Copyright

by

Qiyang Ding

2023

The Thesis Committee for Qiyang Ding
Certifies that this is the approved version of the following Thesis:

Cache Management Policy in gem5

APPROVED BY
SUPERVISING COMMITTEE:

Calvin Lin, Supervisor

Dam Sunwoo

Cache Management Policy in gem5

by

Qiyang Ding

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2023

Dedicated to my parents.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Calvin Lin, for his invaluable guidance, support, and insights throughout my research and academic journey. His expertise and encouragement have been instrumental in shaping my understanding and passion for computer architecture research, and I am truly grateful for his mentorship.

I also extend my gratitude to Dr. Dam Sunwoo, who served as the reader of my thesis. His computer architecture course at the University of Texas at Austin was a pivotal experience that sparked my interest in this field, and I am honored to have had him as a reader.

I would like to thank Dr. Zhao Zhang for providing me with opportunities to conduct high-performance computing research and work as a Graduate Research Assistant in the Texas Advanced Computing Center. His guidance and friendly demeanor have been invaluable to me.

I am deeply appreciative of Molly's assistance and insightful suggestions on my research about the gem5 simulator. Her support saved me countless hours of time and helped me better understand the simulator.

I am also grateful to Mingyu, Weiran, and Zhengtang, who have been my dear friends and provided me with an unforgettable experience during my time here.

Finally, I dedicate this thesis to my parents, whose unwavering love, support, and encouragement have made all of my achievements possible. Without them, I would not be where I am today.

Abstract

Cache Management Policy in gem5

Qiyang Ding, M.S.E

The University of Texas at Austin, 2023

Supervisor: Calvin Lin

The rapid development of computing platforms has widened the gap between the computing system and memory system, placing more pressure on cache, which is an integral part of the memory system. Despite numerous studies on cache management policies to optimize resource usage, some of them cannot keep up with the fast-paced trends in computing devices. Many of the state-of-the-art cache replacement policies and prefetchers in our research group are based on simulators with simple hardware abstraction for easy development and prototyping, but they do not support more realistic environments, such as cache coherence and heterogeneous systems. This thesis aims to experimentally transplant several cache management policies to more advanced simulators and provide initial experience in dealing with the challenges encountered in the process.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The field of computer architecture has witnessed a significant evolution from single-core systems to multi-core systems, and now towards heterogeneous systems, such as CPU-GPU, CPU-accelerators, and big-Little systems. Despite the advancements in CPU performance over the past few decades, the memory system has not improved as quickly, leading to an increasing gap between computing and memory speed. To address the issue of data loading and storing, various techniques such as efficient cache replacement policies, accurate hardware prefetchers, and 3D-stack caches have been developed. However, with the emergence of heterogeneous systems, these memory techniques face new challenges and need to adapt to handle increasingly complex scenarios within the system.

There exist two major challenges in the vicinity of computing devices: cache coherence and cache partition. Cache coherence pertains to the pattern of data sharing between different devices. In homogeneous platforms, all devices follow the same rules for sharing data, while in heterogeneous platforms, devices have different requirements for data sharing patterns. On the other hand, cache partition refers to the pattern of resource sharing between different devices. For instance, in the current ARM platform, big cores and little cores perform different tasks based on the scheduling of operating systems [1]. The cache resources for each core differ, with some requiring more cache resources to achieve better performance. Big cores generally have more cache requests due to their out-of-order execution and large instruction issue window. However, with less effective cache partition techniques, big cores may use up all the cache resources, leaving little cores with limited cache resources and hampering their performance and quality of service.

Cache partition is a continuously evolving field, and another trend is the change in cache subsystem architecture. With the increase in cache sizes, the assumption of a uniform cache architecture has a detrimental impact on performance. This is particularly evident in large L3 or LLC caches that cannot be placed on a specific die area. A uniform cache architecture assumes that all cache access latencies are the same, resulting in the worst-case access latency being the average for the entire cache. To address this issue, such caches are usually split into different banks to

cater to different requests. Research on wire delay and cache access latency shows that making the cache architecture non-uniform (NUCA cache) results in better average cache access time than using the worst-case access time for the entire cache [2].

Advanced architecture simulators are required to simulate cache coherence, cache partitioning, and NUCA cache as these components significantly increase the memory system's complexity. Simulating NUCA cache also necessitates the simulation of the internal network connecting cache banks. In this thesis, we aim to re-implement some state-of-the-art methods from simple simulators to gem5, an architecture simulator that supports advanced features such as cache coherence and cache banks for NUCA cache [3]. gem5 also enables full-system simulation, allowing the Linux kernel to boot on top of the simulator, resulting in more realistic simulation outcomes. Nevertheless, with the simulator's increased complexity, cache replacement policies and cache partitioning techniques face new challenges that were not considered during their design in simpler environments.

1.1. MOTIVATION

Cutting-edge cache replacement policies such as Hawkeye and Mockingjay have been developed using architecture simulators like Champsim [4]. However, these simulators lack support for cache coherence and realistic memory behavior, thereby hindering the development of more effective cache subsystem techniques. Although significant progress has been made on cache replacement policies and prefetchers using Champsim in our research group, these methods may perform better in more complex systems. To enable further research in this area, an advanced architecture simulator is needed that is suitable for integration with heterogeneous systems and can address issues related to cache coherence and NUCA cache.

gem5 is an advanced architecture simulator that satisfies the aforementioned requirements. It is a highly abstracted simulator that enables full-system simulation, and its interfaces support the integration of customized accelerators, making it convenient for heterogeneous simulation. Furthermore, gem5 provides GPU models and is capable of integrating with current cache designs used in the industry [5].

However, the transplantation of cache-related techniques from other simulators to gem5 poses numerous new challenges. Nevertheless, this endeavor offers an opportunity to enhance the robustness and performance of these techniques in more realistic environments.

1.2. PROBLEM AND CHALLENGE

One of the most significant challenges in re-implementing cache replacement policies and cache partition techniques in gem5 is addressing the differences between gem5 and other simulators. One issue is that, in Champsim or Macsim cache replacement policy interfaces, much of the required information can be directly retrieved, while gem5 does not provide a direct view, making it difficult to obtain the necessary information for these methods. gem5's high level of abstraction also poses a challenge in obtaining ideal information for these techniques, and the simulator's full-system simulation may encounter infrastructure issues such as hanging in the Linux kernel boot program or crashing with strange segmentation faults, which makes debugging extremely difficult.

Additionally, cache replacement policies and cache partition techniques must deal with more cases in the cache subsystem, such as atomic memory access and cache coherence requests, which require handling. Cache partition techniques may change a large portion of codes in cache design and add potential bugs that could disrupt the entire cache behavior.

When re-implementing the cache management policies in gem5, the third challenge is how to improve them for their cache replacement policies and cache partition algorithm. This is mainly in future work, but in this thesis, we will propose some basic ideas that can make these methods more suitable in gem5's memory system and potentially provide better performance.

1.3. CONTRIBUTION

This thesis makes the following contributions:

- We reimplement Hawkeye and Mockingjay cache replacement policy into gem5 and make necessary changes to fit them into gem5 memory system.

- We reimplement Flock, an unreleased cache management policy for heterogeneous multi-core systems, in gem5 .
- We explain how and what we learned when moving simulation infrastructure from Macsim and Champsim to gem5 and propose some new ideas on these methods.
- We evaluate the performance of the Hawkeye and Mockingjay cache replacement policies in gem5 and compare them with their counterparts in Champsim.

1.4. ORGANIZATION OF THESIS

The subsequent sections of this thesis will be structured as follows: Chapter 2 will discuss the related works in cache replacement policy, cache partition, and non-uniform cache architecture. Chapters 3, 4, and 5 will focus on the cache replacement policy, cache partition, and their combinations in this thesis, respectively. Chapter 6 and Chapter 7 will provide a detailed explanation of the memory system of gem5 and the modifications made to all the components to obtain a more realistic simulation result. Chapter 8 will present the results obtained from the experiments. Finally, Chapters 9 and 10 will discuss the future work and conclusion of this thesis.

Chapter 2: Related Work

This thesis presents research on cache replacement policies, cache partition techniques, and uniform/non-uniform cache architectures. The following section will outline different related works in these three areas.

