MSA TERM PROJECT

RESEARCH PROJECT

INTRODUCTION TO CACHE REPLACEMENT POLICY

1. Memory Wall is a big Issue in Computer architecture To Overcome that Optimsation of Cache plays an important role
2. Cache Replacement Policy Can play a major role in Cache Optimisation. Right Replacement policy according to the task can make a huge difference
3. Mainly Cache Replacement comes into play when there is a miss and a block has to be replaced in cache, to choose the victim block Replacement Policy helps.

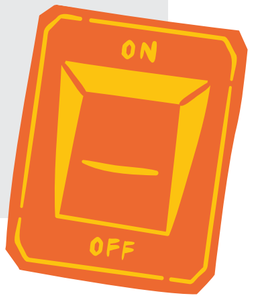
EXISTING REPLACEMENT POLICIES

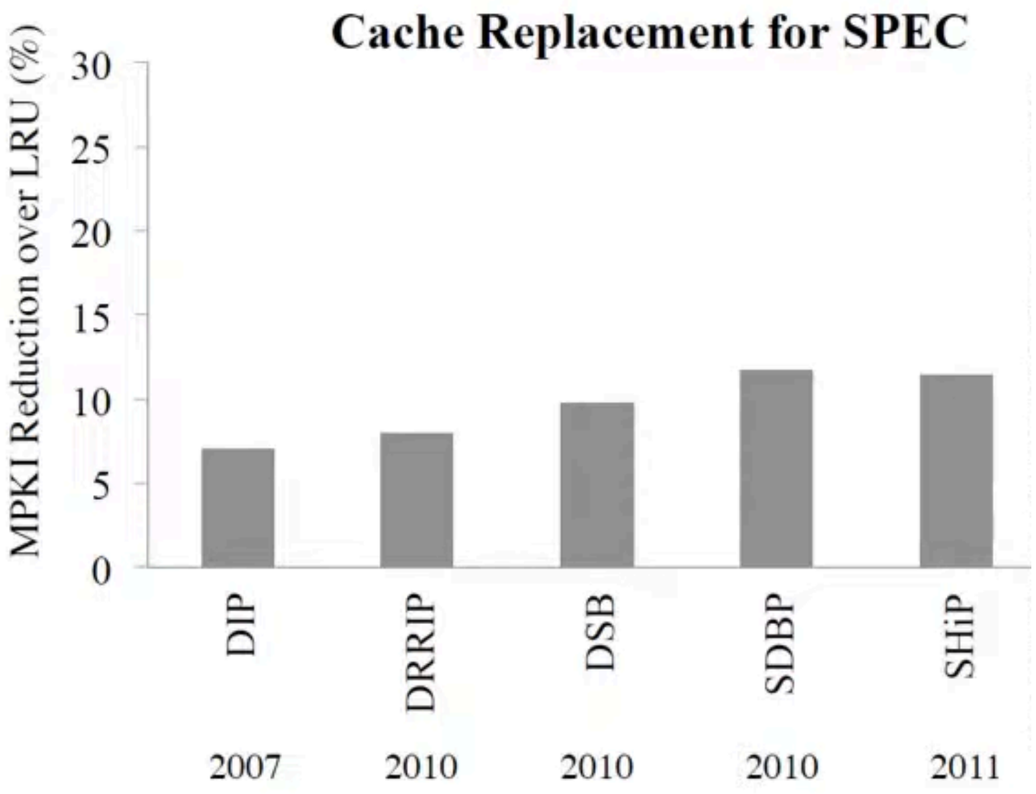
Common Policies

- Random Replacement Policy
- LRU Replacement Policy
- FIFO Replacement Policy
- LFU Replacement Policy
- MRU Replacement Policy

Advanced Policies

- DIP
- DRRIP
- DSB
- SDBP
- SHiP





SHORTCOMINGS OF EXISTING POLICIES

As due to Limited Medium term and Long Term Adaptability, High Hardware Overhead,Sensitivity to Workload Variability we need a new policy that adapt and learn from recent data and adapt according to access pattern also improve in SPEC benchmark from Previous Policies as shown in Figure

HAWKEYE : KEY IDEA

Binary Classification:

- Cache Friendly
- Cache Averse

Past is a good indicator of Future:

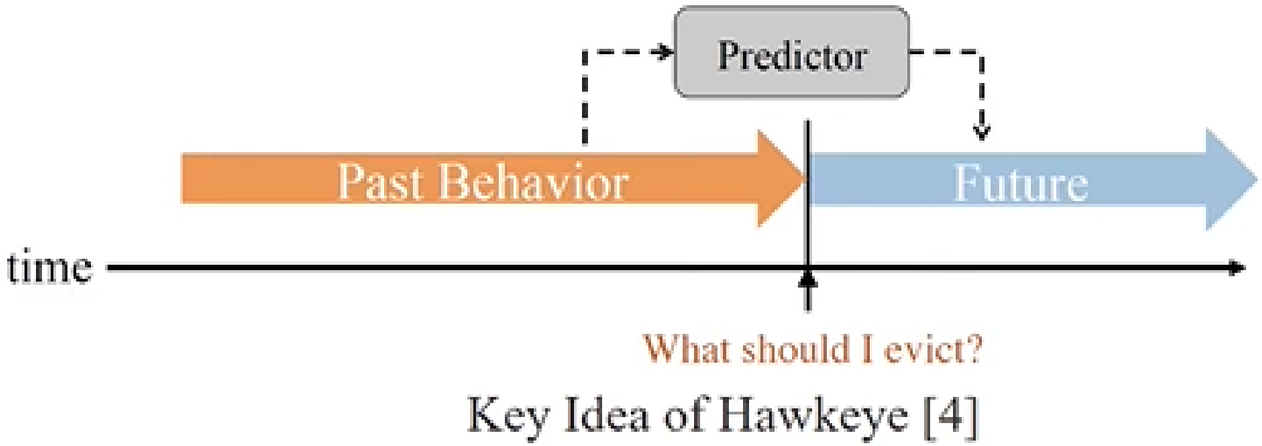
- If a block was hit , then most likely to be hit again and vice - versa

Applies Beladys OPT to past Events

Train predictor per-Pc based on past OPT behaviour.

The Predictor guides on

- Insertion decisions
- Eviction Decisions



Key Idea of Hawkeye [4]

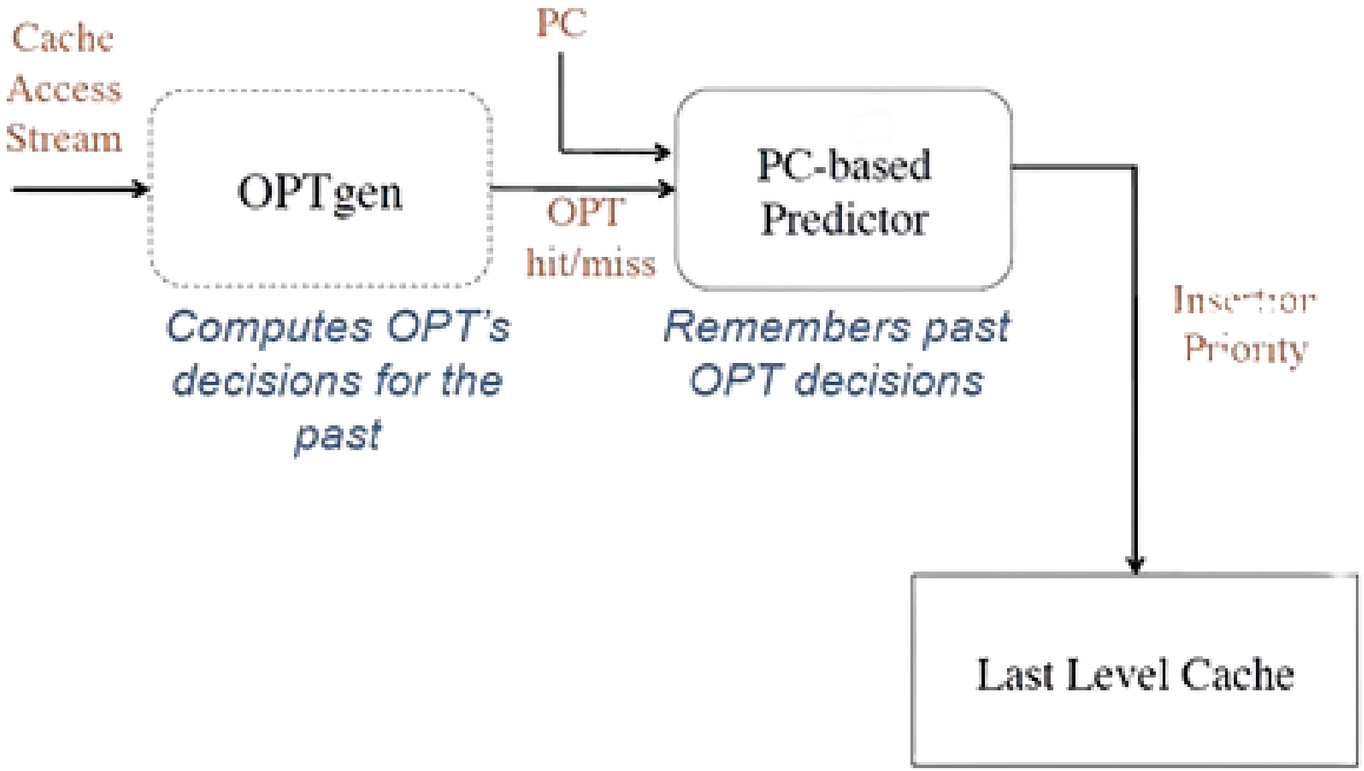
HAWKEYE : COMPONENTS

OPT GEN

- Simuluate the Beladys OPT for the past
- Train the predictor
- Stores the history of past accesses.

