Cache Replacement Improvement on Last-Level Caches Using Belady's Algorithm

Abdullah Abdul Kader
Computer Science and Engineering Department
Texas A&M University
College Station, TX, United States
akader16@tamu.edu

Yerania Hernandez
Computer Science and Engineering Department
Texas A&M University
College Station, TX, United States
hernandez.yerania@tamu.edu

***Abstract*—Our paper implements the Hawkeye cache replacement policy, which learns from Belady's algorithm by using past cache access history to efficiently predict future cache replacement decisions. We demonstrate that the implementation can accurately make predictions when simulating Belady's behaviour to improve cache replacements. We evaluated Hawkeye on a single core with 2MB LLC with and without prefetchers using the memory-intensive SPEC2006 benchmark suite through the ChampSim simulator and comparing the results to the current existing cache replacement policies of LRU and SRRIP. Our solution is able to demonstrate an average speed up of 58.52% compared to LRU's 56.74% and SRRIP's 56.85% on the single-core configuration without prefetchers. In the presence of prefetching, Hawkeye's speedup performance is 62.89% compared to LRU's 61.55% and SRRIP's 61.66%.**

***Index Terms*—Cache replacement, Belady's algorithm**

I. Introduction

Caches are important for a number of reasons. The use of caches reduce long latencies in memory accesses, which calls for an effective cache replacement policy. Cache replacement is a difficult problem compared to a problem like branch prediction. The answer to the question "Which cache line to evict?" is unfortunately not as simple as "Will this branch be taken?". Existing replacement algorithms such as Least Recently Used (LRU) and Most Recently Used (MRU) run based on assumptions and work for only certain type of loads. These algorithms fall short in complex scenarios. Back to the future [3] presents the example of a triply nested algorithm used in matrix multiplication to study the performance of different replacement policies as shown in Figure 1 to demonstrate the reuse patterns of A, B and C. In this paper, we implement the algorithm for cache replacement based on Belady's algorithm [3] and include the improvements provided in [4] for prefetching. Even though Belady's algorithm is optimal, it's impractical since it requires the knowledge of the future. Nevertheless, it is possible to apply a variation of the Belady's algorithm by learning the past behavior and using it to predict the future. The decisions made using the Belady's algorithm is referred to as OPT in this paper.

The proposed cache replacement strategy contains two components. The first reconstructs the Belady's solution for past cache accesses and the second is the predictor, which learns OPT's behavior to inform eviction decisions for future loads. The challenge with reconstructing Belady's is that it looks arbitrarily in the future and requires remembering a long history. However, Belady's algorithm has a better performance even with a reuse window of eight times the cache size which leads to the implementation of the Hawkeye replacement policy [3]. Hawkeye is evaluated using the ChampSim simulator that was released by the Second Cache Replacement Championship (CRC) [1]. Additionally, we compare various replacement policies with Hawkeye using a single core configuration with and without a prefetcher. To account for the prefectch requests, Hawkeye ignores redundant prefetches by considering only the prefetches that follow a demand request. Additionally, separate predictors are used to distinguish between the caching behavior associated with demand and the prefetches by the same load instruction.

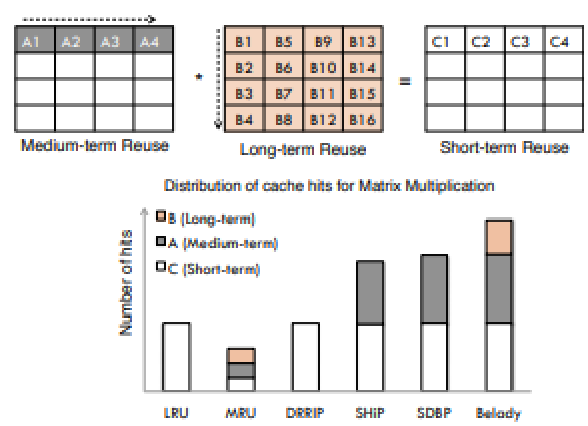


Fig. 1. Existing replacement policies are limited to a few access patterns and are unable to cache the optimal combination of A,B and C

II. Related Work

Some closely related work in the area of cache replacement that is similar to Hawkeye is addressed in this section.

A. Short term history information

Several cache replacement policies rely on short term information to determine which cache line to evict while ignoring the history. One popular example is LRU, which favors the

