THESIS TITLE

Summer Project 1 Report

Submitted in partial fulfillment of the requirements
for the completion of

Summer Project

by

Rishabh
(Roll No. 200260041)

Under the guidance of
Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
October 2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Rishabh
Electrical Engineering
IIT Bombay

Abstract

The cache memory is a resource that does not need to be explicitly managed by the user. Instead, the cache is managed by a set of cache replacement policies (also called cache algorithms) that determine which data is stored in the cache during the execution of a program. To be both cost-effective and efficient, caches are usually several orders-of-magnitude smaller than main memory (e.g., there are typically a few KB of L1-cache and a few MB of L3-cache versus many GB or even a few TB of main memory). As a consequence, the dataset that we are currently working on (the working set) can easily exceed the cache capacity for many applications. To handle this limitation, cache algorithms are required that address the questions of

1. Which data do we load from main memory and where in the cache do we store it?

2. If the cache is already full, which data do we evict?

If the CPU requests a data item during program execution, it is first determined whether it is already stored in cache. If this is the case, the request can be serviced by reading from the cache without the need for a time-consuming main memory transfer. This is called a cache hit. Otherwise, we have a cache miss. Cache algorithms aim at optimizing the hit ratio; i.e. the percentage of data requests resulting in a cache hit.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

1.2 Work Done Till Now

1. Covered literature on the architecture of Graphic Processing Unit, and parallel programming
2. Covered literature on Instruction Level Parallelism, introduced to concepts of VLIW and Superscalar architecture
3. Learnt memory architecture in computers, introducing concepts of virtual memory
4. Extensively analyzed existing literature on Last Level Cache (LLC) optimizations
5. Covered literature on more efficient replacement policies such as RRIP[1], Hawkeye[2] and Mockingjay[3]

1.3 Motivation

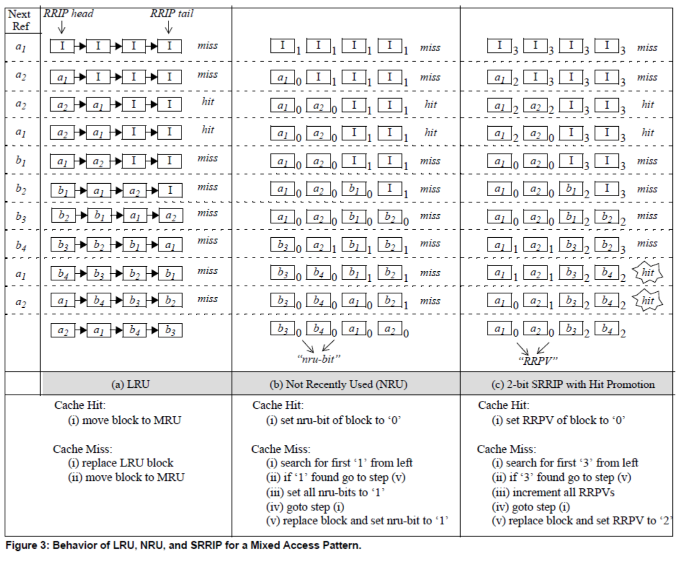
The introduction of memory hierarchy to tackle the long latenciees involved with direct memory access lead to the search for effective replacement policies. The LLC is usually where these policies are implemented as they are relatively large compared to other caches. The LLC's primary objective is to cater to the misses that occur in higher cache levels and due to the large size of main memory compared to cache, the search for efficient replacement policies began. The goal is decrease the number of misses on the LLC so that the requirements of higher cache levels are met immediately thereby decreasing delays associated with access to main memory.

