Project Report

Reena Elangovan and Jeffry Louis

Abstract:
Belady's optimal policy for cache replacement provides the theoretical best performance but requires oracle knowledge for implementation. In this project we implement Hawkeye replacement policy that relies on past accesses to predict future and approaches Belady's performance. Further, we propose to compare the performance of Hawkeye predictor to other replacement policies such as LRU and SDBP. SDBP is a sampling based dead-block predictor. We achieve an average miss reduction of 26.36% over LRU across 8 SPEC-2006 benchmarks. We also achieve a performance improvement of upto 9.9% across these benchmarks. The github links to the implementation of the Hawkeye predictor and SDBP are given below.

Hawkeye Predictor: https://github.com/reena855/gem5.git
SDBP: https://github.com/purdue-ece565-2020/assignment-1-v4-Jeffry-github

1. Introduction

Efficient Cache Replacement policies improve the cache hit-rate, which in turn improves the overall performance of the processor by hiding the DRAM access latency. The existing replacement policies are based on heuristics such as LRU (least-recently used), MRU (most-recently used) etc., and work well only for specific access patters. In contrast, Belady's algorithm provides a theoretical best policy to achieve maximum possible hit-rate in caches. Therefore, a cache replacement algorithm based on Belady's algorithm is highly desired. The Hawkeye cache replacement policy proposed by [1] predicts the future cache access patterns using the decisions made by the Belady's algorithm. In doing so, it approaches Belady's behavior.

Since Belady's algorithm is impractical because it requires oracle knowledge, [1] applies a variant to the history of past cache accesses. Using the Belady's optimal solution (OPT) for the past accesses as a guide, Hawkeye predicts the future accesses.

Hawkeye consists of two parts

- OPTgen algorithm (computes optimal solution for the past)
- Hawkeye Predictor (learns from OPT to make future predictions)

OPTgen determines what accesses would have hit in a cache if the OPT policy had been used. An occupancy vector generated by OPTgen simulates the Belady's policy for the past accesses and keeps record of the liveliness of the cache blocks. Based on OPTgen, the Hawkeye predictor learns whether an access by a given PC would results in a hit or a miss under the OPT policy. The entries of the predictor are indexed by 13-bit hashed PC and uses 3-bit counters for training. The counters are incremented during a hit under OPT policy and decremented during a miss. The MSB of the counter indicates whether a cache line is (1) cache-friendly or (0) cache-averse. The proposed cache replacement policy associates an RRIP value to each cache line and updates it based on the hawkeye prediction to determine the cache line to be evicted.

We propose to compare the effectiveness of the Hawkeye replacement policy with the SDBD policy proposed by [2]. The SDBP policy is based on predicting and replacing the dead blocks of LLCs, thereby reducing the number of off-chip memory accesses and increasing the overall system performance. Most of the current dead block prediction algorithms are based LRU which is expensive to implement in highly associative caches [3]. In contrast, SDBD proposes a simple sampling-based predictor that is comparatively more power efficient. One technique for

predicting dead blocks better is to have a partial tag array (sampler) and a skewed sampling predictor made of three prediction tables [1]. Each set in the sampler corresponds to a selected set in the cache. Each access to the LLC incurs an access to the predictor. However, the predictor is only updated when there is an access or replacement in a cache set with a corresponding 3 sampler set. This reduces the power compared to other predictors as the predictor and sampler are updated only on a small fraction of cache accesses and replacements

2. Related Work

Cache replacement policies can be divided into three types depending on the type of information they use. They are Short term History information, Long term history information and Future Information. The Short-term history solutions are based on heuristics specific to cache access patterns. For example, LRU prioritizes recently used cache blocks as they might be used again. Jaleel et.al., used 2 bits of re-reference interval prediction which helps in avoiding cache pollution due to streaming access [4]. Thrash resistant policies are most suitable for workloads with large reuse distances [5]. Qureshi et al. proposed Dynamic Set Sampling (DSS) [5], a mechanism that chooses the best policy after sampling the cache to assess the efficiency of the desired policy.

SHiP and DBP are examples of policies that use Long term history information for predictions. SHiP identifies instructions that load streaming accesses [6] while Dead Block Prediction (DBP) collects the trace of instructions which accesses a particular block [7]. An improvement over DBP is SDBP which samples certain cache blocks and uses the PC to predict if a cache block is dead or not [2]. The future-information based class of replacement policies takes inspiration from victim caches [8] and defers replacement decisions to the future when more information is available. For example, the Shepherd Cache [9] emulates OPT by deferring replacement decisions until future reuse can be observed, but it cannot emulate OPT accurately because it uses an extremely limited window into the future.