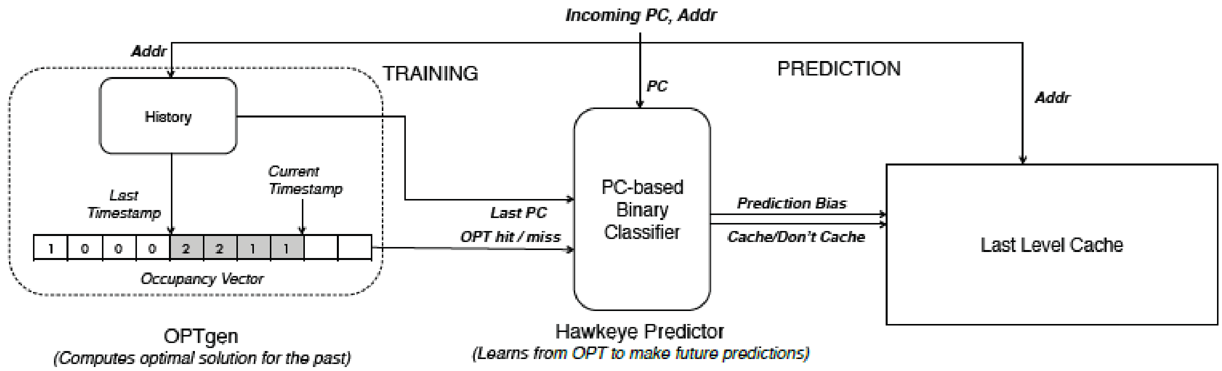


Fig. 2. Overall structure of Hawkeye replacement algorithm

recently used line assuming it will be used again soon in the future. Some other policies rely on access frequencies to decide which line to evict with the assumption that a high frequency access line will soon be accessed again. With the use of re-reference interval prediction, Jaleel et al. [5] enhances recently friendly policies by using 2-bits for prediction. Hawkeye uses RRIPs idea of aging to adapt to changes in phase behavior. Other policies include hybrid replacements which allow the flexibility of choosing a policy for different cache lines and Hawkeye exploits this approach by using different policies for different load instruction.

B. *Long term history information*

Contrast to the short term information approach, some replacement policies exploit long term information to inform their eviction decisions. For instance, some policies use an expensive approach of reuse distances to predict their decisions. On the contrary, Hawkeye heavily relies on liveliness interval which is considerably inexpensive. SHiP [2] uses a predictor to identify instructions that load stream accesses. Hawkeye builds on the prior work done by SHiP to inform future decisions based on past load instructions. Hawkeye uses a longer history (8x the size of cache) compared to SHiP to inform evict decisions.

III. Design Overview

The purpose of Hawkeye is to be able to predict if incoming lines are cache-friendly or cache-adverse. If a line is predicted to be cache-friendly, it will be inserted with high priority in the cache, while if a line is predicted to be cache-averse, it will be inserted with low priority in the cache. In order to provide this prediction, Hawkeye attempts to create a Belady's optimal solution based on the past accesses and applies these solutions to predicting the caching behavior of future loaded instructions.

Figure 2 summarizes the structure of the Hawkeye replacement algorithm. The two main components are the OPTgen algorithm, which is used to compute the optimal solution from past access, and the Hawkeye Predictor, which uses OPTgen's solutions as input in order to train a PC-based predictor to provide an eviction decision. We will describe each component of the Hawkeye in further detail.

A. *OPTgen*

OPTgen is the algorithm used to compute the OPT of past history caches. OPT is the decision made by Belady's algorithm by relying on history to predict if a load instruction, for example, will bring a line in the future based on it's past behavior. This helps inform eviction decisions for future load instructions. The terms *usage interval* and *livelines interval* are used to further define the functions of OPTgen. *Usage interval* is the time period that starts with a reference to X and leads up to but not including it's next reference X'. Alternatively, *usage interval* can also be defined as the demand of X in a cache. *Livelines interval* is the time the line resides in the cache under the OPT policy. Figure 3 demonstrates the usage interval of X which proceeds up to it's next reference. The liveliness intervals of A, B and C start at their respective first reference and end immediately at their second reference. These liveliness intervals do not overlap, therefore the cache capacity is not reached leading to a cache hit for X'. The question of whether a line would be a hit or miss is based on both the reuse distances and their overlap.

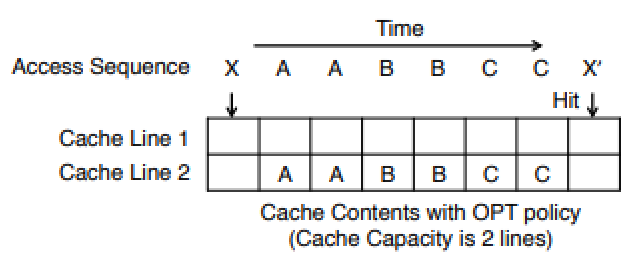
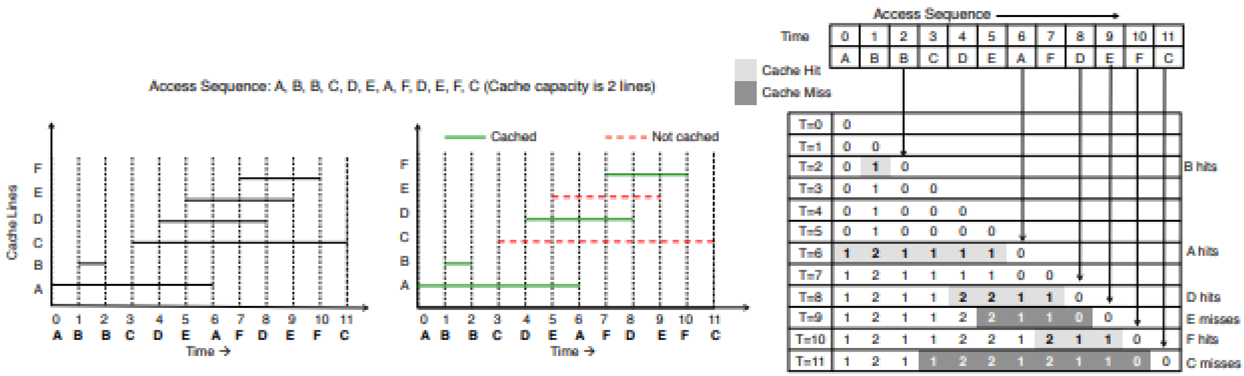


Fig. 3. Intuition behind OPTgen.