Chapter 2

Literature Survey

2.1 RRIP [1]

A large amount of research is dedicated to improving the performance of replacement policies and the implementation of these policies in inclusive LLCs. The most commonly used replacement policy is the Least Recently Used (LRU) policy which always predicts a near-immediate re-reference interval. Therefore, the performance of applications whose re-references only take place in the far future performs poorly. These applications often have bursts of references to non-temporal data (or scans) or a working set that is greater than the cache (or thrashing access patterns). Since the re-reference of a scan block is in the distant future, scans do not receive cache hits after their initial reference. Re-reference Interval Prediction (RRIP) [2] replacement policy is employed to enhance the performance of the workloads by making the policy scan resistant and thrash resistant. This study proposes Static RRIP (SRRIP) policy which is scan-resistant and Dynamic RRIP (DRRIP) policy which is both scan-resistant and thrash-resistant. RRIP prevents cache blocks with a distant re-reference interval (i.e., scan blocks) from evicting blocks that have a near-immediate re-reference interval (i.e., non-scan blocks). RRIP statically predicts the re-reference interval of all missing cache blocks to be an intermediate re-reference interval that is between a near-immediate re-reference interval and a distant re-reference interval. In a scan-resistant SRRIP, it updates the re-reference prediction to be shorter than the previous prediction upon a re-reference. The paper further proposes two SRRIP policies:



Next Ref
RRIP head
RRIP tail
miss
hit
"nru-bit"
"RRPV"
(a) LRU
(b) Not Recently Used (NRU)
(c) 2-bit SRRIP with Hit Promotion

Cache Hit:
(i) move block to MRU

Cache Miss:
(i) replace LRU block
(ii) move block to MRU

Cache Hit:
(i) set nru-bit of block to '0'

Cache Miss:
(i) search for first '1' from left
(ii) if '1' found go to step (v)
(iii) set all nru-bits to '1'
(iv) goto step (i)
(v) replace block and set nru-bit to '1'

Cache Hit:
(i) set RRPV of block to '0'

Cache Miss:
(i) search for first '3' from left
(ii) if '3' found go to step (v)
(iii) increment all RRPVs
(iv) goto step (i)
(v) replace block and set RRPV to '2'

Figure 3: Behavior of LRU, NRU, and SRRIP for a Mixed Access Pattern.

Figure 2.1: Behavior of LRU, NRU, and SRRIP for a Mixed Access Pattern

1. SRRIP-HP(Hit Priority): This policy predicts that any cache block that receives a hit will have a near-immediate re-reference and thus should be retained in the cache for an extended period.
2. SRRIP-FP (Frequency Priority): This policy predicts that the frequently referenced cache blocks will have a near-immediate re-reference and thus they should be retained in the cache for an extended period.

DRRIP provides both scan-resistance and thrash-resistance by using set dueling to dynamically select between inserting all missing cache blocks with an intermediate re-reference interval or with a distant re-reference interval. RRIP policy outperforms LRU policy in terms of performance, while RRIP requiring 2X less hardware than LRU.

2.2 Hawkeye [2]

Since, policies such as LRU and RRIP are a heuristic-based replacement policy, they only performs well for a limited class of access patterns and are unable to perform well in more complex scenarios. The Hawkeye policy [3] proposes a fundamentally different approach that is not based on heuristics. The paper proposes a cache replacement algorithm which learns from Belady's optimal solution (OPT) algorithm by applying it to past cache accesses to predict the future cache replacement decisions. To learn from the past behavior of Belady's algorithm, it observes that if with the OPT solution, a load instruction has historically brought in lines that produce cache hits, then in the future, the same load instruction is likely to bring in lines that will also produce cache hits. The implementation of this cache replacement policy consists of two components. The first component reconstructs Belady's optimal solution for the past cache accesses. Since a large history would be required to compute OPT, this challenge is solved by use of liveness intervals. This explicitly conveys information about both reuse distance and the demand on the cache, which are both essential for making proper eviction decisions. The second is a predictor that learns from the OPT behaviour of previous PCs in order to inform eviction choices for subsequent loads by the same PCs. It is observed that Hawkeye policy does not increase the number of cache misses for any of the evaluated workloads. It also substantially improves performance upon state-of-the art replacement policies, while the hardware overhead is more than LRU and DRRIP. Its main components are the Hawkeye Predictor, which makes eviction decisions, and OPTgen, which simulates OPT's behavior to produce inputs that train the Hawkeye Predictor.