2.1. Cache Replacement Policy

The upper bound of cache replacement policy is Belady's MIN algorithm that needs the future information to make the best cache line arrangement [6]. However, in run-time environment, it's impossible to know the future cache access. Therefore, there are many works on practical cache replacement algorithms and here we only focus on ones that are helpful for this thesis.

Cache replacement policies can be classified into two categories: memory-less policies and prediction-based policies. LRU is the most popular memory-less cache replacement policy, and it remains popular today [7]. Jaleel et al. proposed SRRIP and DRRIP to utilize the behavior between LRU and Most Recently Used (MRU) to achieve thrash-resistant property [8]. Memory-less replacement policies are simple, but they target specific access patterns and cannot deal with the complexity achieved by Belady's MIN policy.

Prediction-based cache replacement policies predict whether or not to cache a particular line. SHiP and SDBP are two prediction-based policies that correlate load instructions with either the first one or the one causing reuse, respectively [9, 10]. Hawkeye simulates Belady's MIN behavior based on past history and classifies cache lines into two categories [11]. Mockingjay uses a different approach to mimic Belady's MIN policy by predicting the reuse distance of each cache line and selecting those that will be accessed in the near future [12]. There is also a perception predictor for cache replacement policy that trains the learner for different cache access patterns [13].

2.2. CACHE PARTITION

Qureshi and Patt published the Utility-Based Cache Partition (UCP), which provides an effective way to partition the cache based on the LRU replacement policy [14]. It also uses set dueling to significantly limit the hardware budget in the partition monitor. Zhan et al. optimize locality and utility by being aware of thread behavior [15]. They use the BIP replacement policy and set the insertion position based on the partition size. Kaseridis et al. propose an advanced partition method that partitions cache-friendly cache lines more effectively and distinguishes the thrashing access pattern of the application [16]. Other methods attempt to partition the sets instead of ways but require redesigning the cache architecture compared to the common one [17].

2.3. NON-UNIFORM CACHE ARCHITECTURE

Kim et al. were the first to propose the concept of a non-uniform cache architecture, which demonstrated that wire delay has a significant impact on cache access latency, especially with larger cache sizes [2]. They also demonstrated that separating a large cache into banks and using different access latencies for each bank improves cache performance. Hardavellas et al. subsequently published a new data placement approach in a distributed shared cache, allocating cache banks to the nearest cores to achieve better cache access performance [18]. They also addressed coherence problems within the NUCA cache using page and TLB support. Kandemir et al. designed a NUCA cache architecture suitable for data migration to improve performance [19]. There are also software-controlled NUCA cache designs. Jigsaw is one such design that partitions the cache based on the operating system and colors the cache banks through page and TLB support [20]. It solves interference and scalability problems in shared caches. Authors then scaled Jigsaw to fit into large distributed caches that use a more aggressive algorithm to color the optimal cache bank allocations for each thread [21]. They also design a hardware monitor that outperforms the utility monitor in UCP and can approximate the hit curve greatly. Schwedock and Beckmann improved Jigsaw to make it suitable for datacenters by addressing security and tail-latency problems [22]. In scientific computing, NUCA

cache is also popular since many applications require larger cache sizes to increase execution speed effectively. TD-NUCA is a method that targets optimizing NUCA performance for clusters at runtime [23].

Chapter 3: Cache Replacement Policy

3.1. Overview

This thesis considers three cache replacement policies, namely Least Recently Used (LRU), Hawkeye, and Mockingjay. LRU is the most common cache replacement policy in current cache designs, as it can effectively utilize temporal locality. Hawkeye and Mockingjay, on the other hand, are methods that mimic Belady's MIN policy to obtain a better choice of victim in the cache. All these three methods are based on a homogeneous system. Hawkeye and Mockingjay both target the Last Level Cache (LLC) to improve performance since LLC has larger cache sizes and a cache replacement policy requiring much hardware budget will not significantly be a great portion of cache design.

3.2. Least Recently Used (LRU)

The Least Recently Used (LRU) cache replacement policy is widely used in modern CPU cache designs as it is simple yet effective [7]. It takes advantage of temporal locality by only evicting the least recently used cache lines when the cache set is full. Additionally, it has a low hardware budget as it only requires a few bits per cache line to determine its priority within the set. As a result, the LRU policy is commonly used in the L1 and L2 cache levels, which are located closest to the CPU and depend heavily on frequently accessed data to improve performance.

3.3. Hawkeye

The Least Recently Used (LRU) cache replacement policy is simple and widely adopted in current cache designs, but it struggles with increasingly complex access patterns. A new, smarter method for cache replacement is needed. Belady's MIN algorithm offers the upper bound of cache replacement performance, but it requires future information that is impossible to predict in online cases [6]. One solution to this problem is the Hawkeye cache replacement policy, which simulates local Belady's MIN behavior and uses this information to make decisions on cache replacement state [11].

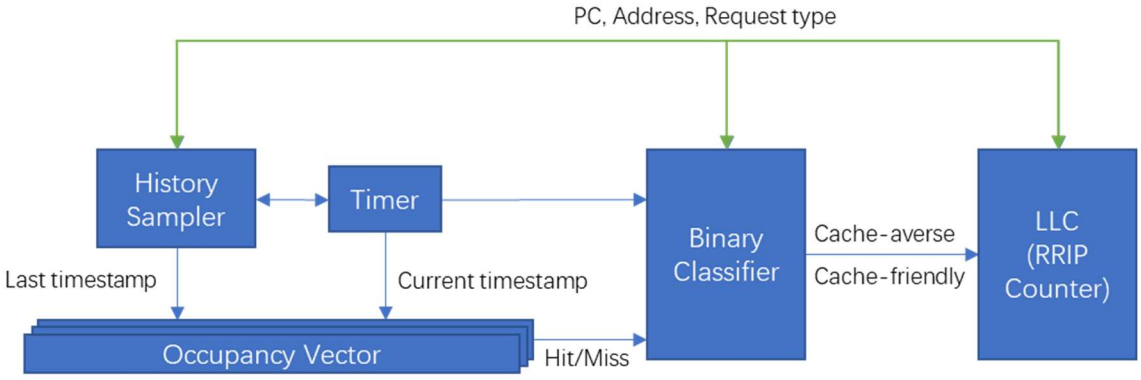

Figure 1: Hawkeye Structure

Hawkeye has three major components: OPTgen, PC-based Binary Classifier, and Re-reference Interval Prediction (RRIP). OPTgen simulates Belady's MIN behavior and provides caching results under this policy. The PC-based Binary Classifier uses the information from OPTgen to classify cache lines into two categories: Cache-friendly and Cache-averse. Cache-friendly cache lines are those that are better to be cached, while cache-averse cache lines are those that will not be used in the near future and have the highest priority to be evicted. RRIP maintains the status of each cache line and builds the priority rank of all cache lines in one set.

The OPTgen module comprises two primary components, namely the history sampler and the occupancy vector. The history sampler stores the history of cache sets, while the occupancy vector records the cache state under Belady's MIN algorithm. An 8x history is necessary to reduce the error of OPTgen's prediction, but this requires a substantial hardware budget. One possible solution to this problem is to increase the granularity of sampling history. For instance, by sampling history once per four cache accesses, the sampling history can only be twice the number of cache ways for a single set.

Another challenge is the difficulty of sampling all the cache sets. In the case of Hawkeye, it is typically implemented on the Last Level Cache (LLC), which is always at least 16 Mbyte in today's chip and includes more than ten thousand sets. To overcome this challenge, the authors proposed a set dueling technique to randomly select some sets while still maintaining accuracy for the entire cache. In practice, only

64 sampled sets are required to depict the state of the entire cache, significantly reducing the hardware budget of OPTgen.

The history sampler is designed as an 8-way set associative cache, with each entry including a 2-byte address tag, a 2-byte hashed PC, and a 1-byte timestamp. The first 2-byte address tag indicates whether the sample line is accessed or not, the second 2-byte hashed PC records the last PC that accesses this cache line, and the final 1-byte timestamp is used to index the occupancy vector with the last access location.

