Diploma Project

GENETIC SEARCH ON RRIP REPLACEMENT POLICIES

Nikolas Papaki

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2023

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

Genetic Search on RRIP Replacement Policies

Nikolas Papaki

Professor
Yiannakis Sazeides

The personal dissertation is submitted in partial fulfillment of requested obligations for receiving the degree of computer science from the department of Computer Science of the University of Cyprus

May 2023

Acknowledgments

At this point I would like to express my special thanks to my Professor, Mr. Yiannakis Sazeides, for giving me this great opportunity to work with him on this topic and for the guidance and support throughout the entire process of my Diploma Thesis.

Abstract

Cache replacement policies play a crucial role in optimizing cache efficiency and performance in modern computer architectures. Traditional policies like LRU and FIFO have limitations in meeting the demands of complex workloads and diverse access patterns. This research investigates the viability of automating the search for specialized cache replacement policies using a genetic algorithm. The objective is to optimize cache hit rates and minimize cache trashing for specific access patterns.

Through a population-based evolutionary algorithm, the genetic search approach explores a vast search space of potential cache replacement policies and evaluates their performance through simulations. The algorithm evolves and refines the policies over multiple generations, converging towards more effective solutions. Specifically, this thesis focuses on the application of genetic search to RRIP (Re-Reference Interval Prediction) replacement policies.

The methodology encompasses a detailed investigation of the genetic search process, including the experimental setup. Simulations are conducted to compare the performance of evolved specialized policies against traditional replacement policies. The results demonstrate the superiority of the evolved policies, achieving an average improvement of 1.38% over LRU and 1.05% over SRRIP in terms of average instructions per cycle (IPC) over 20 Benchmarks. These findings highlight the potential benefits of specializing a replacement policy for different cache access types and provide valuable insights into cache replacement policy optimization.

Overall, this research contributes to the advancement of cache replacement policy optimization. By automating the search for specialized policies, it opens up new possibilities for improving cache hit rates and overall system performance. Future research directions are discussed, emphasizing the need for continued exploration and refinement of specialized cache replacement policies to meet the evolving demands of modern computer architectures.

Contents

Chapter 1

Introduction

1.1 Introduction

Cache replacement policies play a crucial role in the efficiency and performance of cache systems in modern computer architectures. These policies control the selection of cache lines to be evicted when the cache becomes full when a new line is fetched. Traditional policies such as LRU (Least Recently Used) and FIFO (First-In-First-Out), have been widely studied and implemented. However, with the increasing complexity of workloads and diverse access patterns, there is a growing need for specialized cache replacement policies tailored to specific cache access types.

The objective of this research is to investigate the viability of automating the search for a specialized cache replacement policy through the use of a genetic algorithm and potentially yield performance benefits. By employing genetic search algorithms, we aim to discover efficient cache replacement policies that optimize the cache hit rate and minimize cache trashing for specific types of cache accesses.

The motivation behind this research comes from the limitations of existing cache replacement policies. While traditional policies are general-purpose and can provide reasonable performance in many scenarios, they may not be optimal for specific cache access patterns. By automating the search for specialized policies,

we can explore new possibilities and potentially achieve higher cache hit rates and improved overall system performance.

To accomplish our research goal, we will utilize a genetic search approach that makes use of a population-based evolutionary algorithm. This approach allows us to explore a large search space of potential cache replacement policies and evaluates their performance through simulations. The genetic search algorithm will evolve and refine the cache replacement policies over multiple generations, gradually converging towards more effective solutions.

In this thesis, we will present a detailed investigation of the genetic search on RRIP (Re-Reference Interval Prediction) replacement policies. We will outline the methodology, describe the experimental setup, and present the results of our simulations, comparing the performance of evolved specialized policies against traditional replacement policies. Finally, we will discuss insights that derive from our findings, and possible future research in cache replacement policy optimization.

1.2 Outline

This paper consists of seven chapters and is organized as follows. Chapter 1 was the introduction which presents the significance of optimizing cache replacement policies and posing the research question whether automating the search for specialized policies can yield performance benefits. Chapter 2 presents the background knowledge someone will need in order to understand this work. Chapter 3 presents the frameworks and methodology, it describes the benchmark suite used to evaluate candidate policies, analyzes the frameworks and the evaluation methodology used in our research. Chapter 4 describes how we represent a replacement policy as a genome and the interface created to join the genetic algorithm to the microarchitecture simulator. In Chapter 5 we evaluate the solutions from our search and compare them to traditional replacement policies. In Chapter 6 we discuss research related to ours and what differentiates us. Chapter 7 concludes this work and discusses future work.

Chapter 2

Background

2.1 Modern CPUs

Modern central processing units (CPUs) play a crucial role in driving the performance of today's computing systems. These microprocessors have seen great advancements in the past decades to deliver exceptional processing power and efficiency. When a program is executed, the CPU undertakes a series of operations. It starts by fetching instructions from memory, utilizing various levels of cache to minimize data retrieval time and improve overall performance. The CPU then decodes the instructions in order to understand their intended action and proceeds to execute the necessary calculations or operations. Throughout this process, the CPU effectively manages the flow of data, utilizing the cache hierarchy to store frequently accessed data and instructions closer to the CPU cores for faster retrieval. By intelligently utilizing caches, modern CPUs allow for faster program execution, resulting in better performance and enabling computers to efficiently handle a wide range of tasks with speed. In addition to these advancements, modern CPUs also feature simultaneous multithreading (SMT), also known as hyper-threading. SMT enhances the CPU's multitasking capabilities by allowing a single

CPU core to handle multiple threads simultaneously. This technology has an impact on cache utilization as well. With SMT, the CPU core is shared among multiple threads, which means that each thread has access to a portion of the cache resources. The cache is divided and dynamically shared between the threads, ensuring that frequently accessed data and instructions are still stored closer to the CPU cores for faster retrieval. However, the cache resources are distributed among the threads, and if multiple threads compete for the same cache space, it can potentially lead to cache thrashing or increased cache misses, which may impact performance. By intelligently utilizing caches, modern CPUs optimize program execution speed. This results in enhanced overall performance, enabling computers to handle a wide range of tasks more efficiently. The combination of fast instruction fetching, decoding, and execution, along with cache utilization, allows modern CPUs to deliver swift and seamless computing experiences.

2.2 Multilevel Cache Hierarchy

Processors nowadays typically have three levels of cache L1, L2, and L3 (LLC) with each level providing different performance characteristics and serving a specific purpose in the memory hierarchy. L1 cache is the smallest and fastest cache of them all, located closest to the processor. It is usually split into separate instructions and data caches and is designed to provide very fast access to frequently used instructions and data. L2 cache is larger than L1 , which makes it slower than L1. It is designed to provide a large cache for frequently accessed data and instructions. Its larger size allows it to store more data than L1, which reduces the number of cache misses and improves overall performance. L3 cache is even larger than L2 and shared across all the cores on a processor. It is slower than L2 but still faster than main memory and is designed to provide a large cache for shared data between cores. This can reduce the amount of traffic to main memory and improve overall performance. By using different sizes of caches, we can optimize performance by keeping frequently accessed data and instructions close to the processor while also reducing the number of cache misses and main memory

accesses. This can significantly improve overall performance and reduce power consumption. Figure 2.1 shows a 1-core processor with private IL1, DL1 and L2 caches and a shared L3 cache. Each individual cache in the figure has its own stream buffer used to prefetch data from lower levels of the cache hierarchy and a miss status holding registers (MSHR) used to track outstanding requests that had a miss in the cache.

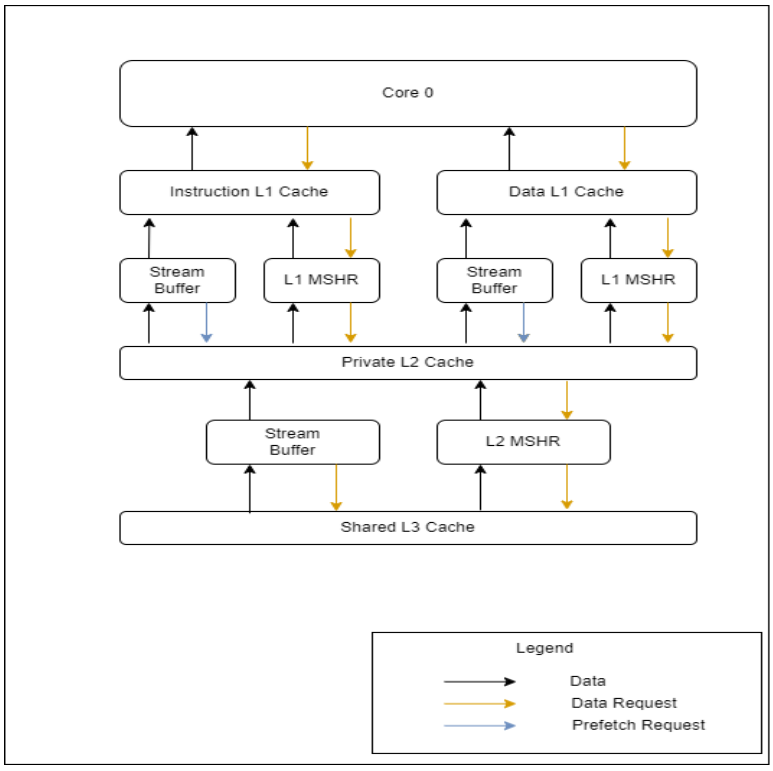

Figure 2.1 Single Core Multilevel Cache

2.3 Replacement Policies - SRRIP

With programs nowadays requiring large amounts of data, caches are essential components in computer systems. Caches are small, fast memory systems that store frequently accessed data for quick access by the CPU. However, caches have limited size and capacity, and as a result they can become full quickly. To make room for new data, it is necessary to replace some of the existing data in the cache, this process is described as a replacement policy. Choosing the right replacement policy for a cache is critical because it can significantly impact the caches and the

overall system's performance. The primary objective of a replacement policy is to maximize the cache hit rate, which is the percentage of memory accesses that are satisfied by the cache. There are various replacement policies currently available, such as LRU (Least Recently Used), PLRU (Pseudo Least Recently Used), LFU (Least Frequently Used), MRU(Most Recently Uses), RRIP (Re-reference Interval Prediction) which have their own advantages and disadvantages. The best choice of policy will depend on the specific requirements of the system and the workload being executed. Moreover, when choosing a cache replacement policy, we should also note that the properties of the cache (write-through, write-back, write-allocate, write-no-allocate) also have an impact on what is the best choice of policy, as these characteristics change the behavior of the cache and in turn the optimal replacement policy for it.

In this paper we will use the Re-reference interval prediction policy (RRIP) and most specifically SRRIP which is a cache replacement policy that uses a history-based approach to determine the usefulness of a cache block. Traditional replacement policies such as LRU and FIFO focus on the recency or frequency of a block accesses, but SRRIP goes a step further by tracking the re-reference intervals of each block. The re-reference interval refers to the time between two consecutive references to a block, and it provides valuable information about the temporal locality of the block. SRRIP divides the re-reference intervals into different classes, where each class corresponds to a different level of usefulness. The class of a block is determined by its re-reference interval value and ranges from 0 to N, with 0 being the most recently referenced and N being the least recently referenced. Whenever a block is accessed, its value is reset to its minimum value (highest class), indicating that the block is recently used. As time progresses and the block remains untouched, the counter gradually increases through the process of finding which block to evict. An eviction occurs when the cache is full and a new block needs to be inserted, the block with the maximum value is evicted but if there not blocks currently matching these criteria then the value of all blocks in the set is increased until a block matches the eviction criteria. Figure 2.2 shows how LRU, NRU and SRRIP behave on the same access pattern. This approach has

proven effective in adapting to changing access patterns and making informed eviction decisions. In addition, SRRIP is relatively easy to implement and has low overhead compared to other replacement policies, e.g. In a 16way cache that implements LRU we need 4 bits per way to encode the position in the LRU stack, on the other hand using 2-bit RRIP requires only 2 bits per way to encode the 4 different classes. RRIP has shown promising results in reducing cache misses and improving system performance, particularly in workloads with irregular access patterns. However, it may not perform as well as LRU in workloads with high temporal locality, as LRU is better suited to exploit this type of access pattern.

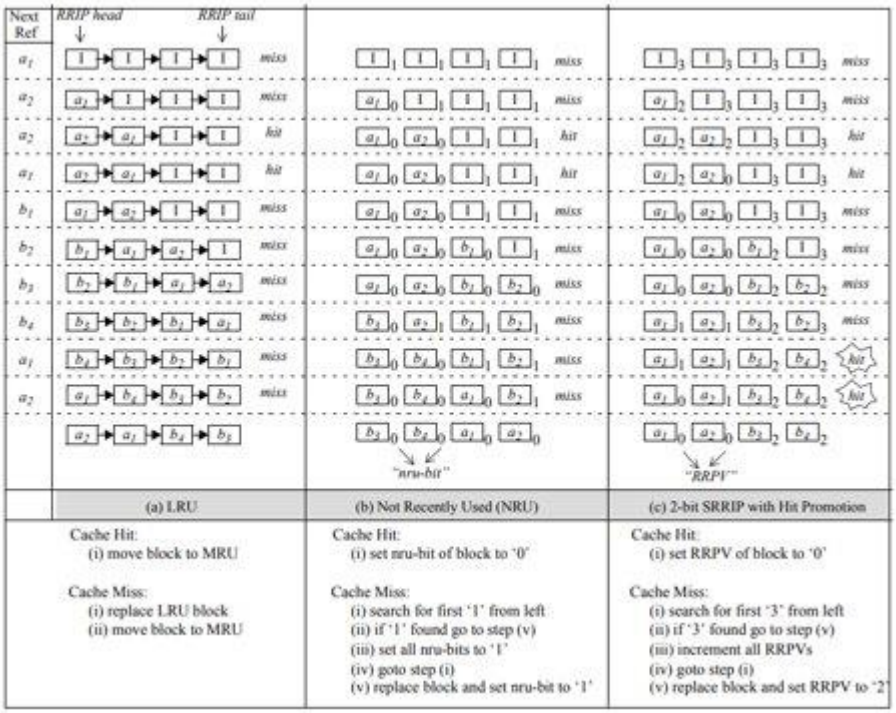

Figure 2.2 Example of SRRIP and comparison with LRU and NRU

2.4 Genetic Algorithms

Genetic Algorithms are a heuristic based approach inspired by the process of natural selection. They work by repeatedly generating new solutions through the combination and mutation of existing solutions, then evaluating their fitness before selecting the most promising ones for the next iteration. We can see the flow of a

genetic algorithm in Figure 2.3. The use of a genetic algorithm can be beneficial in narrowing down the search space. The algorithm can generate and evaluate a large number of candidate solutions efficiently, making it well-suited for problems with a large search space such as ours. Moreover, genetic algorithms are capable of exploring diverse regions of the solution space, increasing the likelihood of finding an optimal solution. This is because it uses a stochastic approach to solution generation, incorporating randomness in the selection and mutation of candidate solutions. The algorithm can identify and retain the most promising candidate solutions while discarding the poor ones, helping to narrow down the search space. In order to evaluate each candidate solution, we calculate the fitness value of each one based on the average IPC scored in 20 different benchmarks. In this paper we will be using GeST, a framework created for automatic generation of stress tests that was modified to work for the purpose of our research.

