Diploma Project

EXAMINING SYNERGISTIC BEHAVIOR OF TWO CACHE REPLACEMENT POLICIES

Anastasios Chatzikyriakou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Examining Synergistic Behavior Of Tow Cache Replacement Policies

Anastasios Chatzikyriakou

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 2024

Acknowledgments

I would like to express my special thanks to my Professor, Mr. Yiannakis Sazeides, for giving me this great opportunity to work with him and explore an area of Computer Science that I would not have otherwise. Also, I would like to thank Ioannis Constantinou for his help and support. Lastly, I would like to express my gratitude to Panteleimonas Chatzimiltis for his help on the process of better understanding the genetic algorithm.

Abstract

This thesis builds upon previous research on cache replacement policies by exploring the potential performance benefits of combining two distinct replacement strategies, aiming to discover the optimal solution for enhancing cache performance in modern computer architectures. Traditional approaches such as LRU (Least Recently Used) and FIFO (First In, First Out) often do not address the complexities of varied workloads and diverse access patterns. The prior research utilizes a population-based genetic algorithm to systematically evolve a broad spectrum of potential policies across generations, enhancing their effectiveness progressively.

A significant innovation of that study is its focus on the different impact of cache misses, recognizing that not all misses are equal and each affects performance differently. The motivation stems from the earlier work that individually addressed cache replacement policies with the objective of identifying a single replacement policy that maximizes the average instructions per cycle (IPC). In contrast, this thesis investigates whether a strategic combination of two replacement policies can further enhance system performance.

Through extensive simulations, we assess the performance of hybrid policies, particularly examining IPC improvements. The study introduces an analysis of maximum IPC speed-ups by evaluating the top-performing hybrid policies from each benchmark versus the overall fittest singular policy from prior experiments.

The results are very promising. The best synergy of policies (dual-policy individual) yielded a performance benefit of 7.58% over the baseline synergy established.

This analysis underscores the potential for achieving higher performance gains by integrating multiple replacement strategies tailored to specific workload scenarios, rather than adhering to a one-size-fits-all policy. The findings pave the way for future research to refine these hybrid strategies further, addressing the continuously evolving demands of modern computing environments. This work contributes not only to the theoretical understanding of cache replacement policy optimization but also offers practical insights for enhancing system performance.

Contents

Chapter 1 Introduction……………………………………………………………………1

1.1 Introduction……………………………………………………………..1

1.2 Outline....……………………………………………………………..2

Chapter 2 Background………………………………………………………………3

2.1 Modern CPUs…………………………………………………………...3

2.2 Multilevel Cache Hierarchy……………………………………………5

2.3 Replacement Policies - RRIP...…………………………………………6

2.4 Genetic Algorithms……………………………………………………10

2.5 Microarchitectural Simulators…………………………………………11

2.6 Prior work and Motivation..……………………………………………...11

Chapter 3 Frameworks and methodology…………………………………...…..14

3.1 The HPC system………………………………………………………14

3.2 Benchmark suit…………………………………………………….. 14

3.3 ChampSim simulator…………………………………………………..15

3.3.1 The simulator…………………………………………………...15

3.3.2 The configuration……………………………………………....16

3.3.3 Focus on the LLC………………………………………………17

Chapter 4 DRRIP-based Replacement Policy…………………………………..19

3.2.4.1 Cache Scan and Thrash………………………………………………19

3.2.4.2 Set Dueling……………………………………………………………..20

3.2.4.3 SRRIP-DRRIP…….…………………………………………………….20

Chapter 5 Gest: The Genetic Algorithm……………………………………..…….22

5.1 Traditional Approach………………………………..…………....22

5.2 Policy encoding……………………………………………………...…...23

5.3 The search space……………………………………..……………..26

Chapter 6 Dual-Policy Modifications……………………………….………………......27

6.1 Modifications……………………………………………………………….27

6.2 Fitness evaluation, Mutation and Crossover……………………………..28

Chapter 7 Experiments and Results………..………………………………………32

7.1 Experiments' Set up………………………..………………………32

7.2 Results.……………………………………….……………………..34

7.2.1 First experiment: Proof of concept……………………………..34

7.2.1.1 Benchmark Analysis..………………….………...36

7.2.1.2 Conclusions..……………………………………...42

7.2.2 Second experiment: Evolve dual-policy individuals...………...43

7.2.2.1 Analysis of the best…….…………………………...44

7.2.2.2 Benchmarks examination..…………………………..46

7.2.2.2.1 Dive into Lbm…....………………………46

7.2.2.3 Conclusions……...………………...……………...52

Chapter 6 Future work and Conclusions………………………………………………..53

6.1 Future work…………………………………………………………...53

6.2 Conclusions…………………………………………………………...54

References ……….…………………………………………………………………..56

Appendix A……………………….……..…………………………….………….…...57

Appendix B……………………….……..…………………………….………….…...58

Chapter 1

Introduction

1.1 Introduction ……………………………………………………………………..1
1.2 Outline ………………………………………………………………………..4

1.1 Introduction

This thesis extends the study of cache replacement policies by examining the integration of two replacement policies to determine the optimal approach for enhancing cache performance in modern computer architectures. Traditional methods such as Least Recently Used (LRU) and First In, First Out (FIFO) are commonly used but often fall short in managing complex workloads and diverse access patterns effectively. This limitation highlights the need for more dynamic and specialized cache replacement strategies capable of adapting to the changing demands of modern applications.

Leveraging the groundwork laid by previous research [5], this study innovates by using the same population-based genetic algorithm as the previous work, to foster the systematic evolution of hybrid policies. These hybrid policies are unique in that each individual in the current study represents a combination of two individuals from prior work. This means that each individual of this research is a combination of two policies that would be tested to find the (one) best policy. The goal is to explore the potential benefit on performance when using the best combination of two policies over the use of the best policy given by the previous work.

Through simulations, the performance of these composite policies is evaluated, focusing specifically on improvements in the average instructions per cycle (IPC). This thesis introduces a pioneering analysis method that measures maximum IPC speed-ups, comparing the best-performing hybrid policies for each benchmark against the optimal single policy identified in earlier experiments.

The findings from this analysis not only facilitate future developments in hybrid cache replacement strategies but also significantly enhance both the theoretical and practical understanding of cache policy optimization. It sets a foundation for further investigation into adaptive and context-specific solutions, providing practical insights that could lead to substantial improvements in the efficiency and effectiveness of computer systems. This thesis marks a critical step forward, offering new perspectives and methodologies that could transform cache replacement strategies in contemporary computing environments.

1.2 Outline

This thesis is divided into six chapters, each focusing on different aspects of the research. Chapter 1 introduces the topic, highlighting the importance of optimizing cache replacement policies and outlining the main research question: What are the performance benefits of applying the most effective cache replacement policies for each benchmark out of pairs of policies? Chapter 2 provides necessary background information on key concepts in computer architecture that are vital for this research. Chapter 3 details the tools, their configurations and the methods used in this research. Chapter 4 discusses the changes made to test the main ideas behind the study. Chapter 5 describes how the experiments were conducted and shares the findings. Finally, Chapter 6 suggests directions for future research and summarizes the conclusions of the thesis.

Chapter 2

Background

2.1 Modern CPUs……………………………………………………………………3
2.2 Multilevel Cache Hierarchy……………………………………………………...5
2.3 Replacement Policies - SRRIP ……………………………………………………6
2.4 Genetic Algorithms…………………………………………………………..10
2.5 Microarchitectural Simulators……………………………………………………11
2.6 Prior work……………………………………………………………………11

2.1 Modern CPUs

Modern CPUs (Central Processing Units) represent the brain of computer systems, orchestrating the execution of programs by performing various computational tasks. As technology has advanced, the architecture and functionality of CPUs have evolved significantly to meet the demands of increasingly complex software applications and processing requirements. A key feature of modern CPUs is their use of a pipeline architecture to improve efficiency and performance. A CPU pipeline is similar to an assembly line in a factory, where each part of the line handles a specific task. This method allows multiple instructions to be processed concurrently, each at a different stage of execution, thus speeding up the overall processing time. The basic stages of a modern CPU pipeline include:

1. **Fetch**: The CPU reads an instruction from the memory.
2. **Decode**: The instruction is broken down and interpreted.
3. **Execute**: The CPU carries out the instruction.
4. **Memory Access**: If the instruction requires data from memory, this is the stage where the CPU fetches or stores data.
5. **Write Back**: The result of the instruction's execution is written back to a register or memory.

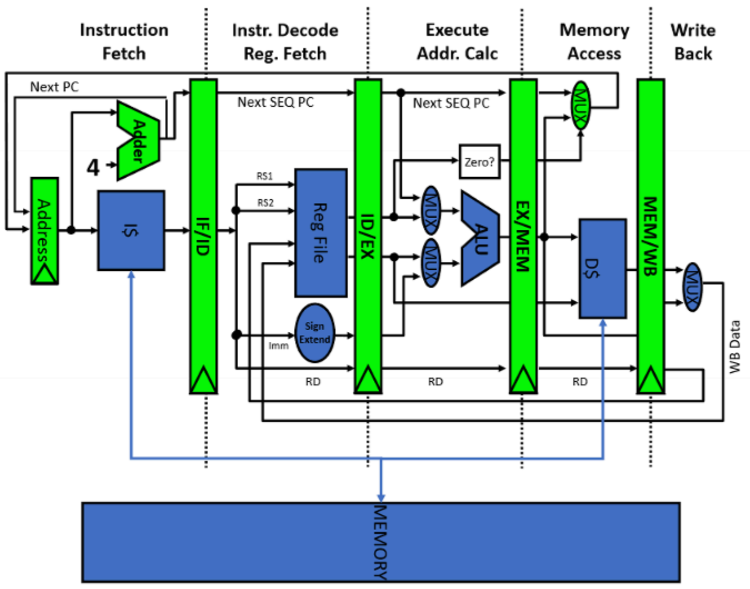


Figure 2.1: A simple five stage pipeline

The Fetch stage is particularly critical as it is the starting point of the instruction processing cycle. During this stage, the CPU retrieves an instruction from the memory, which introduces the necessity for efficient interaction with the memory hierarchy. The speed and efficiency of fetching instructions are heavily influenced by the effectiveness of the cache memory. Cache memory plays a pivotal role in the performance of a CPU overall. They are smaller, faster types of memory located closer to the CPU cores, designed to store copies of data from frequently accessed main memory locations. Efficient caches reduce the time it takes for the CPU to access data and instructions, which can significantly boost overall system performance. If the cache is efficient and well-managed, the fetch stage is expedited because instructions are readily available at high speeds. On the other hand, inefficient cache systems can lead to increased latency as the CPU may need to fetch instructions from the slower main memory, which negatively impacts the CPU's performance and can cause a bottleneck in the processing pipeline. The Memory Access stage is another critical component of the CPU pipeline, occurring after the instruction has been executed. This stage involves the CPU either fetching data from memory to use in computation or storing data back to memory after computation. The efficiency of this stage is crucial for the overall performance of the CPU, as it directly affects the speed at which a program can run. Memory accesses in modern computer systems can be categorized into four primary types, each signifying a different interaction mechanism between the processor and memory. These access types are essential for understanding system performance and designing efficient cache management strategies. 1) Load occurs when the processor retrieves data from memory to use in computations. 2) Read For Ownership (RFO) involves acquiring data for modification, ensuring exclusive rights to alter it, which is crucial in multi-threading environments to prevent data conflicts. 3) Prefetch is a proactive mechanism where data is fetched and stored in the cache before it is actually needed, aiming to reduce latency by anticipating future requests. Lastly, 4) Writeback is the process of writing modified data back to memory from the cache, which is critical for maintaining data integrity and consistency across the system. Each type of access has distinct implications on cache replacement policies and overall system performance, making their understanding pivotal in the architecture of efficient computing systems. Therefore, optimizing cache replacement policies (section 2.3) and ensuring efficient cache management is crucial. It allows modern CPUs to meet their performance potentials by minimizing delays in instruction fetch and overall execution times. This optimization is particularly important in systems handling complex and diverse workloads, where access patterns may vary significantly, challenging traditional cache management strategies.

2.2 Multilevel Cache Hierarchy

Modern processors generally have three levels of cache: L1, L2, and L3, each offering varying performance benefits. The L1 cache, the smallest and fastest, is located closest to the processor cores. It is typically divided into separate caches for instructions and data, ensuring rapid access to frequently used resources. The L2 cache, though larger and thus slower than L1, holds more data, which helps in reducing the frequency of cache misses and boosts performance. The L3 cache, often referred to as Last Level Cache (LLC), larger than both L1 and L2, is shared among all cores, making it slower than L2 but faster than main memory. Its size benefits the system by storing data shared across cores, minimizing the need for main memory access, enhancing performance. The hierarchical arrangement of varying cache sizes helps maintain frequently accessed data and instructions near the processor, minimizes cache misses, and decreases accesses to main memory, thereby significantly enhancing performance.