3. Hawkeye Replacement Policy

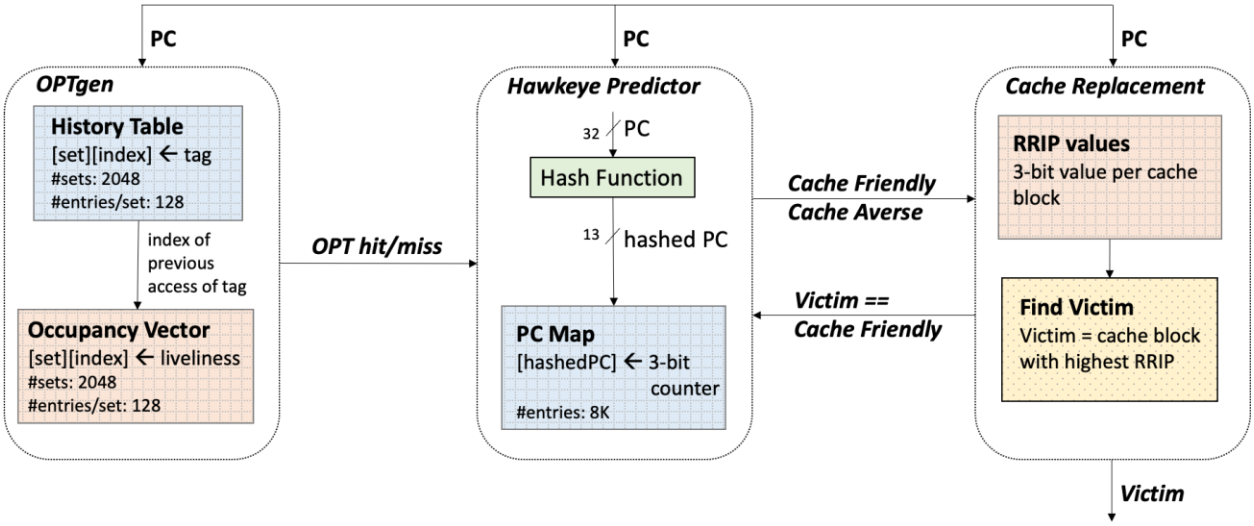


Figure 1: Hawkeye Replacement Policy Implementation in Gem5

We detail the implementation of the main components – OPTgen, Hawkeye Predictor, Cache Replacement in this section. Figure 1 shows a high-level implementation of this policy. The OPTgen algorithm recreates the Belady's optimal policy (OPT) and determines OPT hit/miss based on PC and previous access information from the history table. The Hawkeye Predictor updates the PC map of 8K 3-bit counters indexed by 13-bit hashed PC based on the

OPT hit/miss. Further, it classifies the current access as cache-friendly or cache-averse based on the counter value in the PC map. Finally, the cache replacement policy associates an RRIP value to each cache block based on the cache friendliness prediction. During a miss, the cache block with the highest RRIP value is selected as the victim and if the victim block was previously predicted as cache-friendly, the corresponding entry in the PC map of the Hawkeye Predictor is decremented.

Note that the sizes and assumptions for each of the components is exactly the same as in [1]. Let us now discuss the implementation of each of the components in detail.

3.1. OPTgen

The OPTgen algorithm utilizes a history table to track the previous accesses. One history table is maintained per set and each history table contains 128 entries (8 times the associativity of the LLC). The occupancy vector (OV) determines the liveliness of a cache block in LLC based on the history table. Similar to the history table, one OV is maintained per set and each OV has 128 entries. Each element of the occupancy vector has the total number of overlapping liveliness intervals at a given time.