An *occupancy vector* is used to keep a track of the occupied cache capacity over time. The number of liveliness intervals that overlap at any time is recorded in each entry of the vector. Figure 4 shows the use of an occupancy vector through the view of access stream in Figure 4(a), the optimal solution to



(a) Timeline view of the Access Stream

(b) Optimal Solution (4 hits) [Cache hits marked as solid lines]

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

Fig. 4. Example to illustrate OPTgen

the access stream is presented in Figure 4(b) and the Figure 4(c) shows how the occupancy vector gets calculated and modified over time. For each time starting at 0 (most recent entry) when line X is not loaded for the first time, the OPT checks to see if every element corresponding to the usage interval is less than the cache capacity. If yes, then X is placed in the cache.The occupancy vector is incremented and it is shaded. Consider the time T=8, the usage interval starts at 4 and ends at 7 shown by the shaded portion. At time T=7 the cache capacity at all entries starting from 4 to 7 are less than cache capacity (2). This results in a hit for D as shown in T=8 while E is a miss in time T=9 because the an element in the usage interval for E has a value of 2, already at the cache capacity. See Appendix A for the code implementation of OPTgen.

B. Hawkeye Predictor

The second major component of this design is the Hawkeye Predictor, which classifies loaded instructions as either cache-friendly or cache-averse. It learns from the OPT policy whether a loaded instruction would have caused a hit or miss. If the OPTgen determines that there was a hit for a specific line, then the PC of that recently accessed line is trained positively under the predictor. Otherwise, if there was miss instead, then the PC of that recently accessed line is trained negatively. Overall, the Hawkeye Predictor will consist of 2K entries, 5-bit counters to train, and indexed by an 11-bit hashed PC. See Appendix C for the code implementation of the Hawkeye Predictor and Appendix D for the code implementation of the hash algorithm.

C. Cache Replacement

To efficiently implement Hawkeye, each of the cache lines are associated to a 3-bit RRIP counter. This counter maintains the eviction priorities, where a high RRIP value (RRPV) represents a high eviction priority and a low RRPV represents a low eviction priority. As a result, the RRIP counter is

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

Fig. 5. Hawkeye's policy based on predictor.

maintaining information on both the Hawkeye's prediction and the age for each line. The Hawkeye predictor will generate a prediction on every cache access. If the line is predicted to be a cache-friendly line, then the RRIP counter will update with an RRPV of 0. However, for a newly inserted cache-friendly line, the RRIP counters for all the other cache-friendly lines are also updated by increasing their value by one (aging). If the line is predicted to be a cache-averse line, then the RRIP counter will update with an RRPV of 7. Figure 5 summarizes how the RRIP counter is updated with Hawkeye's predictions.

With this setup, cache replacement is decided based on the RRPV. Any line with the RRPV of 7 is selected as an eviction candidate. However, if there is no line that contains an RRPV of 7, then the Hawkeye selects the line that contains the next highest RRPV as the eviction candidate. The implementation of this cache replacement aspect is still in progress considering it is integrating the OPTgen and Hawkeye Predictor. However, the initial setup of the replacement state has been created and we only have the update replacement state left to develop.

D. Prefetch Awareness

In order to accommodate prefetch requests, the OPT gen and the Hawkeye predictor need to be slightly modified. In the case of no prefetching as shown in Figure 3, the start and end of liveliness intervals are both demand accesses.

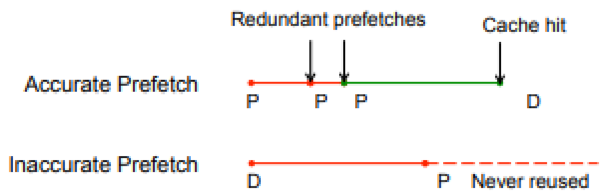


Fig. 6. Liveness intervals that end with a prefetch are not cached by prefetch-aware OPTgen.

However, when prefetching is introduced, it essentially adds up 4 different intervals: Prefetch to Prefetch P-P, Prefetch to Demand P-D, Demand to Prefetch D-P and Demand to Demand D-D. Treating all these intervals similarly would result in poor performance since they contribute to redundant caches as shown in Figure 6. Therefore, we modify the liveliness interval to consider only the intervals that end with a demand access P-D or D-D. As seen in Figure 6, P-P and D-P result in inaccurate fetches when they are not followed by a demand access. The implementation of prefetch awareness in Hawkeye is still in progress considering it will take place in the integration of OPTgen and Hawkeye Predictor, as described in the previous subsection. Once we have developed the cache replacement aspect of Hawkeye, we will easily be able to accommodate the modifications for prefetch requests.