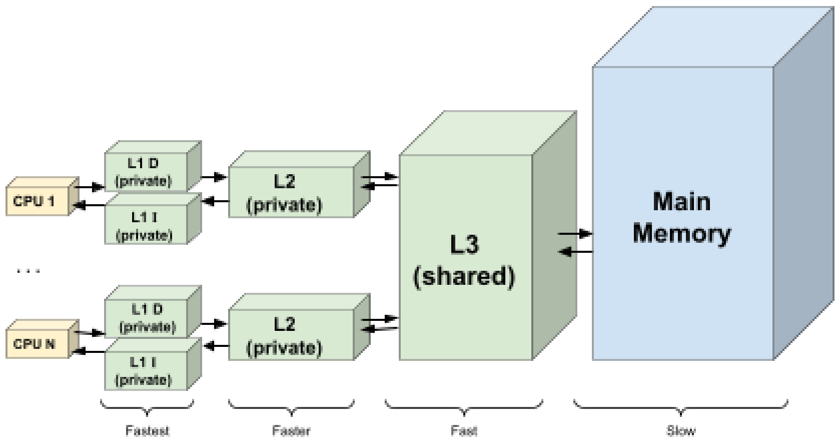


Figure 2.2: Memory Hierarchy

2.3 Replacement Policies - RRIP

Cache replacement policies are essential mechanisms within computer architectures designed to manage how data is stored, replaced, or removed in a cache when it becomes full. The necessity for such policies stems from the limited size of cache memory coupled with the need for high-speed data access by the CPU. Efficient cache management ensures that the most relevant data is available in the cache, thereby reducing the number of cache misses - situations where the data requested by the CPU is not found in the cache, leading to slower access times as data is fetched from a lower level memory. By doing so, an efficient cache replacement policy maintains an optimal mix of data in the cache that is most likely to be needed soon, based on various criteria such as frequency of access or the “age” of the data. First In, First Out (FIFO) is a cache replacement policy that removes the oldest data from the cache to make space for new data, regardless of how frequently or recently the data has been accessed, making it a bad decision in the majority of situations. Least Recently Used (LRU) is another cache replacement policy that discards/evicts the least recently accessed data from the cache to make room for new data, assuming that data not used recently will likely not be needed soon. The performance and efficiency of the LRU are generally high in scenarios with predictable access patterns. However, its effectiveness diminishes in more complex or varied scenarios where access patterns are less predictable. Moreover, LRU can be computationally expensive to implement in large cache systems because it requires tracking the recency of access for each cache line. Other replacement policies are PLRU (Pseudo Least Recently Used) [4], LFU (Least Frequently Used), MRU(Most Recently Used), RRIP (Re-reference Interval Prediction) which have their own advantages and disadvantages. Unfortunately, the best replacement policy is not a one-fits-all decision. The best choice of policy will depend on the specific requirements of the system and the workload being executed. Also, the characteristics of the cache system will be a concern when choosing a replacement policy.

The experiments we contact have as their base the RRIP (Re-reference Interval Prediction) [1] which is a cache replacement policy that uses a history-based approach to determine the usefulness of a cache block. As its name suggests, RRIP is not implemented in the conventional way, which is replacing blocks of data based on the frequency of access or the “age” of the data. RRIP policy 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 categorizes re-reference intervals into various classes, each representing a different level of a block's utility. Each block in the cache set has an associated counter that tracks its re-reference interval value (RRIP value). The classification of a block is based on its RRIP value , which can range from 0 to N; with 0 indicating the most recently referenced and N the least; typically, N is set to 3. The four different utility classes, represented by the RRIP values for N set to 3 are: 0 (highest utility), 1 (high utility), 2 (lower utility), and 3 (lowest utility). When a block is accessed, its value is reset to the minimum (indicating the highest utility class), showing that the block has been recently used. Over time, if the block is not accessed, its counter incrementally increases as the system determines which block to remove. Eviction happens when the cache needs space for a new block and the one with the highest value is chosen for eviction. If no blocks meet this criterion, then the value of all blocks in the set is increased until one qualifies for eviction.

Figure 2.3 which comes from the paper High performance cache replacement using re-reference interval prediction (RRIP) [1], compares the behaviors of LRU, NRU, and

SRRIP under the same access pattern. The LRU (Least Recently Used) policy evicts the block that has not been used for the longest time. On a cache hit, the block is moved to the most recently used (MRU) position. On a cache miss, the least recently used block is replaced, and the new block is placed in the MRU position. This ensures that the most recently accessed data is retained in the cache, but it can lead to suboptimal performance if the access patterns change rapidly.

Pseudocode for LRU:

```
On Cache Hit:
    1. Move the accessed block to the MRU (Most Recently Used) position.

On Cache Miss:
    1. Identify the LRU (Least Recently Used) block.
    2. Replace the LRU block with the new block.
    3. Move the new block to the MRU position.
```

The NRU (Not Recently Used) policy divides blocks into two categories: recently used and not recently used. Each block has an "nru-bit" indicating its status. On a cache hit, the nru-bit of the block is reset to '0', marking it as recently used. On a cache miss, the policy searches for the first block with an nru-bit of '1' to replace. If no such block is found, it sets all nru-bits to '1' and repeats the search. This approach simplifies the replacement decision but might not be as adaptive to varying access patterns.

Pseudocode for NRU:

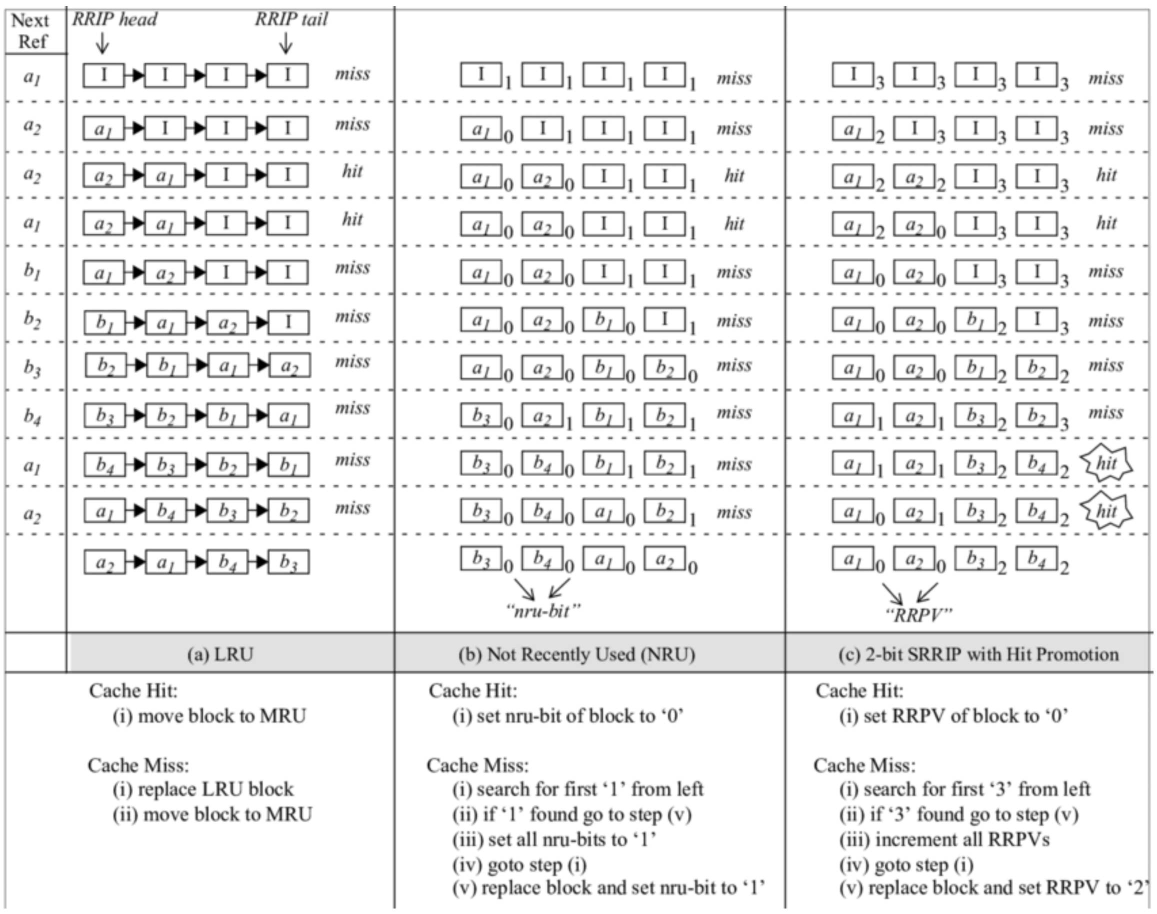
```
On Cache Hit:
    1. Set the nru-bit of the accessed block to '0'.

On Cache Miss:
    1. Search for the first block with nru-bit '1'.
    2. If found, replace the block.
    3. If not found:
        a. Set all nru-bits to '1'.
        b. Repeat the search for the first block with nru-bit '1'.
```

The SRRIP (Static Re-Reference Interval Prediction) policy uses a counter for each block to track re-reference intervals, with values ranging from 0 (indicating highest utility) to 3 (indicating lowest utility). On a cache hit, the block's RRIP value is reset to 0, showing that it has been recently used and thus has high utility. On a cache miss, the policy searches for a block with an RRIP value of 3 to replace. If none is found, it increments the RRIP values of all blocks and repeats the search. This mechanism ensures adaptability to changing access patterns and efficient eviction decisions. Additionally, SRRIP's implementation is straightforward and is characterized by low overhead. For instance, a 16-way cache using LRU requires 4 bits per way to encode positions, whereas 2-bit RRIP only needs 2 bits per way to encode the four utility classes (0-3).

<u>Pseudocode for SRRIP:</u>

```
    On Cache Hit:
        1. Set the RRIP value of the accessed block to 0.

    On Cache Miss:
        1. Search for the first block with RRIP value of 3.
        2. If found, replace the block.
        3. If not found:
            a. Increment the RRIP values of all blocks.
            b. Repeat the search for the first block with RRIP value of 3.
```

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.


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

(a) LRU

Cache Hit:
(i) move block to MRU

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

(b) Not Recently Used (NRU)

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

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

(c) 2-bit SRRIP with Hit Promotion

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

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

Figure 2.3 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 generation. We can see the flow of a genetic algorithm in Figure 2.4. 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 (see section 3.4.3). 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 [3], a framework created for automatic generation of stress tests that was modified to work for the purpose of our research.

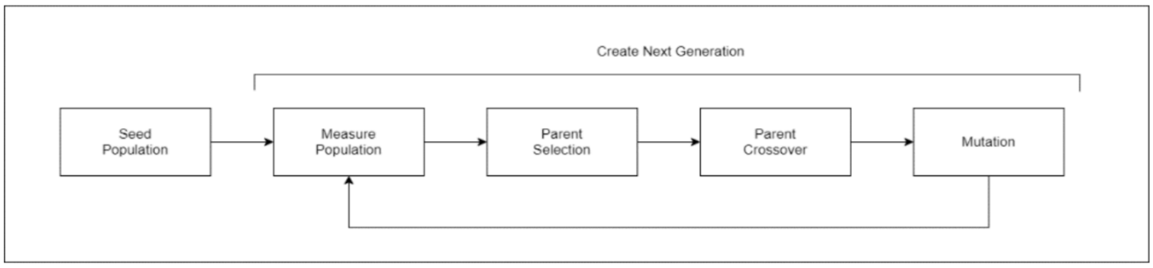


Figure 2.4:The flow of a genetic algorithm

2.5 Microarchitecture Simulators

Microarchitecture simulators are software tools designed to emulate the behavior and functionality of a microarchitecture, enabling researchers to analyze, test, and optimize hardware designs without physical prototypes. These simulators offer several benefits over real hardware in development and testing. Their modular design allows users to easily alter specific components or try out different configurations without needing to make physical changes. Additionally, unlike physical hardware experiments that can be influenced by environmental conditions, simulators provide a stable and reliable means to precisely replicate memory access patterns and other critical workload characteristics. For research purposes, using a microarchitecture simulator facilitates quicker experimentation and evaluation of numerous potential policies in a consistent and reproducible setting. One of the key advantages of using simulations is the cost-free flexibility it provides; parameters such as CPU core count, cache sizes, and replacement policies can be adjusted effortlessly, enabling extensive and economical testing possibilities.

2.6 Prior work and Motivation

As already mentioned, this research builds upon prior work. This prior work is Nikolas' Papaki Thesis: Genetic Search on RRIP Replacement Policies [5]. Nicolas' thesis on

RRIP replacement policies is an approach that aims to automate the search for efficient cache replacement policies using a genetic algorithm. His work, with its primary objective of optimizing IPC, delves into the efficiency of various replacement policies and their evolutionary refinement over successive generations. His research places a particular emphasis on the RRIP model, investigating its viability in different workload scenarios and comparing its performance with standard policies such as LRU (Least Recently Used). With a focus on genetic algorithms' heuristic-based approach, Papaki's thesis illuminates their potential in generating replacement policies through mutation and crossover, two terms that have their roots in Darwinian evolution - and will be explained further in the paper (see section 3.4.1) - , assessing their fitness, and iteratively selecting the most promising candidates for subsequent generations. Genetic approaches prove particularly efficient at exploring expansive search spaces and harnessing randomness effectively to increase the probability of optimal solution discovery. Papaki's thesis establishes a foundational framework that informs this current study.