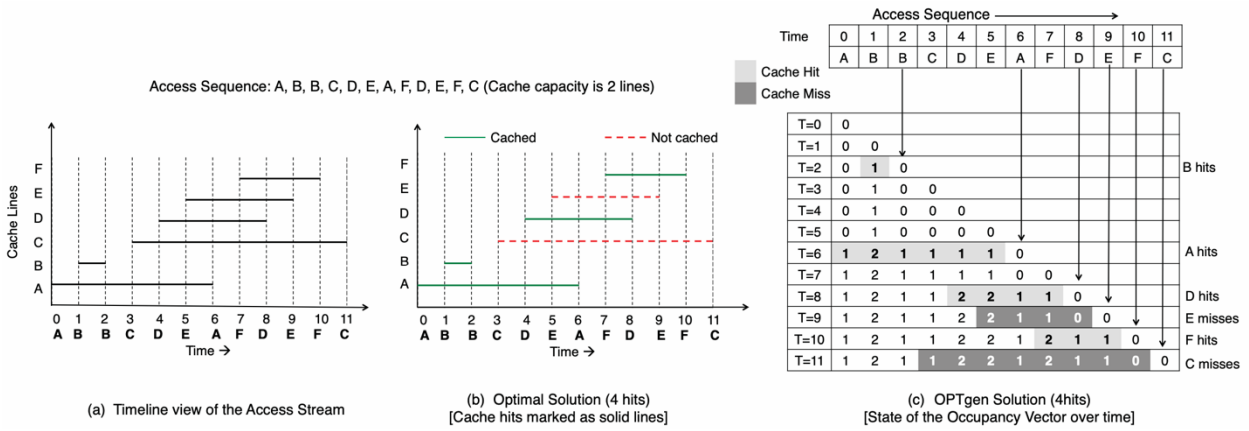


Figure 2: OPTgen algorithm to re-create Belady's optimal policy using past access history

For instance, Figure 2.a shows an example access pattern along with the usage interval (time between first and last access) of the cache blocks A to F. In a bypassing cache, only the blocks A, B, D and F are predicted by the OPTgen algorithm to be cache hits. Therefore, only these blocks are cached as shown in Figure 2.b. The OPTgen algorithm relies on the occupancy vector (OV) to predict hits/misses under the Belady's policy. For each set in the LLC, an OV of length 8 times the associativity of the LLC is maintained. The update methodology of the OV is detailed in Figure 2.c. The OV is updated during every access of the LLC. The OV element corresponding to the current access is initialized to zero. During a re-reference of a cache block, if the value of the OV elements staring from the past reference of the block to the current reference are less than the capacity (associativity in a set-associative cache) of the cache, OPTgen classifies the current reference as OPT hit. Otherwise, it is an OPT miss. An example hit scenario is shown in Figure 2.c at Time = 6 when block A is re-referenced. We can see from the OV at T=5 that the elements starting from the first reference of A to the current reference is less than 2 (Here the cache capacity = 2). So, at T=6 the corresponding OV elements are incremented to record the OPT hit for A. The access to block E misses at Time = 9 since the OV has a value of 2 (full capacity) at Time = 5 (i.e, between the previous and current access of E).

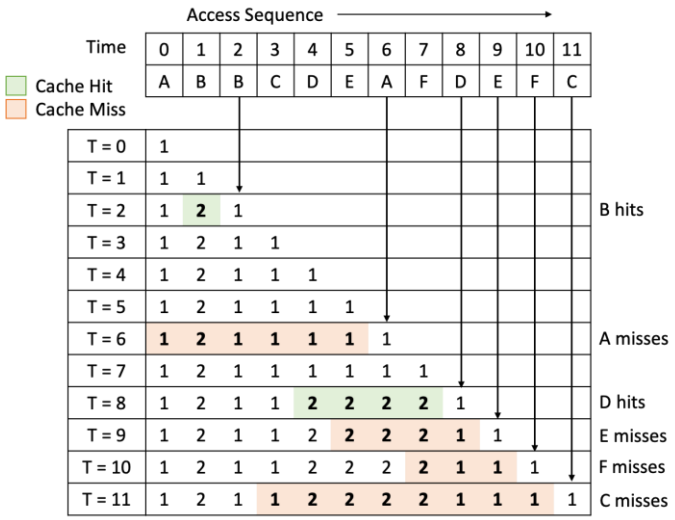


Figure 3: Modified OPTgen algorithm for non-bypass cache

The original Hawkeye implementation has assumed bypassing last-level cache. However, since gem5 does not currently support bypassing cache, we assumed non-bypassing cache instead. We follow the guidelines in the paper to implement OPTgen with non-bypassing cache and initialize the OV element corresponding to the current access to 1 instead of 0. Now, for the same access pattern as Figure 2.c, we show the OV update in Figure 3. We obtain 2 OPT hits and 4 misses for the non-bypassing cache. Although this is worse than the bypassing cache, it is better compared to our baseline LRU policy which would result in just 1 hit. The pseudo-code for this modified OPTgen algorithm is shown in Figure 4.

