Literature review

High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel Emer

Presented by Shubhayu Das, IMT2018523
Processor Architecture(VL-803) @IIITB

---

https://dl.acm.org/doi/10.1145/1816038.1815971

Cache replacement policy

- ➔ Policy finds the ideal cache block to evict(replace) with new data
- ➔ Ideal replacement policy has perfect knowledge of the future, always knows *exactly* which block to remove(called Belady's OPT)
- ➔ Real world policies are not so fortunate, use past history to make better predictions

---

1. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement, Akanksha Jain, Calvin Lin [Hawkeye replacement policy]
2. Hawkeye implementation for ChampSim on GitHub

Different Terminology

- ➔ **Re-Reference Interval(RRI)** - time* after which a particular cache block will be accessed again
- ➔ This is equivalent to the older concept of *recency.*

Most recently used(MRU) → near-immediate RRI

Least recently used(LRU) → distant RRI

* Time refers to the number of overall accesses to the cache, rather than the number of CPU cycles.

The need for new predictors - Cache access patterns (1/2)

```
for uint16 j:1 → n
        for float32 i:1→ 32:
                updateParams(array1[i], array2[j]);
        endfor
endfor
```

➔ Same elements(array1) accessed repeatedly, all elements fit in cache(128kb/core).
➔ LRU works perfectly fine.
➔ Eg: digital filters(FIR/IIR)

1. Recency friendly access pattern

```
for uint32 j:1 → n
        for float32 i:1→ 10^4:
                updateWeights(array1[i], array2[j]);
        endfor
endfor
```

➔ Same elements accessed repeatedly, all elements **do not** fit in cache.
➔ Eg: ML training workload

2. Cache thrashing access pattern

The need for new predictors - Cache access patterns (2/2)

while (dataStream[i] is not NULL)

    function(dataStream[i])

    i++

endwhile

- ➔ No re-use of data at all
- ➔ No replacement policy works here.
- ➔ Eg: playing a wav music file

3. Streaming access pattern

for uint32 j:1 → n

    array2 = convolution(array1, reference);

    digitalFilter(array2, coeffs);

endfor

- ➔ Some elements accessed repeatedly, all elements **do not** fit in cache.
- ➔ Elements that don't fit lead to *scans* where the relevant data might get replaced depending on the replacement policy

4. Mixed access pattern

Attributes of new replacement policy

- ➔ Preserve LRU's performance for recency favouring workloads
- ➔ Reduce cache thrashing
- ➔ Be scan resistant *to a certain extent,* preventing replacement of useful data with other data

Contents in this slide are not directly mentioned in paper

Cache replacement policy

- ➔ Policy finds the ideal cache block to evict(replace) with new data
- ➔ Ideal replacement policy has perfect knowledge of the future, always knows *exactly* which block to remove(called Belady's OPT)
- ➔ Real world policies are not so fortunate, use past history to make better predictions

---

1. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement, Akanksha Jain, Calvin Lin [Hawkeye replacement policy]
2. Hawkeye implementation for ChampSim on GitHub

Different Terminology

- ➔ **Re-Reference Interval(RRI)** - time* after which a particular cache block will be accessed again
- ➔ This is equivalent to the older concept of *recency.*

Most recently used(MRU) → near-immediate RRI

Least recently used(LRU) → distant RRI

* Time refers to the number of overall accesses to the cache, rather than the number of CPU cycles.

The need for new predictors - Cache access patterns (1/2)

```
for uint16 j:1 → n
        for float32 i:1→ 32:
                updateParams(array1[i], array2[j]);
        endfor
endfor
```

➔ Same elements(array1) accessed repeatedly, all elements fit in cache(128kb/core).
➔ LRU works perfectly fine.
➔ Eg: digital filters(FIR/IIR)

1. Recency friendly access pattern

```
for uint32 j:1 → n
        for float32 i:1→ 10^4:
                updateWeights(array1[i], array2[j]);
        endfor
endfor
```

➔ Same elements accessed repeatedly, all elements **do not** fit in cache.
➔ Eg: ML training workload