His thesis pioneered the use of genetic algorithms for automating the search for efficient cache replacement policies, with a particular focus on finding the single best policy for a given set of twenty benchmarks. His approach systematically evaluated a population of potential policies by simulating their performance and refining them through the genetic algorithm's processes of selection, crossover, and mutation. The end goal was to identify one optimal replacement policy that would maximize the IPC speedup over the baseline LRU and RRIP policies.

Papaki's work laid the foundation for understanding how individual cache replacement policies could be optimized. However, it operated under the assumption that a single policy could be sufficiently robust across various workloads. This approach is akin to seeking a universal key that fits multiple locks — a challenging task, given the diversity of application behaviors and access patterns.

Building upon Papaki's methodology, the current research takes the exploration a step further by investigating the potential benefits of combining two distinct replacement policies. Rather than searching for a universal solution, it acknowledges that different workloads may require different strategies and that a combination of policies might better adapt to these varying demands. In this expanded approach, the genetic algorithm generates pairs of policies (combinations-of-two / dual) and tests each pair's effectiveness together. The simulation assesses which paired policies yield the highest cache performance, particularly in terms of IPC, across the same suite of benchmarks as Papakis. The critical innovation here is the wonder if the combined strengths of two policies will outperform the single best policy for certain workloads. This approach recognizes that cache misses have different characteristics and costs, and thus, a more nuanced strategy may provide a more tailored and efficient caching solution. It's an exploration of cache replacement policy synergy — how two policies might complement each other to cover a broader spectrum of workload behaviors more effectively than one policy alone.

In summary, while Papaki's research sought the best singular policy, the current thesis expands the horizon to consider pairs of policies, exploring the concept of hybrid vigor in cache replacement policies to achieve potentially superior performance across diverse computational scenarios.

Chapter 3

Frameworks and methodology

3.1 The HPC system..........14
3.2 Benchmark suit..........14
3.3 ChampSim simulator..........15
  3.3.1 The simulator..........15
  3.3.2 The configuration..........16
  3.3.3 Focus on the LLC..........17

3.1 The HPC system

The experiments were conducted on a high-performance computing (HPC) system that is hosted in the University of Cyprus. It is powered by Intel® Xeon® Gold processors, each operating at a base frequency of 2.90 GHz. The system is structured with a dual-socket setup featuring 32 physical cores, optimized for parallel computation. It runs on CentOS Linux version 7.8.2003, offering a stable and efficient environment for the extensive simulations required in this research. This robust computing platform is instrumental in effectively exploring and evaluating cache replacement policies.

3.2 Benchmark suit

In this research, we employ the same benchmark suite as the prior work (section 2.6) for consistent result comparison. Using different traces might render the results incomparable. We utilize the SPEC 2017 suite, known for stressing the system processor and memory subsystem — ideal for evaluating cache replacement policies. Conducting simulations with all benchmarks from the suite would be impractical due to time constraints, so Papaki selected 20 benchmarks for the experiments. Table 3.1 lists the benchmarks used under “All Benchmarks”. Moreover, in his research he observed that some benchmarks were not significantly influenced by the change in the replacement policy and ended up excluding them from the benchmark suit and conducting another experiment on the narrowed down suit. Table 3.1 lists the narrowed down suit under “Final Benchmarks”. The executed benchmarks are Bwaves, Exchange, Leela and Povray.

All Benchmarks		**Final Benchmarks**	
Blender	Bwaves	Blender	Imagick
cactusBSSN	Cam4	cactusBSSN	Omnetpp
Exchange	Fotonik3d	Gcc	Perlbench
Gcc	Imagick	Lbm	Roms
Lbm	Leela	Mcf	x264
Mcf	Omnetpp	Parest	Xz
Parest	Perlbench	Wrf	———
Povray	Roms	Xalancbmk	———
Wrf	x264	Fotonik3d	———
Xalancbmk	Xz	Cam4	———

Table 3.1 The slice of SPEC 2017 that is used

3.3 ChampSim simulator

3.3.1 The simulator

ChampSim [2] is a trace-driven microarchitectural simulator extensively utilized in various competitions to evaluate new concepts. Designed to replicate the behavior of out-of-order processors with a three-level cache hierarchy, ChampSim is an ideal tool for our research, offering a robust platform to assess the impact of cache replacement policies on system performance. It generates critical metrics essential for evaluating these policies and analyzing their effectiveness. Key metrics provided include hit and miss rates for writebacks, RFOs (Read For Ownership), and loads, as well as Instructions Per Cycle (IPC), the primary metric used to evaluate the efficiency of each candidate cache replacement policy. This plethora of information facilitates a comprehensive analysis of which policies perform best and explores the underlying reasons for their performance.

3.3.2 The configuration

For our research, we adopted a consistent simulation setup using ChampSim to ensure the validity and comparability of our results. By maintaining a uniform hardware configuration, any variations in performance outcomes can be directly attributed to the differences in cache replacement policies being evaluated, rather than variations in hardware settings.

We modeled a single-core, out-of-order processor equipped with a three-level cache hierarchy. This specific setup reflects a typical modern CPU environment, allowing us to accurately assess the performance impacts of various cache replacement strategies under realistic conditions.

Here is a breakdown of our simulation configuration:

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	256KB, 64B Blocks, 8-way

(Private)	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 Simulator's configuration

3.3.3 Focus on the LLC

ChampSim includes a significant feature that is essential for the analysis that follows the simulations: changes to the replacement policy only affect the Last Level Cache (LLC) and not the higher-level caches. This specific behavior is demonstrated in Figures 3.1 and 3.2, which visualize the output of a targeted experiment conducted to confirm this feature. This is the case for every metric of caches L1D (L1 Data), L1I (L1 Instructions) and L2. I choose to provide the Total Misses which is the first metric of interest when evaluating the performance on cache replacement policies. Recognizing this allows us to concentrate our analysis exclusively on data from the LLC, ignoring other cache levels. This focus helps simplify and accelerate the process of evaluating individual cache replacement policies, making our analysis more effective and efficient. Note that 1-2 and 2-3 are just the names I gave to the policies I used to run the experiment.

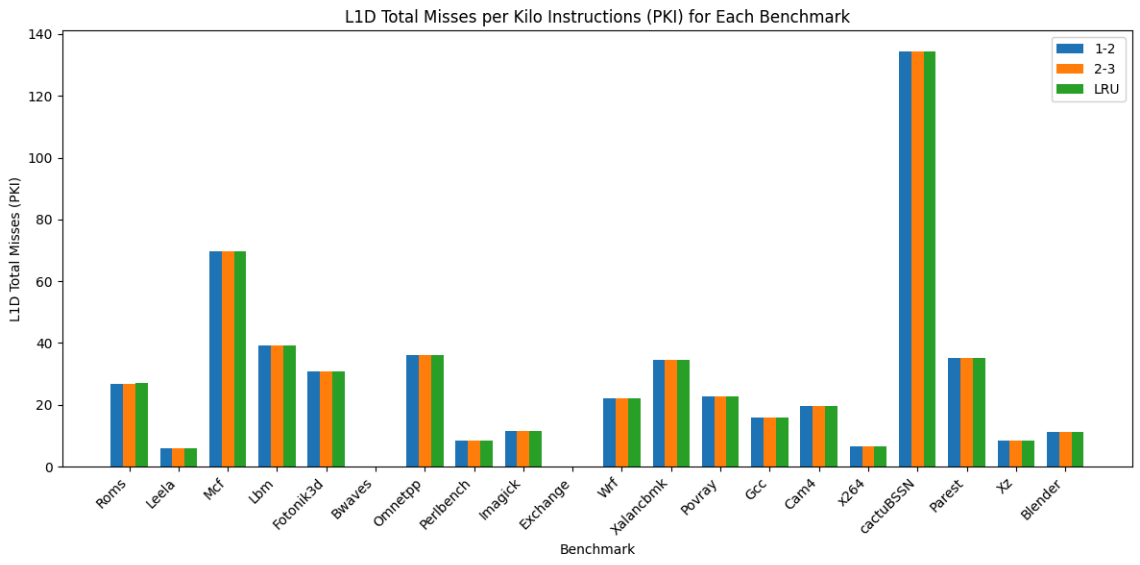


Figure 3.1 L1 Data cache Total Misses

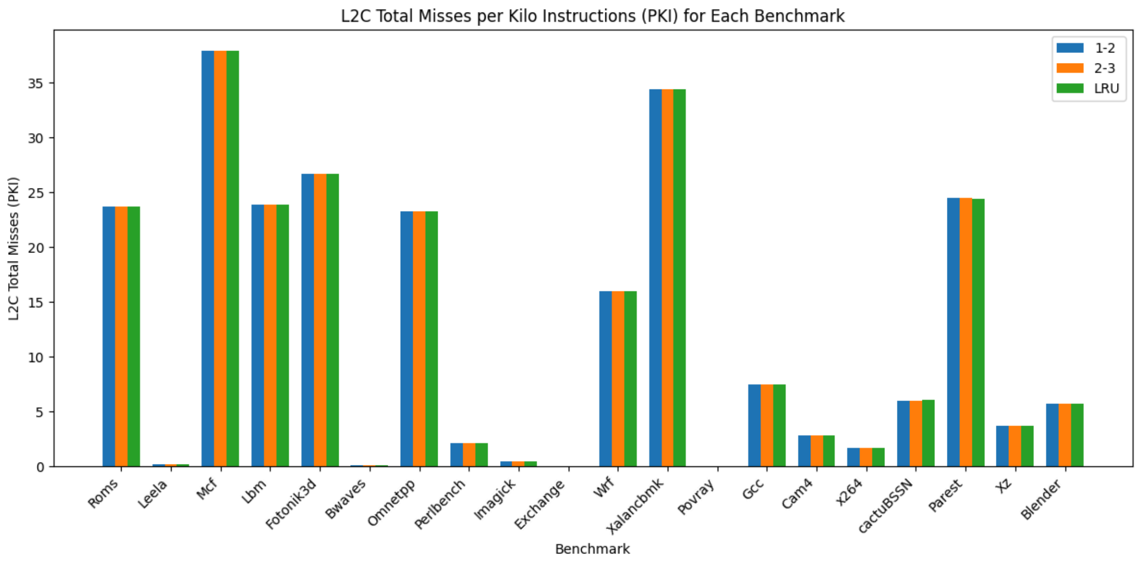


Figure 3.2 L2 cache Total Misses

Chapter 4

DRRIP-base Replacement policy

4.1 Cache Scan and Thrash……………………………………………………………19
4.2 Set Dueling...…………………………………………………………………...20
4.3SRRIP-DRRIP…………………………………………………………………..20

To understand the functionality of the Modular DRRIP model within the simulator's environments, it is beneficial to first familiarize ourselves with certain key terms and phenomena related to cache mechanisms. This foundational knowledge will enhance our understanding of how the model operates and its implications for our research.

4.1 Cache Scan and Thrash

Scans and thrashes are two phenomena that significantly impact the performance of cache memory in computing systems, especially in how they manage data.

A scan occurs when a sequence of cache accesses involves data blocks that are not likely to be reused soon after they are accessed. These are typically sequential or one-time access patterns where each block is used briefly and then not needed for a considerable period. Scans can disrupt cache performance because they tend to fill the cache with data that has a long re-reference interval, potentially displacing other data that will be needed more immediately. This can lead to increased cache misses for subsequent accesses that involve frequently used data. Replacement policies need to be scan-resistant, meaning they should recognize and minimize the retention of blocks used in scans.

Thrashing happens when the working set of a program — the set of data it needs to access frequently — is larger than the cache can hold. As a result, the cache experiences a high rate of cache misses, leading to frequent loading of data into the cache and eviction of existing data. This excessive loading and eviction cycle can severely degrade the performance, as the system spends more time managing cache contents rather than executing instructions. Thrashing indicates a fundamental mismatch between the cache capacity and the demands of the workload.

4.2 Set Dueling

Set Dueling involves using some sets of the cache, known as dedicated sets, to track the performance of competing replacement policies. Each policy operates in its dedicated sets to provide real-time data on its effectiveness. A Policy Selector counter (PSEL) adjusts based on the relative performance of these policies; increases or decreases depending on which policy shows fewer cache misses. The policy that shows better performance in its dedicated sets becomes the 'winner' and is applied to the majority of the cache, known as follower sets.

4.3 SRRIP - DRRIP

The Re-Reference Interval Prediction (RRIP) cache replacement policy is designed to address limitations found in traditional policies like Least Recently Used (LRU), particularly its vulnerability to cache thrashing and scanning. RRIP operates by assigning a re-reference value (RRIP value) to each cache block, indicating the predicted time until the block will be needed again (see section 2.3). This value helps decide which blocks to replace upon a cache miss. RRIP is divided into two subversions, Static and Dynamic.