Haweye Predictor

- A simple look-up table
- Tracks cache behaviour as per PC
- Classifies cache insertions as :
  - Cache friendly
  - Cache Averse
- RRIP Values
  - 0x7 : Cache Averse
  - 0x0 - 0x6 : Cache friendly



HAWKEYE : OPT GEN

Recreates belady's algo

Occupancy vector (OV)

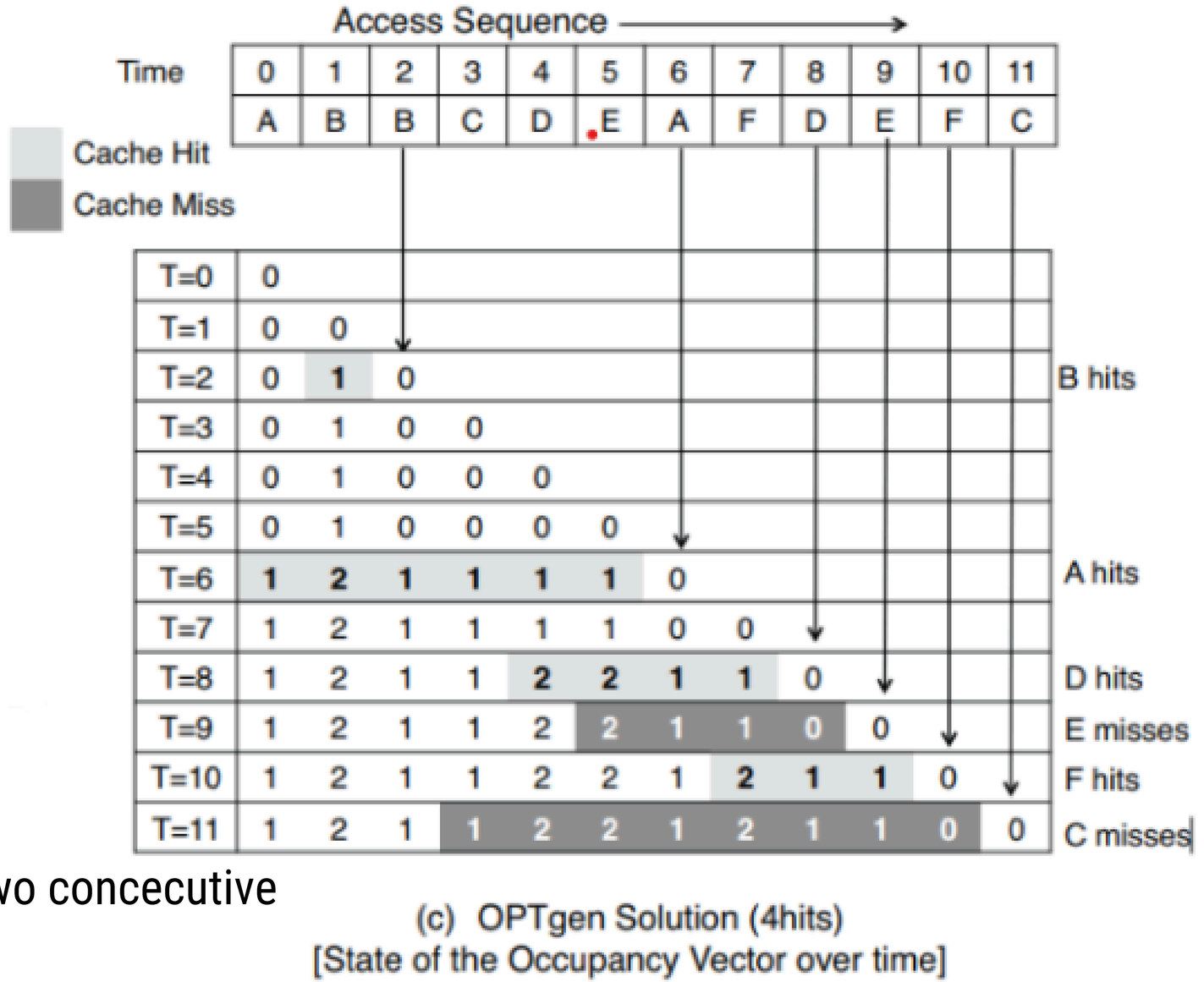
- Number of active lines within that set

Sampled cache

- To track usage history
- obtain timestamp of last access

Two main parameters

- Usage interval : Time interval between two concecutive access of same block
- Liveness Interval : Amount of time for which a line is active in a cache



(c) OPTgen Solution (4hits)
[State of the Occupancy Vector over time]