The occupancy vector is a critical component that aids in making decisions under Belady's MIN policy. It is designed as a circular buffer, and each new access sets the most recent location to 0. If the access is the first time (no hit in the history sampler), no changes are required in the occupancy vector. However, if it is not the first time, OPTgen retrieves the old location of the same address accessed in the occupancy vector. If the values in the region from the last access location to the most recent one are greater than the cache capacity, it indicates that under Belady’s MIN policy, the requested block will not be present in the cache. Therefore, the output of OPTgen to the classifier will be a cache-miss case.

Conversely, if there are no values greater than the cache capacity, the output will be a cache-hit case, and these values will increase by 1, representing caching in Belady's MIN behavior. In this way, the occupancy vector helps optimize cache performance by enabling accurate predictions of whether a requested block will exist in the cache or not.

The second component in Hawkeye is the PC-based Binary Classifier, which plays a vital role in classifying cache lines into two categories. This component is implemented as a lookup table that includes a saturation counter in each entry. Whenever OPTgen produces an output, the classifier updates its entries by either increasing or decreasing the counter. Specifically, when OPTgen identifies a cache line access as a cache-hit case, the entries indexed by the last hashed PC are incremented. On the contrary, when OPTgen recognizes a cache line as a cache-miss, the entries indexed by the last hashed PC are decremented. Based on the value of the most significant bit in the saturation counter, the incoming cache line is categorized as

either cache-friendly (1) or cache-averse (0). By effectively distinguishing between these two categories, the PC-based Binary Classifier helps optimize cache performance and improve overall system efficiency.

The third component in Hawkeye is RRIP, which is implemented inside the cache tag and plays a critical role in determining which cache lines to evict. When a cache line is classified as cache-averse, it is always assigned the highest value of RRIP, which indicates that it has the highest priority to be evicted. In contrast, cache-friendly cache lines initially have an RRIP value of 0, indicating the lowest priority to be evicted. Among the cache-friendly cache lines, the victim is chosen following the LRU manner. As new cache-friendly lines are inserted, the RRIP values of existing lines will be incremented by 1, getting closer to be evicted.

3.4. MOCKINGJAY

Mockingjay is a cache replacement policy that mimics Belady's MIN policy, but with an ETA-based method, in contrast to Hawkeye's classification method [12]. The key advantage of the ETA-based method is that it is more resistant to errors. Wrong predictions of reuse distance only affect the order of that cache line within the timeline of the cache, whereas incorrect classifications could affect the global order in the cache replacement process. For instance, Hawkeye might mistakenly identify a cache-friendly cache line as such, leading to a false positive that remains in the cache set for a long time until all other cache-averse lines are evicted.

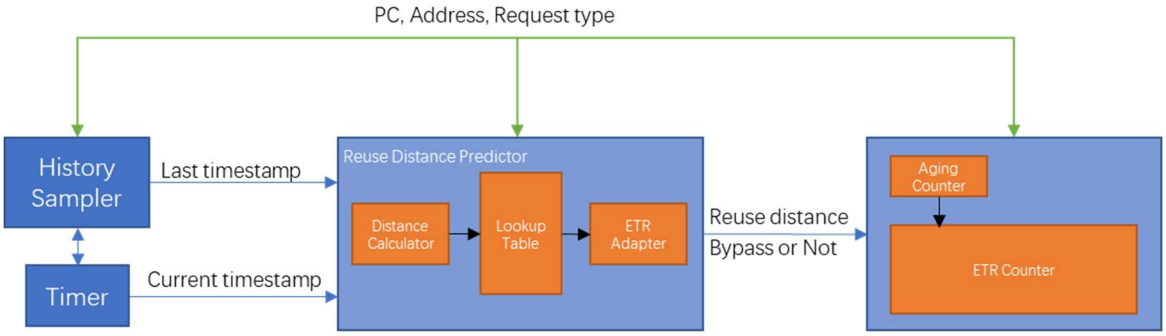

Figure 2: Mockingjay Structure

The estimate time of arrival (ETA)-based method is a cache replacement policy that predicts the time of cache access and uses this information to arrange the cache order, keeping more useful cache lines inside the cache. Mockingjay uses this method to achieve performance that is closest to Belady's MIN policy.

There are three components in Mockingjay: Sampled cache, Reuse distance predictor, and ETR counter. The Sampled cache is the same as what Hawkeye has and provides the necessary history to calculate the reuse distance of a particular cache line. The Reuse distance predictor determines the next access time of an incoming cache line. Finally, the ETR counter arranges the order of each cache line in one set for eviction.

Sampled cache used in Mockingjay is similar to the one in Hawkeye, but with a different indexing policy. It combines bits from set id and address tag to index the set in the sampled cache. In their design, sampled cache is 5-way set-associative cache with 512 sets and the number of sampled sets is 32. Therefore, 5-bit will be chosen from set id to form choices of 32 sample sets and the other 4-bit will be chosen from address tag so that totally 90 history samples can be recorded for one sampled set. Even if it needs 8x address history, which means that 16-way set-associative cache needs 128 access history, some accesses are repeated and 90 entries are enough to record all the accesses. Unlike Hawkeye, the entry in the sampled cache includes a 10-bit address tag, an 11-bit hashed PC, and an 8-bit timestamp.

The Reuse Distance Predictor is the most significant component in Mockingjay, as it is responsible for predicting the next access time of incoming cache line requests. The predictor consists of a lookup table that is indexed by a hashed PC, with each entry representing the reuse distance for the cache request. The predictor performs two functions: predict and train.

During prediction, the predictor first checks if the reuse distance for an incoming cache request is greater than the threshold or if it has not been initialized in a multi-core environment. If either of these conditions is true, the cache line is regarded as a scan line and assigned the maximum distance value. Otherwise, the predictor retrieves the reuse distance for the cache request from the entry in the

lookup table indexed by the hashed PC. Finally, the predictor right-shifts some bits to match the number of bits in the ETR counters to generate the final result.

For the training part, there are two cases: sampled cache hit and sampled cache miss. When there is a sampled cache hit, the predictor will retrieve the last timestamp of the cache line's access and the current timestamp for the entire set. If the counter in the predictor has not been initialized before, it will be directly set to the difference of the two timestamps. If it has already been set, temporal difference sampling will be used to limit the effect of outliers. In the case of a sampled cache miss and one entry being evicted from the sampled cache, indicating that the cache request has not been touched for a long time, the cache line will be treated as a scan line and set to the maximum distance prediction value in the reuse distance predictor. Additionally, the predictor will determine whether the cache line should be cached or not. If the reuse distance is larger than any other ETR values in the current set or it is larger than the threshold, the cache line will not be cached at this level.

The last part is estimated time remaining (ETR) counter. It is used to manage the priority of each cache lines in the cache set. Lower ETR values means that this cache line is predicted to be reused in the nearer future. To lower the space of ETR counters, the granularity of aging the ETR counters are 8. This can help limit the space of ETR counters into 5-bit compared with the size of reuse distance (8-bit). However, the problem occurs when the ETR counter is 0. An out-of-date cache line will keep in the lowest priority to be evicted. The solution is to allow the counter to be negative and use the absolute value to determine the priority of each cache line.

To simplify the work in this thesis, we will not consider prefetcher and leave this into future work.

Chapter 4: Cache Partition Techniques

4.1. Overview

Various cache partitioning techniques have been proposed in the current research, including software-based and hardware-based policies. Software-based techniques use operating systems to pre-set cache budgets for each core based on the executing application, and can use aggressive algorithms to find optimal cache budgets. However, the reconfiguration overhead is high, as it requires communication between the operating system kernel and low-level hardware [24]. Hardware-based techniques, on the other hand, face the challenge of being aware of the different requirements of all applications.

To address this challenge, the Mixture of Experts method in machine learning can be used, where multiple learners work together to divide a problem into homogeneous cases [25]. Cache replacement policies also exhibit learning behavior, and each policy can provide an overview of the access pattern for the device it belongs to. Thus, each policy can be regarded as an expert for that device. Cache partitioning techniques can use this information to make decisions on cache budgets for each device and manage cache lines inside the cache.

This thesis describes two cache partitioning techniques, namely the classic Utility-based Cache Partition (UCP), published in 2005, which determines cache budgets for different cores using the LRU policy, and Flock, an unreleased technique designed to address the challenge of cache partitioning in heterogeneous systems [14, 26].