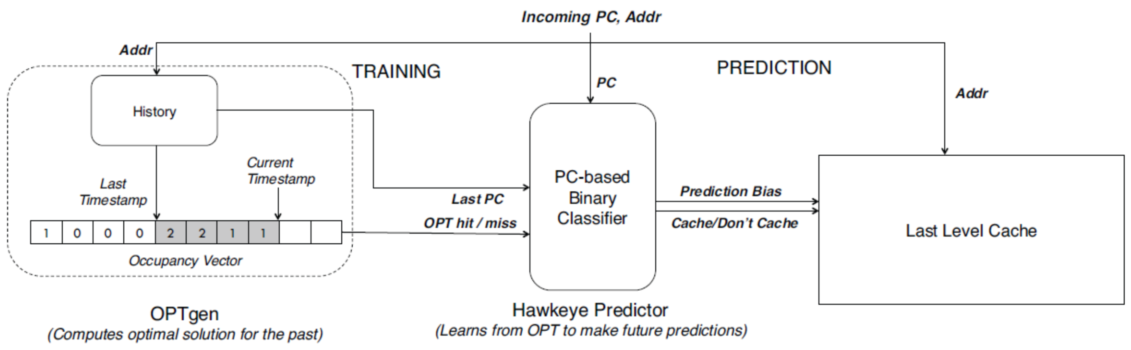


Figure 2.2: Block diagram of the Hawkeye replacement algorithm

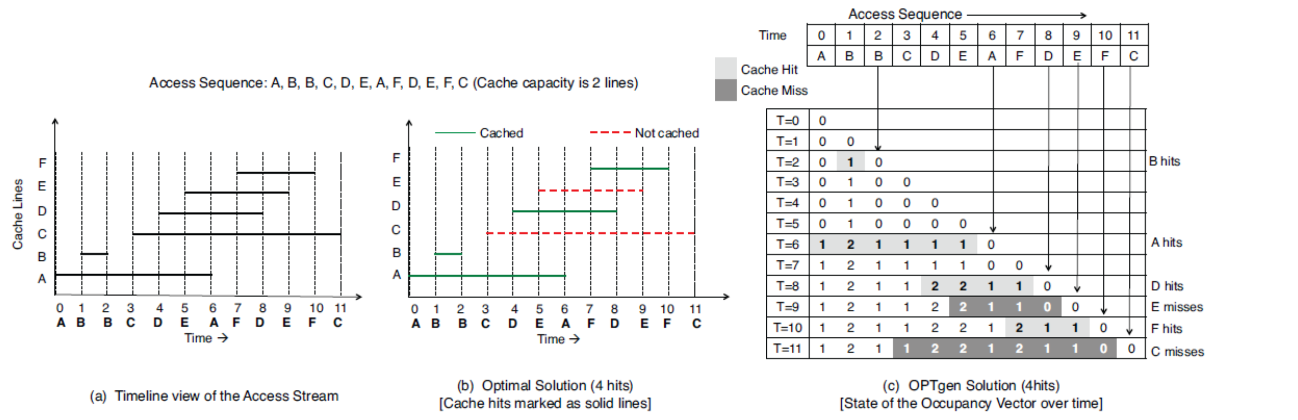

Access Sequence: A, B, B, C, D, E, A, F, D, E, F, C (Cache capacity is 2 lines)
Cache Lines
Time →
(a) Timeline view of the Access Stream
Cached
Not cached
(b) Optimal Solution (4 hits)
[Cache hits marked as solid lines]
Access Sequence
Time
Cache Hit
Cache Miss
B hits
A hits
D hits
E misses
F hits
C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