HAWKEYE : HAWEYE PREDICTOR

Indexed by current PC

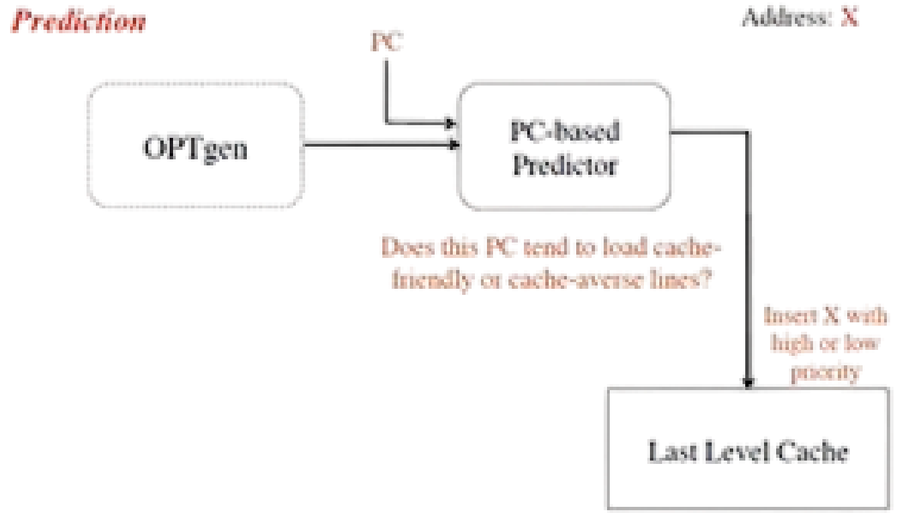
3-bit Counter

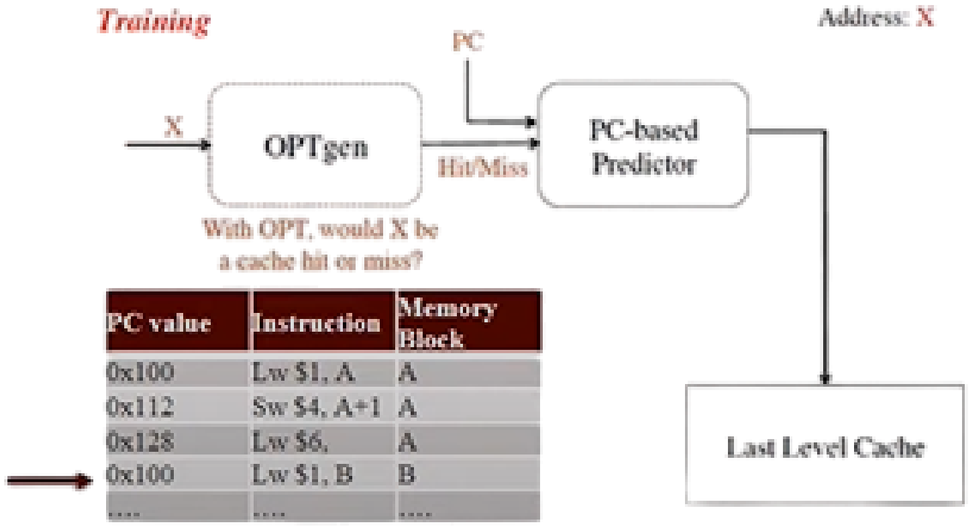
- 0 -> 3 : Cache Averse
- 4 - > 7 : Cache friendly

OPT gen

- Hit -> Train positively
- Miss -> train negatively

Predict before training





PC value	Instruction	Memory Block
0x100	Lw $1, A	A
0x112	Sw $4, A+1	A
0x128	Lw $6,	A
0x100	Lw $1, B	B
....	....	....

HAWKEYE : SUMMARY

Only affects Access or Insertion

- Prediction is cache Averse -> RRIP = 7
- Prediction is cache friendly -> RRIP = 0

Hawkeye Prediction | Hit or Miss	Cache Hit	Cache Miss
Cache Averse	RRIP = 7	RRIP = 7
Cache Friendly	RRIP = 0	RRIP = 0; Age all lines i.e. If (RRIP <6) RRIP++

- Train Haweye Predictor

Victim Selection

- Based on RRIP value
- If cache friendly is evicted, train last PC negatively

SCOPE OF IMPROVEMENT IN HAWKEYE

1. **FINE-TUNING PARAMETERS**
   Analyze the performance of Hawkeye across different workloads and adjust its parameters accordingly.
2. **MULTI-LEVEL HIERARCHICAL APPROACH**
   Extend Hawkeye to support a multi-level hierarchical cache structure, where different replacement policies are applied at each cache level.
3. **INTEGRATION WITH PREFETCHING MECHANISMS**
   Integrate Hawkeye with prefetching mechanisms to proactively fetch data into the cache before it is accessed.
4. **PARALLELISM AND SCALABILITY**
   Optimize Hawkeye for parallel execution and scalability in multi-core and distributed computing environments.

