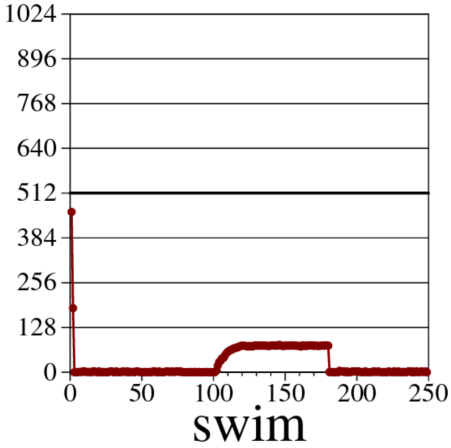


Figure: Vertical: PSEL value, horizontal: instructions in millions

Analyzing PSEL changes: ammp

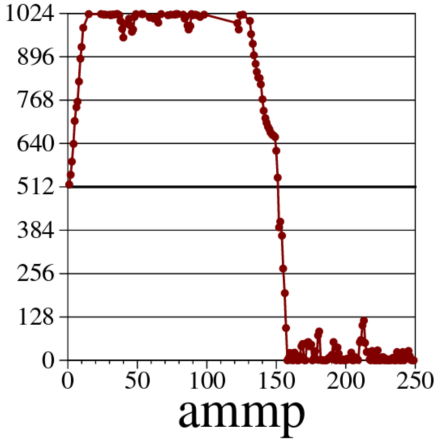


Figure: Vertical: PSEL value, horizontal: instructions in millions

Analyzing the effects of policy change and cache size change

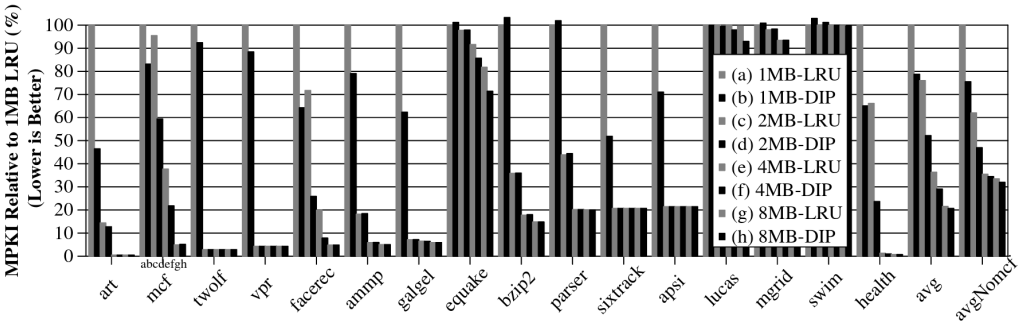


Figure: avg: Arithmetic mean MPKI for all the benchmarks, avgNomcf: Same as avg, but without mcf

Additional consideration: bypassing instead of LIP

In BIP as proposed so far, lines were inserted in either the MRU or LRU position. What if in the latter case we skipped inserting the line at all?

Additional consideration: bypassing instead of LIP

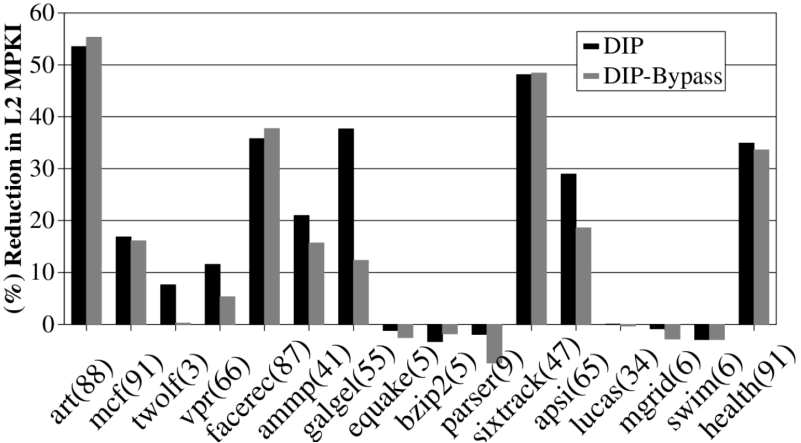


Figure: (n) means n% bypassed misses in DIP-Bypass.