Figure 2.3: Figure to illustrate OPTgen

2.3 Mockingjay [3]

The past decade has seen the rise of highly successful cache replacement policies that are based on binary prediction. For example, the Hawkeye policy learns whether lines loaded by a given PC are Cache Friendly (likely to remain in the cache if Belady's MIN policy had been used) or Cache Averse (likely to be evicted by Belady's MIN policy). In this paper, we instead present a cache replacement policy that is based on multiclass prediction, which allows it to directly mimic Belady's MIN policy in a surprisingly simple and effective way. Our policy uses a PC-based predictor to learn each cache line's reuse distance; it then evicts lines based on their predicted time of reuse. We show that our use of multiclass prediction is more effective than binary prediction because it allows for a finergrained ordering of cache lines during eviction and because it is more robust to prediction errors. Binary classification leads to two shortfalls: First, a small prediction error will flip a prediction from one class to the other. Second, there can be many ties among lines in the same class, which are typically broken by falling back on the same LRU heuristic that these polices attempt to improve upon. Mockingjay has three main components which are the Sampled Cache, the Reuse Distance Predictor (RDP), and the ETA Counters that reside in the LLC.

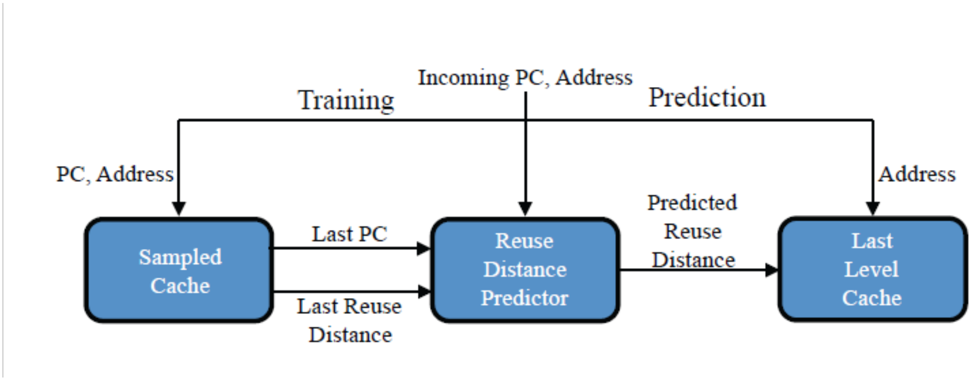


Figure 2.4: Overview of the Mockingjay cache replacement policy

Sampled Cache: The goal of the Sampled Cache is to track past reuse distances to train the RDP, which predicts future reuse distances
Reuse Distance Predictor (RDP): The RDP is a PCbased predictor that learns reuse distances for loads initiated by a given program counter (PC). The RDP is organized as a direct-mapped cache and is indexed by a PC signature. Mockingjay uses separate predictor entries—and therefore learns separate reuse distances—for loads by a PC that hit in the cache and for loads by that same PC that miss in the cache
ETA Counters: Finally, the cache itself maintains the ETA for each line. Upon insertion into the cache, a line's predicted reuse distance is converted to an ETA, and these ETA values are used to make eviction decisions for future cache misses. Cache insertions that are predicted to have ETA values larger than any existing line in the set are bypassed, which means that they are not inserted in the cache.

Chapter 3

Future Scope

3.1 Future Plans

1. Intend to complete simulation of the Mockingjay LLC replacement policy on SNIPERsim
2. Present the Mockingjay paper
3. Explore more on effective repacement policies

3.2 End Goals

1. Explore new policies to optimally replace cache lines in the LLC
2. Successfully implement the Mockingjay replacement policy on SniperSim

References

[1] S. S. J. E. Aamer Jaleel, Kevin Theobald, “High performance cache replacement using re-reference interval prediction (rrip),” 2010.

[2] A. J. Calvin Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2022.

[3] A. J. Calvin Lin, Ishan Shah, “Effective mimicry of belady’s min policy,” *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.