A Newer Look at Hawkeye

Divyanshu
2021EEB1168
2021eeb1168@iitrpr.ac.in

Shreyansh
2021EEB1055
2021eeb1055@iitrpr.ac.in

***Abstract*—This project explores different possible methods of improving hawkeye using various Software and Hardware Improvement techniques. The main focus was to reduce miss latency and also improve the Instructions per Cycle which was Achieved by effective Cache Partitioning.**

Keywords—OPTGen, HawkeyePredictor, HashMaps, SegmentTree, Cache Partitioning.

I. Introduction

This report outlines enhancements to the Hawkeye cache replacement policy, focusing on implementing superior data structures, cache partitioning, and introducing a virtual buffer. These improvements aim to elevate cache efficiency and overall system performance in response to the growing demands of modern computing workloads.The replacement policy of caches has a major impact on how well they work as crucial mechanisms for cutting down on the lengthy wait times for DRAM memory accesses. Sadly, cache replacement is a challenging issue.

It is difficult to predict which cache line must be evicted earlier and which one to keep. As discussed during the class Belady's algorithm for cache replacement is probably the most efficient policy but is infeasible because it requires knowledge of the future.

The base paper discusses how to implement a better policy than pre-existing policies based on Short-Term History Information and Long-Term History Information as well. Now the pre-existing policies built on heuristics are limited to perform for some specific access patterns and are unable to perform well in more complex scenarios. This paper provides an alternate solution which does not use a heuristic approach but rather an approach based on a variant of Belady's algorithm to the history of past memory accesses. If past behavior is a good predictor of future behavior, then this replacement policy will approach the behavior of Belady's algorithm.

II. Abstract

Belady's algorithm, while theoretically optimal, is impractical in real-world scenarios due to its reliance on knowledge of future cache accesses. This paper explores an alternative approach wherein a cache replacement algorithm learns from Belady's algorithm by leveraging past cache accesses to inform future replacement decisions. We demonstrate that this implementation is remarkably efficient, achieved through a novel method for simulating Belady's behavior efficiently, coupled with established sampling techniques to succinctly represent extensive historical information necessary for accurate predictions. For a 2MB Last-Level Cache (LLC), our approach requires a hardware budget of only 16KB (excluding replacement state in the tag array). When applied to memory-intensive subsets of SPEC 2006 CPU benchmarks, our solution outperforms Least Recently Used (LRU) by 8.4%, surpassing the previous state-of-the-art improvement of 6.2%. In a 4-core system with a shared 8MB LLC, our solution achieves a performance improvement of 15.0%, compared to 12.0% for the previous state-of-the-art approach.

III. A brief idea of Hawkeye

Hawkeye is a sophisticated cache replacement policy that aims to improve cache efficiency by predicting cache-friendly and cache-averse memory access patterns. It combines two key components: the Hawkeye Predictor and OPTgen.

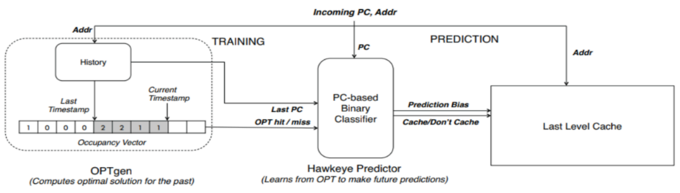


Figure 4: Block diagram of the Hawkeye replacement algorithm.

A. Hawkeye Predictor

This component classifies memory access patterns as either cache-friendly or cache-averse based on historical behaviors. It employs a classification mechanism where the computer is trained to predict whether accessing a particular cache line will result in a cache hit or miss under the optimal (OPT) cache replacement policy. The predictor uses a set of 8K entries indexed by hashed program counters (PCs) and trained with 3-bit counters.

B. *OPT Gen*

OPTgen is responsible for simulating the behavior of Belady's optimal replacement policy (OPT). It determines what cache line would have been cached if the OPT policy were employed. OPTgen operates by analyzing past memory accesses and answering questions about whether the subsequent reference to a cache line would result in a hit or miss under the OPT policy, given the history of memory references.