3.2. Hawkeye Predictor

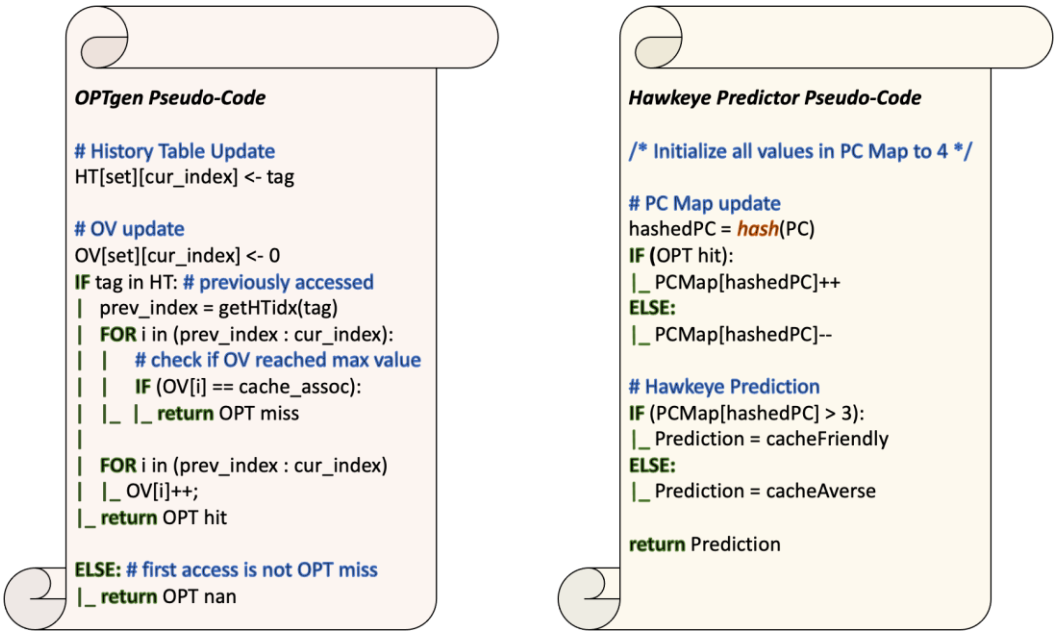
Given the OPT hit/miss prediction from the OPTgen algorithm, The Hawkeye Predictor classifies a cache block to be cache friendly or cache averse. The hawkeye predictor maintains a PC map of 8K entries indexed by 13-bit hashed PC. Each entry of the PC map has a 3-bit counter which is incremented during an OPT hit and decremented during an OPT miss. The most-significant bit of the counter determines if the cache block accessed (hit) or inserted (miss) is cache friendly or cache averse. A pseudo-code of the hawkeye predictor logic is given in Figure 4.

Since the hash function used for hashing the 32-bit PC is not given in the paper, we experimented with two different hash functions – CRC and mod. We found that mod performs slightly better than CRC for several benchmarks. So, we use mod hash function in our experiments. As we will explain later, the hash function could significantly impact the predictor accuracy since it determines which instructions are tracked in the PC map.

3.3 Cache Replacement

Hawkeye Prediction \ Hit or Miss	Cache Hit	Cache Miss
Cache-averse	RRIP = 7	RRIP = 7
Cache-friendly	RRIP = 0	RRIP = 0; Age all lines: if (RRIP < 6) RRIP++;

Figure 5: RRIP update of cache blocks during cache-hit and cache-miss



OPTgen Pseudo-Code

```
# History Table Update
HT[set][cur_index] <- tag

# OV update
OV[set][cur_index] <- 0
IF tag in HT: # previously accessed
|   prev_index = getHTidx(tag)
|   FOR i in (prev_index : cur_index):
|   |     # check if OV reached max value
|   |     IF (OV[i] == cache_assoc):
|   |_  |_ return OPT miss
|
|   FOR i in (prev_index : cur_index)
|   |_ OV[i]++;
|_ return OPT hit

ELSE: # first access is not OPT miss
|_ return OPT nan
```

Hawkeye Predictor Pseudo-Code

```
/* Initialize all values in PC Map to 4 */

# PC Map update
hashedPC = hash(PC)
IF (OPT hit):
|_ PCMap[hashedPC]++
ELSE:
|_ PCMap[hashedPC]--

# Hawkeye Prediction
IF (PCMap[hashedPC] > 3):
|_ Prediction = cacheFriendly
ELSE:
|_ Prediction = cacheAverse

return Prediction
```