POSSILE APPROACHES THAT WE CONSIDERED

Initial thoughts to work on improving hit rate latency

Possibilties that we considered-

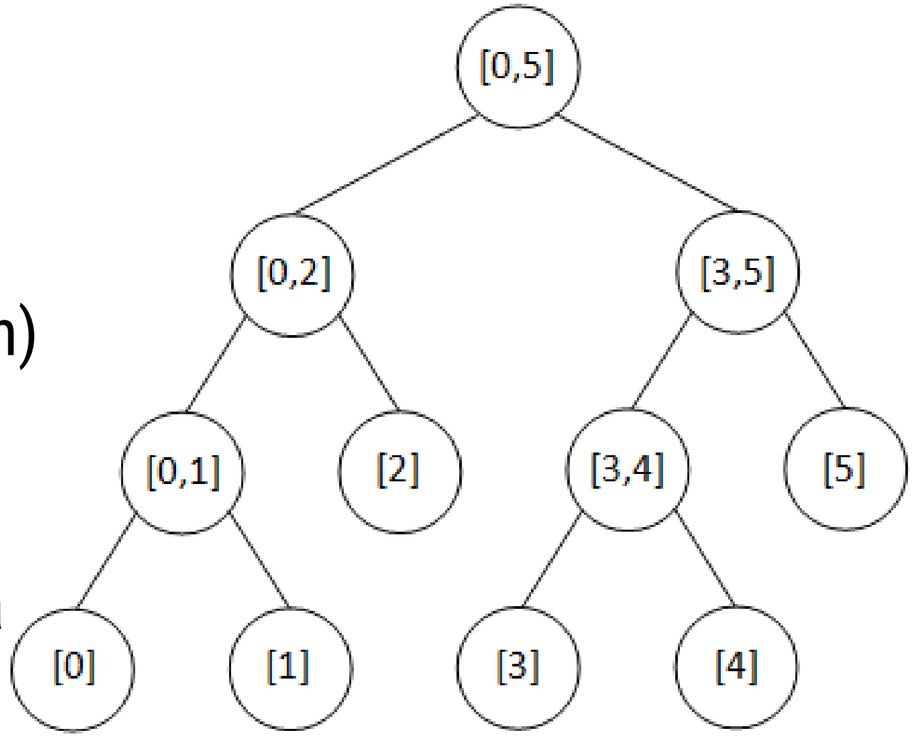
1. **Improve Occupancy Vector of Hawkeye**
2. **Improve The Sampled Cache of Hawkeye**

IMPROVING OCCUPANCY VECTOR

In Code Occupancy Vector Iteratively Look for Previous Element Greater than Size of Cache Taking O(n)

LEVERAGING POWER OF SEGMENT TREES

- Occupancy vector Implemented using vector<deque>
- Greater Lookup is O(n)
- Implement Segment tree with Lazy Propogation Makes a lookup O(log(n))



FIRST ATTEMPT

Given Code looks up iteratively to find the element greater than LLC_WAY

```
for(int32_t index = occupancy_vector[set].size()-2; index >= prev_occurence; --index)
{
    assert(occupancy_vector[set][index] <= LLC_WAY);
    if(occupancy_vector[set][index] == LLC_WAY)
    {
        opt_miss = true; // OPT would have produced a miss
        break;
    }
```

- *Point to Note is,* ***On a miss this improvisation will not reflect****.*
- *The time taken to fetch data from Secondary memory(~200 cycles) will overlap with our improve in Latency*
- *But* ***in Case of a Hit this approach might improve the IPC****.*

SECOND ATTEMPT

After unsuccessful attempt of implementing segment tree we decided to implement a rather simple implementation of vector of hash set

RESULT?

A rather lower hit rate of ~61% as that of 70% of hawkeye on CC6 benchmark.

WHY?

USAGE OF VECTOR OF HASH SET IN OCCUPANCY VECTOR

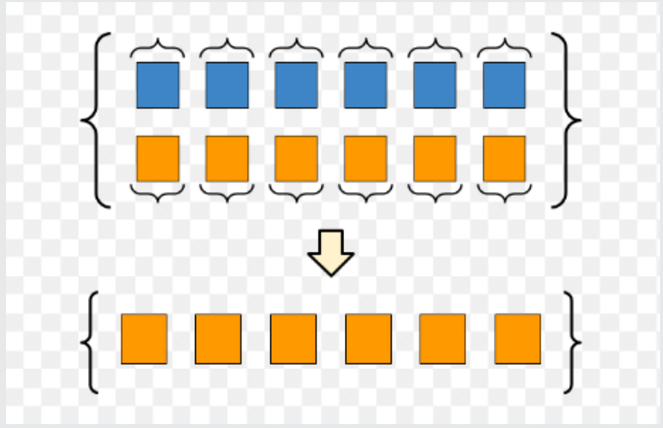
Stores unique values in sorted order but donot retain the indices