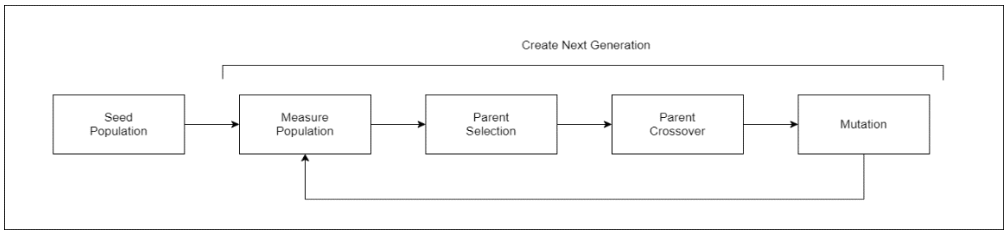

Figure 2.3 The flow of a genetic algorithm

2.5 Microarchitecture Simulators

Microarchitecture simulators are software tools that replicate the behavior and functionality of a microarchitecture, allowing researchers to analyze, test, and optimize hardware designs without the need for physical implementations. They over several advantages over using real hardware for development and testing purposes. Firstly, simulators possess a modular nature, allowing users to modify specific components or simulate different configurations without physical modifications. Furthermore, hardware experiments may be subject to variability due to environmental factors or hardware degradation. In contrast, using a simulator can be an effective alternative as it can accurately capture the memory access patterns and other characteristics of the workload, making it a reliable tool. Moreover, for the purpose of our research a microarchitecture simulator can enable

faster experimentation and evaluation of a large number of candidate policies in a controlled and repeatable environment. One of the most important benefits of simulations is that we can change any parameter we want without any cost. For example, we can use a single core or multicore CPU, we can change the sizes of the caches, and most importantly their replacement policies.

Chapter 3

Frameworks and Methodology

3.1 Evaluation Methodology.

In this subsection, we describe the evaluation methodology employed in this paper aimed to investigate the practicality of automating the search for specialized cache replacement policies tailored to specific cache access types, with the objective to achieve higher cache hit rates and improved overall performance. To accomplish this, a genetic search approach (described in section 3.4) was utilized to generate candidate policies. These candidate policies were evaluated through simulations using a set of 20 benchmarks (described in section 3.2) that represent a range of workload characteristics and cache access patterns. The genetic algorithm employed in this study follows a systematic process to generate and evaluate candidate cache replacement policies. Initially, a set of candidate policies is randomly created. Through the interface the candidate policies of the entire generation are subjected to simulations, with each policy evaluated across a set of benchmarks. The simulations capture the cache performance under various workload scenarios. From the output of each simulation, we can extract metrics that can help us identify the best candidate replacement policies. For the purpose of our research, we decided to evaluate each candidate's replacement policy based

on the Instructions Per Cycle (IPC) value it achieved. Once the simulations for the generation are completed, the interface calculates the average IPC achieved by each candidate policy, that serves as the quantitative measure of their performance. Based on the principles of evolution, a new generation of candidate policies is created through the selection, combination, and mutation of the most promising policies from the previous generation. This iterative process continues until the desired number of generations is achieved, allowing the genetic algorithm to refine and improve the cache replacement policies over time. By employing this evolutionary approach, the algorithm explores and converges towards more effective cache replacement policies tailored to specific cache access types. Figure 3.1 illustrates how the genetic algorithm works in conjunction with the microarchitecture simulator through the interface.

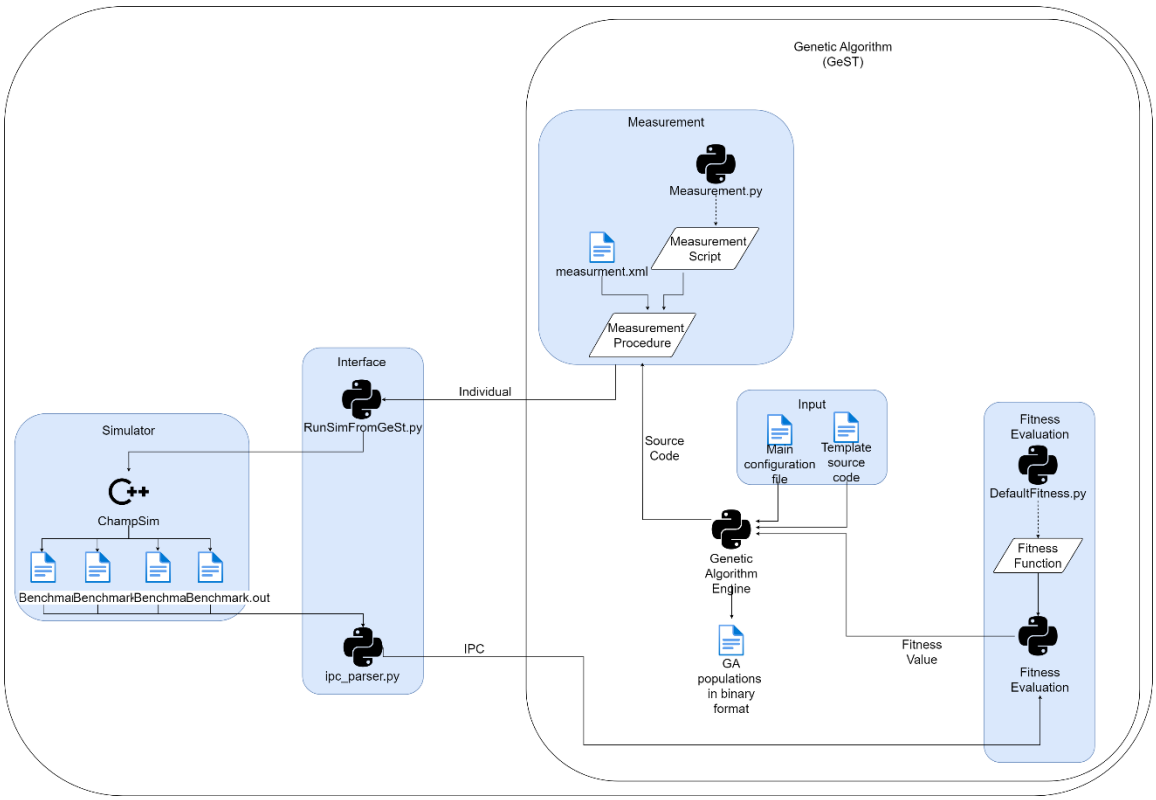

Figure 3.1 Integration of Genetic Algorithm and Simulator

3.2 SPEC CPU 2017 Benchmark Suite

In order to evaluate our candidate policies, we used the traces created from the Benchmarks included in the SPEC 2017 suite, which were designed to stress the system processor and memory subsystem, which is great for evaluating cache replacement policies.

For our study we decided to use 20 traces created from the benchmarks included in the SPEC 2017 suite, shown in Table 3.1.

Benchmarks	
Blender	Bwaves
cactusBSSN	Cam4
Exchange	Fotonik3d
Gcc	Imagick
Lbm	Leela
Mcf	Omnetpp
Parest	Perlbench
Povray	Roms
Wrf	X264
Xalancbmk	Xz

Table 3.1 SPEC CPU 2017 Benchmarks used

3.3 ChampSim

3.3.1 ChamSim Overview

For the purpose of our research, we used ChampSim [4] which is a trace-based simulator for microarchitecture study that was used in various contests, such as the 3rd Data Prefetching Championship and the 2nd Cache Replacement Championship. The simulator is designed to model and evaluate the behavior and performance of modern out-of-order CPUs with a three-level cache hierarchy and uncore memory. This enables us to assess the impact of a candidate replacement policy on the performance of the system. Through its output it provides us with valuable insights and information about the behavior and performance of the simulated system, some of the metrics the simulator generates are instruction per cycle (IPC), execution time, cache hit/miss rates.

3.3.2 ChampSim Configuration

For our experiments we modeled a 1-core out-of-order processor with 3 levels of cache. The configuration used is described in Table 4.3. For the LLC replacement policy, we used a modular version of DRRIP replacement policy.

Parameter	Configuration
L1 I-Cache (Private)	32KB, 64B blocks, 8-way 8 MSHRs, 1 cycle latency LRU Replacement Policy

L1 D-Cache (Private)	32KB, 64B blocks, 8-way 8 MSHRs. 4 cycles latency LRU Replacement Policy Next-Line Prefetcher
L2 Cache (Private)	256KB, 64B Blocks, 8-way 16 MSHRs, 8 cycles latency LRU Replacement Policy IP-Based Stride Prefetcher
L3 Cache (Shared)	2MB per core, 64B Blocks, 16-way 32 MSHRs, 20 cycles latency Modular DRRIP-Based Replacement Policy
Branch Predictor	Perceptron

Table 3.2 ChampSim Simulator Configuration

3.3.3 Modular DRRIP-based Replacement Policy

In order to effectively run our simulation, we need to create a replacement policy that is modular and can change its behavior based on the arguments given to the simulator. The policy is case driven with each case implementing one function (described in Chapter 4). We decided to use a DRRIP-based policy that we will modify in order to achieve the preferred function. DRRIP is a replacement that just like SRRIP predicts the intervals at which cache lines will be accessed. The difference is that it dynamically selects between 2 competing policies using set dueling. Set dueling works by dedicating a few sets of the

cache to each of the competing policies, with 32-64 dedicated sets to be sufficient to choose the best policy. The remaining sets referred to as “follower sets” use the policy that performs better. In order to determine which policy performs better and select it as the primary policy used by the follower sets, Set Dueling uses a Policy Selector counter referred to ”PSEL”. PSEL is a saturating counter that keeps track of which of the two competing policies incurs fewer misses, one policy adds to the counter and the other subtracts if a miss occurs in the sets dedicated to it. The most significant bit of the counter determines the policy that performs better. Figure 3.2 illustrates how Set Dueling will work with LRU and BIP as competing policies.

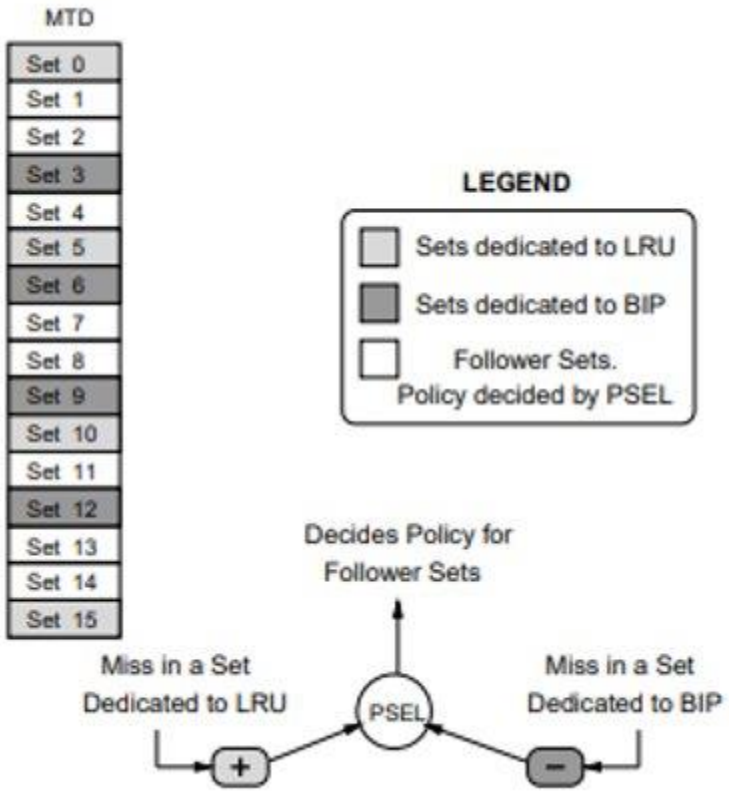

Figure 3.2 Set Dueling between LRU and BIP

3.4 GeST

For our research we used GeST [5] , an automatic framework for generating CPU stress tests. It uses a genetic algorithm, a technique inspired by biological evolution, to create tests that maximize the load on a CPU. The framework operates by evolving a population of test cases through generations. Initially, a set of test cases (individuals), each representing a potential stress test, is randomly generated. The second step involves measuring the metrics of interest and assigning a fitness value to the individual. In the third and final step the algorithm creates a new population by selecting the fittest individuals as parents for the individuals of the next generation, the process includes the exchange of instructions between the two parents (crossover) and performing a mutations operation. A mutation converts an instruction or an operand into another based on a probability referred to as “mutation rate”. The potential stress test created consists of a loop of assembly instructions that continuously executes on the CPU. Within this loop, a series of computationally intensive instructions, such as mathematical calculations or data manipulations, are performed repeatedly. By subjecting the CPU to this intense loop of instructions, GeST aims to identify potential weaknesses, bottlenecks, or instabilities in the processor's performance. With GeST users have the ability to tweak parameters through the main configuration file. With this level of modularity, users can customize various parameters to fine-tune the generated stress tests. They can specify the size and complexity of the instruction loop, select specific instruction types, set the mutation rate, and change the fitness function. By tweaking these parameters, users can focus on specific aspects of the CPU's behavior, target particular architectural features, or emulate real-world workloads that closely resemble their specific use cases. As mentioned, GeST allows users to modify the fitness function through the creation of a new file. In GeST, the fitness function is a crucial component that determines the effectiveness of a stress test in measuring their desired metric. By creating a separate file dedicated to the fitness function, users can customize and define their own criteria for evaluating the quality and effectiveness of the stress tests. This modularity empowers users to adapt GeST to their specific needs and research objectives. They can introduce new

metrics, algorithms, or heuristics to assess the CPU's performance based on their unique requirements. Figure 3.3 illustrates an overview of the framework.