IV. Evaluation Methodology

We will be evaluating our algorithm using the ChampSim simulator that was released by the Second Cache Replacement Championship (CRC) [1]. This simulator is designed to evaluate replacement algorithms for private and shared last level caches (LLCs) with and without a data prefetcher. We will evaluate Hawkeye using the two different configurations that the CRC provides:

- Single core with 2 MB LLC without a prefetcher.
- Single core with 2 MB LLC with L1/L2 data prefetchers.

In order to compare the results of the Hawkeye algorithm with that of the original paper, we will also use the SPEC2006 benchmark suite with the traces that demonstrate more than a 2% improvement with the OPT policy. In order to generate the samples from each benchmark, we used Pin 3.2 with the CRC, creating a total sample of 250 million instructions. This sample is divided between the warm-up stage and behavior stage, where 50 million instructions are used to warm the cache and the other 200 million instructions are used to measure the behavior.

To evaluate the performance, we will be reporting the Miss Per Kilo instruction (MPKI) and the speedup for each benchmark combination. In addition, we will compare our implementation of Hawkeye to other caching systems. We will compare against LRU (Least Recently Used) and SRRIP (Static Re-Reference Interval Prediction). As noted earlier, LRU is a commonly used replacement policy always predicting a "near-immediate re-reference interval on cache hits and misses" [5]. Further improvements have been made on the policy by using SRRIP for a scan-resistant approach. For each of these implementations, we used the code provided on the CRC website and further tuned the parameters to provide an adequate comparison to the Hawkeye.

V. Results and Analysis

Figures 7 and 8 show the simulation results obtained for the no-prefetching configuration on the simulator. Figures 9 and 10 show the simulation results for the prefetching configuration.

A. Single-core, No Prefetching Configuration

Figure 7 shows that Hawkeye reduces the LLC MPKI in comparison with the two other replacement policies by only a slight margin. In particular, Hawkeye achieves an average of 14.02 MPKI throughout the 18 memory-intensive SPEC benchmarks that reported improvement for the OPTgen. In comparison, LRU and SRRIP obtained an average of 14.32 and 14.05 MPKI, respectively. Figure 8 shows that Hawkeye's slight marginal reduction in misses translates to an average speedup of 58.52% in comparison to LRU's average speedup of 56.74%. By contrast, SRRIP only is able to obtain an improvement performance with a 56.85% average speedup. This is significant considering Hawkeye is able to outperform both LRU and SRRIP on a single-core system without prefetching. Figure 8 also demonstrates an important trend. Despite, LRU and SRRIP performing well throughout a few workloads, their performance gains are not consistent throughout the benchmarks. As a matter of fact, Hawkeye performs consistently well across all benchmarks, except calculix where Hakweye loses in comparison to LRU and SRRIP. It can even be observed that Hawkeye does not perform worse than the baseline of LRU except for that same calculix benchmark. These results reinforces the notion that current replacement policies are geared to specific types of access patterns, whereas the Hawkeye algorithm is able to adapt to any type of workload as it learns from the OPT.

B. Single-core, With Prefetching Configuration

When analyzing the prefetching configuration, Figure 9 demonstrates that Hawkeye achieves an average of 17.98 MPKI. Despite the Hawkeye being able to show a slight improvement in comparison to LRU that obtains an average of 17.99 MPKI, it does not outperform SRRIP, which is able to obtain an average of 17.76 MPKI. Figure 10 provides insight on the performance speedup of Hawkeye, which surprisingly outperforms both LRU and SRRIP. Hawkeye is able to obtain an average of 62.89% speedup, compared to the overall average of 61.55% for LRU and 61.66% for SRRIP. Ideally, an efficient LLC policy aims to reduce the MPKI in order to decrease having to go beyond the cache and therefore, shortening the time necessary to execute the instructions. This will overall reducce the time required for computations and would increase the IPC if accomplished. However, our observations demonstrate that Hawkeye has the ability to improve the performance when compared to the other two replacement

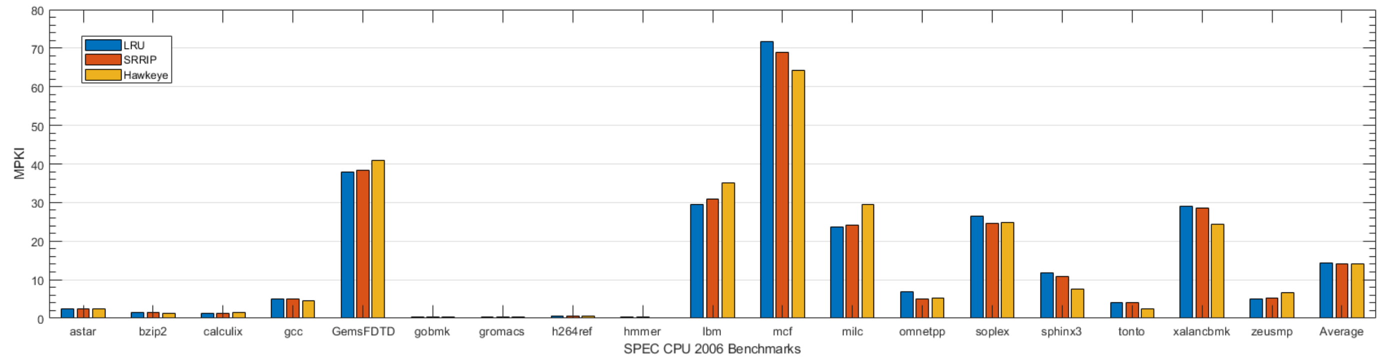


Fig. 7. Hawkeye's policy based on predictor.