THIRD ATTEMPT

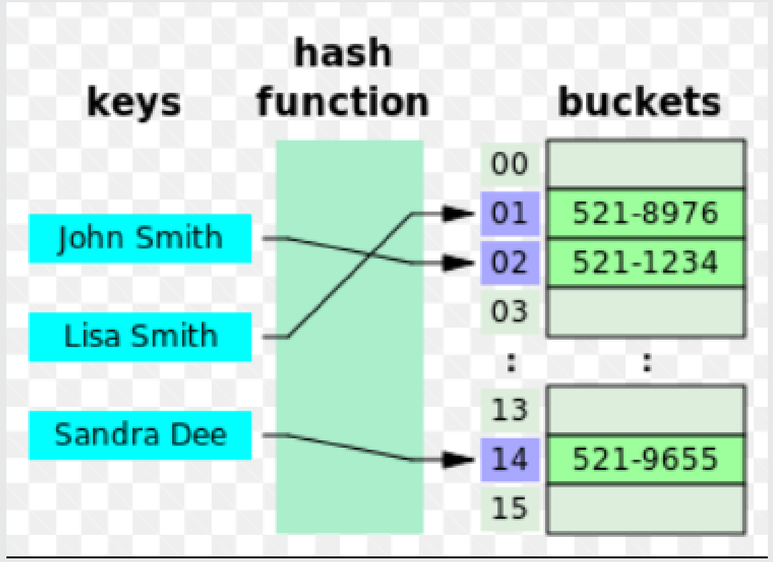
Now we started targeting the sampled cache and in what manner is the last index is retrieved.

```
for(int32_t index = access_history[set].size()-2; index >= 0; --index)
{
    if(access_history[set].back().first == access_history[set][index].first)
    {
        prev_occurence = index;
        break;
    }
}
```

Is there any scope of improvement here?



A vector of pairs



Hash map

```
if(mp.find(val) != mp.end()) prev_occurence = mp[val];
```

A FEW OBSERVATIONS WE MADE

- access_history is essentially a vector of vector of pair consisting of address and timestamp.
- There is no need to reiterate from index n - 2 to 0 each time.

SOLUTION

Hashing

Simply use HASH map to store or rewrite the current address.

if (never occured) map[address] = -1
else map[address] = current time stamp.

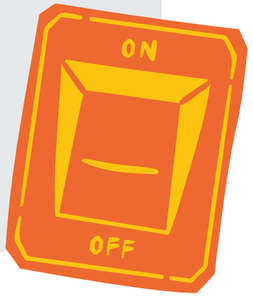
RESULTS

Initially after the simulation the hit rate fell to ~58%.

```
if(access_history[set].size() == history_len)
{
    stats.update.access_history_spill++;
    auto it = access_history[set].front();

    if(mp.find(it.first) != mp.end()) mp.erase(mp.find(it.first));
    access_history[set].pop_front();
}
access_history[set].push_back(pair<uint64_t, uint64_t>(ca_addr, timestamp[set]));
mp[ca_addr] = timestamp[set];
```

We actually never require to
delete an entry from the map

WHY?



RESULTS

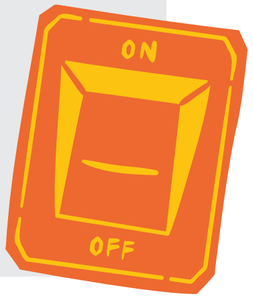
After commenting out that part we got hit rate of astonishing 70%.

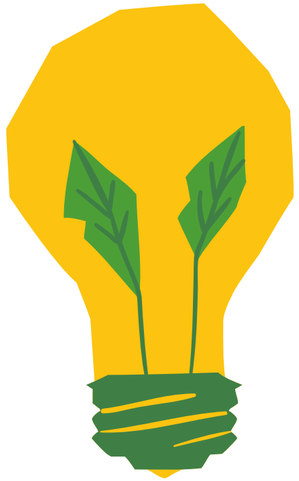
```
Core_0_LLC_total_access 1946680
Core_0_LLC_total_hit 1381877
Core_0_LLC_total_miss 564803
Core_0_LLC_loads 1739200
Core_0_LLC_load_hit 1179649
Core_0_LLC_load_miss 559551
Core_0_LLC_RFOs 0
Core_0_LLC_RFO_hit 0
Core_0_LLC_RFO_miss 0
Core_0_LLC_prefetches 0
Core_0_LLC_prefetch_hit 0
Core_0_LLC_prefetch_miss 0
Core_0_LLC_writebacks 207480
Core_0_LLC_writeback_hit 202228
Core_0_LLC_writeback_miss 5252
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_dropped 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_filled 0
Core_0_LLC_prefetch_useful 0
Core_0_LLC_prefetch_useless 0
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 180.67
```