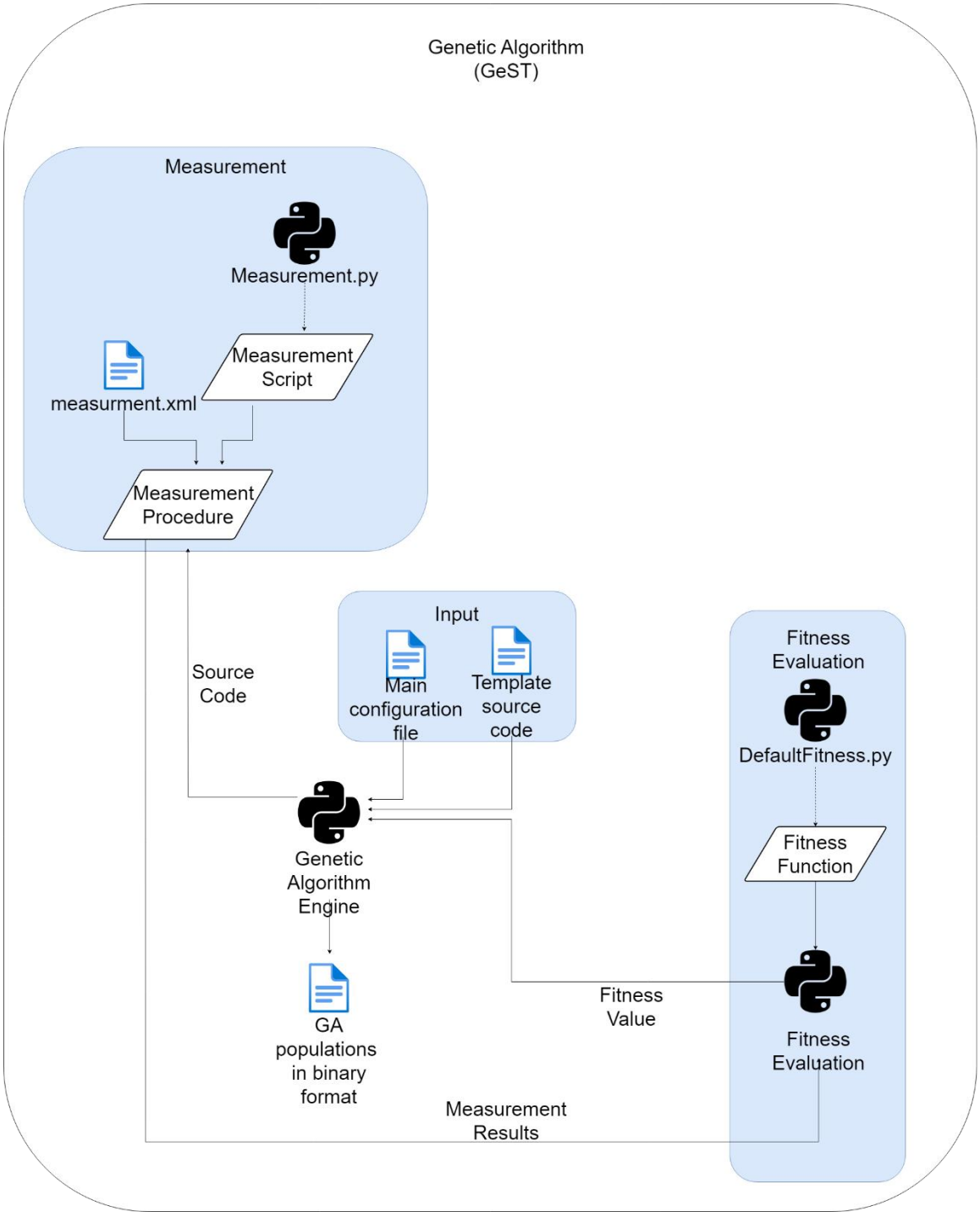

Figure 3.3 Framework Overview

Chapter 4

Representing a cache replacement policy as a genome

4.1 Policy Definition

The first step we needed to overcome was to create an encoding that would allow us to represent a cache replacement policy as a genome. We propose an encoding that consists of multiple values, each value represents a different function for the different aspects of the policy. The genetic algorithm will produce a genome containing these values and those same values will be given as arguments to the simulator. As mentioned in Chapter 3, the replacement policy used in the simulator is case driven, with each case implementing the function it represents. We identified five aspects, which are independent from each other, that need to be encoded (Shown in Table 4.1). In addition, each aspect of the policy if applicable has two states clean / dirty.

Aspect	Definition
Demotion	On a miss what happens to the value of all blocks in the set
Eviction	On a miss which block will be evicted
Insertion	On a miss what value will the newly installed block have
Self – Promotion	On a hit what is the new value of that block
Other-Promotion	On a hit what happens to the value of all other blocks

Table 4.1 Policy aspects that need encoding.

4.1.1 Proposed functions for Demotion

For Demotion we propose four functions:

The first proposed function is Repeating Saturating Increment by Y with ceiling X in which for a given set we increment the RRPV value of all blocks in the set by Y until a block in the set reaches the ceiling set by X.

The second proposed function is Conditional Repeating Saturating Increment by Y with ceiling X if block under Z in which for a given set increment the RRPV value of blocks that currently have an RRPV value smaller than Z by Y until a block in the set reaches the ceiling set by X.

The third proposed function is Conditional Repeating Saturating Increment by Y with ceiling X if block over Z in which for a given set increment the RRPV value of blocks that currently have an RRPV value larger than Z by Y until a block in the set reaches the ceiling set by X.

4.1.2 Proposed functions for Eviction

For Eviction we propose four functions:

The first proposed function is First Block with RRPV of X in which when looking from left to right (Way 0 – Way N) in the set, evict the first block you encounter that has an RRPV value of X.

The second proposed function is Probabilistic First Block with RRPV X or Y based on Z in which when looking from left to right (Way 0 – Way N) in a set, evict the first block you encounter that has an RRPV value of X or Y. For each eviction we choose X or Y based on the probability Z.

The third proposed function is First block with RRPV of X unless a dirty block with RRPV of Y was found before. In this function, while looking from left to right

we evict the first block that has an RRPV value of X but if we find a dirty block with an RRPV value of Y before we evict that.
The last proposed function is First block with RRPV of X unless dirty block before. This function is very similar to the third one but with the difference that it does not take into consideration the RRPV value of the dirty block.

4.1.3 Proposed functions for Insertion

For Insertion we propose two functions :
The first proposed function is Assignment to X, in which when inserting a new block in the set, add the new block with an RRPV value of X.
The second proposed function is Probabilistic Assignment to X or Y based on Z. In this function, when inserting a new block add the new block with an RRPV value of either X or Y based on the probability Z.

4.1.4 Proposed functions for Self-Promotion

For Self-Promotion we propose two functions:
The first proposed function is Saturating Decrement by Y with floor X, in which for a given set decrement the RRPV value of all blocks in the set by Y until a block reaches the floor set by X.
The second proposed function is Probabilistic Saturating Decrement by X or Y with floor R based on Z. In this function, for a given set decrement the RRPV by X or Y until a block reaches the floor set by R based on the probability Z.

4.1.5 Proposed functions for Other-Promotion

For Other-Promotion we propose three functions:

The first proposed function is Do Nothing. In this function, when another block was accessed and promoted don't do anything to the rest of the blocks in the set.
The second proposed function is Saturating Increment by Y with ceiling X. In this function, when another block was accessed and promoted, we increment the RRPV value of all blocks in the set by Y until a block in the set reaches the ceiling set by X.
The third proposed function is Saturating Decrement by Y with floor X. In this function, when another block was accessed and promoted, we decrease the RRPV value of all blocks in the set by Y until a block in the set reaches the floor set by X.

4.2 Representation of a cache replacement policy as a GeST individual

4.2.1 How we represented a policy as a GeST individual

As highlighted in Chapter 3, GeST is an exceptionally modular framework that allows us to harness its capabilities for our intended purpose. We will achieve that by tailoring its parameters such as the fitness function and selecting specific instruction types, to align with the specific requirements of our research. For our research we mapped the instruction types that can be defined in the configuration of GeST to represent the four access types that we will explore (Shown in Figure 4.1). We then mapped the instructions to represent three aspects of the proposed five with Demotion and Self-Promotion having a clean and dirty version (Shown in Figure 4.2). Finally, we mapped the operands of the instructions to represent the functions that each aspect supports (Shown in Figure 4.3).

```
</general_inputs>

<instruction_types>

  <instruction_type
    id="LOAD"
    perc="0.25"
  />

  <instruction_type
    id="RFO"
    perc="0.25"
  />

  <instruction_type
    id="PREFETCH"
    perc="0.25"
  />

  <instruction_type
    id="WRITEBACK"
    perc="0.25"
  />

</instruction_types>
```

Figure 4.1 How the policy access types are represented as GeST instruction types

```
<instructions>
  <!--  LOAD   -->

    <instruction
      name="Demotion_Clean"
      num_of_operands="1"
      type="LOAD"
      operand1="Dem"
      format="L_DemClean op1"
      toggle="False">
    </instruction>

    <instruction
      name="Demotion_Dirty"
      num_of_operands="1"
      type="LOAD"
      operand1="Dem"
      format="L_DemDirty op1"
      toggle="False">
    </instruction>

    <instruction
      name="Insertion"
      num_of_operands="1"
      type="LOAD"
      operand1="Inse"
      format="L_Insert op1"
      toggle="False">
    </instruction>

    <instruction
      name="SelfPromotion_Clean"
      num_of_operands="1"
      type="LOAD"
      operand1="SelfProm"
      format="L_SPromClean op1"
      toggle="False">
    </instruction>

    <instruction
      name="SelfPromotion_Dirty"
      num_of_operands="1"
      type="LOAD"
      operand1="SelfProm"
      format="L_SPromDirty op1"
      toggle="False">
    </instruction>
```

Figure 4.2 How the policy aspects are represented as GeST instructions

```
<instructions_operands>
  <operand
 id="Dem"
 values="%0 %1 %2 %3 %4"
 type="register"
 toggle="False">
  </operand>

  <operand
 id="Inse"
 values="%0 %1 %2 %3"
 type="register"
 toggle="False">
  </operand>

  <operand
 id="SelfProm"
 values="%0 %1 %2"
 type="register"
 toggle="False">
  </operand>

</instructions_operands>

<instructions>
```

Figure4.3 How the functions are represented as GeST operands

4.2.2 Functions explored in this research

Although numerous functions for each of the aspects that comprise a replacement policy were proposed, for the purpose of the research we only explored a subset of them.

We chose to explore only four of the five aspects that comprise a replacement policy, these include Demotion, Eviction, Insertion and Self-Promotion. In addition, we implemented a subset of the available functions for each aspect. In order to simplify the interface, and simulator the encoding assigns each function a number. Tables 4.2 – 4.6 depict the assignment of each function for all aspects of the policy.

Starting With Demotion:

We used five functions for this aspect of the policy, four of which originate from an unpublished paper by Mr. Sazeides.

The first function is Full Demotion, as the name suggests the RRPV value of all blocks in the set increases by until we evict a block.

The second function is Asymmetric Demotion 1, in this function we increase the RRPV value of blocks in the set by 1 until the maximum RRPV value in the set is greater than 1.

The third function is Asymmetric Demotion 2, in this function we increase the RRPV value of blocks in the set by one until the maximum RRPV value in the set is greater than 2 but only for blocks that have an RRPV value less than 2.

The fourth function is Asymmetric Demotion 3, in this function we increase the RRPV value of blocks in the set by one until the maximum RRPV value in the set is greater than 2 but only for blocks that have an RRPV value less than 2 but also if the maximum RRPV value in the set is 0 then all blocks get an RRPV value of 2.

The fifth and final function is Asymmetric Demotion 4, in this function if the maximum RRPV value in the set is 0 then all blocks get an RRPV value of 2 but

also if the maximum RRPV value is equal to 1 then we increase all blocks RRPV by 1.

Number	Function
0	Full Demotion
1	Asymmetric Demotion 1
2	Asymmetric Demotion 2
3	Asymmetric Demotion 3
4	Asymmetric Demotion 4

Moving on to Insertion:

We used 1 function for this aspect of the policy but with different arguments. The function used is Assign to X, in which new blocks get assigned an RRPV value based on X. Because we used a 2-bit SRRIP the possible values for X are 0-3.

Number	Function
0	Assign to 0
1	Assign to 1
2	Assign to 2
3	Assign to 3

Finally on to Promotion:

For Promotion, we yet again used one function similar to Insertion. The function used is Assign to X, but this time X has possible values from 0-2

Number	Function
0	Assign to 0
1	Assign to 1
2	Assign to 2

4.3 Definition of interface between GeST and ChampSim

4.3.1 Interface Definition

This interface is used to convert the individuals that are generated by the genetic algorithm (GeST) to the format required by the microarchitecture simulator (ChampSim). Each individual encodes a replacement policy for the Last Level Cache (LLC) that the simulator will test by simulating multiple benchmarks. In addition, the interface at the end of all simulations will extract the achieved IPC of all benchmarks and will calculate their average. The average IPC will be given back to the genetic algorithm as the fitness value for the individual. In order to simplify the code, we split the interface into two parts, the first part converts the individuals generated by the genetic algorithm to the required format for the simulator and starts the simulations while the second part extracts the achieved IPC of all benchmarks for a specific individual, calculates the average IPC and returns it to the genetic algorithm.