Static RRIP (SRRIP) uses a fixed number of bits per block to manage the re-reference interval prediction. It is resistant to scan operations because it tends to assign high re-reference interval values to newly inserted blocks, indicating they are less likely to be needed soon. This means that when a scan operation brings in a series of new blocks that are accessed once and then not used again, SRRIP quickly identifies these blocks as low-utility (with higher RRIP values) and prioritizes their eviction. This prevents the scan from displacing blocks that have a higher likelihood of reuses. However, it can struggle with cache thrashing when the working set exceeds the cache size. By adjusting the bit size (M), SRRIP can balance between near and distant future references, enhancing its predictive accuracy and retention of the active working set.

Dynamic RRIP (DRRIP) enhances the basic RRIP by dynamically selecting between two different policies, typically a conservative and an aggressive one, using Set Dueling. DRRIP can switch between SRRIP-like behavior and Bimodal RRIP (BRRIP), which tends to assume blocks have distant re-reference intervals unless indicated otherwise. BRRIP is particularly effective against thrashing by favoring the eviction of blocks presumed to be less likely needed soon. This makes the DRRIP scan and thrush resistant.

The modularity and flexibility of DRRIP makes it suitable to use in our experiments as we can change it to achieve the preferred function. It is easy to change its behavior according to the parameters given to the simulator.

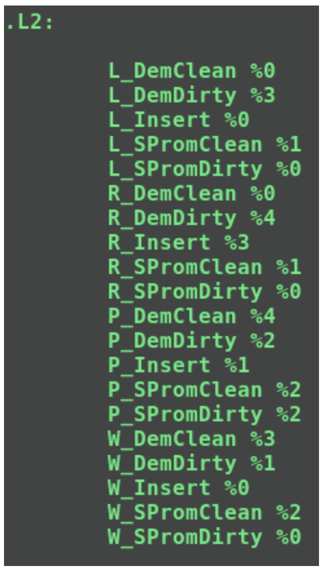
Chapter 5

GeST: The Genetic Algorithm

5.1 Traditional Approach……………………………………………………………...22
5.2 Policy Encoding…………………………………………………………………...23
5.3 The search space…………………………………………………………………..26

5.1 Traditional Approach

GeST (Genetic Stress Tests) is an automatic framework originally designed to generate high-load scenarios for CPU stress testing through the application of genetic algorithms. GeST's core methodology is to evolve a population of test cases, or "individuals", each representing a unique stress test scenario, to systematically explore and optimize CPU performance under intense computational loads. The framework operates by generating an initial population randomly, assessing these individuals based on performance metrics, and iteratively refining them through genetic operations such as selection, crossover and mutation. In the framework of genetic algorithms, the fitness function evaluates each candidate solution, or "individual," assessing its suitability and effectiveness in solving the given problem. This evaluation ensures that the most promising individuals are selected to continue in the evolutionary process. Mutation introduces variability within the population through random changes to individual components of a solution, like instructions or operands, according to a predetermined probability. This randomness helps maintain genetic diversity, preventing the premature convergence to suboptimal solutions. Crossover, a method of genetic recombination, involves merging segments from two parent individuals to create new offspring. This process combines traits from both parents, potentially yielding more viable solutions in the next generation. Together, these mechanisms drive the evolutionary progress of the population towards an optimal solution. In the context of the prior work, GeST was used to generate and evolve a population of single-policy individuals, meaning that each individual in the simulation was representing a single cache replacement policy. A single-policy individual would provide the set of parameters to the simulator which are shown in the figure 5.1.



```
.L2:

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

Figure 5.1 A single-policy individuals' policy encoding

5.2 Policy encoding

It is essential to clarify that a replacement policy is composed of several elemental policies, specifically the insertion policy, self / other promotion policy, eviction policy and demotion policy. The definitions for the listed policies are given in Table 5.4. Figure 5.1 illustrates the encoding of a single replacement policy. This encoding method is also employed in this research to define a single replacement policy. In the figure, “L” stands for Load access, “R” for RFO (Read For Ownership), “P” for Prefetch and “W” for Wrightback which categorize the four types of memory accesses (see section 2.1). The elemental policies within a replacement policy are specified as follows: "Insert" for insertion, "Prom" for promotion, and "Dem" for demotion. Individually, these elemental policies do not define anything substantive; however, when combined, they define the replacement policy. To alter a replacement policy — or to mutate it into another — at least one of the elemental policies must be changed. Additionally, as depicted in Figure 3.3, the policies differentiate between operations on clean and dirty blocks, specifying distinct approaches for demotion and promotion of each. On the other hand, insertion policies are not distinguished by block cleanliness. The range of values for each elemental policy's parameters are detailed in Tables 5.1 to 5.3, as presented in Papaki's work. For a deeper understanding, readers are referred to his paper [5].

Insertion:

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

Table 5.1 The range of values a parameter for insertion policy can take

- Assign to 0 (0): New blocks get an RRIP value of 0, indicating high utility.
- Assign to 1 (1): New blocks get an RRIP value of 1, indicating high to moderate utility.
- Assign to 2 (2): New blocks get an RRIP value of 2, indicating moderate to low utility.
- Assign to 3 (3): New blocks get an RRIP value of 3, indicating low utility.

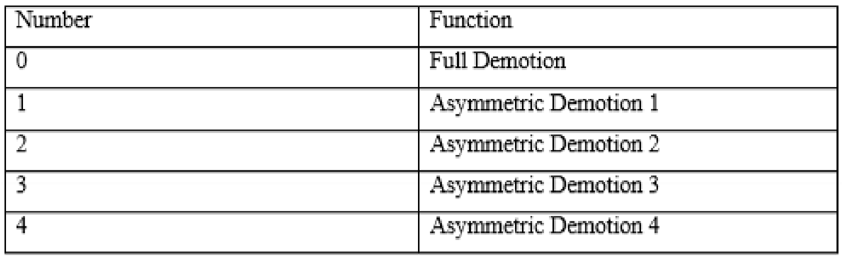
Promotion:

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

Table 5.2 The range of values a parameter for promotion policy can take

- Assign to 0 (0): The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.
- Assign to 1 (1): The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.
- Assign to 2 (2): The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.

Demotion:



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

Table 5.3 The range of values a parameter for demotion policy can take

- Full Demotion (0): Increases the RRIP value of all blocks in the set until a block is evicted.
- Asymmetric Demotion 1 (1): Increases the RRIP value of blocks by 1 until the maximum RRIP value in the set is greater than 1.
- Asymmetric Demotion 2 (2): Increases the RRIP value of blocks by 1 until the maximum RRIP value in the set is greater than 2, but only for blocks with an RRIP value less than 2.
- Asymmetric Demotion 3 (3): Similar to Asymmetric Demotion 2, but if the maximum RRIP value is 0, all blocks get an RRIP value of 2.
- Asymmetric Demotion 4 (4): If the maximum RRIP value is 0, all blocks get an RRIP value of 2. If the maximum RRIP value is 1, all blocks' RRIP values are increased by 1.

Elemental Policy	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 5.4 Definitions of elemental policies

5.3 The search space

The examination of Tables 3.3 to 3.5 reveals a variety of options for configuring replacement policies: four choices for insertion, three for promotion, and five for demotion. Drawing from the details presented in Figure 3.4 and Tables 3.3 to 3.5, we establish the following configurations available for constructing a policy group for one type of access:

- Five options for clean block demotion
- Five options for dirty block demotion
- Four options for insertion
- Three options for clean block promotion
- Three options for dirty block promotion

To synthesize a cache replacement policy, which involves assembling policy groups for all types of accesses, the total search space is calculated as follows:

$5 \times 5 \times 4 \times 3 \times 3 = 900$ combinations (for one type of access),

Given four types of accesses, this results in:

$900^4 = 6.561 \times 10^{11}$ total policies

This is equal to 656,100,000,000, or six hundred fifty-six billion, one hundred million potential policies. To put the enormity of this number into perspective, if each policy evaluation required 30 minutes, it would take approximately 374486 centuries to test every possible policy. Clearly, this number of evaluations is impractical, underscoring the necessity of a method to efficiently reduce the search space. A genetic algorithm serves this purpose effectively, providing a robust tool for navigating and optimizing within this expansive policy landscape.

Chapter 6

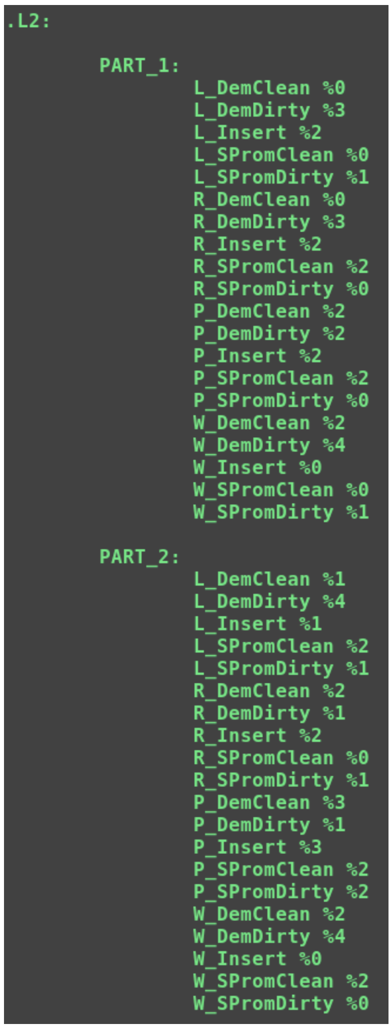
Dual-Policy Modifications

6.1 Modifications……………………………………………………………………...27

6.2 Fitness evaluation, Mutation and Crossover………………………………………28

6.1 Modifications

For our specific research needs, focusing on evaluating dual-policy individuals, significant customizations were implemented within the GeST framework. Unlike traditional single-policy tests, our study required the generation and evaluation of test cases that incorporate two distinct cache replacement policies simultaneously. This approach was aimed at identifying synergistic effects between paired policies that could potentially yield superior performance compared to single-policy configurations. We adapted the genetic algorithm within GeST to handle pairs of policies as single test cases. The class that implements the individuals had to be adjusted. Each individual in the population now consists of two distinct sets of assembly instructions, as shown in figure 6.1. This structure necessitated modifications to the fitness evaluation function. Furthermore new mutation and crossover operations had to be implemented to handle the dual-policy individuals definition (section 6.2 explains dual-policy fitness function, mutation and crossover in depth). These modifications have transformed GeST into an even more dynamic tool capable of handling more complex evaluation scenarios, broadening the scope of possible research into cache replacement strategies. By facilitating the exploration of policy combinations, the adapted framework supports a deeper understanding of how different policies interact within the same operational context, providing insights that are critical for optimizing modern processors. This enhanced capability allows us to closely mimic real-world applications and more accurately predict the performance of dual-policy strategies in practical settings. The ability to simulate and analyze such complex interactions within a controlled environment underscores the flexibility and power of the GeST framework in advancing microarchitectural research and development.

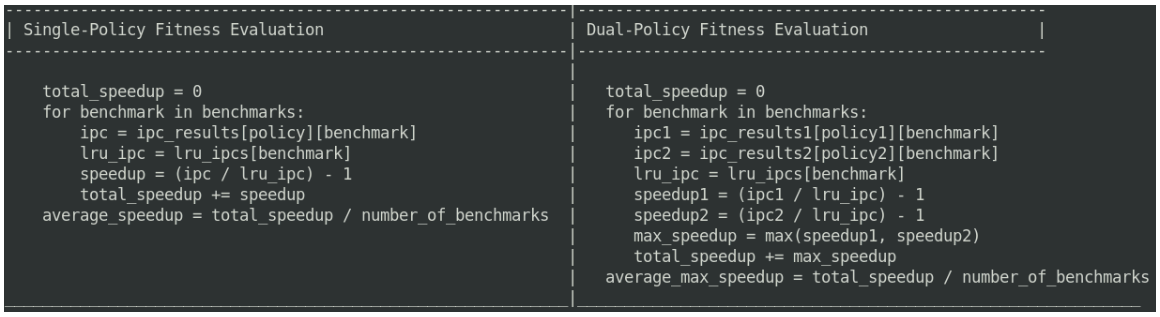


```
.L2:

        PART_1:
                L_DemClean %0
                L_DemDirty %3
                L_Insert %2
                L_SPromClean %0
                L_SPromDirty %1
                R_DemClean %0
                R_DemDirty %3
                R_Insert %2
                R_SPromClean %2
                R_SPromDirty %0
                P_DemClean %2
                P_DemDirty %2
                P_Insert %2
                P_SPromClean %2
                P_SPromDirty %0
                W_DemClean %2
                W_DemDirty %4
                W_Insert %0
                W_SPromClean %0
                W_SPromDirty %1

        PART_2:
                L_DemClean %1
                L_DemDirty %4
                L_Insert %1
                L_SPromClean %2
                L_SPromDirty %1
                R_DemClean %2
                R_DemDirty %1
                R_Insert %2
                R_SPromClean %0
                R_SPromDirty %1
                P_DemClean %3
                P_DemDirty %1
                P_Insert %3
                P_SPromClean %2
                P_SPromDirty %2
                W_DemClean %2
                W_DemDirty %4
                W_Insert %0
                W_SPromClean %2
                W_SPromDirty %0
```

Figure 6.1 A dual-policy individual.
Each part represents one single policy

6.2 Fitness evaluation, mutation and crossover

In the context of the genetic algorithm applied to cache replacement policies, the evolution from single-policy to dual-policy individuals necessitates a modification in the fitness evaluation process. Originally, the fitness of single-policy individuals was assessed by computing the average IPC speedup over the baseline LRU across all benchmarks. With the introduction of dual-policy individuals, the fitness evaluation method has evolved to incorporate a more comprehensive approach. For each dual-policy individual, the IPC value is determined for each benchmark under both policies. Then, the maximum IPC value between the two policies is selected and its speedup over the baseline LRU is the representative fitness value for that particular benchmark. The fitness score for the dual-policy individual is then calculated as the average of these fitness scores across all benchmarks. The rationale for selecting the maximum IPC value for each benchmark is to ensure that each benchmark utilizes the policy that yields the best performance in terms of IPC speedup over the baseline LRU policy. This approach guarantees that the evaluation process considers the highest possible performance of each dual-policy individual for every benchmark, thereby optimizing the selection process for policies that can deliver the best results under varied conditions. This method ensures a robust evaluation by considering the best performance potential of each dual-policy individual across the benchmarks. To illustrate the differences in the fitness function implementation between single-policy and dual-policy individuals, pseudocode examples are provided in Figure 6.2.