4.2. UCP

Utility-Based Cache Partitioning relies on the LRU replacement policy, which follows the stack property. This means that an access hitting in an LRU-managed cache will also hit in its larger LRU-managed cache. To obtain cache hit information under different cache sizes, monitors can be added for cache blocks. An auxiliary cache is also added to store the hit count for each entry in the LRU cache set. The auxiliary cache stores only the address tag and a counter to measure the hit count for

that entry. With the help of the stack property, if the cache size is decreased, it only needs to sum the total hit counter values without several least recently used ones, based on how much size it wants the cache to be. Algorithms can then be used to decide the cache budgets based on this information from the hardware for performance under different cache sizes.

Another important contribution of Utility-Based Cache Partitioning is set dueling, which can significantly reduce the hardware budget required for the utility monitor. Set dueling involves sampling some parts of the cache to approximate the status of the entire cache. For example, in UCP, monitoring 32 sampled sets is sufficient to approximate the results obtained from monitoring all the sets in the cache. This method is also used in other cache partitioning techniques, such as Hawkeye and Mockingjay, to reduce the hardware budget required for sampling the cache access pattern while still obtaining a confident approximation.

The partitioning algorithm used in UCP calculates the total cache miss count for different partition sizes and selects the one with the lowest miss count. It then calculates the increase or decrease in miss count for different applications and chooses the partition budget that minimizes the overall miss count, resulting in the best possible cache performance.

4.3.FLOCK

Flock is an approach to cache partitioning techniques that mainly targets heterogeneous multi-core systems, as different devices have their own memory access patterns [26]. For instance, in a big-Little multi-core system, the big core may require more cache resources than small cores during heavy workloads, while small cores may always use some cache lines for small background tasks. In a CPU-GPU system, such as in today's mobile SoCs, they are all integrated on the same die and share the last-level cache. When a GPU task is running, it always has a streaming access pattern, which means that it will not frequently access the cache line it brought in again, while the CPU still needs temporal locality to increase its memory operation speed. The problem arises when the cache partitioning technique provides too much cache budget to the GPU system.

The proposed solution aims to enable more efficient cache management through a smarter approach that utilizes cache replacement policies. Similar to the Mixture of Experts (MoE) approach in machine learning, this technique adopts a unique cache replacement policy for each core or device, and combines the decisions made by all of them to arrive at final decisions regarding cache lines. It consists of three components: the Partition Allocator, the Cache Replacement Policy, and the Aging Scheme.

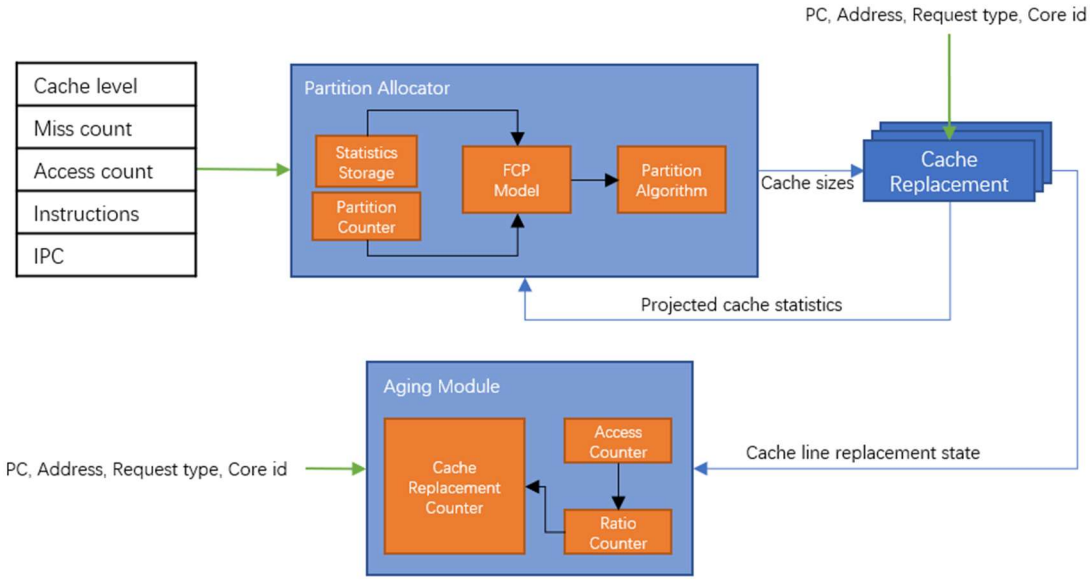

Figure 3: Flock Structure

The Partition allocator is responsible for determining the cache budget. It reads statistics from higher-level caches and lower-level memory systems, including access count, hit count, miss count, and average cache access latency for the former, and row hit count, row miss count, access count, and average DRAM latency for the latter. Additionally, it incorporates a performance model to quantify the current cache subsystem's performance. FCP is the current performance model used, and it is calculated by $FCP = (mr1 - mr2) * T_2 + (mr2 - mr3) * T_3 + mr3 * T_{dram}$, where mr1, mr2, and mr3 denote the miss per instruction in L1, L2, and L3 cache, respectively [27]. This equation is for 3-level caches, which is the most

common type of memory system today. T_2 and T_3 denote the average access latency of L2 cache and L3 cache, respectively, while T_{dram} is the average access latency of DRAM. For different cache partition budgets, the only variables that change are mr3 and T_{dram}, as mr1, mr2, and T_2 are fixed regardless of L3 cache behavior, assuming the cache is non-inclusive and non-exclusive.

To obtain the projected statistics, Additional occupancy vectors in Hawkeye are used. These vectors can provide hit or miss counts under Belady's MIN policy with different cache sizes. However, the original paper computes the projected statistics for all partition choices, which can incur too much hardware overhead since it requires 16 occupancy vectors if the cache has 16 ways for each set. Therefore, in this thesis, to first evaluate the upper bound performance of using occupancy vectors, we assume there are 16 occupancy vectors. We also propose that using one or two extra occupancy vector is possible and will be described in the next chapter.

The calculation of T_{dram} is more complicated since the different miss ratio in L3 can affect the performance of DRAM. A linear approximation is used to determine the projected DRAM access latency by using the ratio of the row miss rate and row hit rate, and also the ratio between the estimated miss count under different cache partition sizes and the current miss count as the factor. This is then multiplied with the current average DRAM access latency. After obtaining all the variables in the equation above, we can then determine the speedup for increasing or decreasing the cache partition budget.

The task of finding the optimal partition budget is challenging since it involves solving an NP-hard knapsack problem [28]. Flock addresses this issue by dividing the cache partition budget into multiple pieces and assigning each piece to the device that can achieve the highest FCP gain. The FCP gain represents the amount of improvement that can be achieved in cache performance by allocating additional cache budget to a device. The algorithm used by Flock is outlined below:

Algorithm 1 Heuristic for Scalable Partitioning

```
1: procedure DECIDE PARTITION
2:     start ← 0
3:     partition[num_cores] ← start
4:     total_credit ← ways
5:     while total_credit > 0 do
6:         minID ← get_max_ipc_gain()
7:         partition[maxID] ← partition[maxID] − 0.1 ∗
   ways
8:         total_credit ← total_credit − 0.1 ∗ ways
9:     end while
10: end procedure
11: procedure GET_MAX_IPC_GAIN
12:     max_core_id ← −1
13:     max_gain ← 0
14:     for each core_id idx do
15:         k ← partition[idx]
16:         gain ← FCP[idx][k+1] − FCP[idx][k]
17:         gain ← gain/CPI[idx]
18:         if max_gain < gain then
19:             max_gain ← gain
20:             max_core_id ← idx
21:         end if
22:     end for
23:     return max_core_id
24: end procedure
```

Figure 4: Flock cache partition algorithm

The cache replacement policy in Flock generates cache replacement results for each device based on the partition size provided by the allocator. The theory is to let the cache replacement policy make decisions assuming its cache capacity is the partition size, which is a flexible way to adapt to dynamic cases with different requirements from devices. The candidate cache replacement policies are Hawkeye and Mockingjay, and their integration into Flock will be explained in the next chapter.

The aging scheme in Flock is based on the access ratio of different devices. If one core has twice the access as another core, then the cache line aging for that core will occur once per two accesses. This balances the aging speed of different cores to avoid one device burning out all the cache resources.