4.3.2 Input and Output formats

The interface gets its input by opening and reading a file that has the format "generation_id.txt". Because we run the simulations for all individuals in a generation simultaneity the interface gets as argument the generation number, while the ability to run a single individual is possible by giving as arguments the generation and id of the specific individual. The interface then creates a batch file that will start the simulations on the cluster. In each file we are only interested in the part of the code that includes the genome (shown in Figure 4.1). The genome consists of 20 genes, placed in the same number of lines. As you can see in Figure 4.1 there are 4 distinct types that correspond to the access types we want to explore in this research. We differentiate the types by the first character in each line, L is

for LOAD, R for RFO, P for PREFETCH and W for WRITEBACK. Each type also has 6 subtypes corresponding to the 4 aspects with Demotion and Self-Promotion having different values for clean and dirty blocks. Moving on to the output format after the interface reads the input in the format described above it will then produce an output that can be given as the input to the microarchitecture simulator (ChampSim). The simulator takes as input three 16digit numbers named plist, dirty_plist and demmask. So, the interface must create these 16digit numbers from the values it read from the input file and then give them as arguments to the simulator. It also gives as argument a unique identifier that consists of the generation and id of the GeST individual.

The plist defines what happens on a promotion and insertion for clean blocks for the four types of cache accesses (LOAD, RFO, PREFETCH, WRITEBACK). The dirty_plist defines what happens on a promotion and insertion for dirty blocks for the four different types of cache accesses. And finally, the demmask defines what happens on demotion of clean and dirty blocks for the four types of cache accesses.

```
.L2:

        L_DemClean %1
        L_DemDirty %4
        L_Insert %3
        L_SPromClean %1
        L_SPromDirty %2
        R_DemClean %3
        R_DemDirty %4
        R_Insert %1
        R_SPromClean %1
        R_SPromDirty %2
        P_DemClean %2
        P_DemDirty %0
        P_Insert %0
        P_SPromClean %0
        P_SPromDirty %2
        W_DemClean %3
        W_DemDirty %3
        W_Insert %1
        W_SPromClean %0
        W_SPromDirty %1
```

Figure 4.1 How the genome is represented as a GeST individual

4.2.3 Processing of Data

The interface, to easily create the desired output format, uses dictionaries to store the values for each gene. The code (Shown in Figure 4.3) first searches the file to find the section of interest that is always located after the ". L2" line. After the section of interest, the algorithm reads each line, the key is always the first word of the line e.g., "L_DemDirty" and the value is the number after the % symbol. After storing all the key pair values in a dictionary, the interface creates the plist, dirty_plist and demmask values. Depending on the arguments given the interface creates a bash file containing the commands that are used to run the simulator with the values calculated given as arguments. After that the interface runs the bash script to start all simulations for all benchmarks.

```
for line in file:
    # Section of interest started
    if '.L2:' in line:
        section_found = True

    # Section of interest ended
    if 'OPromPrefetch' in line:
        break

    # Get the number from each line
    if section_found:
        if line.startswith('\t'):
            # extract number at the end of the line
            text = line.strip().split()[-1]
            key = line.split()[0]
            number = re.search(r"\d+", text).group()
            values[key] = number
```

Figure 4.3 Code that extracts the values for each policy and stores them in a dictionary

Chapter 5

Experimental Evaluation

5.1 Experiments Specifications

In this subsection, we describe how we did our simulations on ChampSim. For each candidate replacement policy, we run a set of benchmarks, with each benchmark running for one hundred million warm-up instructions and five hundred million instructions on the simulator. The experiments are divided into two groups, in the first group we run simulations with the full benchmark suite while the second group consists of a subset of benchmarks as we discovered some of them are not affected by the change in replacement policy.

5.2 Simulations using the full benchmark suite

5.2.1 Simulations with random first Generation

Our first experiment consisted of fifty individuals that were randomly generated and evolved for fifty generations. As mentioned in this search we used the full benchmark suit in order to evaluate the candidate's replacement value. Figure 5.1 illustrates the average IPC for each individual from generation 1 to generation fifty. The goal of this experiment was to see if automating the search for candidate replacement policies with the use of genetic algorithm will yield performance benefits. So, based on the figure below we can see that the average IPC appears to increase through the generations but hits the ceiling of 0.788 around the eighth generation.

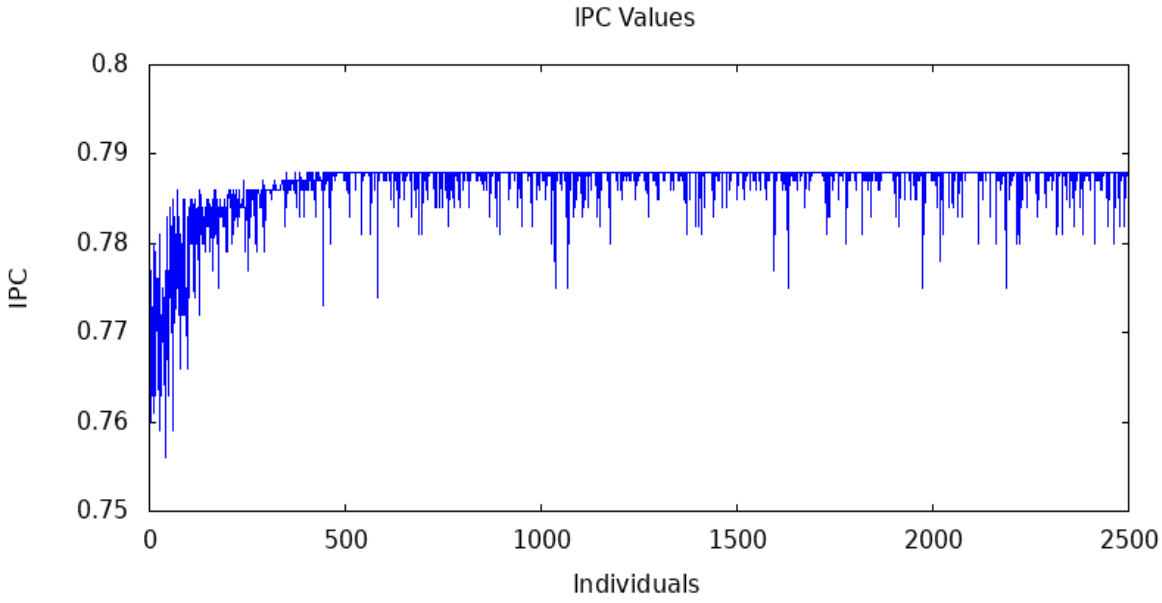

Figure 5.1 Average IPC for each individual from generation 1 to generation 50

5.2.2 Simulations with some promising individuals in the first Generation

We then proceeded to our second experiment that also consisted of fifty individuals from which two were explicitly added and include SRRIP (Table 5.5) and the best assignment of functions from previous work (Table 5.1), with the rest of the individuals being randomly generated. The second run also evolved individuals for fifty generation and used the full benchmark suit in order to evaluate the candidate replacement policy. Figure 5.2 illustrates the average IPC for each individual from generation one to generation fifty. The goal of this analysis is to see if the two explicitly added policies from which one was the most promising from pervious work will help achieve a higher average IPC than our first experiment. The idea behind this run was that adding two promising policies in the first generation will improve the performance of upcoming generations because they have strong "genes". As we can see from Figure 5.2 our suspicion that the promising policies added in the first generation did not work as expected. In addition, we should also point out that this time we reached the ceiling of 0.789 in the sixth generation in contrast to the first run that took eight generations. Finally, we observe that we did gain a small improvement of 0.001 in the average IPC.

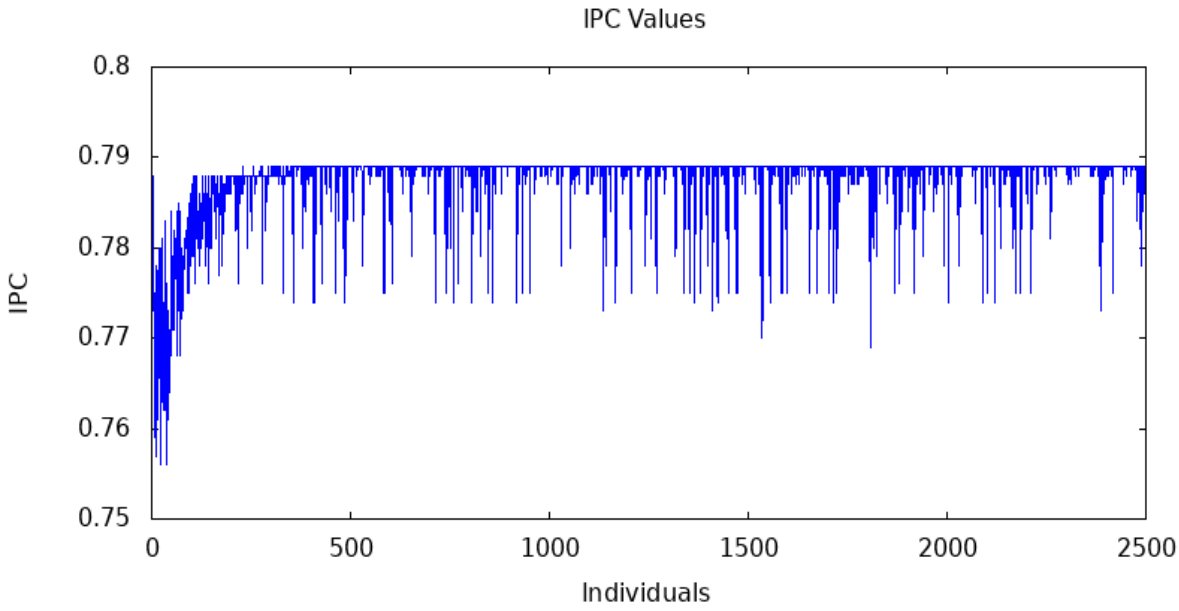

Figure 5.2 Average IPC for each individual from generation 1 to generation 50

L_DemClean	4
L_DemDirty	4
L_Insert	2
L_SPromClean	0
L_SPromDirty	0
R_DemClean	0
R_DemDirty	4
R_Insert	2
R_SPromClean	0
R_SPromDirty	0
P_DemClean	4
P_DemDirty	4
P_Insert	3
P_SPromClean	0
P_SPromDirty	0
W_DemClean	0
W_DemDirty	4
W_Insert	3
W_SPromClean	0
W_SPromDirty	0

Table 5.1 Best policy from previous research

Benchmark	IPC
Blender	0.670
cactuBSSN	0.755
Cam4	0.731
Fotonik3d	0.620
Gcc	0.353
Imagick	2.191
Lbm	0.697
Mcf	0.401
Omnetpp	0.247
Parest	1.024
Perlbench	0.447
Roms	1.046
Wrf	0.825
x264	1.365
Xalancbmk	0.433
Xz	0.899

Table 5.2 IPCs the best policy from previous research achieved

5.3 Simulations with reduced benchmark suite

From the results of the previous experiments, we discovered that some of the benchmarks used to evaluate the candidate replacement policies were not affected by the change in replacement policy. The benchmarks that were removed from the benchmark suite include Bwaves, Exchange, Leela and Povray. By observing the IPC that all candidate policies achieved in these benchmarks we observed that either the value is constant or has minor fluctuations. We then rerun our experiments but removed these benchmarks from the benchmark suite, in order to identify if they are the reason that we did not see much improvement in the IPC from our candidate policies. Figure 5.3 illustrates the IPC achieved in the Bwaves benchmark from all candidate individuals in our first experiment.

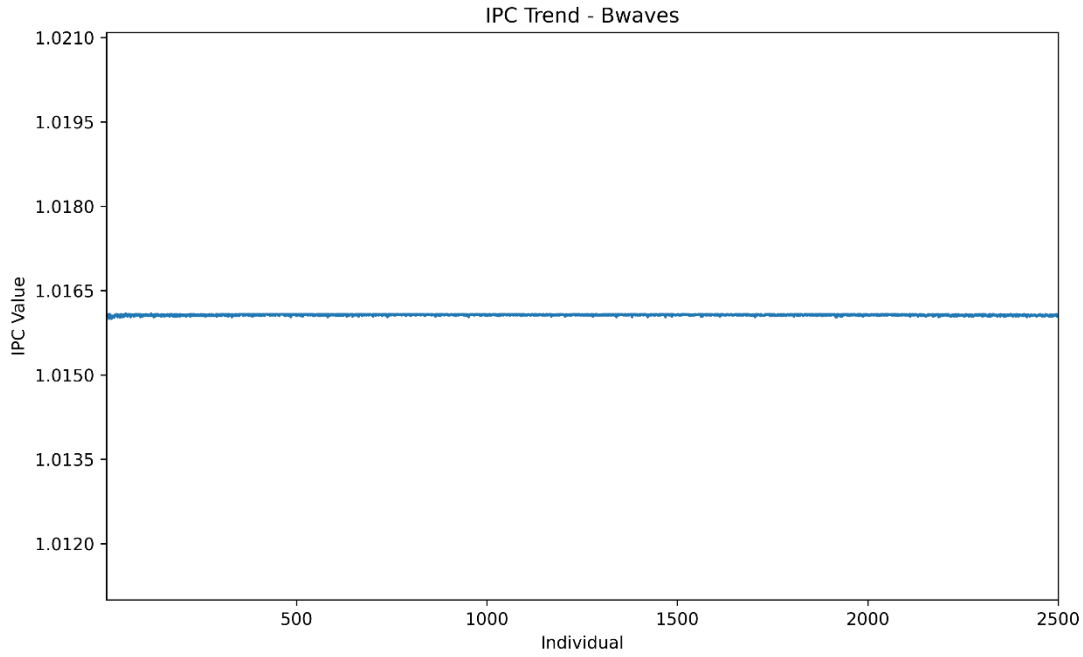

Figure 5.3 IPC achieved from all candidate policies in the Bwaves benchmark

5.3.1 Simulations with random

Moving on to the third experiment, it consisted of fifty individuals that were randomly generated and evolved for fifty generations. The goal of this experiment was to validate if the unaffected benchmarks influence the average IPC negatively and if that was the reason, we did not see much improvement in our previous experiments. Figure 5.4 illustrates the average IPC for each individual from generation 1 to generation 50. At a first glance removing those benchmarks seems to help improve the performance as the average IPC of our best candidate policy is 0.796, but after comparing the IPC achieved in each benchmark we came to the conclusion that the removal of the unaffected benchmark did not help us achieve higher performance. Table 5.4 shows the IPC each policy achieved in the benchmarks excluding the benchmarks that are not affected by the change in replacement policy.