The combination of these two components allows Hawkeye to make informed eviction decisions in the cache. It first attempts to evict cache-averse lines predicted by the Hawkeye Predictor. If no such lines are found, it resorts to a more traditional policy like Least Recently Used (LRU) for eviction. This adaptive approach helps Hawkeye adapt to changes in program behavior, known as phase changes.

Overall, Hawkeye leverages historical access patterns to predict future cache interactions, aiming to reduce cache misses and improve overall cache performance. It incorporates innovative techniques like Set Dueling and granularity modifications to enhance its accuracy and efficiency. While it shows promising results, especially in scenarios with consistent access patterns, its effectiveness may vary in dynamic workloads, and further refinements could be beneficial.

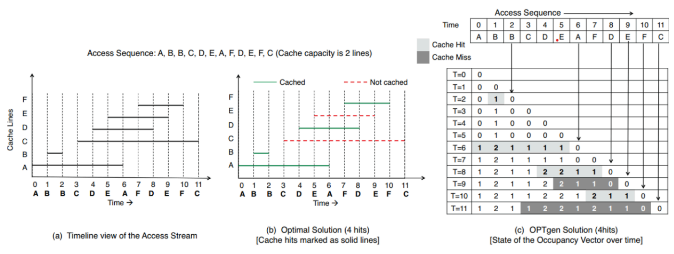

Access Sequence: A, B, B, C, D, E, A, F, D, E, F, C (Cache capacity is 2 lines)
Cache Lines
Time →
Cached
Not cached
Access Sequence
Time
Cache Hit
Cache Miss
(a) Timeline view of the Access Stream
(b) Optimal Solution (4 hits) [Cache hits marked as solid lines]
(c) OPTgen Solution (4hits) [State of the Occupancy Vector over time]

Figure 6: Example to illustrate OPTgen.

IV. Our Developments

In our project to enhance Hawkeye, our primary aim was to optimize its performance in terms of Instructions per Cycle (IPC) and hit rate. To achieve this, we focused on refining the existing Hawkeye design. Our initial approach involved improving the OPTgen (Optimal Prediction Generator) component. We explored two avenues:

- Enhancing the OPTgen vector implementation
- Refining the sampled cache mechanism..

Initially, we attempted to integrate a segment tree into the OPTgen vector to facilitate decision-making regarding cache hits and misses. However, this approach proved unsuccessful due to the complexity associated with changing the overall implementation of Hawkeye, prompting us to pivot towards a map-based implementation for easier decision making of cache Hits and Misses. While this seemed promising, the outcomes fell short of expectations, leading us to explore alternative strategies. This was because of our inability to retain the order of indices.

Subsequently, we implemented a map-based approach within the sampled cache, leveraging hashing techniques to expedite access history retrieval and decision-making processes. This endeavor proved fruitful, as our implementation yielded results comparable to the original code. We were expecting an increase in IPC but due to the inherent parallelism property within the cache our efforts to overcome the latency weren't successful as the time taken for decision making through vector and map were the same.

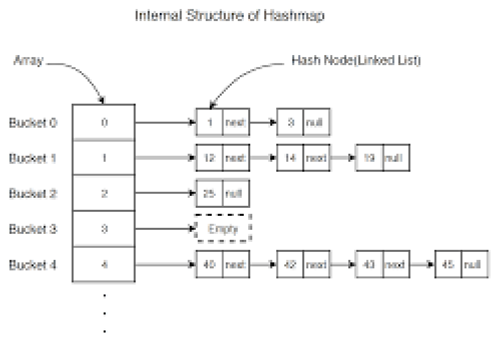


Fig:: Hash maps

Recognizing the limitations of solely focusing on software enhancements, we shifted our attention to improving the cache itself. Initially, we explored static cache partitioning, aiming to allocate cache resources more efficiently. However, due to constraints within the Hermes architecture, traditional cache partitioning was unfeasible, as memory allocation was fixed. To circumvent this limitation, we devised a strategy to allocate virtual buffer memory for cache partitioning purposes.

A. Cache Partitioning

Cache partitioning is a technique used in cache management to improve performance by dividing the cache into multiple partitions. Each partition can be managed independently, allowing for better optimization of cache resources and improved hit rates for specific subsets of data. By isolating working sets, optimizing replacement policies, reducing conflict misses, and improving cache utilization, cache partitioning can lead to overall performance improvements in systems with shared caches.