Chapter 5: Cache Management Policy

5.1. Overview

The cache partition technique is a crucial step in defining the cache budget, but its role is limited to this task and does not extend to the allocation of cache lines to different devices. This allocation is determined by the enforcement policy, which governs the arrangement of cache lines within a set. In the context of Flock, the enforcement policy consists of two key components, namely the cache replacement policy and the aging scheme, which jointly determine the status of each cache line in the set.

5.2. FLOCK with HAWKEYE

Flock employs the cache replacement policy as its enforcement policy for cache partition budget, which is based on the Hawkeye. Hawkeye is responsible for providing the necessary information for the partition allocator. In Flock, each device sharing the cache has a specific Hawkeye component installed, which acts as an expert for that device and does not interfere with other devices. For example, in a big-little multi-core system, the cache request from the big core is learned by the Hawkeye component assigned to that core. The Hawkeye consists of 16 occupancy vectors for a 16-way set-associative cache, which offer statistics that aid in miss count determination under Belady's MIN policy with varying cache capacities. Moreover, the projected cache partition size to the current size ratio can serve as a factor for estimating the total miss count for different cache partition sizes.

However, using 16 occupancy vectors per device incurs significant hardware overhead. To reduce this, an alternative approach is to use only one additional occupancy vector for all partition sizes. In this scheme, the partition process is gradually executed until it reaches the maximum cache budget. Each time the partition algorithm runs, it estimates the partition budget increase for one step since only one occupancy vector is available. This can be similar to a predecessor-successor pair for the current partition size and the next partition size for each core. However, this approach lacks support for decreasing the cache partition budget. To address this,

adding another occupancy vector to track the next larger partition size and the previous smaller partition size may be a potential solution.

After classification by Hawkeye for each core, cache lines in one set are sorted based on their cache-friendliness. The cache-averse cache line will be evicted first since it receives the highest priority of eviction. The cache-friendly cache line will be aged using Flock's aging scheme. Flock's aging scheme differs from the original Hawkeye aging scheme in that it only ages the cache line when a cache-friendly line is inserted.

Flock's enforcement policy relies on its aging scheme for cache line management across different devices. While this approach offers much flexibility for accommodating diverse requirements, it may not achieve the desired partition performance due to its weak enforcement policy. Furthermore, it is worth noting that the performance model FCP utilized in Flock may not be suitable for all computing models and must be reviewed when there are changes in the memory system or CPU/GPU model.

5.3.FLOCK WITH MOCKINGJAY

Mockingjay is also being considered as a candidate for the cache replacement policy in Flock, due to its higher cache performance than Hawkeye. It is also suitable for the partition allocator since it has sampled cache which stores the same information for the occupancy vector. However, integrating Mockingjay into Flock requires changing both the aging scheme and cache replacement policy. Mockingjay is an ETA-based method that uses a timeline to create a relative order for all the cache lines it manages and evicts the ones in the farthest future. In Flock, each Mockingjay component will provide its timeline prediction, which is the reuse distance of a cache line. We will initially keep the same aging scheme and assess whether there is any improvement compared to Hawkeye. However, there are potential issues to consider. Firstly, different Mockingjay components will give different reuse distance predictions, and it cannot be the global timeline order without any offset. Secondly, Mockingjay is not aware of the cache capacity since it only predicts the reuse distance and arranges them through ETR counters.

Therefore, we cannot give a comprehensive evaluation of Flock with Mockingjay and leave it as future work.

Chapter 6: Simulation Infrastructure

6.1. SIMULATOR

In this thesis, we will use gem5 as our experiment simulator [29]. Although Hawkeye and Mockingjay were originally implemented on Champsim and Flock was implemented on Macsim, both of these simulators are more simplistic and were chosen for their development efficiency [4, 30]. However, since our final target is to work on non-uniform cache architectures and CPU-GPU system, which are not available in these simulators, gem5 is a better choice. gem5 is an open-source architecture simulator supported by companies and institutions like AMD Inc., ARM Inc., and UC Davis. It supports more advanced features and provides a more realistic simulation environment, including Linux kernel full-system simulation and CPU models similar to current products. Additionally, it provides a robust simulation infrastructure, including debug classification and statistics output [31].

6.2. GEM5

gem5 is one of the most comprehensive simulators for computer architecture research. In this thesis, we will focus on gem5's memory system, including cache and memory design. All the components in gem5 are highly abstracted and use proper interfaces that allow users to create their own components. gem5 offers two cache system designs: Classic cache and Ruby memory model. The former has a simpler cache architecture but does not simulate multi-bank caches. It only supports MOESI cache coherence and cannot be easily modified. In contrast, Ruby memory model simulates multi-bank cache with a network within the cache system. It also supports more advanced cache coherence protocols and allows users to implement their own. Most importantly, multi-bank cache allows us to simulate non-uniform cache architecture easily without greatly changing the behavior of the cache system. Due to time limitations, this thesis will not use Ruby memory model, but future work will move towards it.

6.3. CLASSIC MEMORY SYSTEM IN GEM5

The classic cache model in gem5 can be found in the src/mem/cache directory. It consists of a base cache class and a base set-associative cache class. For this thesis, we only use the set-associative class and its basic indexing policies (block offset, set, and address tag). This cache implementation includes port implementation and can be connected to XBar or CPU ports. L1I caches and L1D caches are always connected to the CPU Instruction or Data ports. Communication between the CPU and caches occurs through these ports and uses packets to transfer necessary information. Packets include all the information the memory system needs to know, such as command type, request type, address, data, etc. These packets flow through the entire memory system until completion. If a response is required, the packet is flipped from request to response and sent back. Each cache has MSHRs that temporarily store missing cache requests until the response is ready.

There are two modes in the classic cache model: timing mode and function mode. Function mode does not consider any access latency inside the cache, while timing mode is more accurate and considers lookup latency, data/tag access latency, packet waiting latency, etc.

Another essential component is the cache tag class, which includes cache insertion, cache access, and cache replacement. In this thesis, cache replacement will be the most important component, and gem5 provides the necessary interface for the cache replacement policy to function correctly. The interface includes four functions: invalidation, initialization, insertion, and update. The first one is designed for cache coherence, meaning that when there is a cache invalidation request, it will use this invalidation function to reset the replacement state for that cache line. The second one is for the initialization of all the components needed for this cache replacement policy. For example, RRIP values and valid bit will be set to 0 at the start of execution. The third and fourth functions are for cache miss handling and cache hit update. The former will be called when a missing cache line is replied from the low-level memory system, while the latter will be called when an address tag matches any valid cache lines. For the third and fourth function, it provides two overloaded interfaces to

support advanced cache replacement policies that need to know the request information.

The memory system statistics also include all the necessary behaviors, such as cache miss count, cache access count, and average access latency. All the statistics are inherited from statistics::Group and include interfaces for Python to directly get and dump to files when the simulation ends. The problem here is that since these statistics classes are highly abstracted and used for high-level simulation, the values in the statistics are not easy to use in the memory systems. For example, Flock needs statistics from other levels of cache systems and low-level memory, and these statistics are very difficult to transfer to Flock. There are two tricky solutions, which will be discussed in the next chapter.

In the memory system model, there are two base classes: memory interfaces and memory controllers. The former is designed for any type of memory, such as DRAM or non-volatile memory, while the latter is responsible for sending or receiving requests through the memory system and higher cache system.

Therefore, the entire process for a read or write CPU memory request is as follows: the CPU sends the request to its L1 caches through a port connection, and the caches first search the cache tags to determine whether the cache line is hit or not. If it is a hit, the request is replied to directly; if it is a miss, it is inserted into the MSHR and added to the buffer in the port. The port then sends the requests in the buffer to its pair. L2 and L3 caches perform the same actions as L1 caches. From the last-level cache (LLC) to the memory, since they use the same port interface, the complexity is hidden from the cache to the memory, and the memory controller deals with these cache packets and converts them to a new memory packet for the actual memory design. After the memory resolves the packets and retrieves the data, the memory controller replies until it reaches its timing and sends it back to the cache system.