Let's now inspect the best policy from the third experiment. Table 5.3 illustrates the values the best policy consists of. As we can see Load and Writeback blocks are given more chance to "prove themselves" as they are inserted with an RRPV of 2 while RFO and Prefetch blocks are inserted with an RRPV of 3. In addition, we can see that all dirty blocks never get promoted to an RRPV of 0 which is also true for clean RFO and Writeback blocks.

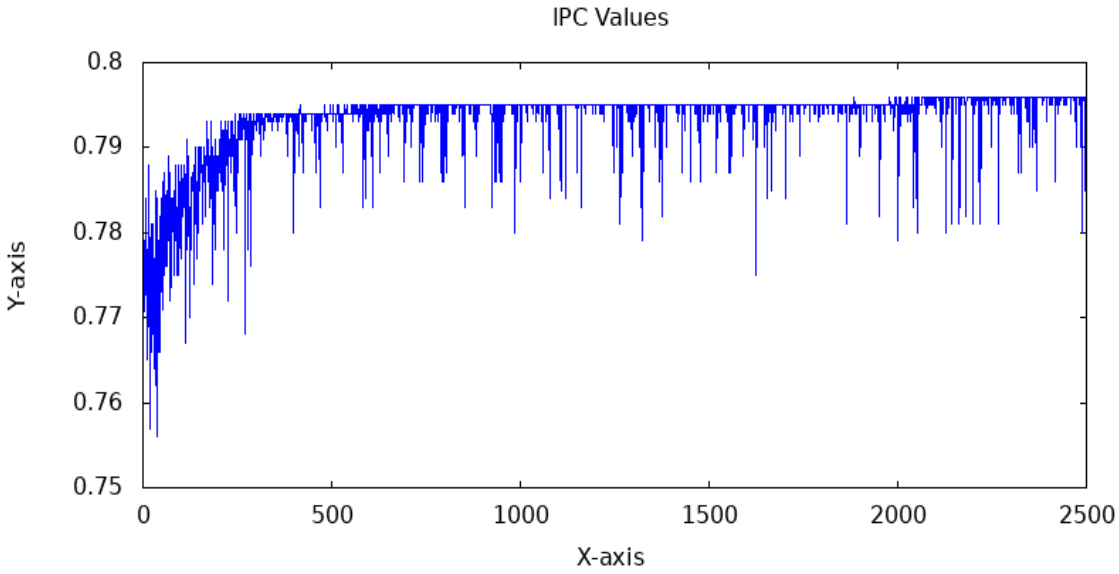

Figure 5.4 Average IPC for each individual from generation 1 to generation 50

L_DemClean	4
L_DemDirty	0
L_Insert	2
L_SPromClean	0
L_SPromDirty	1
R_DemClean	0
R_DemDirty	2
R_Insert	3
R_SPromClean	1
R_SPromDirty	1
P_DemClean	4

P_DemDirty	4
P_Insert	3
P_SPromClean	2
P_SPromDirty	0
W_DemClean	0
W_DemDirty	1
W_Insert	2
W_SPromClean	1
W_SPromDirty	1

Table 5.3 Best policy from experiment 3

Benchmark	IPC from experiment 2 best	IPC from experiment 3 best
Blender	0.672	0.672
Cam4	0.730	0.730
cactuBSSN	0.755	0.755
Gcc	0.353	0.353
Lbm	0.698	0.690
Mcf	0.401	0.401
Parest	1.036	1.035
Wrf	0.826	0.841
Xalancbmk	0.431	0.431
Fotonik3d	0.619	0.616
Imagick	2.191	2.191
Omnetpp	0.247	0.248
Perlbench	0.447	0.447
Roms	1.049	1.051
x264	1.365	1.365
Xz	0.899	0.899
Average IPC	0.795	0.796

Table 5.4 IPC of all benchmarks from experiment 3

L_DemClean	0
L_DemDirty	0
L_Insert	2
L_SPromClean	0
L_SPromDirty	0
R_DemClean	0
R_DemDirty	0
R_Insert	2
R_SPromClean	0
R_SPromDirty	0
P_DemClean	0
P_DemDirty	0
P_Insert	2
P_SPromClean	0
P_SPromDirty	0
W_DemClean	0
W_DemDirty	0
W_Insert	2
W_SPromClean	0
W_SPromDirty	0

Table 5.5 SRRIP used for the comparison with our best policy

5.4 Comparison with SRRIP and LRU

In this subsection we will compare the best policy from experiment 3 as it has the highest average IPC against SRRIP and LRU. This subsection focuses more on the results of the simulations in order to compare the policies, for better comparison we recommend reviewing the source code for each benchmark in order to understand better why we see degradation / improvement in IPC. Table 5.3 illustrates how we encoded SRRIP (Described in Chapter 2) as a GeST individual. As you can see in the encoding on insertion regardless

of the type of access blocks get assigned an RRPV of 2, while on cache hits all type of accesses get promoted to 0. Figure 5.5 illustrates the IPC achieved in each benchmark from the competing policies, Figure 5.6 illustrates the improvement in IPC that out policy achieved over LRU and SRRIP. Improvement was calculated using the formula Improvement = (New IPC – Reference IPC) / Reference IPC * 100.

We will firstly compare our best policy with SRRIP (Table 5.5). We use the Parest benchmark for this comparison as we observed the most improvement in IPC. Looking at Figures 5.5 and 5.6 we can see that our best policy found has a 7.8% improvement in IPC over regular SRRIP. We will now try to determine why we see this improvement using Figures 5.7 – 5.11. When inspecting Figure 5.7, we can see that our policy does better with 13.5 hits PKI opposed to 10.2% achieved by SRRIP with also achieving lower misses PKI, shown in Figure 5.9. From Figure 5.10 we can see that our policy has a higher Hit rate in LLC compared to SRRIP, with 53.80% over 40.72% which is expected. We will then have a closer look to determine what makes our policy perform better than SRRIP. When inspecting Figure 5.11 which illustrate the percentage of hits for each type of access compared to the total number of hits, we can see that our policy decreases the percentage of hits on Writeback accesses by 2% and increases the percentage of hits on Prefetch accesses by the same amount. Load and RFO accesses don't see much of change. This indicates that blocks that originate from prefetch are more important than blocks that originate from writebacks in this benchmark which in turn improves the Hit Rate increasing the performance and the achieved IPC in this benchmark.

We then compare our best policy with LRU. We also used the Parest benchmark for this comparison as we observed the most improvement in IPC. Looking at Figures 5.5 and 5.6 we can see that our best policy found has a 10.78% improvement in IPC over LRU. We will now try to determine why we see this improvement using Figures 5.7 – 5.11. When inspecting Figure 5.7, we can see that our policy does better with 13.5 hits PKI opposed to 7.6% achieved by LRU with also achieving lower misses PKI, shown in Figure 5.8. From Figure 5.10 we can see that our policy has a higher Hit rate in LLC compared to SRRIP, with 53.80% over 30.39 % which is expected. We will then have a closer look to determine what makes our policy perform better than LRU. When inspecting Figure 5.11 that illustrate the percentage of hits for each type of access compared to the total number of

hits, we can see that our policy decreases the percentage of hits on Load accesses and Writeback by 2% and 4% respectively and increases the percentage of hits on prefetch by 6%. RFO accesses don't see much change. This indicates that blocks that originate from Prefetch accesses are more important than blocks that originate from Writebacks and Load accesses in this benchmark which in turn improves the Hit Rate increasing the performance and the achieved IPC in this benchmark.

One interesting result is when we investigate the Lbm benchmark. As shown in Figure 5.6 our best policy yields ~6% improvement over SRRIP and LRU but does so with significantly less hit rate ~10% over the two competing policies that have 42% and 43% respectively, shown in Figure 5.12. We then investigated to uncover the reason why our policy performed better. From Figure 5.9 we can see that Lbm benchmark has a high intensity of RFO and Writeback accesses ~ 17 accesses PKI in both types, with load and RFO only having ~3 accesses PKI. We then investigate the Hit percentage for each access type across the competing policies shown in Figures 5.13. Looking at the percentages of all LLC hits, LRU and SRRIP Writeback hits make up 96% of all LLC cache hits and RFO only 1%. When looking at the same percentages when using our best policy, we can see that the percentage of Writeback hits decreases to 42.5% and RFO hits increase to 42%. From these results we can see that our policy balanced the LLC cache hits for RFO and Writeback which as we discussed have the same intensity PKI, this in turn reduced our Hit Rate but increased the performance.

Moving forward, we can see that our policy did not perform better in all benchmarks. From Figure 5.6 we can see that for benchmarks cactuBSSN, Roms, x264 our policy was outperformed by both SRRIP and LRU. Most notably, in benchmark Roms we saw a decrease of performance by 2% over SRRIP and 2.6% from LRU. When inspecting the misses PKI in LLC for the Roms benchmark we can see that our policy has 15 misses PKI in the LLC while LRU and SRRIP managed ~12 misses PKI. This indicates that our policy increased the number of LLC cache misses by 3 for every 1000 instructions, which in turn contributes to the decrease in performance. Similarly, the same behavior can be observed in benchmarks x264 and cactuBSSN.

We then explore if the ability to use different policies for each benchmark was possible how much improvement will each benchmark see, and how much would the overall improvement we will see. In order to do determine the improvement, we first found which policies performed the best in each benchmark, Table 5.4 shows for each benchmark which policy performed best at it. As we can see we have 13 unique policies with Imagick and Perlbench having the same policy as well as Mcf and Roms also having the same policy. We then calculated the improvement in each benchmark using the formula Improvement = (Best IPC – Reference IPC) / Reference IPC * 100. Figure 5.14 shows the best IPC we achieved in each benchmark while Figure 5.15 shows the improvement compared to LRU and SRRIP. As we can see most benchmarks achieved better performance with Imagick and Perlbench being the only benchmark that had 0% improvement. Nevertheless, when calculating the average improvement over the SRRIP and LRU, using different policies improves the IPC by 2.11% on average over SRRIP and 2.5% on average over LRU. In this point we would like to point out that running different experiments where the genetic algorithm would search for the optimal policy for each individual benchmark, we would probably see better improvement as the policies would be more specialized for each benchmark as opposed to the policies created in this experiment that were specialized to perform better in all benchmarks.

Benchmark	Generation	ID
Blender	50	2483
cactuBSSN	13	628
Cam4	45	2248
Fotonik3d	3	132
Gcc	10	488
Imagick	1	1
Lbm	43	2127
Mcf	33	1618
Omnetpp	38	1862
Parest	47	2305
Perlbench	1	1
Roms	33	1618

Wrf	1	48
x264	4	157
Xalancbmk	1	46
Xz	40	1973

Table 5.6 Policies that achieved the best IPC for each benchmark

5.4.1 Figures used in the comparison

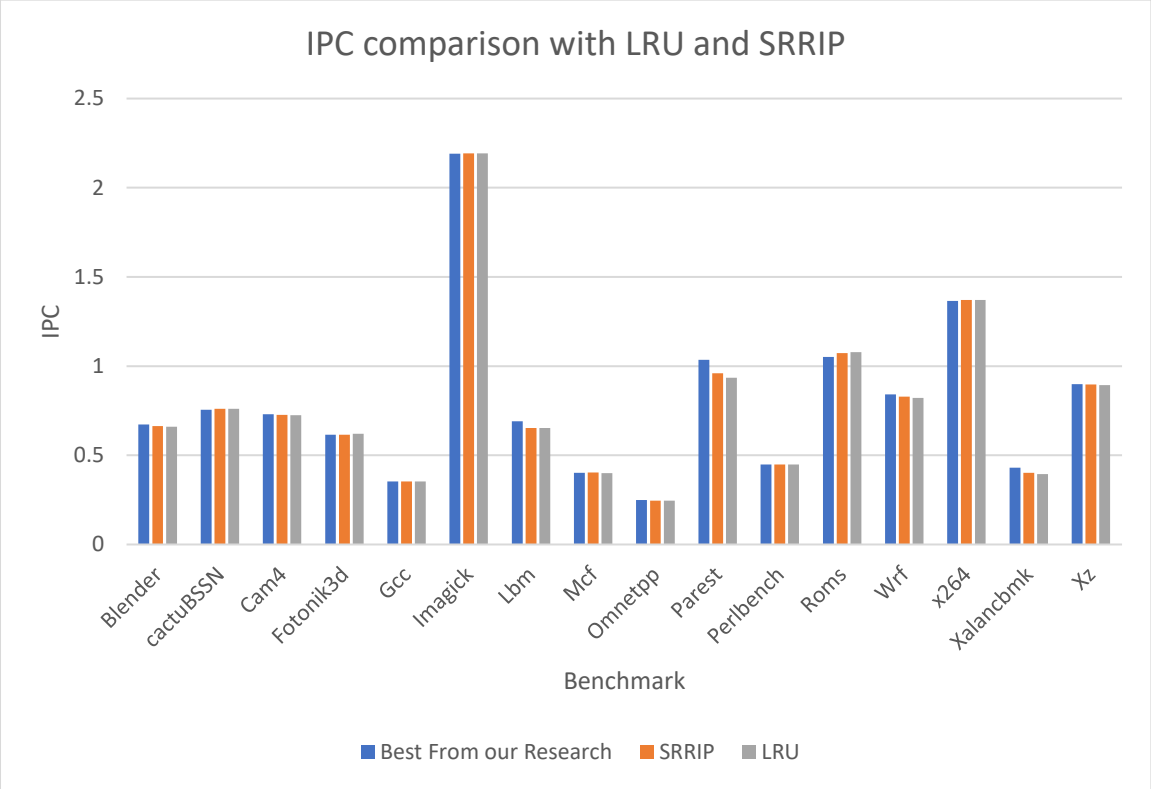

Figure 5.5 IPC comparison between best policy with SRRIP and LRU in benchmark suite