Our implementation

```
Core_0_LLC_total_access 1946680
Core_0_LLC_total_hit 1381880
Core_0_LLC_total_miss 564800
Core_0_LLC_loads 1739200
Core_0_LLC_load_hit 1179723
Core_0_LLC_load_miss 559477
Core_0_LLC_RFOs 0
Core_0_LLC_RFO_hit 0
Core_0_LLC_RFO_miss 0
Core_0_LLC_prefetches 0
Core_0_LLC_prefetch_hit 0
Core_0_LLC_prefetch_miss 0
Core_0_LLC_writebacks 207480
Core_0_LLC_writeback_hit 202157
Core_0_LLC_writeback_miss 5323
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_dropped 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_filled 0
Core_0_LLC_prefetch_useful 0
Core_0_LLC_prefetch_useless 0
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 180.64
```

Base code





BUT,

We wondered even after optimising the code why wasn’t the result that promising?

The answer lies on the below two reasons:

- Authors did not account for the latency in iterating over the loop (Huge liveness is less likely)
- Hardware implementation is not based on STL ds rather than on multiplexers

YOUR SUGGESTIONS

IMPLEMENTING CACHE PARTIONING

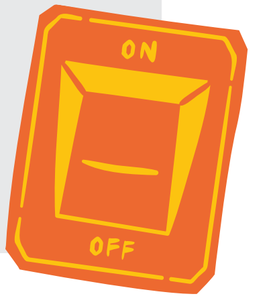
Set-Wise Partitioning

1. Approced the Partitioning Problem thorugh Dynamic Partitioning
2. Expected an increase in Hit Rate because of Late eviction of Important Block
3. Finally Settled to Static Partitioning with Total of 4 Partitions.

- Implemented the Partition based on Instruction Pointer “ip”.
- Calculated hash value of “ip” then modulus operator decides the partition

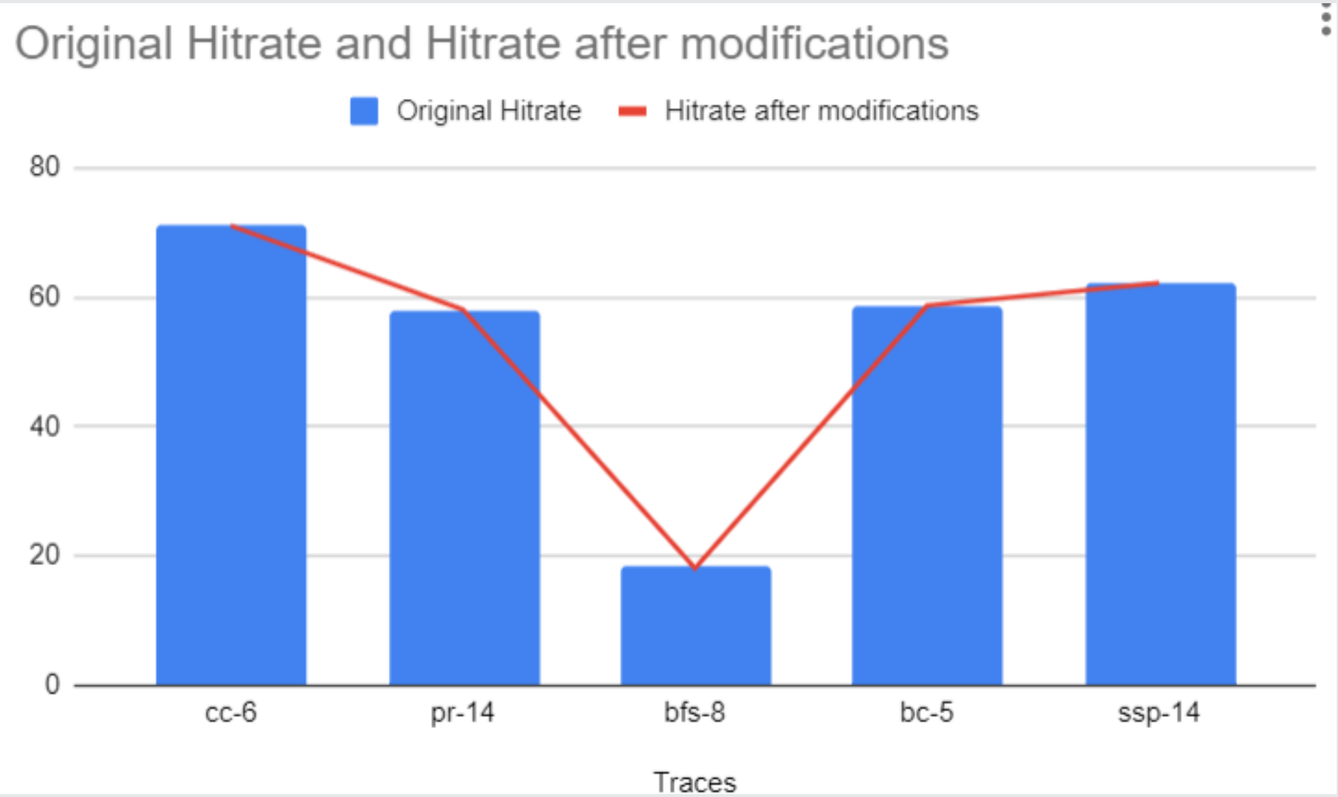
```
uint32_t determine_partition(uint64_t ip) {
    // Calculate a hash value based on the instruction pointer
    uint32_t hash = static_cast<uint32_t>(ip);

    // Modulo hash by the number of partitions to determine the partition index
    uint32_t partition = hash % knob::num_partitions;

    return partition;
}
```