```
------------------------------------------------------------|------------------------------------------------
| Single-Policy Fitness Evaluation                          | Dual-Policy Fitness Evaluation                |
------------------------------------------------------------|------------------------------------------------
                                                            |
    total_speedup = 0                                       |   total_speedup = 0
    for benchmark in benchmarks:                            |   for benchmark in benchmarks:
        ipc = ipc_results[policy][benchmark]                |      ipc1 = ipc_results1[policy1][benchmark]
        lru_ipc = lru_ipcs[benchmark]                       |      ipc2 = ipc_results2[policy2][benchmark]
        speedup = (ipc / lru_ipc) - 1                       |      lru_ipc = lru_ipcs[benchmark]
        total_speedup += speedup                            |      speedup1 = (ipc1 / lru_ipc) - 1
    average_speedup = total_speedup / number_of_benchmarks  |      speedup2 = (ipc2 / lru_ipc) - 1
                                                            |      max_speedup = max(speedup1, speedup2)
                                                            |      total_speedup += max_speedup
                                                            |   average_max_speedup = total_speedup / number_of_benchmarks
____________________________________________________________|________________________________________________
```

Figure 6.2 Pseudo code that shows the fitness function differences. In Appendix B you can see the full code.

The mutation operation, while largely preserved from its original form, required slight modification to accommodate dual-policy individuals. For single-policy individuals, mutation involves altering a parameter, as depicted in Figure 5.1, based on a predetermined probability defined in the inputs for the GeST framework. To extend this operation to dual-policy individuals, the procedure now begins with a random selection between the two policies as illustrated in Figure 6.2. Mutation is then applied exclusively to the parameters of the selected policy. This targeted approach moderates the introduction of randomness within the evolutionary process, a crucial adjustment given that the GeST framework inherently incorporates a significant degree of randomness.

Significant changes were necessary for the crossover operation, due to the complexity introduced by handling two sets of policy parameters per individual. Previously, for single-policy individuals, as shown in Figure 6.3, the method involved splitting each parent's parameters in half and mixing these halves to create two new individuals, ensuring that segments matched to preserve the parameter order.

Transitioning to dual-policy individuals introduces more complicated options for crossover. 1) The first approach is similar to the earlier method, splitting both policies of each individual into two, creating four halves, and then combining these to produce two new individuals. A child would inherit the upper or lower halves of both policies from the same parent. 2) Another method considers each policy as a unit for crossover, where each child receives one complete policy from each parent, mixing policies between two parents. 3) The most complex approach proposed involves cutting each policy into halves and examining all combinations of these quarters. This method generates four potential children from two parents, aiming to increase the diversity of the resulting individuals. 4) In an intuitive sense, combining the last two approaches appears to offer the most effective strategy. During the initial stages of evolution, where diversity and exploration are crucial, employing approach 3) encourages a broader exploration of potential solutions. As the individuals evolve and achieve higher fitness scores, transitioning to approach 2) could be beneficial.

The intention behind examining such varied crossover configurations was to identify the most effective strategy. Due to time constraints, a full exploration of all potential configurations was not possible. The decision to adopt the last method was influenced by the need to enhance genetic diversity among the offspring. It was anticipated that simpler crossover methods might result in offspring too similar to their parents, thus requiring more generations to achieve an optimal solution. This was a concern given the limited number of generations available in the study conducted by Papakis. Figure 6.4 outlines the revised mutation and crossover processes, reflecting the second suggestion from the previous paragraph while Figure 6.5 the third. The combination of these two Figures (6.4 and 6.5) show the final crossover strategy used in my dual-policy experiments. For the first half of generations (the first 25, section 5.1) the suggestion 3) is employed and for the rest (last 25) the suggestion 2) is employed.

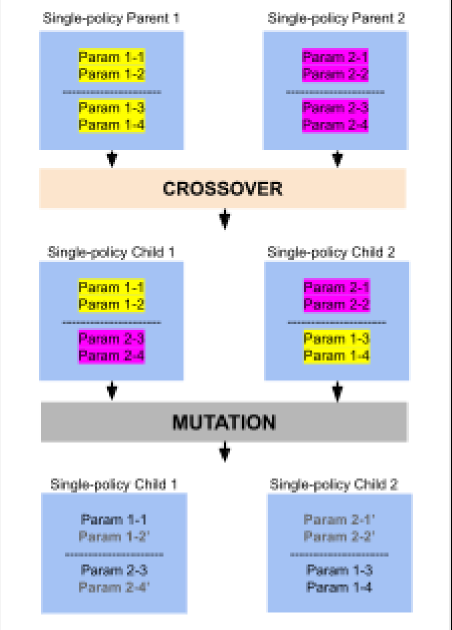


Figure 6.3 Single policy mutation and crossover

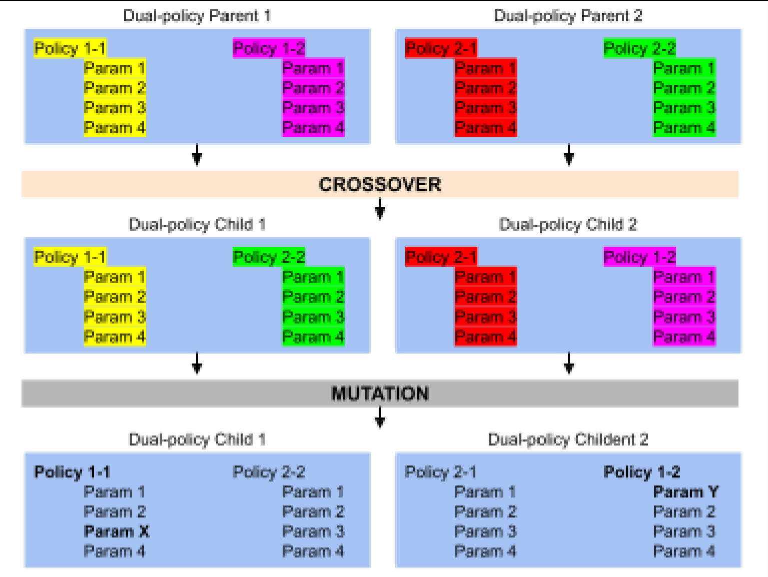


Figure 6.4 Dual-policy mutation and crossover suggestion 2) of section 6.2

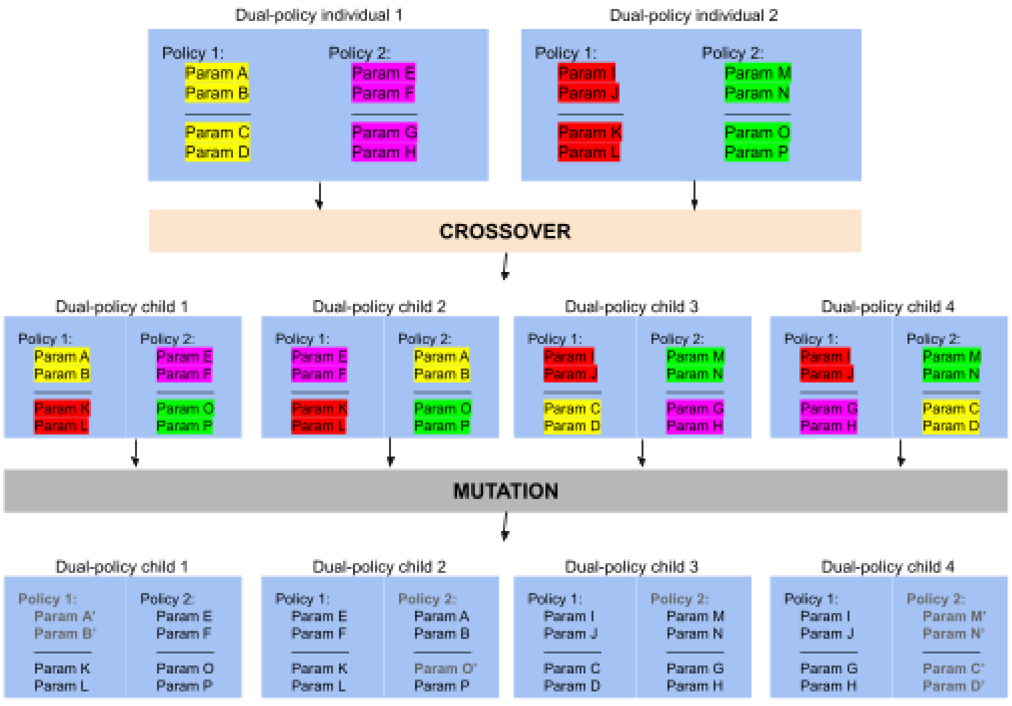


Figure 6.5 Dual-policy mutation and crossover as described in the approach 3) of the section 6.2. In Appendix A you can see a crossover operation on actual individuals from the experiment.

Chapter 7

Experiments and Results

7.1 Experiments' Set up……………………………………………………………….32

7.2 Results…………………………………………………………………………..34

7.2.1 First experiment: Proof of concept……………………………………...34

7.2.1.1 Benchmark Analysis..…………………………………………...36

7.2.1.2 Conclusions……………………………………………………...42

7.2.2 Second experiment: Evolve dual-policy individuals……………………...43

7.2.2.1 Analysis of the best……………………………………………...44

7.2.2.2 Benchmarks examination………………………………………..46

7.2.2.2.1 Dive into Lbm…...…………………………………….46