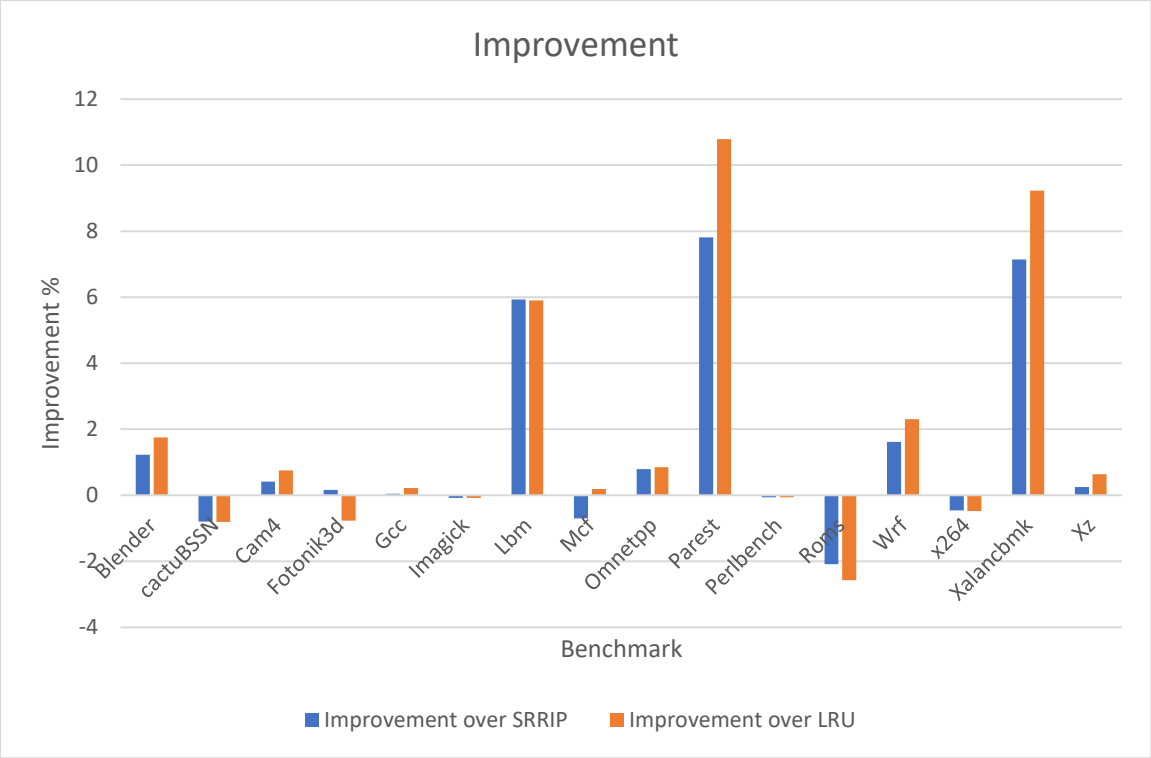

Figure 5.6 Improvement in IPC

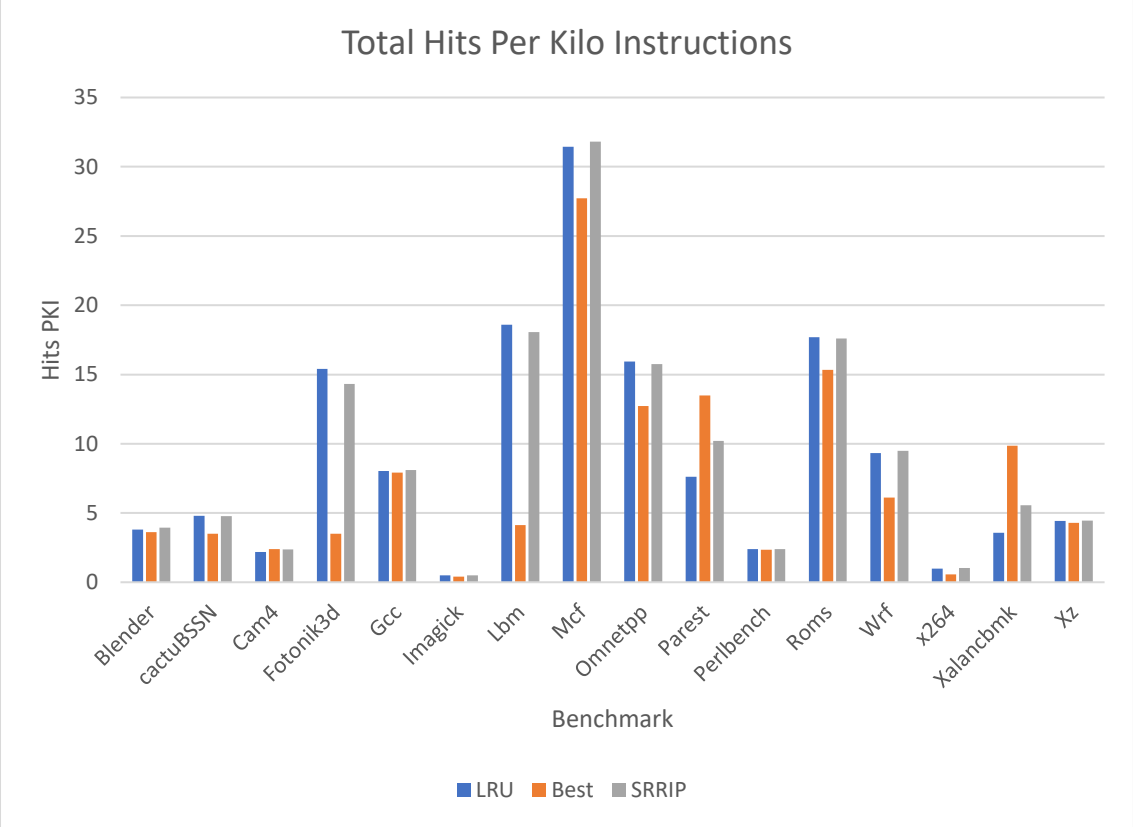

Figure 5.7 Total Hits PKI comparison between best policy with LRU and SRRIP in benchmark suite

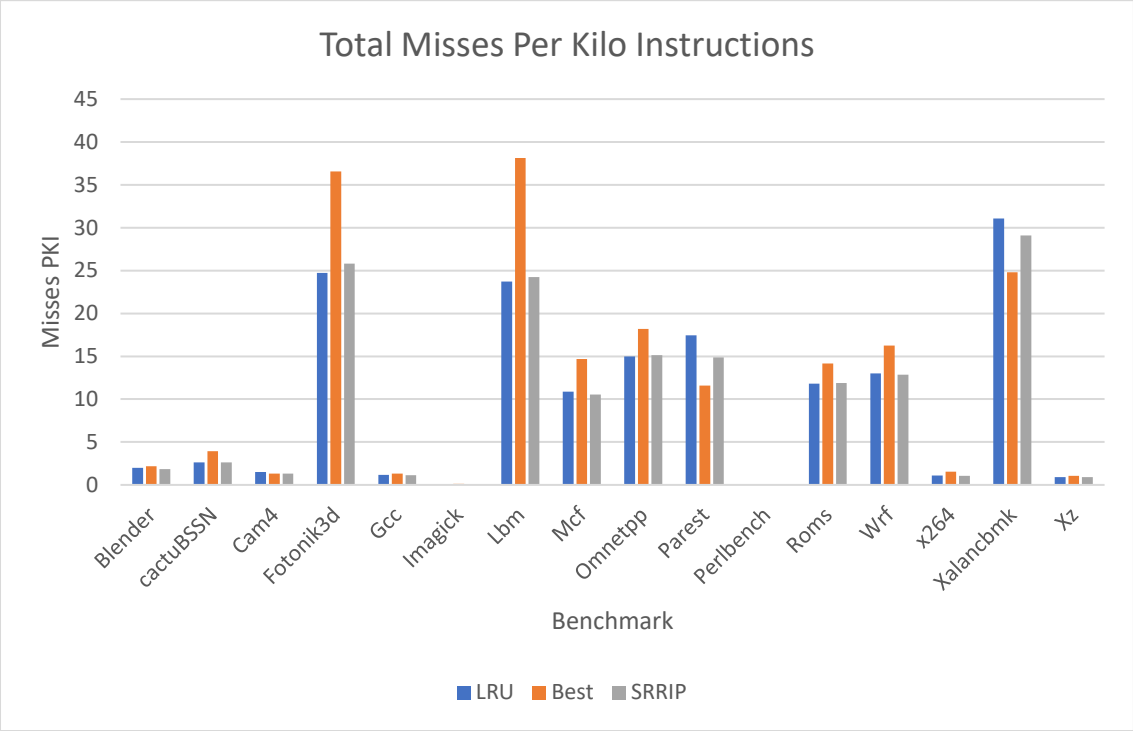

Figure 5.8 Total Misses PKI comparison between best policy with LRU and SRRIP in benchmark suite