Effect on instructions per cycle

Simulated system configuration

Machine width	4 instructions/cycle, 4 functional units
Inst. window size	32 instructions
Branch predictor	Hybrid 64k-entry gshare, 64k-entry PAs misprediction penalty is 10 cycles min.
L1-I cache	16kB, 64B block, 2-way with LRU repl.
L1-D cache	16kB, 64B block, 2-way, 2 cycle hit
L2 unified cache	1MB, 64B block, 16-way, 6 cycle hit
Main memory	32 banks, 270 cycle bank access
Off-chip bus	Proc. to bus speed ratio 4:1, 8B/bus-cycle

Effect on instructions per cycle

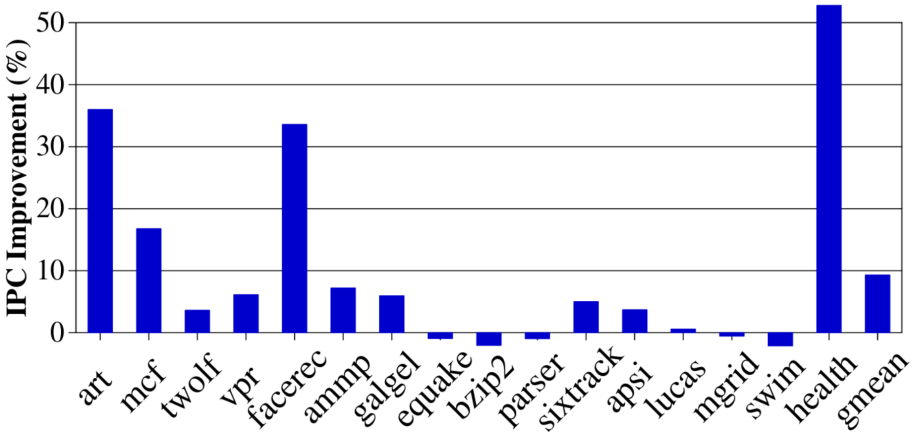
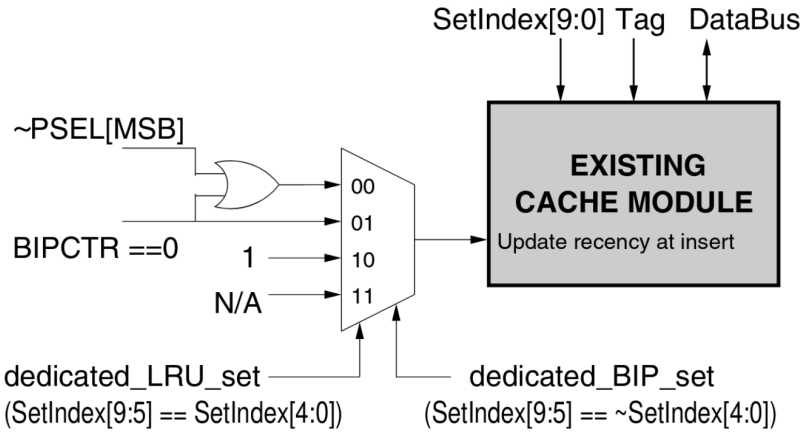


Figure: IPC improvement with DIP over LRU. gmean: geometric mean.

Hardware considerations



Conclusions

- The commonly used (Pseudo-)LRU cache policy performs well for working sets that are no larger than the cache, but suffers from thrashing otherwise.
- BIP solves the thrashing cases, but at the cost of severe performance loss with workloads that benefit from traditional LRU.
- DIP, a hybrid policy which adapts to the workload, performs better than either of the two policies it's made up of with minimal hardware changes.

References

R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, San Diego, California, USA. ACM, 2007.

Wikipedia. Pseudo-LRU — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Pseudo-LRU&oldid=912193587, 2019. [Online; accessed 01-December-2019].