7.2.2.3 Conclusions……………………………………………………...52

7.1 Experiments' Set up

Following the methods from the previous research, I tested each candidate cache replacement policy by running it through a series of benchmarks using ChampSim (section 3.2). Each benchmark included two key phases: a warm-up phase with one hundred million instructions and a main simulation phase that used five hundred million instructions. The warm-up phase is critical in simulation environments as it allows the cache to reach a state representative of typical workload conditions, thereby ensuring that subsequent measurements are accurate and reflective of steady-state behavior. A large number of warm-up instructions helps to make sure that the initial adjustments in the simulation don't affect the final results. For my thesis, I followed the same setup to allow for direct comparison with the earlier research. This setup is important for making sure that any results we observe are due to differences in the replacement policy and not differences in how the experiments were run. However, aiming at conducting a more time efficient, focused, and thorough analysis, I focused my experiments on the narrowed-down set of benchmarks (see section 3.2).

In my experiment, the individuals that constituted the first generation were generated randomly. Consequently, the results from this setup are directly compared with outcomes from the previous study that employed the same approach. The notion of starting with a first generation that includes some pre-selected, promising individuals, which has been previously tested, presents an intriguing avenue for future research (see section 8.1). In the prior research, experiments were structured around evolving the initial randomly created replacement policies across 50 generations, with each generation consisting of 50 individuals. This approach might suggest a straightforward replication — 50 generations with 50 individuals each. To understand the scale of the experiments, consider the basic calculation for single-policy individuals: 50 generations multiplied by 50 individuals per generation gives us a total of 2500 replacement policies to be tested (50 × (50 × 1) = 2500). Applying the same numbers to experiments involving dual-policy individuals — where each individual encapsulates two distinct replacement policies — would double the total count. Here, the calculation would be 50 generations multiplied by 50 individuals per generation, each representing two policies, resulting in 5000 replacement policies to be tested (50 × (50 × 2) = 5000). This doubling in the number of tested policies could skew the results. If the best-performing dual-policy individual exhibited superior performance, it could simply be due to the increased number of policies tested, rather than an intrinsic improvement in policy effectiveness. To ensure that the results are directly comparable with those of the prior study, it was necessary to adjust the experiment's setup to equalize the total number of tested policies. Thus, I conducted the experiments over 50 generations, similar to the previous study, but each generation consisted of 25 dual-policy individuals. This configuration results in the same total number of replacement policies being tested as in the single-policy scenario:

50 generations × (25 dual-policy individuals × 2 policies per individual) = 2500 policies.

Lastly, in the genetic algorithm, we use a method called tournament selection to pick two individuals from the current generation [6]. These individuals will be the parents for the next generation's children. In this method, we randomly pick some individuals and split them into two groups of the same size. Then, we choose the best performer from each group as a parent. Previously, each group had five individuals. So, from 50 single-policy individuals, we picked 10 to form parent pairs for the next generation, which is one-fifth of the total. But now we have 25 dual-policy individuals (half of the original 50), so we needed to change the group size for choosing parents. Since we couldn't pick 2.5 individuals (half of five), we had to choose between 2 or 3. I chose 2 individuals for each group to avoid picking the same pair of parents repeatedly thus generating the same children.

7.2 Results

7.2.1 First experiment: Proof of concept

In our initial experiment, we focused on evaluating single-policy individuals to establish a performance baseline. This approach involved assessing each policy's effectiveness in improving IPC compared to the baseline LRU policy. The results of this experiment identified the best-performing individual policy and quantified its IPC speedup over the LRU. The average IPC speedup of the best policy over the LRU was found to be 1.258%, which aligns with the findings of prior research. To illustrate the potential benefits of adopting dual-policy individuals, we conducted a follow-up analysis. Specifically, we compared the IPC performance of the top individual policy to a hybrid approach where the best combination of two single-policies of the total 2500 individuals was used as the baseline dual-policy. The average IPC speedup over the LRU of that baseline was found to be 1.63%. This translates to 29.6% improvement over the Best single-policy. The individual speedups of the two single policies used were -0.33% and 1.257% respectively.

The plot in Figure 7.1 provides a visual representation of our findings. It plots the IPC speedup for each benchmark, comparing the maximum IPC obtained from the two single-policies used to form the baseline dual-policy against the LRU. The chosen policy's speedup value is annotated in the corresponding color for each benchmark. The observation is that the two single-policies are complementary to each other meaning that for benchmarks one has low IPC speedup the other has high IPC speedup. Some cases are benchmarks Xalancbmk and Parest. Figure 7.2 showcases the curve of the baseline dual-policy with green while Figure 7.3 compares the Baseline dual-policy (green) with the Best single-policy (orange).