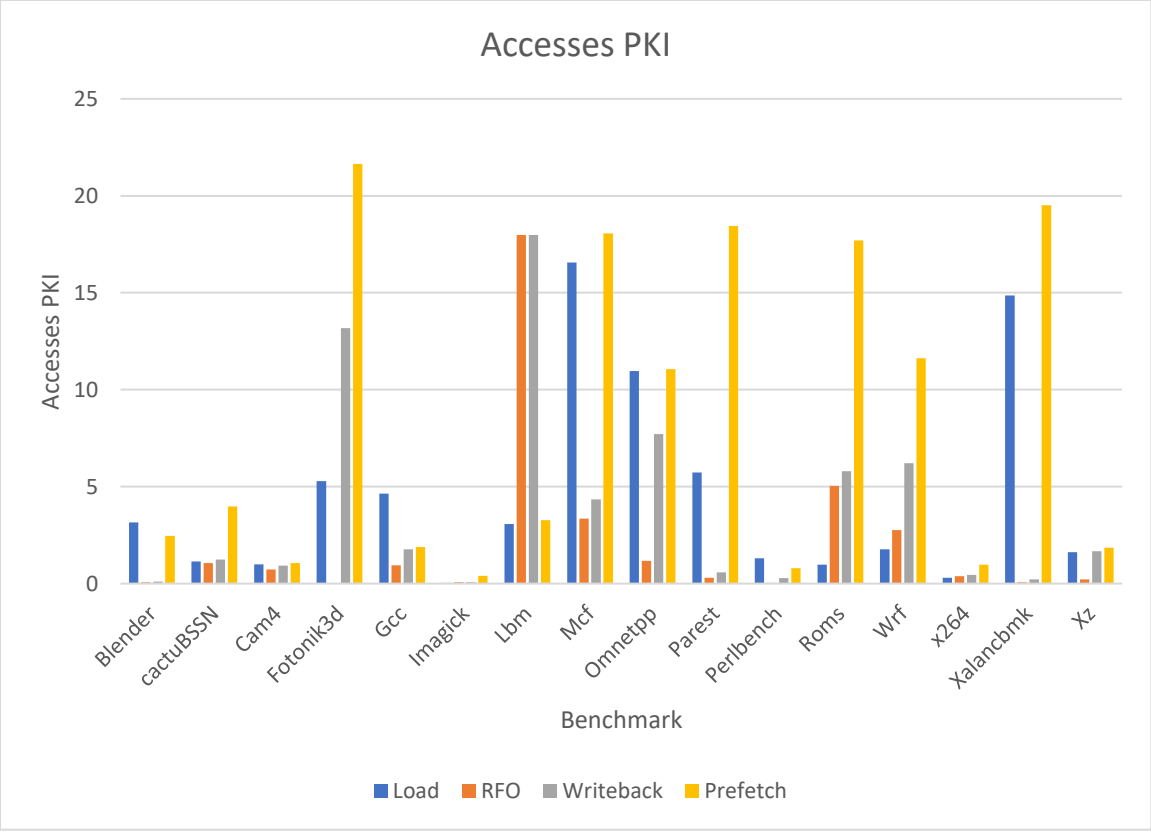

Figure 5.9 Accesses PKI for each access type

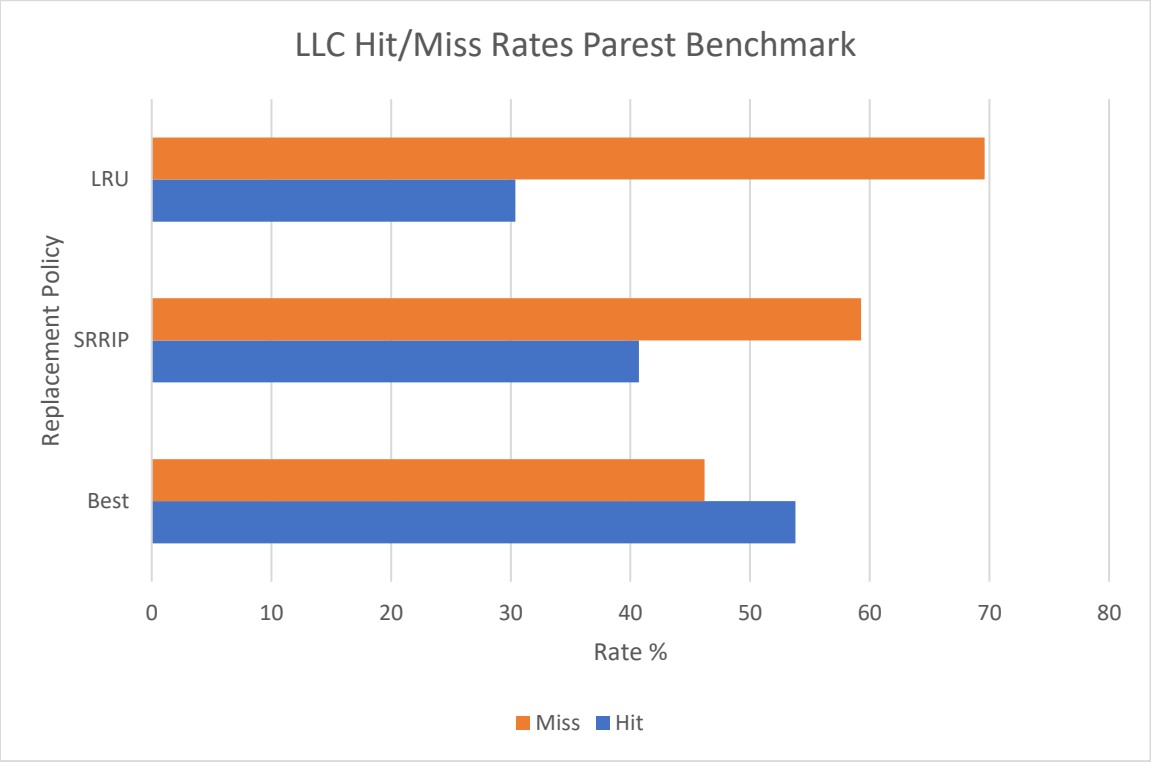

Figure 5.10 Parest benchmark Hit / Miss Rates

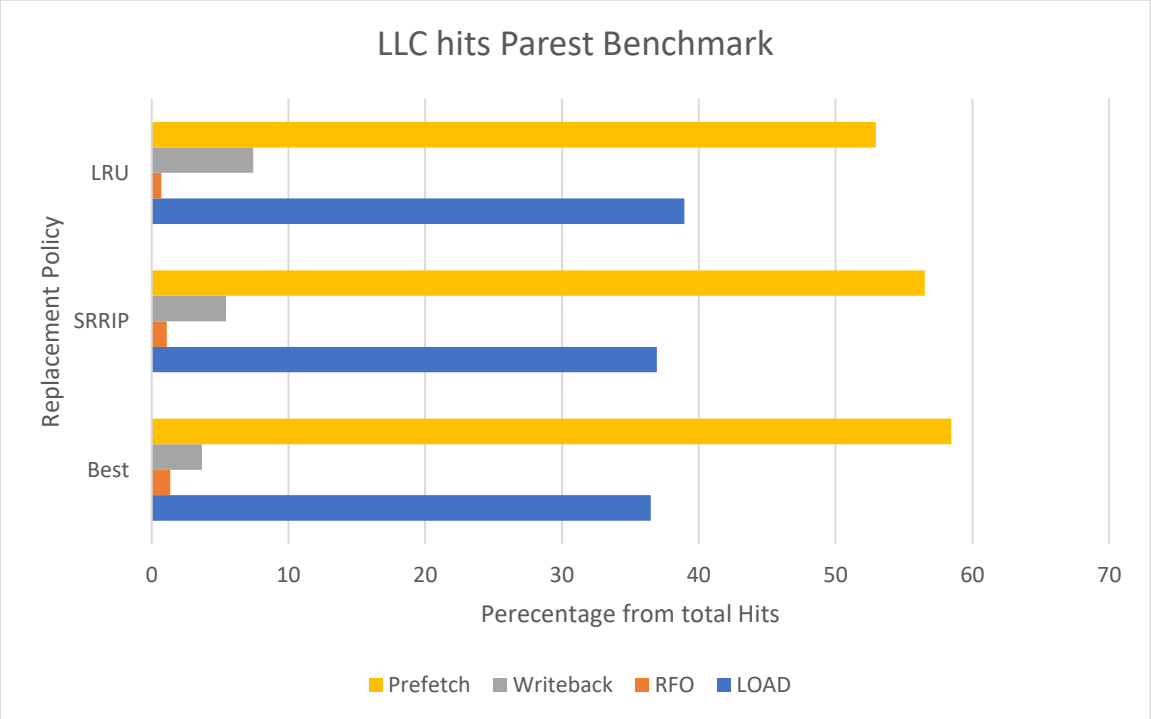

Figure 5.11 Parest benchmark percentage of LLC hits from different access types

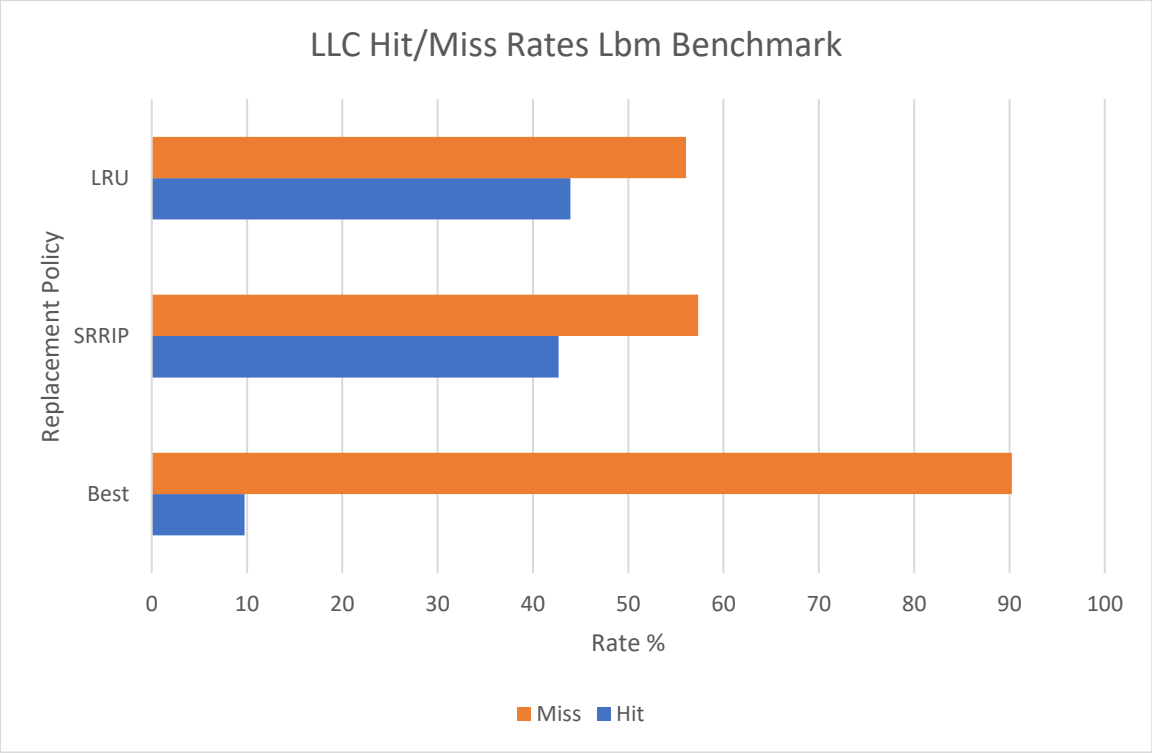

Figure 5.12 Lbm benchmark Hit / Miss Rates in LLC

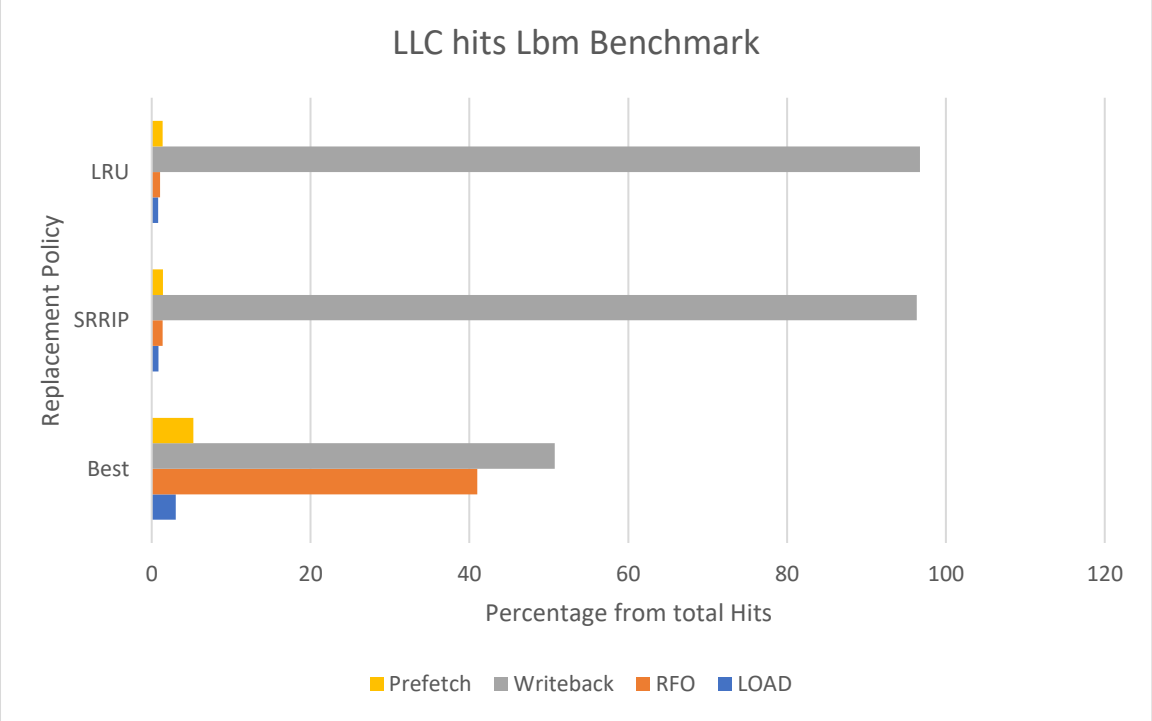

Figure 5.13 Lbm benchmark percentage of LLC hits from different access types

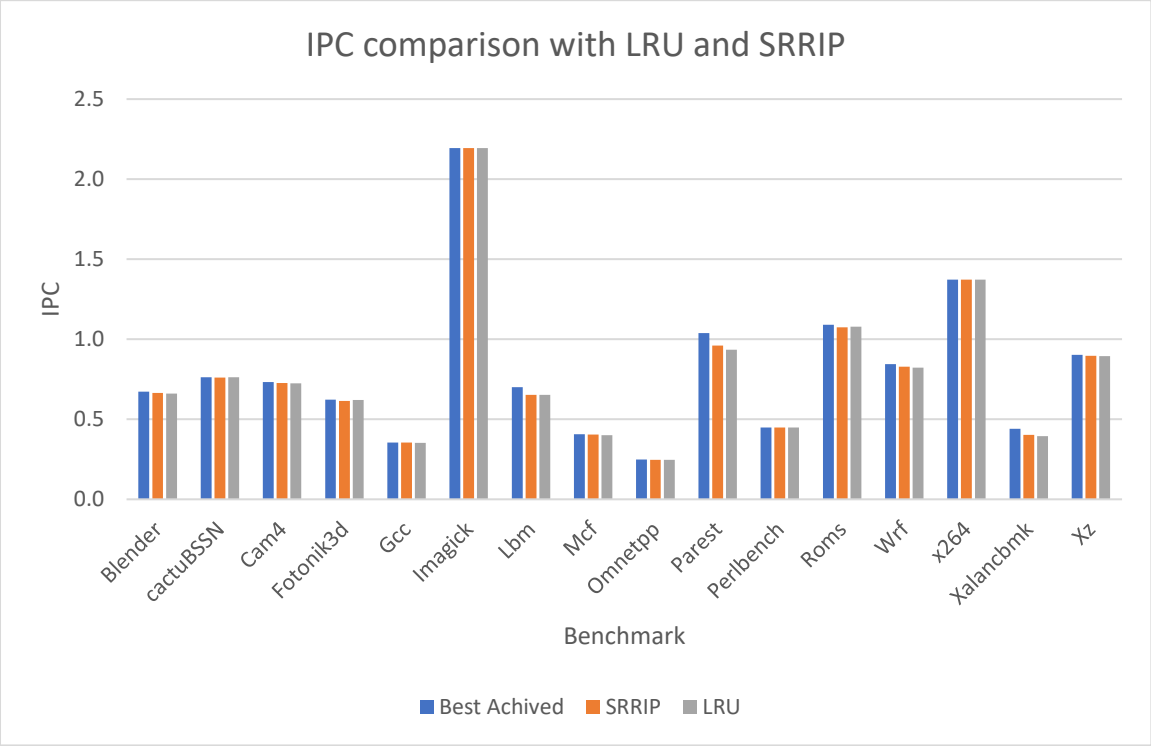

Figure 5.14 IPC comparison between best policy for each benchmark with SRRIP and LRU

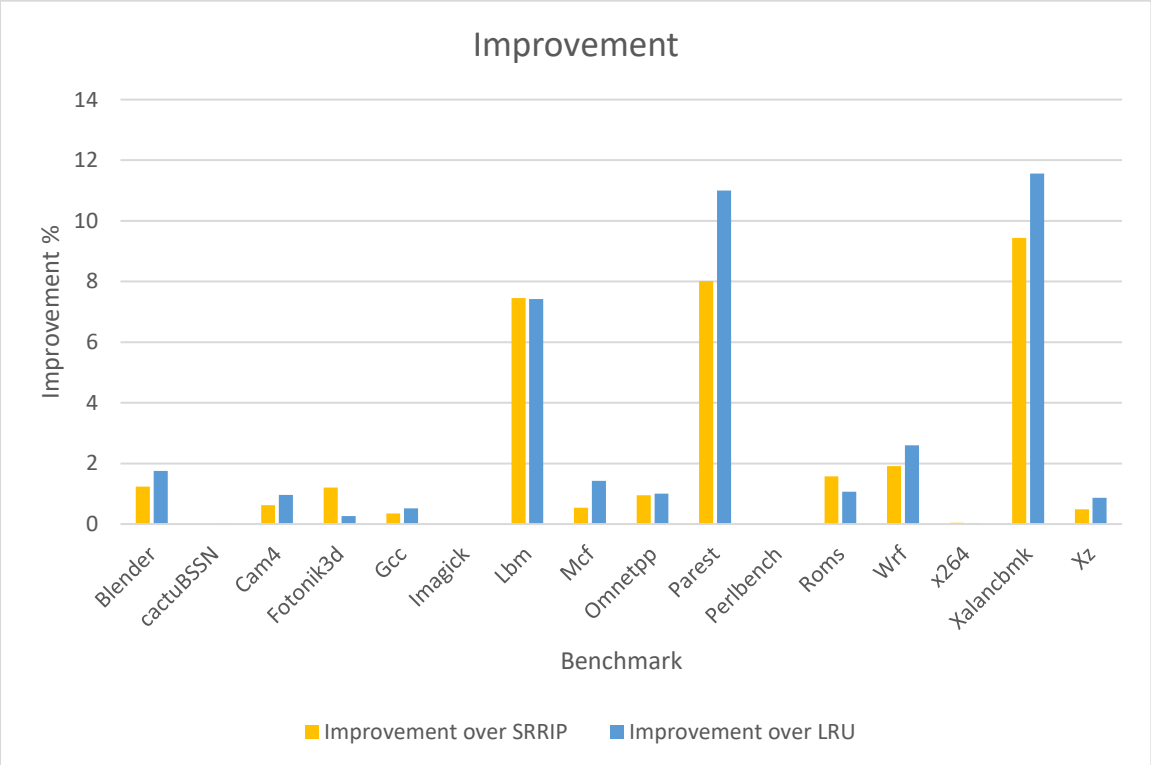

Figure 5.15 Improvement in IPC when using the best policy for each benchmark

5.5 Comparison with best from previous work

In this subsection we will compare the best policy created from the genetic algorithm (Table 5.3) with the best policy created from previous work (Table 5.1) using SRRIP as the base for our comparison. From Figure 5.15 we can see the IPC each policy achieved in each benchmark. We first compare the improvement each policy achieved over SRRIP using Figure 5.16 which illustrates the improvement in IPC achieved by each policy from SRRIP. As we can see from the figure Both policies appear to perform better in the same benchmarks with the exception of Wrf in which the best policy created from the genetic algorithm shows 1.6% improvement over SRRIP wile the best policy from the previous work appears to have 0.2% decrease in improvement over SRRIP. Moving forward, we will compare the two policies head on, to do so we calculated the improvement in IPC the best policy from the genetic algorithm achieved over the best policy from previous work. Figure 5.17 illustrates the improvement, as we can see our policy generated by the genetic algorithm shows improvement over 7 benchmarks while the best policy from previous work is outperforming it in 5 benchmarks.