By partitioning the cache, we can isolate different working sets of data from interfering with each other. Each partition can be dedicated to a specific subset of data, reducing contention and improving hit rates for that particular subset.

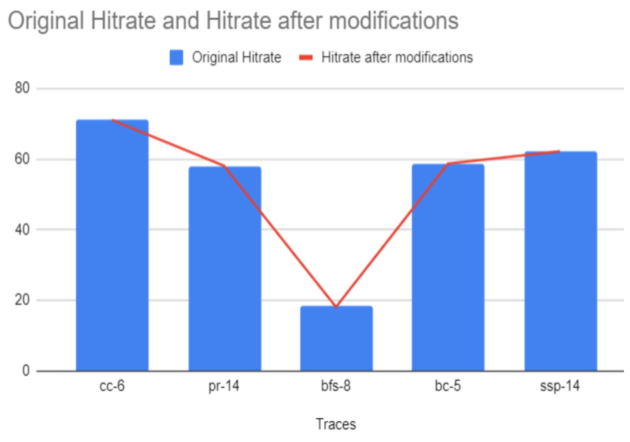
V. Results

We used Hermes setup by Rahul Bera for our analysis.

The Gap benchmark was utilized to compare the performance of our code and the traces used were:

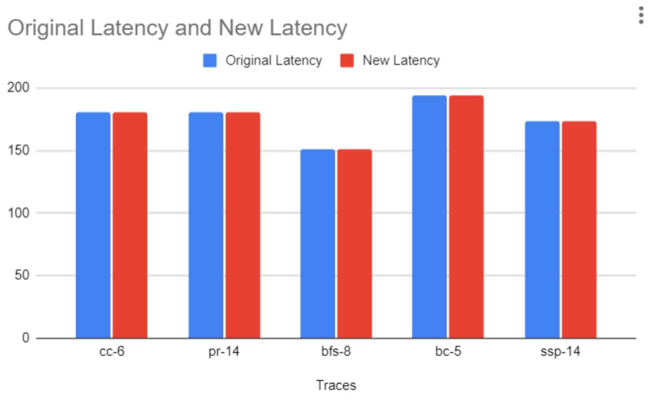
- CC6
- PR-14
- BFS-8
- SSSP-14
- BC-5

A. Comparison between total hit rates for all the traces.



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

B. Comparison between miss latency for all 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
ssp-14	173.61	173.53

As visible hit rate for the new code has increased almost in all cases expert for the trace of bfs. The increase in hit rate across most cases suggests that our optimizations have succeeded in enhancing cache locality and reducing cache misses. The decrease in case of bfs might be due to the diverse nature of access in graph based algorithms.

Similar to hit rates latency has decreased in all cases except bfs.The decrease in latency for all cases except BFS indicates that our optimizations have reduced memory access latency and improved overall system responsiveness.

The increase in hit rate and decrease in latency observed for all cases except the BFS trace indicate that the optimizations implemented in the code have generally improved cache performance and memory access efficiency.

Acknowledgment

We extend our heartfelt gratitude to Mr. Kalyan, Mr. Saurabh, and Mr. Prathmesh for their invaluable contributions and unwavering support throughout the duration of this project. Their combined expertise, guidance, and encouragement have been instrumental in the successful completion of this endeavor.

Mr. Kalyan's mentorship and deep understanding of cache optimization techniques have been instrumental in shaping the direction of this project. His insightful feedback and dedication to fostering an environment of learning have inspired us to push the boundaries of our capabilities.

References

[1] Back to the Future: Leveraging Belady's Algorithm for improved Cache Replacement

[2] https://github.com/CMU-SAFARI/Hermes

[3] Exploiting Secrets by Leveraging Dynamic Cache Partitioning of Last Level Cache

[4] Enforcing Last-Level Cache Partitioning through Memory Virtual Channels

[5] Belady's Algorithm: The foundational work on cache replacement algorithms, providing a baseline for comparison.

[6] RRIP : A notable recent work introducing the Re-Reference Interval Prediction for cache replacement, influencing subsequent research.

[7] PACMan: Prefetch-aware cache management, reflecting the recent trend in optimizing cache performance.