Figure 4: Pseudo-code for OPTgen algorithm and Hawkeye Predictor

The victim block during cache replacement is determined based on the cache-friendly or cache-averse classification by the Hawkeye Predictor. We associate an RRIP value to each cache block and update it during each cache access. During a cache hit, the RRIP value of the accessed block is set as 0 when it is classified as cache-friendly by the Hawkeye Predictor. If it was classified as cache-averse, we set its RRIP value to 7. Similarly, during a cache miss, we update the RRIP value of the new block inserted in place of the victim. During eviction, the cache block with the highest RRIP value is chosen as the victim. If the victim was previously classified as cache-friendly by the hawkeye predictor, we decrement the corresponding entry in the PC map. Furthermore, the cache-friendly entries other than the victim are 'aged' during a cache miss i.e, we increment their respective RRIP values upto a value of 6. Note that only cache-averse blocks can have an RRIP value of 7. This way, the cache-averse blocks are always prioritized over the cache-friendly blocks for eviction. The cache replacement methodology described thus far is summarized in Figure 5.

The Hawkeye Predictor assumes that the past is a good predictor of the future. Therefore, an instruction that resulted in cache-hits in the past is likely to produce more hits in the future. Also, using the OPTgen algorithm, it learns to prioritize the instructions that would result in a hit under OPT policy. The cache blocks accessed by the instructions that are likely to hit under the OPT policy are classified as cache-friendly and set an RRIP value of 0. When an instruction is likely to miss under the OPT policy, the hawkeye predictor classifies the block accessed by this instruction as cache-averse and sets the associated RRIP value to 7. Note that the RRIP values are set based on its cache-friendliness, regardless of whether it actually resulted in a hit or miss. In this way, the predictor learns to follow the Belady's OPT policy.

During the start of a program, the predictions are likely to be wrong since the predictor has not seen enough accesses to mimic the Belady's policy. Therefore, we set the initial value of the Hawkeye Predictor counters to 4 so that all the cache blocks that are accessed initially are set an RRIP value of 0 and the victim is selected based on the ageing of RRIP. That is, the victim selection relies on the LRU policy where the recent usage is tracked using RRIP values of the cache blocks. The cache-averse predictions start to occur when enough OPT misses are seen. In this phase, the hawkeye predictions are likely to be correct. Following this, the phase of incorrect predictions is likely to re-occur since we are only tracking 8K instructions in the PC map. The many-to-one mapping in this table would extend the prediction of a known instruction to an unknown instruction that maps to the same location. This phase

sustains incorrect predictions until the predictor trains itself to adopt to the phase change. Because of this reason, the hash function that does the many-to-one mapping is likely to affect the performance of the simulator significantly. Also, because of the incorrect phases of the predictor the actual misses seen in the LLC are likely to be greater than the OPT misses.

4. SDBP Replacement Policy

A cache block is said to be live in the cache from the time it is inserted until the time of its last reference. From the time of its last reference to the time it is evicted the block is said to be dead. Cache blocks are dead on average 86.2% of the time [2] so the miss rate can be reduced by reducing the number of dead blocks in the cache. Cache efficiency can be improved by replacing dead blocks with live blocks as soon as possible after a block's last reference instead of waiting for that block to be evicted.

Dead block prediction is a technique which predicts whether to evict a block or not at each access to that block. However, current dead block prediction methods have many problems. Firstly, they require a significant overhead in terms of prediction structures as well as extra cache metadata. Thus, the improvement in cache efficiency is compensated with an increase in power and area.

Secondly, due to the large number of instruction references tracked by these predictors, they must be very large or experience destructive interference in their prediction tables which reduces accuracy of their prediction. Instruction trace based predictors do not work in memory systems involving L1, L2, and L3 caches because most of the temporal locality is filtered out by a moderately-sized mid-level cache.