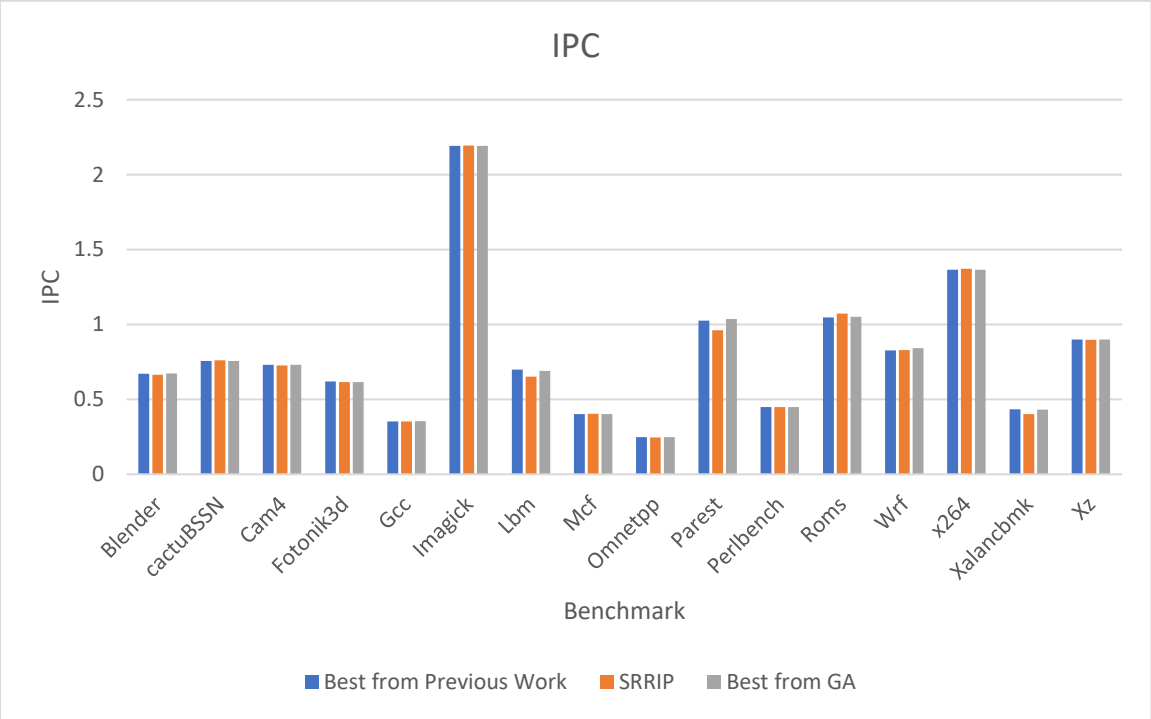

Figure 5.15 IPC comparison between best policy from GA with best policy from previous work and SRRIP

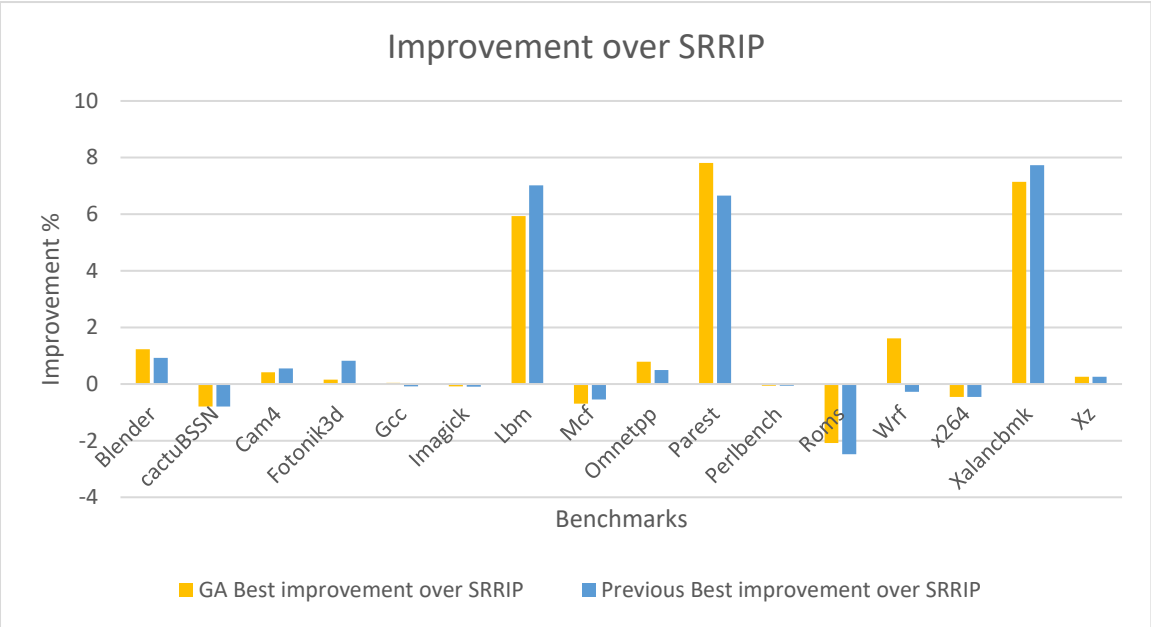

Figure 5.16 Improvement comparison between best policy from GA with best policy from previous work over SRRIP

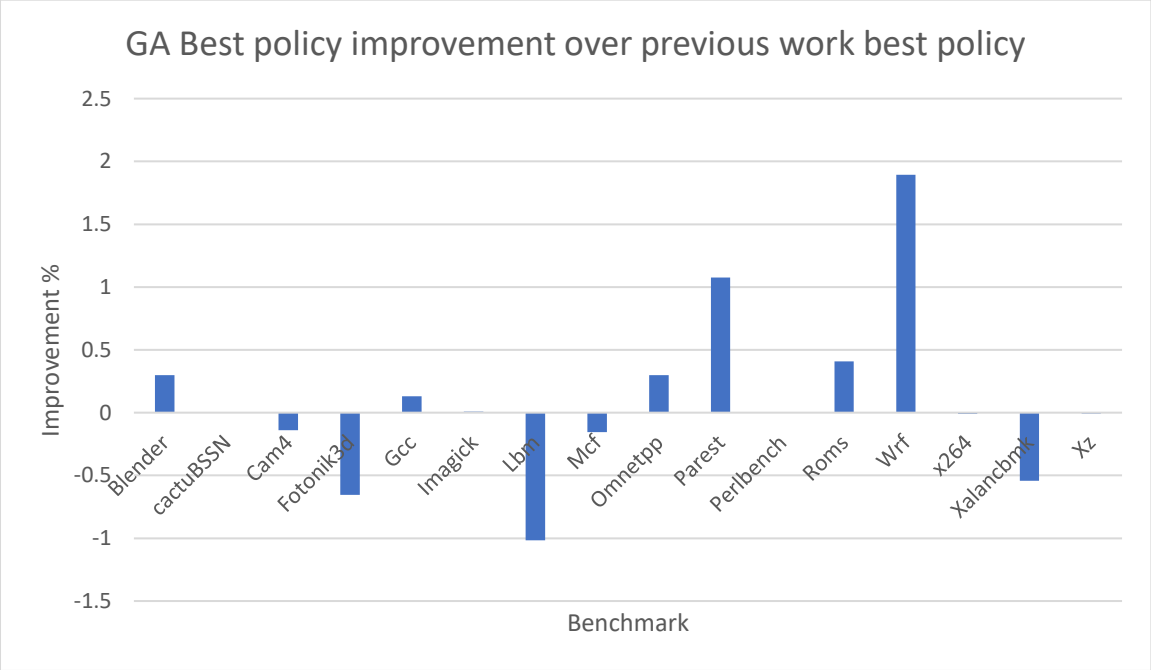

Figure 5.17 Improvement of best policy from GA over best policy from previous work

Chapter 6

Related Work

6.1 Insertion Promotion Vectors

Another relevant work in the field of cache replacement policies for last-level caches is “Insertion and Promotion for Tree-Based pseudoLRU Last-Level Caches” [3]. IPVs propose to approach cache management by utilizing a tree-based pseudoLRU (PLRU) scheme. Unlike traditional LRU, PLRU leverages a hierarchical tree structure to overcome the challenges associated with cache size scalability. A drawback of PLRU is the fact that is based on LRU, meaning it has an inheritance insertion and promotion policy. However, in the paper that are many choices in insertion and promotion that are not exploited by LRU or PLRU. The basic idea of the paper is to explore the space of possible insertion and promotion policies to choose a policy that will maximize the performance. The paper generalizes the notion into the concept of an insertion/promotion vector or IPV. An IPV vector that for a K-way associative cache has length of k+1 entries, each entry has an integer ranging from 0...k-1 that determines what new position a block in the recency stack should occupy when it is re-referenced and what position a new block should occupy. Exploring using an exhaustive search for the vector providing the best speedup would be practically impossible as for a k-way set associative cache there are k^{k+1} possible insertion and promotion policies, e.g., for k=16 there are 2.95×10^{20} different IPVs. Thus, the researchers used a heuristic technique and

to be more specific they used a genetic algorithm to evolve a good IPV. In figure 6.1 you can see the best insertion/promotion vector found by the genetic algorithm for a 16-way set associative last-level cache. We will now discuss the similarities and what differentiates our work from the aforementioned. Firstly, our research uses a very similar general approach in searching for a better replacement policy for last-level cache, we also use a genetic algorithm in order to evolve a good replacement policy as the search space is impossible to explore without a heuristic approach. What differentiates our work besides than we search for an RRIP based replacement policy rather than LRU one is that we specialized our replacement policy to perform differently for each cache access type, in addition we defined 5 aspects that comprise a replacement policy and each one has the ability to perform differently based on the function it implements.

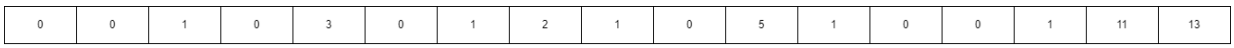

Figure 6.1 Best IPV found by the genetic algorithm

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In conclusion, this thesis has addressed the importance of cache replacement policies in modern computer architectures and the need for specialized policies to optimize cache performance for specific access patterns. Through the use of a genetic algorithm, we have explored the viability of automating the search for such specialized cache replacement policies. The results of our simulations have provided valuable insights into the performance benefits of evolved cache replacement policies. We have observed improved cache hit rates and overall system performance compared to traditional policies, indicating the potential of automated search techniques in optimizing cache efficiency. The findings presented in this thesis contribute to the existing body of knowledge on cache replacement policies and demonstrate the effectiveness of genetic search algorithms in discovering specialized solutions. These results open avenues for further research in cache replacement policy optimization, considering different access patterns and evaluating the performance of evolved policies in real-world scenarios.

7.2 Future Work

In order to advance the field of cache replacement policies and optimize cache performance in modern computer architectures, future work can be directed towards several promising directions. Firstly, conducting experiments on real hardware platforms would provide more accurate and practical insights into the effectiveness of evolved policies. Additionally, expanding the search space by using a larger set of candidate functions for cache replacement would allow for a larger space of potential solutions. Furthermore, incorporating set dueling techniques, which involve utilizing multiple policies in parallel and dynamically selecting the most suitable one, could lead to further improvements in cache efficiency. Moreover, exploring the application of machine learning techniques, such as reinforcement learning or neural networks, as an alternative to genetic algorithms would open new possibilities for automating the discovery of optimized cache replacement policies. By following these avenues of investigation, we can push the boundaries of cache optimization and contribute to the development of smarter and more effective cache replacement strategies.

References

[1] Jaleel, A. *et al.* (2010) *High performance cache replacement using re-reference interval prediction (RRIP)*. Available at: https://people.csail.mit.edu/emer/media/papers/2010.06.isca.rrip.pdf

[2] Moinuddin K. Qureshi The University of Texas at Austin *et al.* (2007) *Adaptive insertion policies for high performance caching*, *ACM SIGARCH Computer Architecture News*. Available at: http://www.jaleels.org/ajaleel/publications/isca2007-dip.pdf

[3] Jiménez, D. (2017) *Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches.* Available at: https://people.engr.tamu.edu/djimenez/pdfs/p284-jimenez.pdf

[4] ChampSim: Architectural Simulation. *Available at: https://github.com/ChampSim/ChampSim*

[5] Z. Hadjilambrou, S. Das, P. N. Whatmough, D. Bull and Y. Sazeides, (2019).GeST: An Automatic Framework For Generating CPU Stress-Tests. Available.at : https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8695639&isnumber=8695630