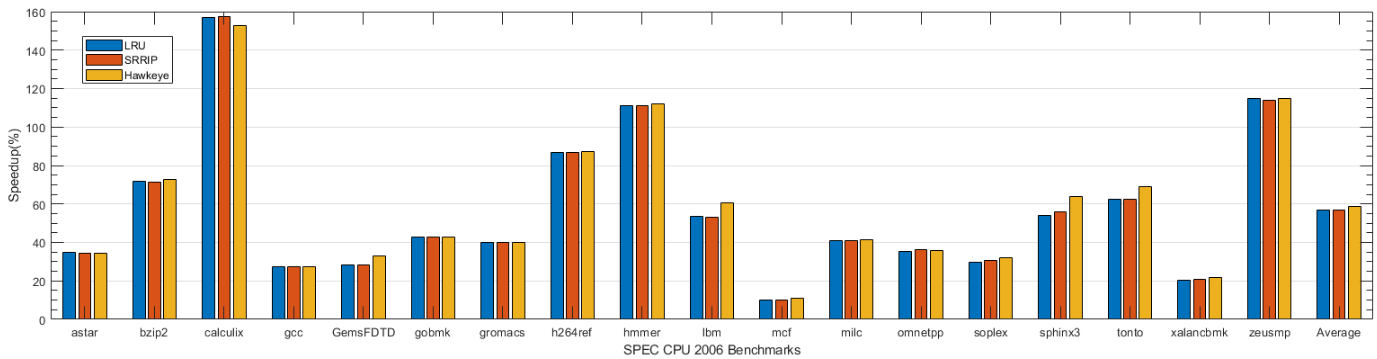


Fig. 8. Hawkeye's policy based on predictor.

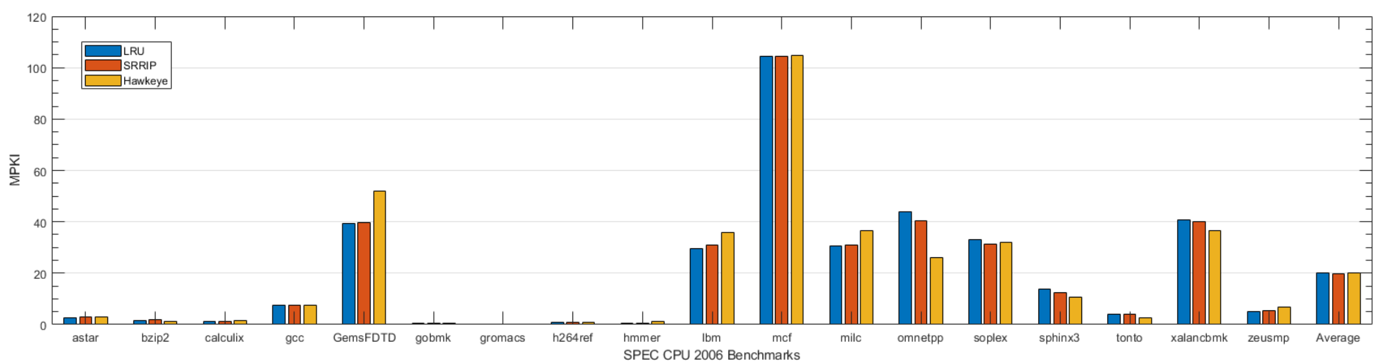


Fig. 9. Hawkeye's policy based on predictor.

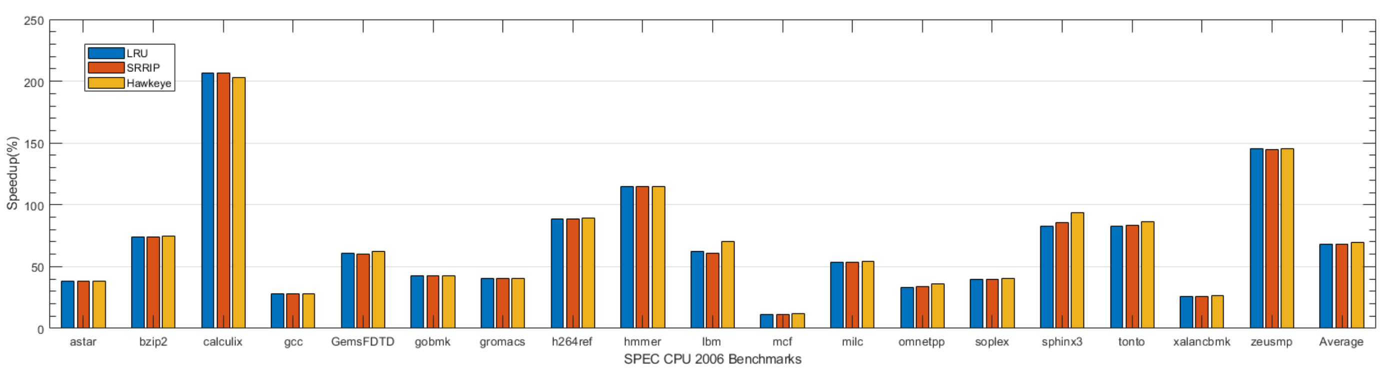


Fig. 10. Hawkeye's policy based on predictor.