As it depicted in Figure 7.3 the, the Baseline Dual-policy demonstrates relatively similar performance as the Best single-policy for most benchmarks. However, in the case of benchmark Lbm the Baseline dual-policy exhibits significantly better performance than the Best single-policy. Specifically, the Best single-policy achieves a 2.07% speedup over the LRU, whereas the Baseline dual-policy attains a 5.21% speedup, translating to a remarkable 151.7% improvement for the Lbm benchmark.

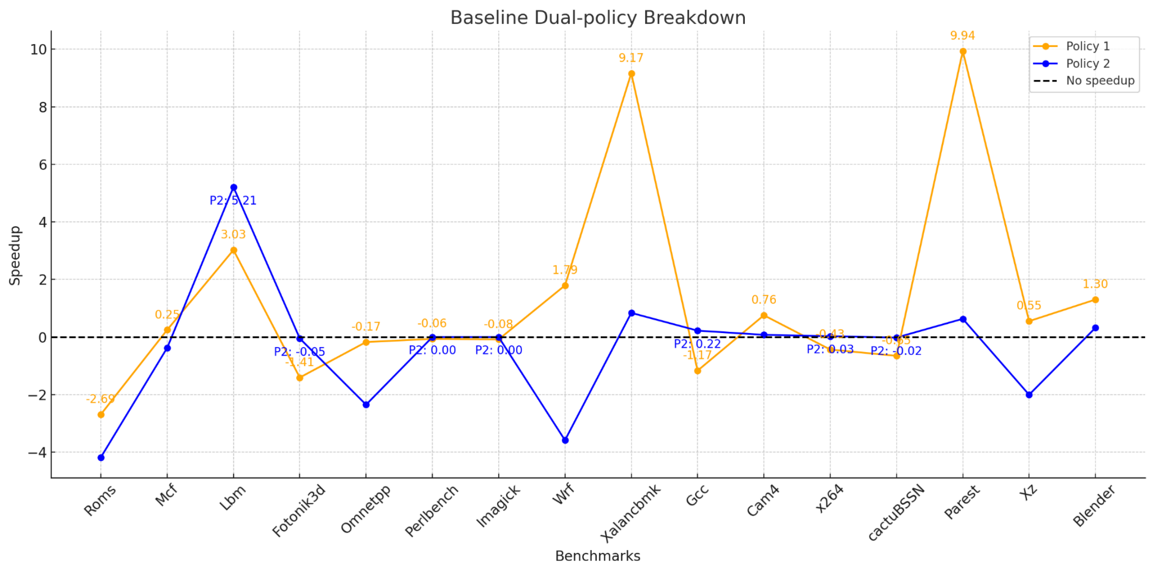


Figure 7.1 Baseline Dual-policy breakdown

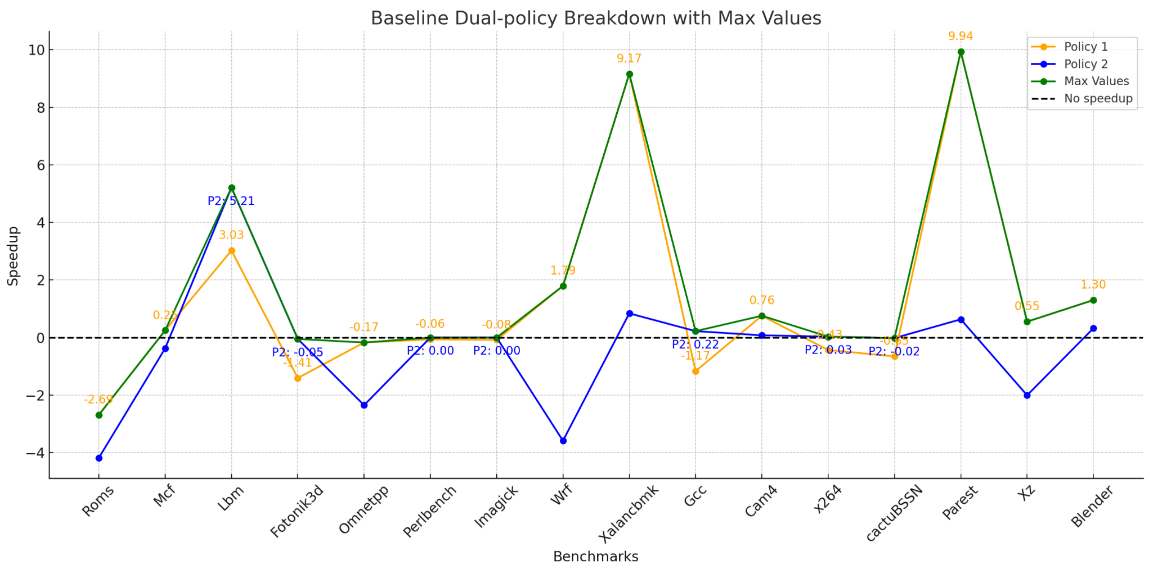


Figure 7.2 Baseline Dual-policy vs Best single policy

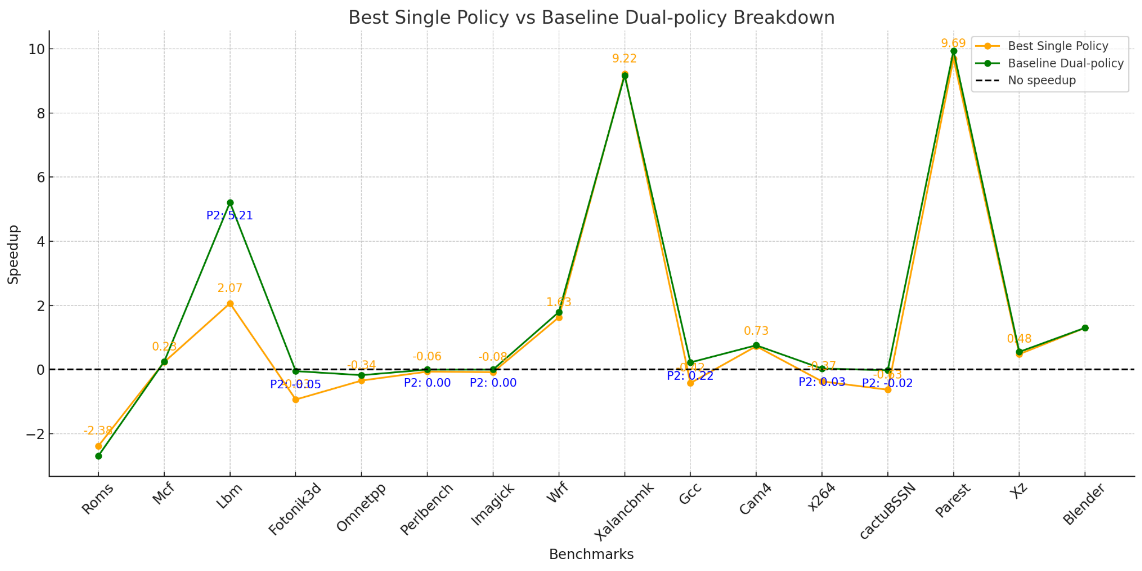


Figure 7.3 Best Single-policy VS Baseline Dual-policy P2 means Policy 2 from figure 7.1

7.2.1.1 Benchmark Analysis

To understand why the Baseline-dual policy yields better performance for the Lbm benchmark, we present a detailed comparison of LLC metrics (see section 3.3.3 to see why only LLC). These metrics include total accesses, hits, misses, and specific types of accesses (load, RFO, prefetch, writeback) for both policies. This snapshot of the LLC provides insights into the behavior and efficiency of each policy. In Figure 7.4, the metrics for both the Best single-policy and Baseline dual-policy are shown side by side, allowing for a clear comparison. Each metric is transformed to per kilo-instruction (PKI) values for better readability and comparison. The L2 cache prefetch misses are included in the Total LLC misses as well. An L2 cache prefetch miss occurs when the L2 wants to prefetch data from the L3 cache and the data is not found thus the data has to be (pre) fetched from the main memory. This renders the prefetch operation of the L2 cache an LLC miss. The total L2 cache prefetch misses are included in the LLC Total Misses metric which is the reason behind the fact that the misses appear to be more than the total accesses to the LLC.

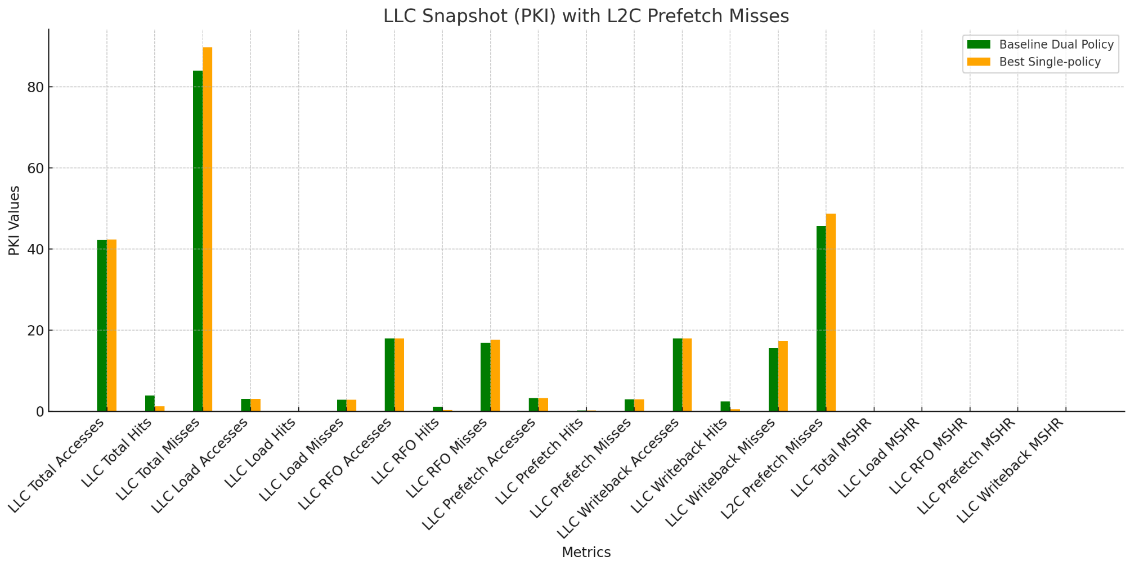


Figure 7.4 Full snapshot of the LLC for Lbm under both the best and the baseline policies

Upon examining the data, it is evident that the Baseline Dual-policy exhibits a lower number of Total LLC misses which has an effect on the overall performance of the policy. However this alone cannot explain the scale of the improvement shown for Lbm. Figure 7.5 shows a zoomed-in snapshot of the LLC making it clear that the access that affect the Total LLC misses are RFO and Wrightbacks.

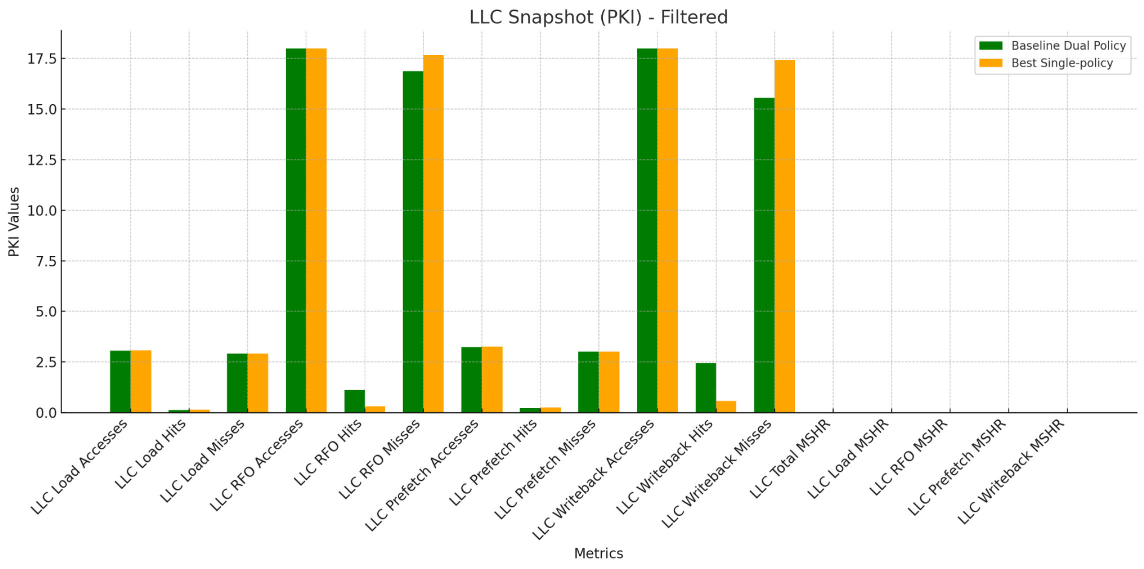


Figure 7.5 A "zoomed in" snapshot of the LLC from figure 5.3

First we examine Loads and RFOs, which are demand accesses. Both policies show the same number of load misses, indicating similar efficiency in handling load accesses. However, the vaseline dual-policy exhibits a lower number of RFO (Read for Ownership) misses, which are critical for performance as these demand accesses must be served for the program to proceed. The reduction in RFO misses under this policy directly contributes to its improved performance.

Then, we have Prefetch and Writeback accesses, which are not demand accesses. The prefetch metrics are identical for both policies, indicating similar behavior in prefetch handling. However, a difference is observed in the writeback metrics. The Baseline dual-policy demonstrates better performance in managing writebacks, with fewer misses compared to the best policy. On a writeback miss, the system must fetch the missing data from the main memory, which intoduces additional latency and increases memory traffic. This not only slows down the write operation but also forces the processor to wait for the data to be fetched, thus stalling it. Such stalls can prevent the processor from executing other instructions that depend on the completion of the write operation, leading to a decrease in overall system efficiency. By reducing the number of writeback misses, the Baseline dual-policy decreases the overhead of write operations, leading to improved overall performance.

When a miss occurs, the request is logged in an MSHR entry. If other misses occur for the same block of data before the first miss is serviced, these requests can be merged into the same MSHR entry rather than creating new ones. This method reduces the amount of accesses to lower in the hierarchy memories, in this case the slower main memory thus, yielding better IPC. The merged MSHRs for the two policies are shown in Figure 7.6. The Baseline dual-policy is not as efficient as the Best single-policy in terms of merged MSHR entries, but the total number of such entries for the two policies is too low to have a significant impact on IPC. Therefore, this observation is not a solid proof of any performance difference.

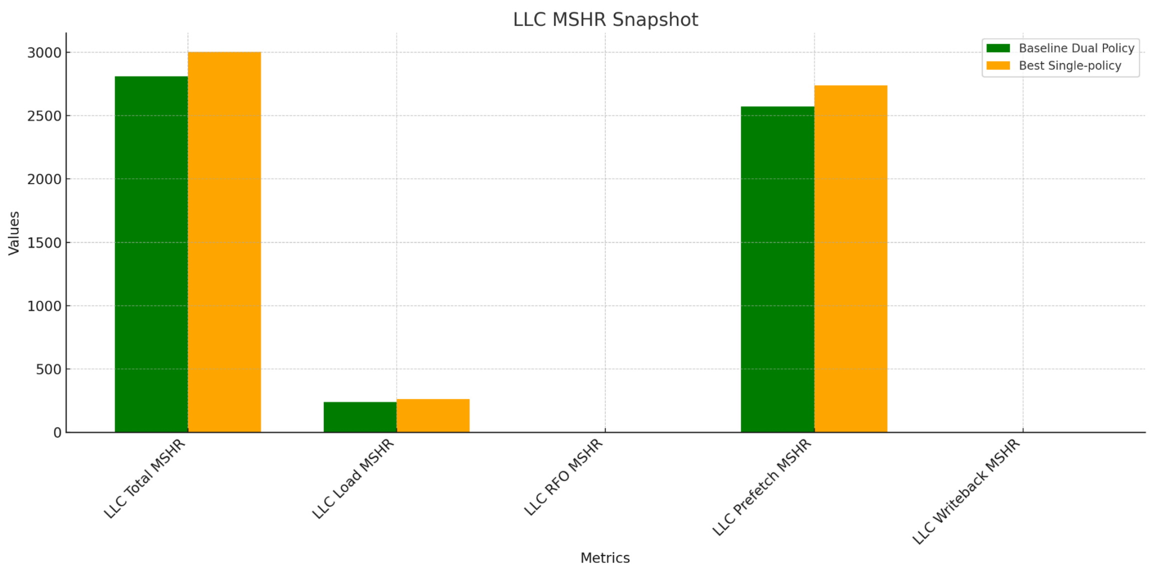


Figure 7.6 Merged MSHR entries

In order to come to a conclusion about the performance difference of the two policies for Lbm, we move to examine the number of useful prefetched blocks under each of them. We can see in Figure 7.7 that the two policies have very similar performance rendering this statistic not suitable to explain results.

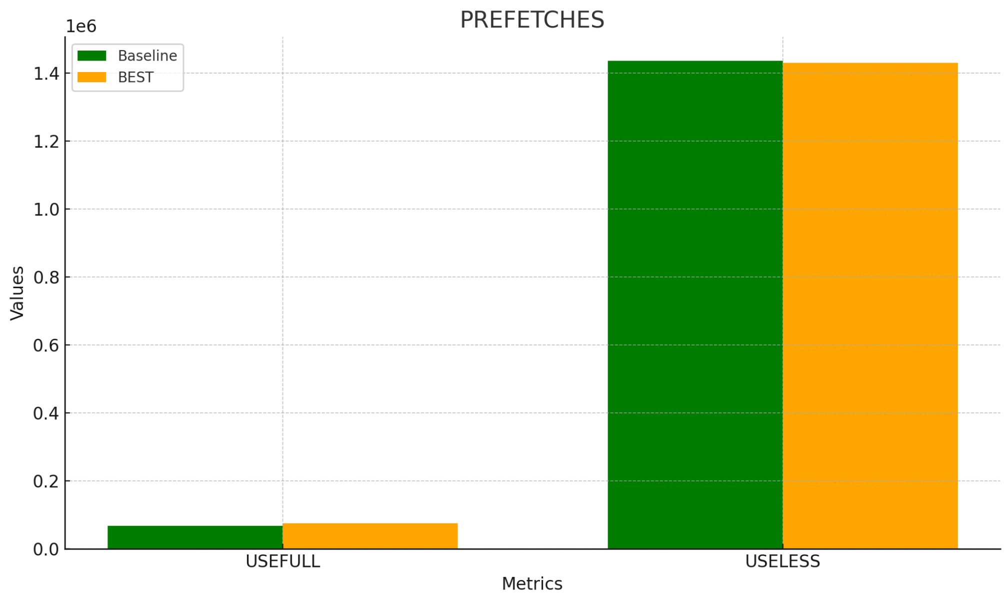


Figure 7.7 Useful VS Useless Prefetches

The last thing left to do is to examine the DRAM (main memory) statistics. First we examine the WQ and RQ roe buffer misses. The Write Queue (WQ) row buffer temporarily holds write data before it is written to DRAM. A row buffer miss occurs when the required data is not in the buffer, causing additional latency due to the need to write back the current row and read a new one. This increased latency reduces memory throughput, causes processor stalls, and ultimately lowers the Instructions Per Cycle (IPC), thereby degrading overall system performance. The same goes for the RQ (Read Queue). As depicted in Figure 7.8, the Baseline Dual-policy experiences fewer RQ and WQ row buffer misses, thus achieving better IPC due to lower latency when accessing the main memory.