Chapter 7: Implementation

7.1. HAWKEYE

We have implemented Hawkeye based on the paper and created three new classes to support the interface provided by gem5. For invalidation, we now directly invalidate the cache line instead of invalidating the same history inside the sampled cache. In future work, we will explain how to make Hawkeye support cache coherence. The fields we added for each cache line in Hawkeye are the valid field and the RRIP counter. During initialization, the former is set to false and the latter to 0.

The two most important functions in Hawkeye are insertion and update, which are called reset and touch in the gem5 replacement policy interface. We have chosen to use the interface with packets since Hawkeye needs to know information from the CPU. The accepted packet to the reset function is the cache line with PC, context ID, and response type. Since the cache miss handling type is response, Hawkeye handles these requests. It then sets the corresponding RRIP value and inserts this cache line's replacement information into the cache tag. If it hits in the history sampler, it updates the corresponding entry in the history sampler and gets the last timestamp and current timestamp. Then, the classifier is trained through the last PC index and its cache hit or miss under Belady's MIN policy. Finally, this new access is added to the occupancy vector. The touch function shows the same behavior, except for the setting of the RRIP value. When a cache-friendly line is hit in Hawkeye, the RRIP value is reset, while if a cache-averse line is hit, the RRIP value is kept as the maximum one.

7.2. MOCKINGJAY

In gem5, an abstract cache replacement policy interface is provided, which both Mockingjay and Hawkeye utilize. However, the main difference between the two lies in their distinct internal components and behaviors.

7.3. FLOCK

To simplify Flock implementation in gem5, most of its functions are implemented through the replacement policy interface. This is because modifying the

base cache class behavior will break the abstraction layers and creating a new cache child requires numerous changes, potentially leading to problems in simulating cache behavior. Therefore, all components are limited to the replacement policy interface.

One significant issue in Flock design is statistics. In Macsim, CPU and DRAM performance counters can be easily obtained, providing a convenient albeit unrealistic way to test. However, gem5 is more realistic and highly abstract, and there is no direct connection between CPU, DRAM, and caches. There are two solutions: connecting LLC with other caches, CPU, and DRAM directly in Python Object or transferring information through packets. The former is tricky since it breaks the abstraction layers between the system, and it ignores the latency of obtaining performance counters. The latter is more realistic but incurs another problem: delayed statistics are used to determine partition size.

To facilitate Flock aging behavior implementation, we add a function called access() to the cache replacement policy interface. This function records statistics from higher-level caches and lower-level memory before adding its own statistics for FCP calculations. Determining which cache statistics are used is a problem. For L1, two caches are combined as the total L1 access miss count. We also add cache level parameters to caches indicating whether they are L1, L2, L3, or other caches. The aging scheme is executed in this function, determined by the ratio counters, aging all cache lines belonging to the core if the counter reaches its maximum value. The calculation of a new cache partition and aging counter reset is also implemented in this function. There are two counters for these functions, and the threshold values are self-customized in the code.

The insertion and update for Flock are similar to Hawkeye, except for the number of components. In Flock, the number of components is based on the number of cores or devices input by the user. If there are four cores, then all the components in Hawkeye will be four times as much as in one Hawkeye. These components are stored in a 1-D vector container and indexed by the core ID (context ID in the packet). Occupancy vectors are 16 times as much as the number of cores and indexed by 16 times the number of cores. For each cache hit or miss for a particular core, its

corresponding projected occupancy vectors are updated to match the current partition size.

The FCP performance model can be estimated once all the statistics are ready. The current implementation hard-codes the calculation to be done at the L3 cache level (LLC) and it is based on the statistics obtained from each packet. This performance model estimation is called when the partition algorithm tries to find the cache budgets for each core. The update of ratio counter is based on the access performance counters in the L3 cache. It will first choose the smallest access count and then use that one to set the ratio counter for the other cores. Both partition allocation and aging are executed repeatedly based on the number of access counts.

Cache coherence is also a future work in Flock design, and thus, the invalidation function will only invalidate the corresponding line without any further action.

7.4. LESSONS FROM SIMULATION INFRASTRUCTURE

This thesis focuses on experimental transplantation from simpler simulators with user-friendly interfaces to more complex ones. Through this process, we have learned several valuable lessons, including:

- Highly abstracted simulator design is convenient when interfaces are explained well, but it will add more overheads when we want to break the layers and increase the development or prototype speed.
- Trace-based simulators like Champsim are easier to debug and can accurately reproduce problems, while execution-based simulators like gem5 often encounter internal problems like segmentation faults or library bugs and may not always be able to reproduce issues.
- Full-system simulation in gem5 requires a deep understanding of the operating system, as the simulator must first boot the OS before executing benchmarks. However, the booting process can encounter various bugs that may cause the simulation to hang indefinitely.

Chapter 8: Evaluation

We conducted an evaluation of Hawkeye and Mocking in the gem5 simulator using the configurations outlined in Table 1 for single-core and Table 2 for multi-core scenarios. Due to time constraints, we selected nine memory-sensitive benchmarks from SPEC2006 and employed statistical sampling to approximate performance evaluation in gem5, as described in [32]. gem5 supports a fast-forward mode that allows for the quick execution of parts of the benchmark, enabling simulation speed to be increased. To implement this, we first utilized the KVM version of the CPU, which directly executes instructions on the local host CPU. We then sampled three points in the benchmark by fast-forwarding 20 billion lines of code for single-core and 40 billion lines of code for multi-core, warming up 50 million lines of code for single-core and 100 million lines of code for multi-core, and executing 200 million lines of code for single-core and 400 million lines of code for multi-core. Following the fast-forward phase, gem5 switched the CPU model to the timing model used in the experiment.

L1 Cache	32Kbyte, 8-way associative, tag/data latency 4, 8 MSHR, Private Cache
L2 Cache	256Kbyte, 8-way associative, tag/data latency 14, 32 MSHR, Private Cache
LLC Cache	2Mbyte, 16-way associative, tag/data latency 44, 256 MSHR, Shared Cache
DRAM	3GByte, DDR4-2400

Table 1: Evaluation configurations for 1-core simulation

8. L1 Cache	32Kbyte, 8-way associative, tag/data latency 4, 8 MSHR, Private Cache
L2 Cache	256Kbyte, 8-way associative, tag/data latency 14, 32 MSHR, Private Cache
LLC Cache	4Mbyte, 16-way associative, tag/data latency 44, 256 MSHR, Shared Cache
DRAM	3GByte, DDR4-2400

Table 2: Evaluation configurations for 2-core simulation

8.1. SINGLE-CORE EVALUATION

In this thesis, most of the experiments were conducted on a 1-core system due to time constraints. The results are presented in Figure 5 and Figure 6, with the former showing the system performance in terms of instructions per cycle (IPC), while the latter indicates the Miss Count Per Kilo-Instructions (MPKI), which represents the cache's performance. From the figures, it is evident that the sphinx3, bzip, and tonto benchmarks experience significant overhead when using Mockingjay, while tonto shows a reduction in MPKI. One possible explanation for this discrepancy is the interference of the operating system, which can affect CPU core instructions and generate more requests not related to benchmark execution. Conversely, hmmer demonstrates the largest reduction in MPKI and the best system performance, which is consistent with the statistics in the two figures.