policies based on the speedup achieved, but in comparison to SRRIP's MPKI, it is causing more misses. It is important to note that Hawkeye is able to learn different priorities on a load and therefore it is common for cache-friendly lines and cache-averse lines to occur simultaneously. This is significant because it means that our algorithm's implementation needs to improve how it predicts a cache-averse and cache-friendly line. If further improvements were made in how the algorithm computes its prediction and learns from the OPTgen, then we could possibly reduce the number of misses and obtain an overall better performance. When evaluating benchmarks individually, Figure 10 demonstrates that Hawkeye obtains a better performance in comparison to the other two replacement policies in twelve different workloads and it does not perform worse than the baseline of LRU across all benchmarks except in calculix. Hawkeye's loss in this benchmark for both configurations can be possibly explained due to the window size. The calculix benchmark seems to benefit from long reuse intervals and therefore, would require a larger interval than the eight size window we accommodated for in our simulator.

VI. Conclusion

In this paper, we explained the original Hawkeye cache replacement policy that was developed for the CRC in 2016 [3] and implemented this algorithm with the improvements presented in the 2017 paper [4]. This algorithm consisted of three major advantages compared to current work in cache replacement. From one aspect, the Hawkeye is able to demonstrate that by looking at a long history of memory accesses, it can learn and predict the optimal behavior for the instructions loaded. Additionally, unlike other policies, the decisions provided by the Belady's algorithm reuses all types of workloads, not solely based on recently used or mostly used. The last advantage of using Belady's algorithm is that it is able to consider both reuse distance and cache demand.

The simulator results have shown that Hawkeye is able to demonstrate an overall improvement in comparison to LRU and SRRIP for single-core simulations on no-prefetching and prefetching. Despite not outperforming SRRIP's average MPKI in the prefetching configuration, Hawkeye's modifications in accounting for prefetching by distinguishing among accurate, redundant, and inaccurate prefetches did significantly improve the speedup. However, even with these changes, Hawkeye's efficiency still needs improvement in line prediction to reduce the number of misses. It leads to an understanding that Hawkeye has not yet answered the question of finding an optimal cache solution when prefetching occurs. Once there is an ideal solution, then a Hawkeye version of the solution could comparably perform better.

For future work, we expect to see improvements in cache predictors due to the trend of sophisticated branch predictors. There is a strong potential of improving the Hawkeye predictor while using the OPTgen in order to model the optimal OPT's behavior. It would also be interesting to consider multi-core configurations considering the current system improvements and usage of multi-cores. If Hawkeye is able to outperform current existing cache replacement policies in this setting, then it would have demonstrated that it is indeed possible to obtain optimal behavior based on the past actions of cache. Additionally, considering Hawkeye is able to provide long-periods of history, the information provided could be useful in other fields other than caching, such as optimizing shared memory systems. Finally, further evaluation on the relationship between replacement policies and the presence of prefetchers will demonstrate that Hawkeye could provide an optimal solution.

References

[1] "Champsim Source." THE 2ND CACHE REPLACEMENT CHAMPIONSHIP, 2017, crc2.ece.tamu.edu/?page_id=41.

[2] 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 430441, 2011.

[3] J. Akanksha, 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), 2016, doi:10.1109/isca.2016.17.

[4] J. Akanksha, and C. Lin, "Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement," CS-University of Texas, 2017, www.cs.utexas.edu/ lin/papers/crc17.pdf.

[5] J. Aamer, et al, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," Proceedings of the 37th Annual International Symposium on Computer Architecture - ISCA 10, 2010, doi:10.1145/1815961.1815971.

Appendix A
OPTgen

```
struct OPTgen{
    vector<unsigned int> liveness_intervals;
    uint64_t num_cache;
    uint64_t access;
    uint64_t cache_size;

    //Initialize values
    void init(uint64_t size){
        num_cache = 0;
        access = 0;
        cache_size = size;
        liveness_intervals.resize(OPTGEN_SIZE, 0);
    }

    //Return number of hits
    uint64_t get_optgen_hits(){
        return num_cache;
    }

    void set_access(uint64_t val){
        access++;
        liveness_intervals[val] = 0;
    }

    void set_prefetch(uint64_t val){
        liveness_intervals[val] = 0;
    }

    //Return if hit or miss
    bool is_cache(uint64_t val, uint64_t endVal){
        bool cache = true;
        unsigned int count = endVal;
        while (count != val){
            if(liveness_intervals[count] >= cache_size){
                cache = false;
                break;
            }
            count = (count+1) % liveness_intervals.size();
        }

        if(cache){
            count = endVal;
            while(count != val){
                liveness_intervals[count]++;
                count = (count+1) % liveness_intervals.size();
            }
            num_cache++;
        }
        return cache;
    }
};
```