2. Cache thrashing access pattern

The need for new predictors - Cache access patterns (2/2)

while (dataStream[i] is not NULL)

    function(dataStream[i])

    i++

endwhile

- ➔ No re-use of data at all
- ➔ No replacement policy works here.
- ➔ Eg: playing a wav music file

3. Streaming access pattern

for uint32 j:1 → n

    array2 = convolution(array1, reference);

    digitalFilter(array2, coeffs);

endfor

- ➔ Some elements accessed repeatedly, all elements **do not** fit in cache.
- ➔ Elements that don't fit lead to *scans* where the relevant data might get replaced depending on the replacement policy

4. Mixed access pattern

Attributes of new replacement policy

- ➔ Preserve LRU's performance for recency favouring workloads
- ➔ Reduce cache thrashing
- ➔ Be scan resistant *to a certain extent,* preventing replacement of useful data with other data

Contents in this slide are not directly mentioned in paper

Proposals for:

Static Re-Reference Interval Prediction(SRRIP)
&
Dynamic Re-Reference Interval Prediction(DRRIP)

Setup for SRRIP

- **RRPV:** Re-reference prediction value - an M-bit counter, which indicated the re-reference interval(RRI), equivalent to recency
- Low RRPV(closer to all 0s) → near-immediate RRI
- Long/Intermediate RRPV → RRPV = 2^M - 2
- Distant RRPV→RRPV = 2^M - 1

Tag	Dirty	Data[63:0]	M(=4) bit RRPV
…..	0/1	…….	0000
…..	0/1	…….	0010
…..	0/1	…….	1111

SRRIP algorithm

On a cache miss:

1. Search for distant RRPV(2^M - 1) in a particular order
2. If found, skip to step 5
3. Increment all RRPV
4. Go to step 1
5. Replace block, set RRPV to intermediate RRPV value(2^M - 2)

On a cache hit:

1. **SRRIP-Hit Priority(HP):** Set RRPV of block to *0* (most recently used, so lowest RRI)
2. **SRRIP-Frequency Priority(FP):** Set RRPV of block to 2^M - 2, to prevent long occupation of unnecessary data blocks

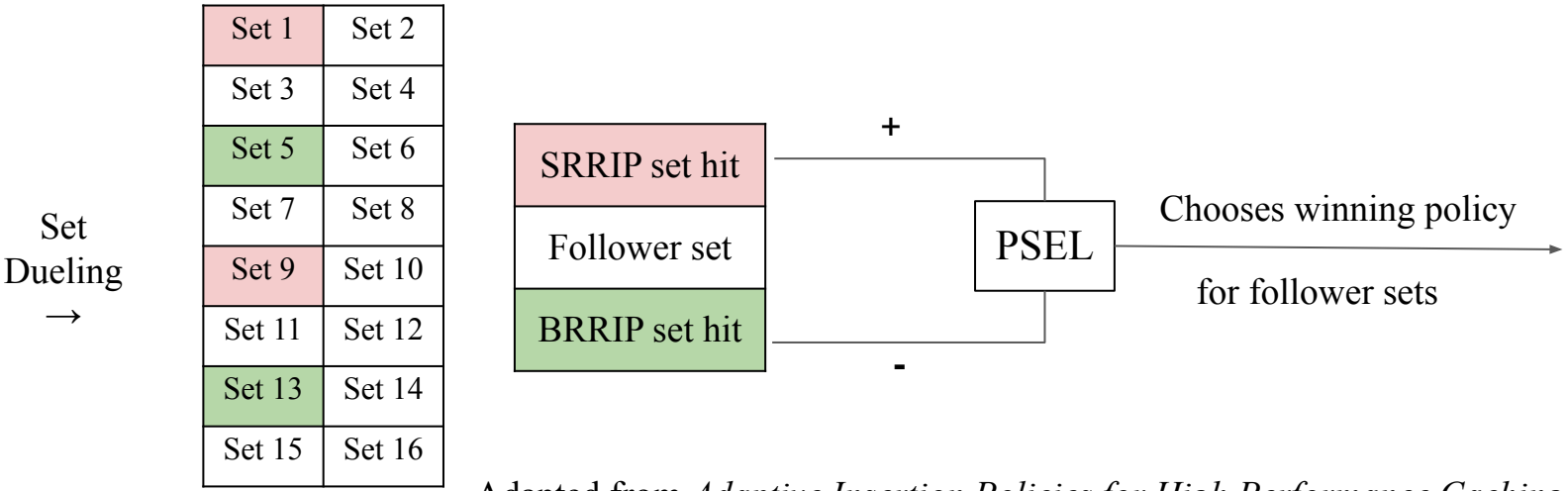
For an example, see Figure 3 in the paper