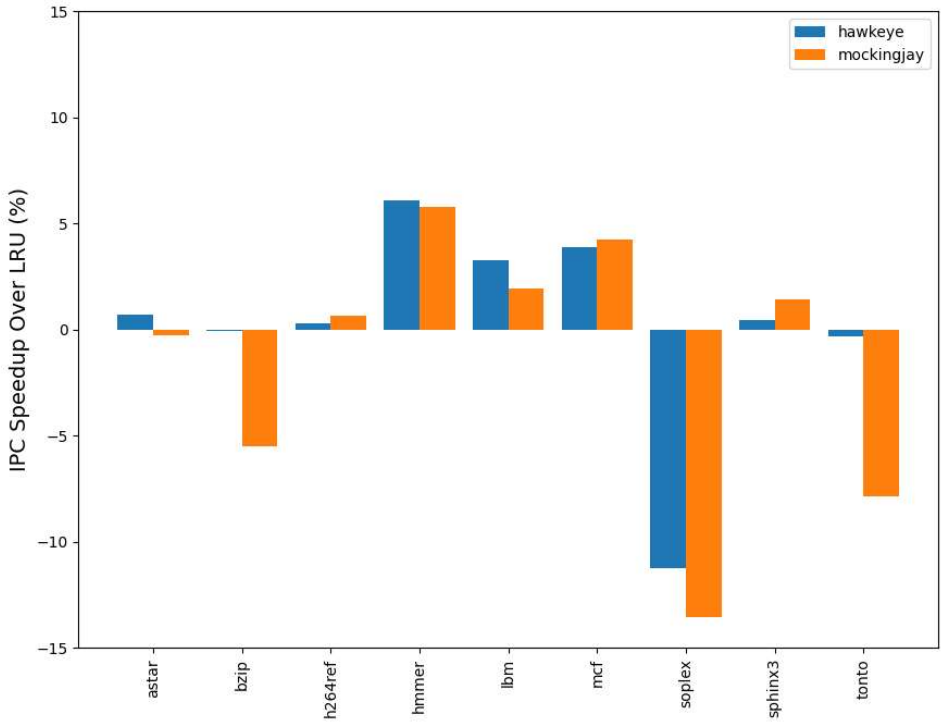

Figure 5: IPC Speedup for Hawkeye and Mockingjay (gem5)

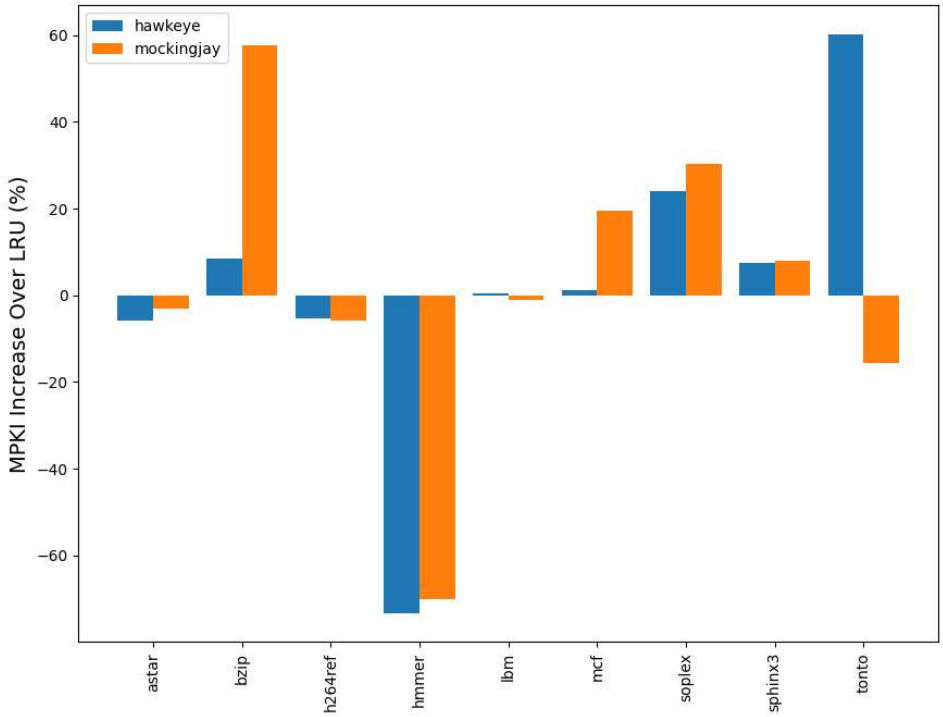

Figure 6: MPKI Reduction for Hawkeye and Mockingjay (gem5)

We also present the system performance of running Hawkeye and Mockingjay on Champsim, which uses similar LLC cache configurations but different L1, L2, CPU and memory models.

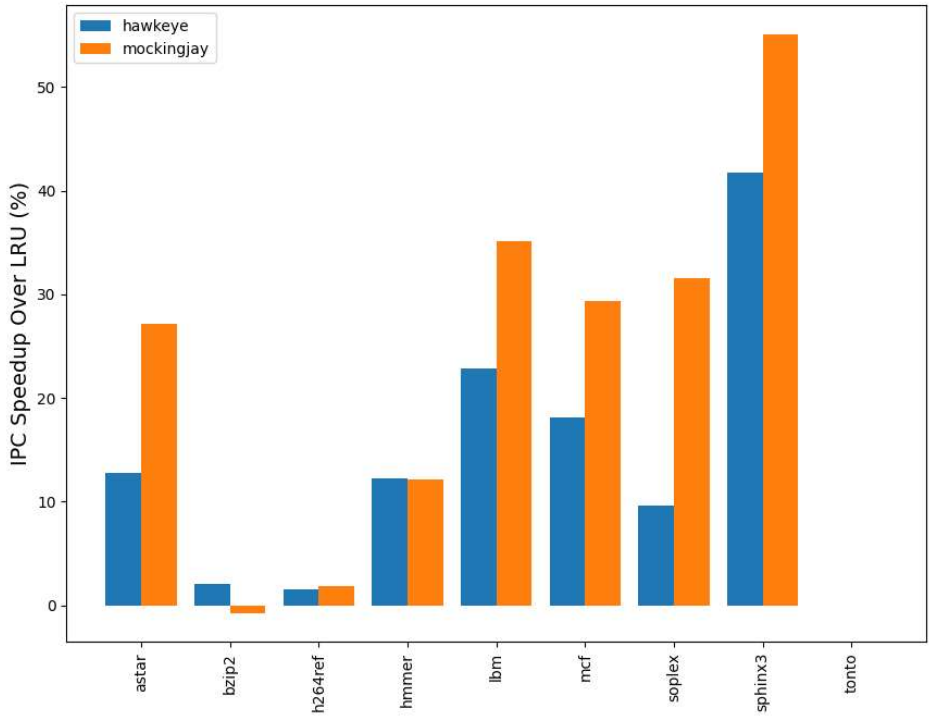

Figure 7: IPC Speedup for Hawkeye and Mockingjay (Champsim)

Figure 7 shows the IPC speedup for Hawkeye and Mockingjay on Champsim. The tonto benchmark shows almost zero improvement in system performance with LRU, Hawkeye, and Mockingjay cache replacement policies. However, for other benchmarks except bzip2 with Mockingjay, there is a significant improvement in system performance, which does not match the cases running in gem5. Since full-system simulation is vastly different from trace-based simulation, more interference such as cache coherence requests inside the memory system, memory behavior, operating system, or cache replacement policy implementation issues may be present. Different configurations may also incur large mismatch cases between Champsim and gem5 results. This discrepancy will be further analyzed in future work.

8.2. MULTI-CORE EVALUATION

Furthermore, we evaluate the performance of Hawkeye and Mockingjay using a 2-core multi-programmed workload in gem5 full-system simulation mode. Two benchmarks, SPEC2006 astar and h264ref, are run, and the LLC cache is increased to 4Mbytes. Figure 8 shows the IPC speedup for Hawkeye and Mockingjay under two different cache replacement policies. Core 1 performance shows a significant increase, while Core 2 performance is slightly lower than LRU. This difference may be due to the PC indexed behavior in Hawkeye and Mockingjay, and operating system scheduling also plays a crucial role in determining which core executes the benchmark. Additionally, the MPKI reduction in LLC over LRU for Hawkeye, and Mockingjay is 89.2% and 87.9%, indicating excellent performance when running two benchmarks on a 2-core system.