Appendix B
Hawkeye Predictor

```
class Hawkeye_Predictor{
private:
  map<uint64_t, int> PC_Map;

public:
  //Return prediction for PC Address
  bool get_prediction(uint64_t PC){
    uint64_t result = CRC(PC) % PCMAP_SIZE;
    if(PC_Map.find(result) != PC_Map.end() && PC_Map[result] < ((MAX_PCMAP+1)/2)){
      return false;
    }
    return true;
  }

  void increase(uint64_t PC){
    uint64_t result = CRC(PC) % PCMAP_SIZE;
    if(PC_Map.find(result) == PC_Map.end()){
      PC_Map[result] = (MAX_PCMAP + 1)/2;
    }

    if(PC_Map[result] < MAX_PCMAP){
      PC_Map[result] = PC_Map[result]+1;
    }
    else{
      PC_Map[result] = MAX_PCMAP;
    }
  }

  void decrease(uint64_t PC){
    uint64_t result = CRC(PC) % PCMAP_SIZE;
    if(PC_Map.find(result) == PC_Map.end()){
      PC_Map[result] = (MAX_PCMAP + 1)/2;
    }
    if(PC_Map[result] != 0){
      PC_Map[result] = PC_Map[result] - 1;
    }
  }

};
```

APPENDIX C
HAWKEYE IMPLEMENTATION

```
// Initialize replacement state
void InitReplacementState()
{
    cout << "Initialize Hawkeye replacement policy state" << endl;

    for (int i=0; i<LLC_SETS; i++) {
        for (int j=0; j<LLC_WAYS; j++) {
            rrip[i][j] = MAXRRIP;
            sample_signature[i][j] = 0;
            prefetching[i][j] = false;
        }
        set_timer[i] = 0;
        optgen_occup_vector[i].init(LLC_WAYS-2);
    }

    cache_history_sampler.resize(SAMPLER_SETS);
    for(int i = 0; i < SAMPLER_SETS; i++){
        cache_history_sampler[i].clear();
    }

    predictor_prefetch = new Hawkeye_Predictor();
    predictor_demand = new Hawkeye_Predictor();

    cout << "Finished initializing Hawkeye replacement policy state" << endl;
}

// Find replacement victim
// Return value should be 0 ~ 15 or 16 (bypass)
uint32_t GetVictimInSet (uint32_t cpu, uint32_t set, const BLOCK *current_set,
    uint64_t PC, uint64_t paddr, uint32_t type)
{
    //Find the line with RRPV of 7 in that set
    for(uint32_t i = 0; i < LLC_WAYS; i++){
        if(rrip[set][i] == MAXRRIP){
            return i;
        }
    }

    //If no RRPV of 7, then we find next highest RRPV value (oldest cache-friendly
    line)
    uint32_t max_rrpv = 0;
    int32_t victim = -1;
    for(uint32_t i = 0; i < LLC_WAYS; i++){
        if(rrip[set][i] >= max_rrpv){
            max_rrpv = rrip[set][i];
            victim = i;
        }
    }

    //Asserting that LRU victim is not -1
    //Predictor will be trained negaively on evictions
    if(SAMPLED_SET(set)){
        if(prefetching[set][victim]){
```

```
            predictor_prefetch->decrease(sample_signature[set][victim]);
        }
        else{
            predictor_demand->decrease(sample_signature[set][victim]);
        }
    }

    return victim;
}

//Helper function for "UpdateReplacementState" to update cache history
void update_cache_history(unsigned int sample_set, unsigned int currentVal){
    for(map<uint64_t, HISTORY>::iterator it = cache_history_sampler[sample_set].begin(); it != cache_history_sampler[sample_set].end(); it++){
        if((it->second).lru < currentVal){
            (it->second).lru++;
        }
    }

}

// Called on every cache hit and cache fill
void UpdateReplacementState (uint32_t cpu, uint32_t set, uint32_t way, uint64_t paddr, uint64_t PC, uint64_t victim_addr, uint32_t type, uint8_t hit)
{
    paddr = (paddr >> 6) << 6;

    //Ignore all types that are writebacks
    if(type == WRITEBACK){
        return;
    }

    if(type == PREFETCH){
        if(!hit){
            prefetching[set][way] = true;
        }
    }
    else{
        prefetching[set][way] = false;
    }

    //Only if we are using sampling sets for OPTgen
    if(SAMPLED_SET(set)){
        uint64_t currentVal = set_timer[set] % OPTGEN_SIZE;
        uint64_t sample_tag = CRC(paddr >> 12) % 256;
        uint32_t sample_set = (paddr >> 6) % SAMPLER_SETS;

        //If line has been used before, ignoring prefetching (demand access operation)
        if((type != PREFETCH) && (cache_history_sampler[sample_set].find(sample_tag) != cache_history_sampler[sample_set].end())){
            unsigned int current_time = set_timer[set];
            if(current_time < cache_history_sampler[sample_set][sample_tag].previousVal){
                current_time += TIMER_SIZE;
            }
```

```
            uint64_t previousVal = cache_history_sampler[sample_set][sample_tag].previousVal % OPTGEN_SIZE;
            bool isWrap = (current_time - cache_history_sampler[sample_set][sample_tag].previousVal) > OPTGEN_SIZE;

            //Train predictor positvely for last PC value that was prefetched
            if(!isWrap && optgen_occup_vector[set].is_cache(currentVal, previousVal)){
                if(cache_history_sampler[sample_set][sample_tag].prefetching){
                    predictor_prefetch->increase(cache_history_sampler[sample_set][sample_tag].PCval);
                }
                else{
                    predictor_demand->increase(cache_history_sampler[sample_set][sample_tag].PCval);