BRRIP algorithm - an intermediate

- ➔ SRRIP is scan resistant to a better extent
- ➔ SRRIP's performance drops when all data blocks have a large RRI
- ➔ As a solution, we enter majority of entries with RRPV=2^M - 1, and the rest at RRPV=2^M - 2.(termed Bimodal RRIP, BRRIP)
- ➔ BRRIP is significantly more thrash resistant

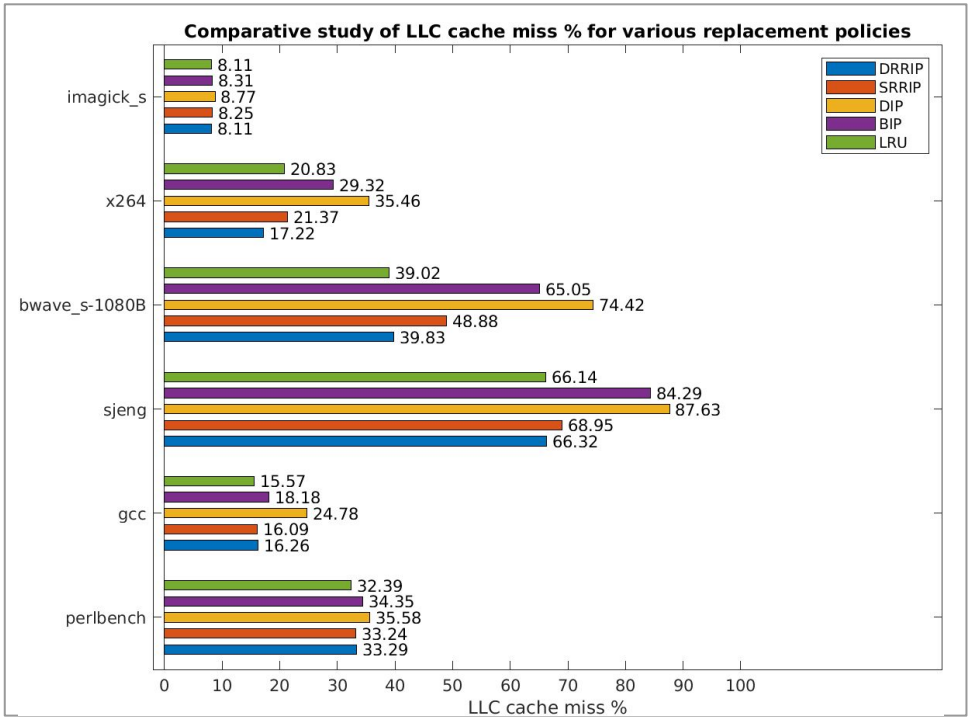
DRRIP algorithm

- ➔ SRRIP is scan resistant, BRRIP is thrash resistant
- ➔ Combine both using *Set dueling* to form DRRIP
- ➔ Best of both worlds



Adapted from *Adaptive Insertion Policies for High Performance Caching, Qureshi et.al.*

Performance analysis - ChampSim simulation



With few exceptions, DRRIP and SRRIP always seem to perform better than other policies like LRU, BIP and DIP.

The exact simulation details, code and results can be found here: https://github.com/vsdevaraddi/PA_Champsim_assignment

Results in the paper also have similar findings.

Lower is better, gshare branch predictor was used, with no prefetching