4.1 Sampling Dead Block Predictor

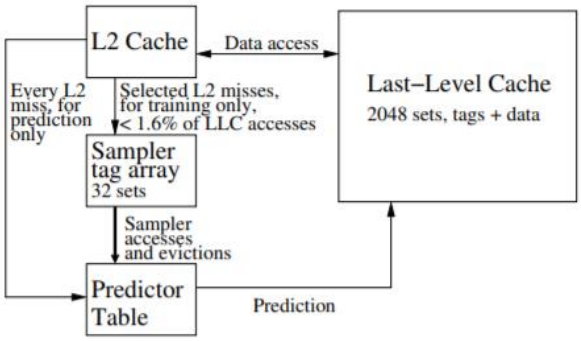


Figure 6: SDBP sampler and Predictor

The SDBP consists of a small partial tag array(sampler). Each sampler set corresponds to a selected cache set; e.g. If a sampler keeps 32 sets for a 2048 set cache then corresponding to every 64th cache set there is a sampler set. Only the lower-order 15 bits of tags are stored to reduce area and energy. The predictor is updated on selected L2 misses which are sampled by the sampler while each access to the LLC incurs an access to the predictor.

4.2 Predictor

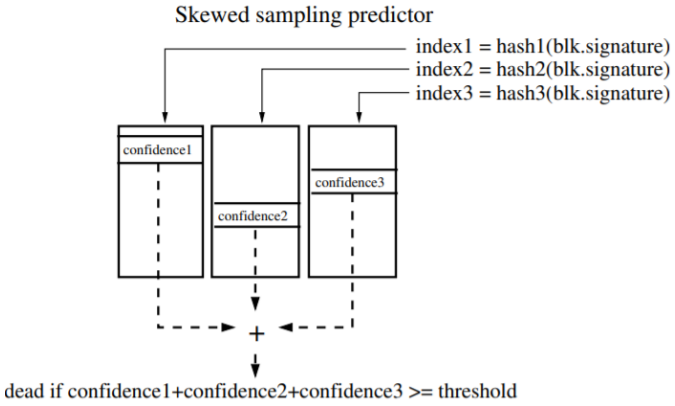


Figure 7: Skewed Sampling Predictor

The sampling predictor uses skewed organization [10] to reduce conflicts in the table. The predictor keeps three 4,096-entry tables of 2- bit counters, each indexed by a different hash of a 15-bit signature. Each access to the predictor yields three counter values. The sum of these values a confidence compared with a threshold. If the confidence is greater than the threshold then the corresponding block is predicted dead. If there had been only one table then unrelated signatures would have conflicted. Since we have three tables, the case of a conflict is very less likely, so the effect of destructive conflicts is reduced.

SDBP Sampler and Predictor Pseudo-Code

```
IF set in Sampler:
|  updatePC
|  Train Predictor (PC)
|_ Test Predictor (PC)
ELSE:
|_ Test Predictor (PC)

Test Predictor (PC):
   Confidence = Table1[hash1(PC)] + /
                Table2[hash2(PC)] + Table3[hash3(PC)]
   IF (Confidence>Threshold)
   |_ evict block

Train Predictor(PC):
   FOR i in (1:4096):
   |  IF (i == hash1) :
   |  |  IF (0<Table1[i]<3):
   |  |_ |_ Table1[i] -1
   |  ELSE:
   |  |  IF (0<Table1[i]<3)
   |_|_|_ Table1[i] +1
    repeat for Table 2 and Table 3
```

Figure 8: Pseudo-code for SDBP sampler and predictor

5. Evaluation Methodology

We use a memory hierarchy with the following configuration from [1] in the Gem5 simulator. The LLC is assumed to be a non-bypassing cache. The Hawkeye replacement policy is applied only to LLC. L1 and L2 caches use LRU policy. We performed our experiments using single-core out-of-order CPU and 8 SPEC 2006 benchmarks such as sphinx3, soplex, mcf, hmmer, tonto, bzip2, astar and gromacs.

L1 I-Cache	32 KB 4-way, 1-cycle latency
L1 D-Cache	32 KB 4-way, 1-cycle latency
L2 Cache	256KB 8-way, 10-cycle latency
Last-level Cache	2MB, 16-way, 20-cycle latency
DRAM	200 cycles

Assumptions:

- Since [1] does not consider pre-fetching in its implementation, we do not use pre-fetching for our LLC
- No cache bypassing: Although [1] assumes LLC cache in bypass mode, we assume no bypass since gem5 does not support bypassing caches for the O3 CPU.

6. Results

In this section, we present the MPKI reduction and performance improvement of Hawkeye policy over LRU. We also compare the Hawkeye policy to BRRIP. Finally, we present the breakdown of cache-friendly and cache-averse victim blocks for further insights into the working of the Hawkeye predictor.