                }
            }
            //Train predictor negatively since OPT did not cache this line
            else{
                if(cache_history_sampler[sample_set][sample_tag].prefetching){
                    predictor_prefetch->decrease(cache_history_sampler[sample_set][sample_tag].PCval);
                }
                else{
                    predictor_demand->decrease(cache_history_sampler[sample_set][sample_tag].PCval);
                }
            }

            optgen_occup_vector[set].set_access(currentVal);
            //Update cache history
            update_cache_history(sample_set, cache_history_sampler[sample_set][sample_tag].lru);

            //Mark prefetching as false since demand access
            cache_history_sampler[sample_set][sample_tag].prefetching = false;
        }
        //If line has not been used before, mark as prefetch or demand
        else if(cache_history_sampler[sample_set].find(sample_tag) == cache_history_sampler[sample_set].end()){
            //If sampling, find victim from cache
            if(cache_history_sampler[sample_set].size() == SAMPLER_HIST){
                //Replace the element in the cache history
                uint64_t addr_val = 0;
                for(map<uint64_t, HISTORY>::iterator it = cache_history_sampler[sample_set].begin(); it != cache_history_sampler[sample_set].end(); it++){
                    if((it->second).lru == (SAMPLER_HIST-1)){
                        addr_val = it->first;
                        break;
                    }
                }
                cache_history_sampler[sample_set].erase(addr_val);
            }

            //Create new entry
            cache_history_sampler[sample_set][sample_tag].init();
```

```
            //If preftech, mark it as a prefetching or if not, just set the demand access
            if(type == PREFETCH){
                cache_history_sampler[sample_set][sample_tag].set_prefetch();
                optgen_occup_vector[set].set_prefetch(currentVal);
            }
            else{
                optgen_occup_vector[set].set_access(currentVal);
            }

            //Update cache history
            update_cache_history(sample_set, SAMPLER_HIST-1);
        }
        //If line is neither of the two above options, then it is a prefetch line
        else{
            uint64_t previousVal = cache_history_sampler[sample_set][sample_tag].previousVal % OPTGEN_SIZE;
            if(set_timer[set] - cache_history_sampler[sample_set][sample_tag].previousVal < 5*NUM_CORE){
                if(optgen_occup_vector[set].is_cache(currentVal, previousVal)){
                    if(cache_history_sampler[sample_set][sample_tag].prefetching){
                        predictor_prefetch->increase(cache_history_sampler[sample_set][sample_tag].PCval);
                    }
                    else{
                        predictor_demand->increase(cache_history_sampler[sample_set][sample_tag].PCval);
                    }
                }
            }
            cache_history_sampler[sample_set][sample_tag].set_prefetch();
            optgen_occup_vector[set].set_prefetch(currentVal);
            //Update cache history
            update_cache_history(sample_set, cache_history_sampler[sample_set][sample_tag].lru);

        }
        //Update the sample with time and PC
        cache_history_sampler[sample_set][sample_tag].update(set_timer[set], PC);
        cache_history_sampler[sample_set][sample_tag].lru = 0;
        set_timer[set] = (set_timer[set] + 1) % TIMER_SIZE;
    }

    //Retrieve Hawkeye's prediction for line
    bool prediction = predictor_demand->get_prediction(PC);
    if(type == PREFETCH){
        prediction = predictor_prefetch->get_prediction(PC);
    }

    sample_signature[set][way] = PC;
    //Fix RRIP counters with correct RRPVs and age accordingly
    if(!prediction){
        rrip[set][way] = MAXRRIP;
    }
    else{
        rrip[set][way] = 0;
```

```
        if(!hit){
            //Verifying RRPV of lines has not saturated
            bool isMaxVal = false;
            for(uint32_t i = 0; i < LLC_WAYS; i++){
                if(rrip[set][i] == MAXRRIP-1){
                    isMaxVal = true;
                }
            }

            //Aging cache-friendly lines that have not saturated
            for(uint32_t i = 0; i < LLC_WAYS; i++){
                if(!isMaxVal && rrip[set][i] < MAXRRIP-1){
                    rrip[set][i]++;
                }
            }
        }
        rrip[set][way] = 0;
    }

}
```

Appendix D
CRC Hash Algorithm

```
//Hashed algorithm for PC: Cyclic Redundancy Check (CRC)
uint64_t CRC(uint64_t address){
  unsigned long long crcPolynomial = 3988292384ULL;  //Decimal value for 0xEDB88320
   hex value
   unsigned long long result = address;
   for(unsigned int i = 0; i < 32; i++ )
     if((result & 1 ) == 1 ){
       result = (result >> 1) ^ crcPolynomial;
     }
     else{
       result >>= 1;
     }
   return result;
```