Implementing this Improved the Cache Stats



RESULTS

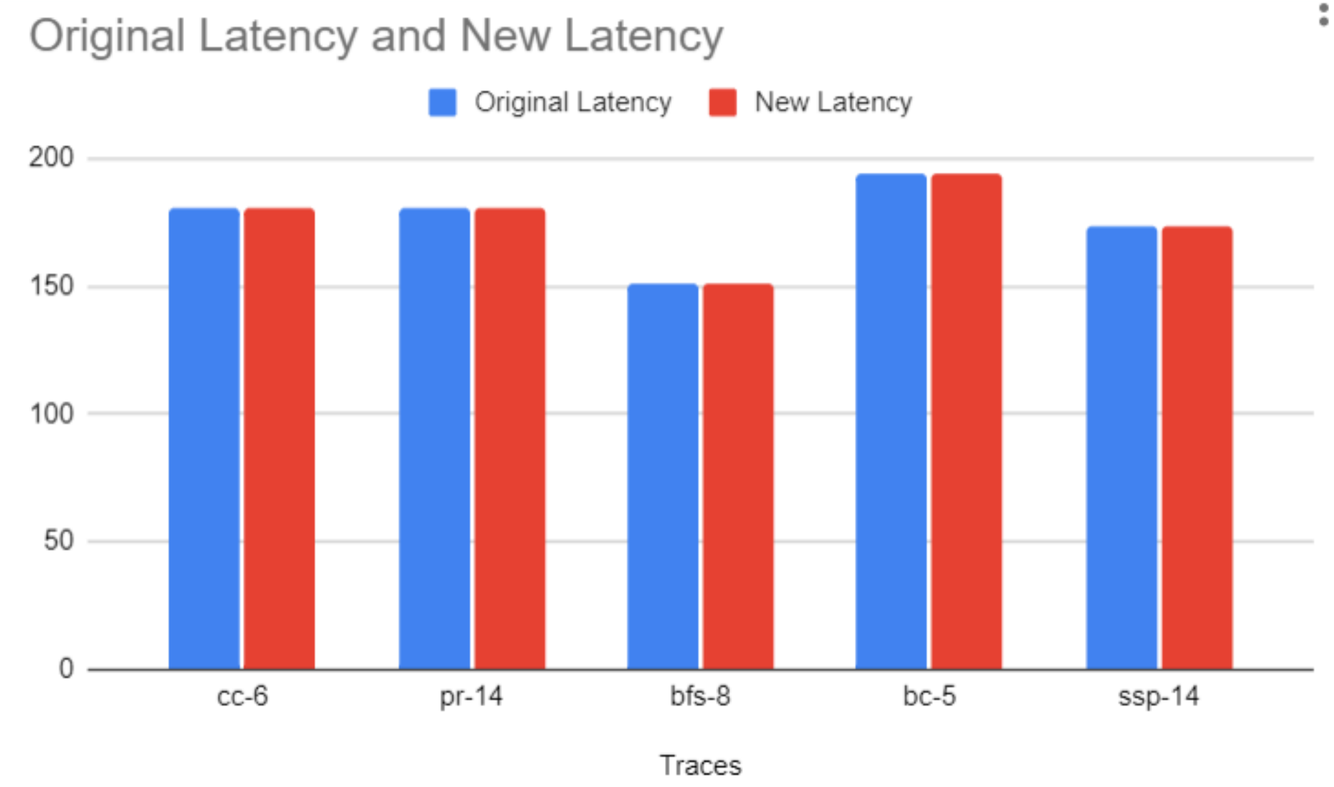
We observed an increase in hit rate for most of the traces



Traces	Original Hitrate	New Hitrate
cc-6	70.98	71.02
pr-14	57.82	58.05
bfs-8	18.27	18.11
bc-5	58.65	58.7
sssp-14	62.19	62.2

RESULTS

We observed an decrease in miss latency for most of the traces



Traces	Original Latency	New Latency
cc-6	180.64	180.58
pr-14	180.75	180.5
bfs-8	150.62	151.03
bc-5	193.95	194.27
sssp-14	173.61	173.53

THANK YOU ALL