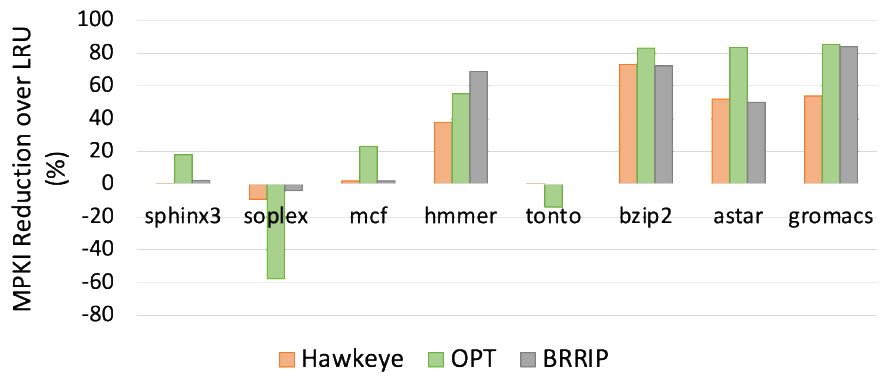


Figure 9: Improvement in overall miss reduction

The overall LLC miss reduction achieved by Hawkeye compared to LRU is shown in Figure 9. On an average, our implementation of Hawkeye achieves 26.36% reduction across the benchmarks considered. The reduction in the number of misses compared to LRU is because of relying on Belady's optimal policy rather than only the last access time of cache blocks. The optimal misses (OPT misses) computed by the OPTgen algorithm is shown in Figure 9 for comparison. We find that hawkeye predictor approaches the OPT miss reduction in most of the benchmarks. In soplex and tonto, we find that both the OPT and Hawkeye policies perform worse than LRU. This is in contradiction to the results in [1] where significant improvement was reported across all the benchmarks. We attribute this

discrepancy to the fact that we have assumed a non-bypassing cache whereas [1] assumes a bypassing cache. Also, we have assumed the mod function as the hash function to map the 32-bit PCs to 13-bit indices in the PC map of the Hawkeye predictor. (We assumed mod function since [1] has not mentioned what hashing function was used in their experiments). As discussed in section 3.3, a better choice of hash function may significantly improve the performance of the Hawkeye policy. Additionally, we compare the Hawkeye policy to BRRIP policy and find that the MPKI reduction of BRRIP is better than Hawkeye in several benchmarks.

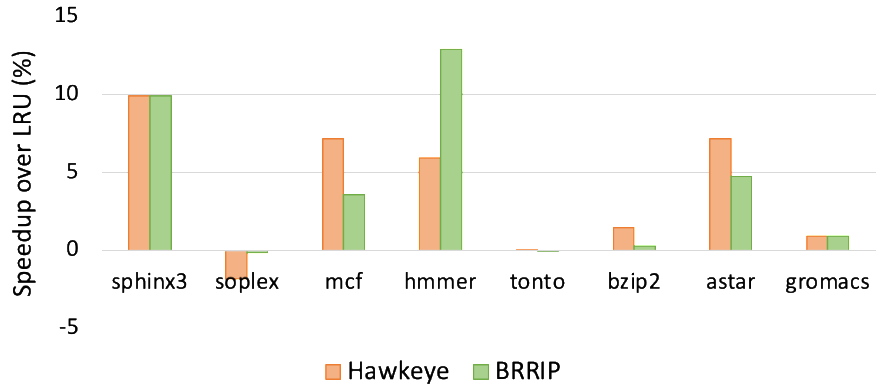


Figure 10: Performance Improvement

The performance improvement of Hawkeye policy over LRU is shown in Figure 10 along with a comparison to BRRIP. We achieve upto 9.9% improvement over LRU and upto 3.45% improvement over BRRIP. We attribute the performance improvement to the MPKI reduction achieved by the Hawkeye policy. Furthermore, the performance improvement achieved by Hawkeye in our experiments are lesser than that reported in [1] for the same reasons discussed above.