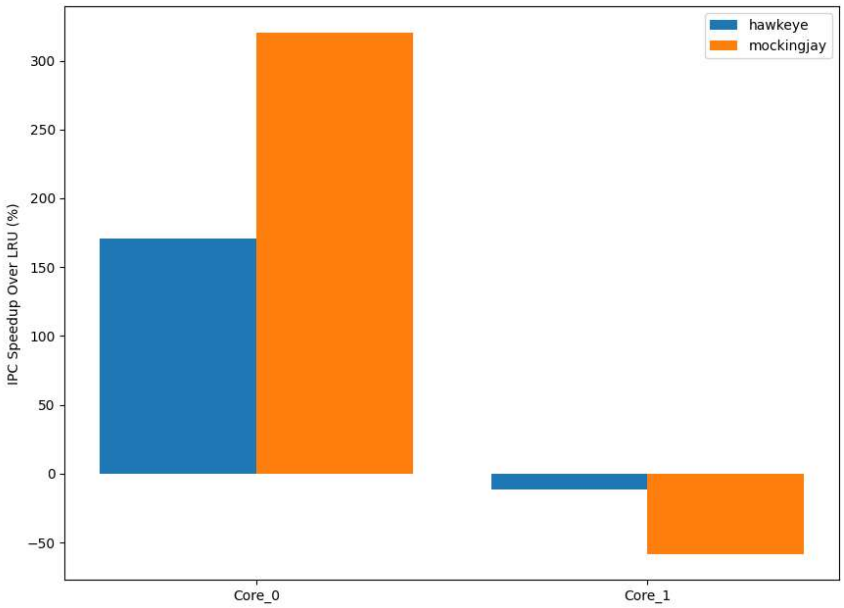

Figure 8: IPC Speedup for Hawkeye and Mockingjay (gem5, 2-core)

Chapter 9: Future Work

9.1. Flock With Mockingjay

The Flock implementation allows for diverse cache replacement policies; however, limitations in hardware overhead constrain the range of feasible options. In order to acquire anticipated statistical projections for various partition sizes, the partition allocator necessitates an occupancy vector. Given that the architecture of Mockingjay bears similarities to Hawkeye, it is likely that it will be harmonious with the Flock framework. Nevertheless, a key issue that remains unresolved is the arrangement of diverse reuse distances stemming from multiple Mockingjay predictors. In the preceding chapter, initial ideas were presented, and these will be further investigated in subsequent research.

9.2. Cache Partition in NUCA

One of the primary objectives of this research is to identify and implement an effective cache management policy that incorporates cache replacement policies and cache partition techniques in Non-uniform Cache Architecture (NUCA). Although NUCA is widely used in current chip Last Level Cache (LLC) designs, these designs typically adopt a static NUCA (S-NUCA) approach. S-NUCA performs similarly to uniform cache and utilizes the same indexing policy of offset, set, and tag. The sole distinction is that each bank may have differing access latencies. However, S-NUCA does not fully exploit the advantages of NUCA cache, such as data replication, data migration, and near-core storage. Dynamic NUCA (D-NUCA) presents an alternative approach that can fully capitalize on these benefits, but it incurs significant overhead and requires support from the Operating System. The architecture design significantly impacts the LLC structure, and current hardware-based cache partition techniques do not take this into consideration. One of the proposed future ideas is to use Flock in D-NUCA for shared cache banks between different cores.

9.3. CPU-GPU Simulation Infrastructure

The availability and usage of CPU-GPU simulators pose a significant challenge, as there are only a limited number of GPU simulators currently accessible, such as GPGPU-sim. However, integrating these simulators with other CPU simulators can prove to be complex and may result in compatibility issues. Furthermore, GPGPU-sim is designed based on Nvidia desktop GPUs and is not suitable for our use case where the CPU and GPU are on the same die and share the LLC cache [3]. The latest version of gem5 now includes a GPU model, which presents an opportunity to continue research in this area [5]. However, the GPU model in gem5 is not yet mature and may lead to problems during development.

9.4. Cache Coherence

The current state-of-the-art cache replacement policies, namely Hawkeye and Mockingjay, are designed without considering cache coherence. The reimplementation of these policies in gem5 provides a promising approach for incorporating cache coherence, given that gem5 offers advanced cache coherence simulation support. Although Hawkeye and Mockingjay are mainly designed for the LLC and do not consider cache coherence, they need to deal with invalidation requests when utilized in L2 or L1 cache. For instance, when Hawkeye encounters an invalidation request, it may set certain bits in the sampled cache and label the cache line as a shared line. Upon bringing it in again, the cache line can be bypassed or given high eviction priority, thereby reducing the cache coherence overhead.

9.5. Operating System Support

The technique of software-based cache partitioning necessitates the support of the operating system to allocate cache budgets for individual cores. However, this approach lacks the necessary dynamism to adapt to the changing behavior of various applications at run-time. Conversely, Dynamic Non-Uniform Cache Architecture (D-NUCA) requires operating system assistance to assign cache banks to each core and enhance access performance. In this regard, hardware-based cache partitioning can exploit operating system hints to more effectively allocate cache budgets to each core,

with the operating system capable of providing more comprehensive information than real-time cache statistics about the contents of applications.

Chapter 10: Conclusion

In conclusion, this thesis provides a comprehensive overview of the transplantation of cache management policies into new simulators and highlights the modifications necessary to achieve this goal, particularly with respect to the gem5 architecture simulator. Additionally, it presents novel research directions for cache management policies, including their implementation in Non-uniform Cache Architecture (NUCA) and their integration with cache coherence policies. The implementation of these approaches in real-world scenarios poses significant challenges. Nevertheless, we are confident that this work and future research in this area will lead to more robust and effective solutions for managing the Last Level Cache in modern heterogeneous systems.

References

[1] "big.LITTLE Technology: The Future of Mobile." https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf .

[2] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 211-222.

[3] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020: IEEE, pp. 473-486.

[4] N. Gober *et al.*, "The Championship Simulator: Architectural Simulation for Education and Competition," *arXiv preprint arXiv:2210.14324,* 2022.

[5] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters,* vol. 14, no. 1, pp. 34-36, 2014.

[6] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal,* vol. 5, no. 2, pp. 78-101, 1966.

[7] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *Acm Sigmod Record,* vol. 22, no. 2, pp. 297-306, 1993.

[8] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH computer architecture news,* vol. 38, no. 3, pp. 60-71, 2010.

[9] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 430-441.

[10] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010: IEEE, pp. 175-186.

[11] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," *ACM SIGARCH Computer Architecture News,* vol. 44, no. 3, pp. 78-89, 2016.

[12] I. Shah, A. Jain, and C. Lin, "Effective Mimicry of Belady's MIN Policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022: IEEE, pp. 558-572.

[13] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001: IEEE, pp. 197-206.

[14] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006: IEEE, pp. 423-432.

[15] D. Zhan, H. Jiang, and S. C. Seth, "CLU: Co-optimizing locality and utility in thread-aware capacity management for shared last level caches," *IEEE transactions on computers,* vol. 63, no. 7, pp. 1656-1667, 2012.

[16] D. Kaseridis, M. F. Iqbal, and L. K. John, "Cache friendliness-aware managementof shared last-level caches for highperformance multi-core systems," *IEEE transactions on computers,* vol. 63, no. 4, pp. 874-887, 2013.

[17] K. Varadarajan *et al.*, "Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006: IEEE, pp. 433-442.

[18] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "R-NUCA: Data placement in distributed shared caches," *Computer Architecture Lab at Carnegie Mellon Technical Report,* 2009.

[19] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son, "A novel migration-based NUCA design for chip multiprocessors," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008: IEEE, pp. 1-12.

[20] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013: IEEE, pp. 213-224.

[21] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015: IEEE, pp. 538-550.

[22] B. C. Schwedock and N. Beckmann, "Jumanji: The case for dynamic NUCA in the datacenter," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020: IEEE, pp. 665-680.

[23] P. Caheny, L. Alvarez, M. Casas, and M. Moreto, "TD-NUCA: runtime driven management of NUCA caches in task dataflow programming models," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022: IEEE Computer Society, pp. 1153-1167.

[24] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Computing Surveys (CSUR),* vol. 50, no. 2, pp. 1-39, 2017.

[25] S. E. Yuksel, J. N. Wilson, and P. D. Gader, "Twenty years of mixture of experts," *IEEE transactions on neural networks and learning systems,* vol. 23, no. 8, pp. 1177-1193, 2012.

[26] M. Asri, A. Hsu, A. Jain, and C. Lin, "Effective Cache Management for Heterogeneous Multi-Core Systems."

[27] R. E. Matick, T. J. Heller, and M. Ignatowski, "Analytical analysis of finite cache penalty and cycles per instruction of a multiprocessor memory hierarchy using miss rates and queuing theory," *IBM Journal Of Research And Development,* vol. 45, no. 6, pp. 819-842, 2001.

[28] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[29] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news,* vol. 39, no. 2, pp. 1-7, 2011.

[30] "Macsim." https://github.com/gthparch/macsim.

[31] B. R. Bruce *et al.*, "Enabling Reproducible and Agile Full-System Simulation," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021: IEEE, pp. 183-193.

[32] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 84-97.