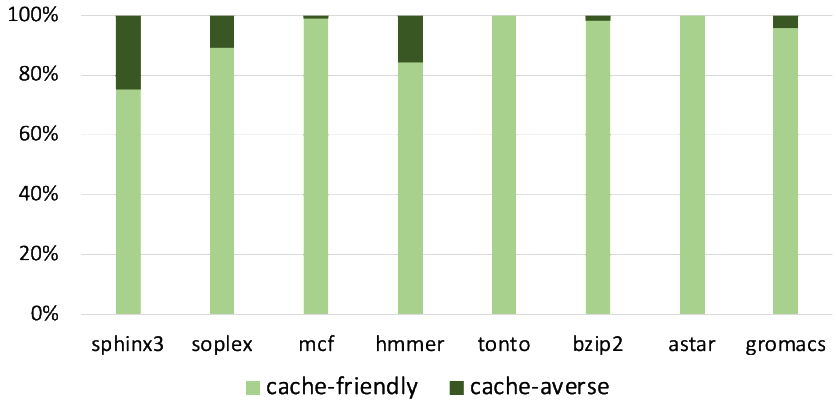


Figure 11: Breakdown of Cache-Friendly and Cache-Averse victim blocks

We now present the breakdown of the cache-friendly and cache-averse victim blocks in Figure 11. We can see that in our experiments the number of cache-friendly victims are significantly greater than the number of cache-averse victims. This trend results in sub-optimal performance improvement compared to [1], since most of the victim blocks must be cache averse to achieve Belady's performance. This behavior can be explained by noting that the hawkeye predictor goes through phases of correct and wrong predictions as explained in section 3.3. The wrong-prediction phase happens because of the many-to-one mapping of PC in the PC map. During this phase, the

hawkeye predictor relies on the LRU policy by predicting most cache blocks as cache-friendly. In Figure 12, we can see that only few blocks were predicted as cache-averse across benchmarks. Therefore, only a few victims are evicted based on the OPT policy and others are evicted based on LRU (through the ageing of RRIP). By using a better hash function than mod, and by using bypassing caches we can overcome this limitation to achieve evictions dominated by cache-averse victims.

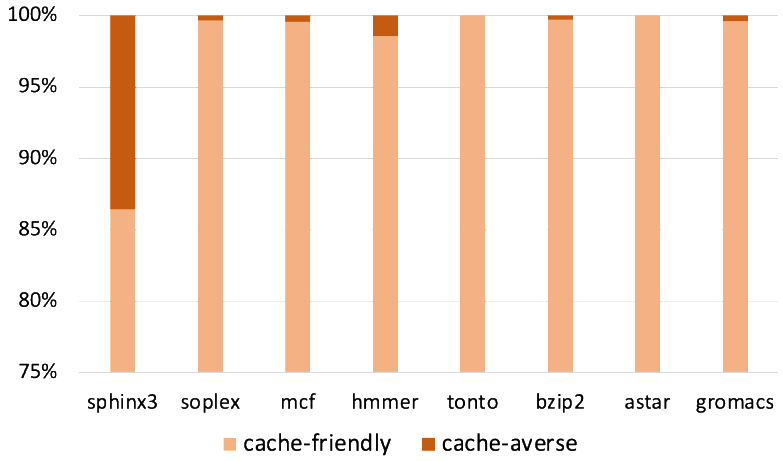


Figure 12: Breakdown of Cache-Friendly and Cache-Averse cache blocks

7. Conclusion

In this project we implemented the Hawkeye Replacement Policy [1] and the SDBP policy [2] to reduce the LLC misses. We find that the Hawkeye policy reduces the LLC misses by 26.36% compared to the LRU policy across 8 SPEC 2006 benchmarks. Futher, across the same benchmarks we achieve upto 9.9% improvement in performance over LRU using the Hawkeye policy.

8. References

[1] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 78-89, doi: 10.1109/ISCA.2016.17.

[2] S. M. Khan, Y. Tian and D. A. Jiménez, "Sampling Dead Block Prediction for Last-Level Caches," 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, 2010.

[3] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon Stelly Jr., and Joel Emer, “Adaptive insertion policies for managing shared caches,” In Proceedings of the 2008 International Conference on Parallel Architectures and Compiler Techniques (PACT), September 2008.

[4] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In International Symposium on Computer Architecture (ISCA), pages 60–71. ACM, 2010.

[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In International Symposium on Computer Architecture (ISCA), pages 381–391. ACM, 2007.

[6] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. SHiP: Signature-based hit predictor for high performance caching. In 44th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 430–441, 2011.

[7] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. SIGARCH Comput. Archit. News, 29(2):144–154, 2001.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In International Symposium on Computer Architecture (ISCA), pages 364–373, 1990.

[9] K. Rajan and R. Govindarajan. Emulating optimal replacement with a shepherd cache. In the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 445–454, 2007.