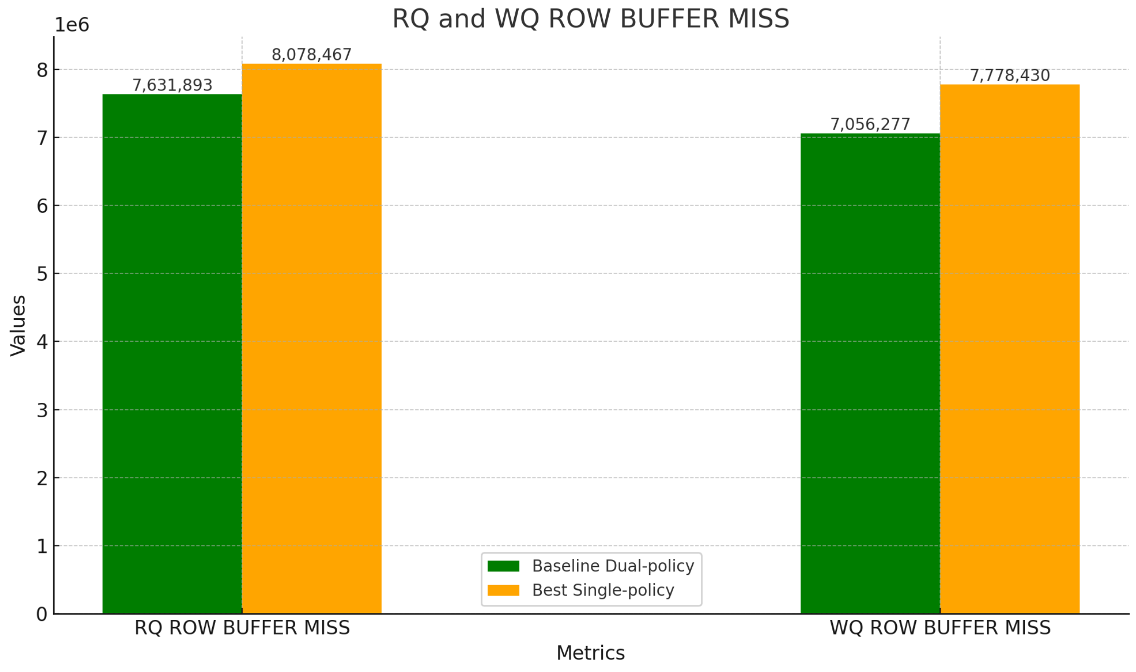


Figure 7.8 RQ and WQ row buffer misses

And lastly we examine the average number of cycles that each policy had to wait for the congested BUS of the DRAM. DBUS congestion occurs when the data bus is overloaded with requests, leading to delays in data transfer and increased latency due to queuing. Figure 7.9 shows the average wait time for the congested DBUS for the two policies. It is clear that the Baseline Dual-policy encountered DBUS congestion less frequently than the Best Single-policy, thus leading to better performance.

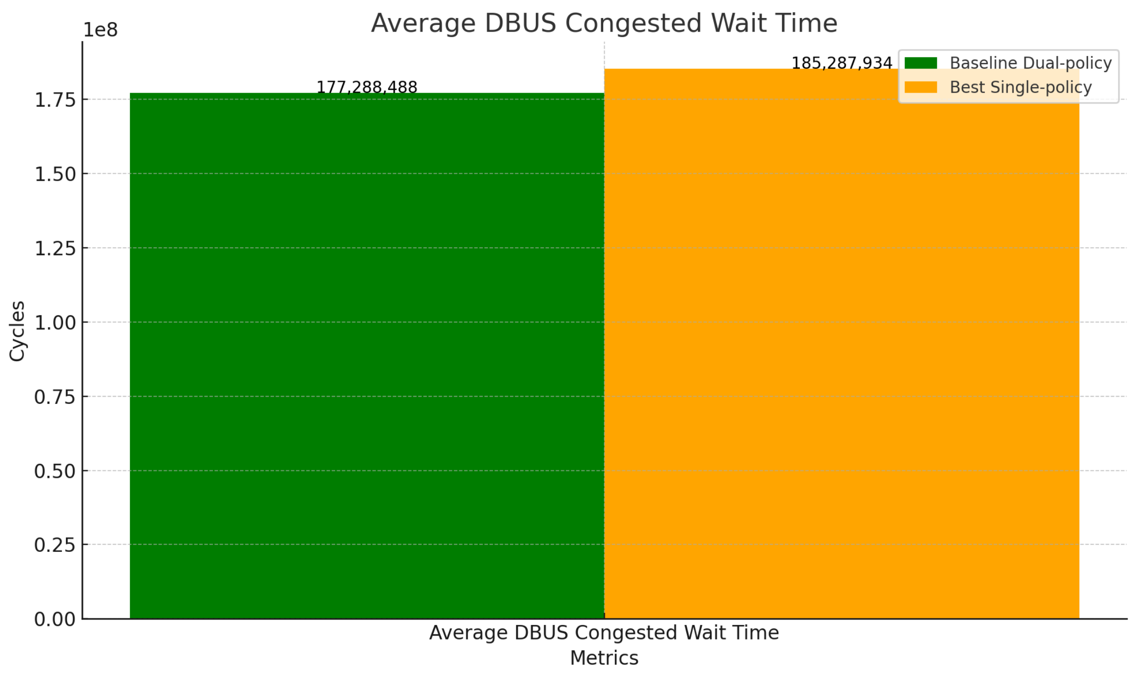


Figure 7.9 Average wait time for the congested DBUS in cycles

7.2.1.2 Conclusions

This detailed comparison of LLC and DRAM statistics highlights the importance of tailored cache replacement policies in optimizing specific workloads. It demonstrates that using dual-policy strategies has the potential to yield better performance compared to single-policy strategies. By evaluating different cache replacement policies, it was observed that combining two policies can adapt more effectively to varying access patterns and workloads. The dual-policy approach showed improved performance by achieving higher speedups and better cache and main memory utilization compared to a single-policy approach. This evidence supports the concept that leveraging the strengths of multiple policies, can provide more flexible and efficient cache management, ultimately leading to enhanced overall system performance. The next step is to run an experiment with dual-policy individuals to see if that approach will yield better performance than the baseline performance we established in the first experiment.

7.2.2 Second experiment: Evolve dual-policy individuals

In the second experiment, we ran GeST with dual-policy individuals, hoping to achieve better IPC speedup performance over the LRU, than the baseline established in the first experiment (Section 7.2.1). The results were not as expected. The best dual-policy individual achieved an average IPC speedup of 1.42% over the LRU. In comparison, the baseline dual-policy from the initial experiment achieved a 1.63% speedup. This indicates that the Baseline dual-policy improved by 12.9% more than the Best dual-policy. Yet, it achieves 13.8% improvement over the Best single-policy from the initial experiment as it achieved 1.25% speedup over the LRU. The plot in Figure 7.10 illustrates the IPC speedup over LRU of the best individual per generation. The conclusion is that for the first 10 generations an improvement was observed but after the 10th generation we reach a ceiling at 1.42% IPC speedup. Given that GeST promotes the best performing individual to the next generation, we can see that after the 10th generation there was not a combination of policies that could achieve to outperform the performance of the best individual of the 10th generation.vReasons for that might be the too much or too little randomness introduced by the evolution operations (mutation and crossover). Also, the hybrid crossover I used for the experiment might have been limiting the evolution as after the 25th generation the crossover strategy used included less randomness (section 6.2).

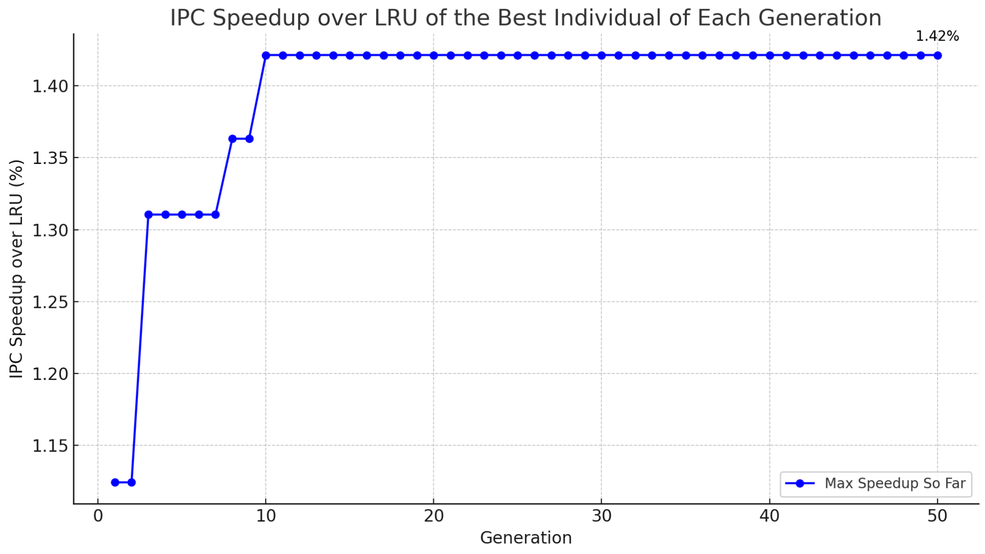


Figure 7.10 Best fitness score of each generation

7.2.2.1 Analysis of the best

In Figure 7.11, we observe the IPC speedups achieved by the best dual-policy individual for each of the benchmarks against the baseline LRU across both its policies. As mentioned in Section 7.2.2, the best dual-policy individual achieved an average IPC speedup of 1.42% over the LRU, which corresponds to an improvement of 13.6% over the use of the single best policy that achieved a 1.25% speedup over the LRU. In Figure 7.12, the curve of the best speedup used by the best dual-policy individual is highlighted in red.

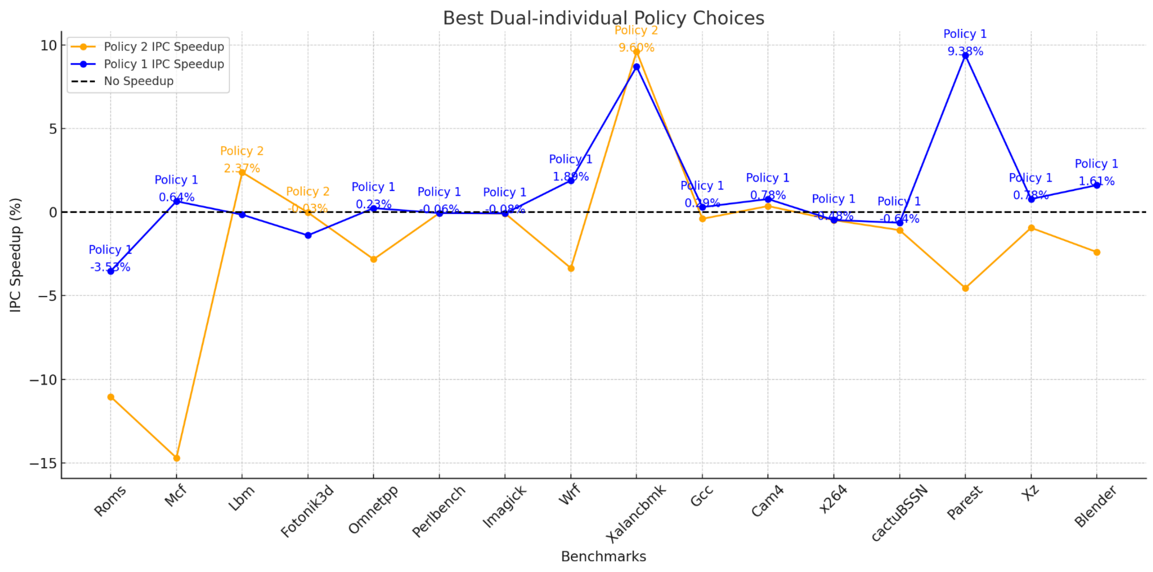


Figure 7.11 Best Dual-policy individual policy choices

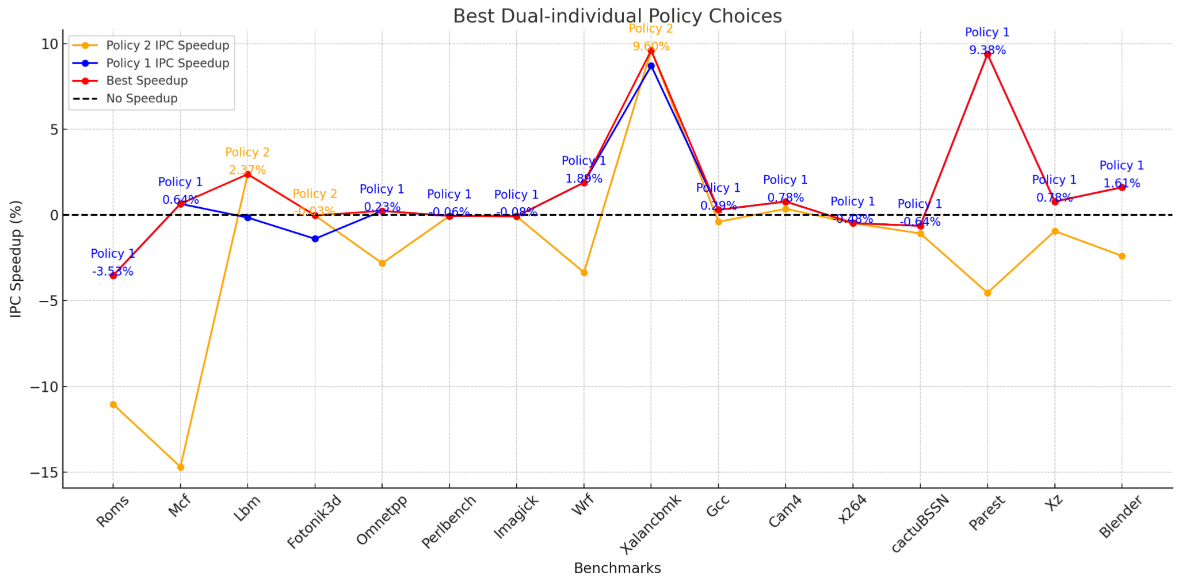


Figure 7.12 Best speedups in red color

In Figure 7.3, we can observe the curve of the baseline dual-policy depicted in green. For a comprehensive comparison, it is essential to examine the curve of the best dual-policy individual, in the same frame as the baseline dual-policy from Section 7.2.1. In Figure 7.13, we present a comparison between the IPC speedups achieved by the baseline dual-policy and the best dual-policy individual across all benchmarks. The green line represents the baseline dual-policy speedup, while the red line denotes the best speedup obtained by the best dual-policy individual.

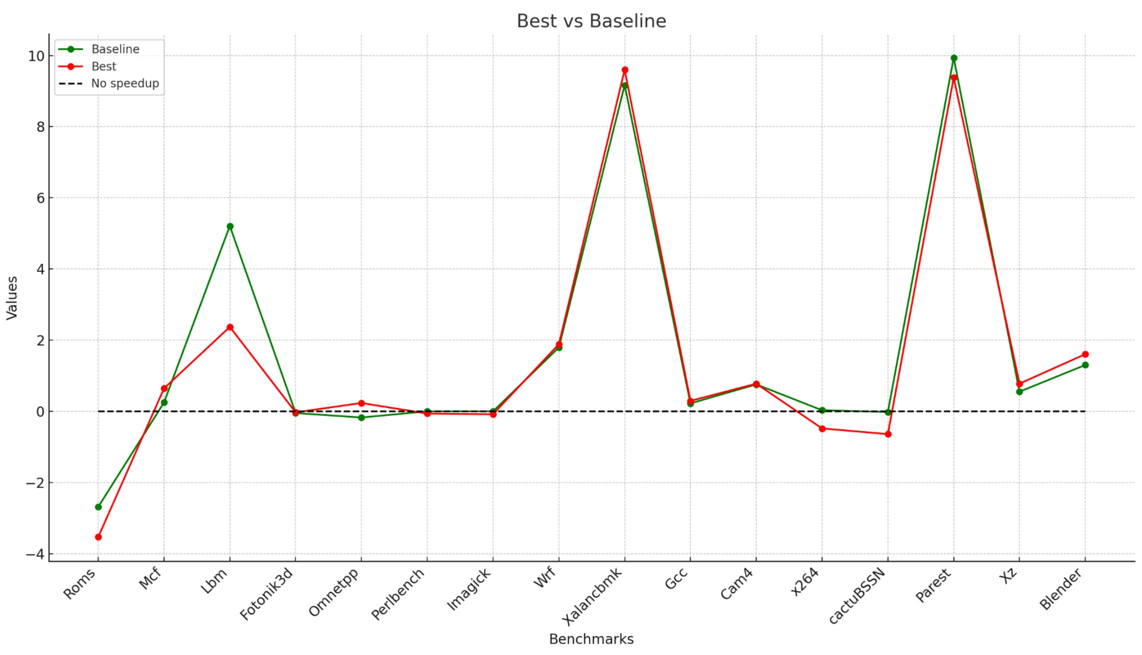


Figure 7.13 Baseline Dual-policy from the first experiment vs the Best dual-policy individual form the second

As observed, the best dual-policy individual generally performs better or similarly to the baseline dual-policy across most benchmarks. However, there are exceptions where the baseline dual-policy outperforms the best dual-policy individual, such as in Lbm and Roms. There are not significant improvements by the best dual-policy showcasing the superiority of the baseline dual-policy. In the case of Lbm, the Best dual-policy exhibits an IPC speedup of 2.37% while the Baseline 5.21%.

7.2.2.2 Benchmarks Analysis

In this section, we examine the benchmark of particular interest which is Lbm. Our goal is to understand the reasons behind the performance behaviors of the best performing policy. between the Baseline dula-policy from the initial experiment (section 7.2.1) and the Best dual-policy that was given as the results from GeST.

7.2.2.2.1 Dive into Lbm

In Figure 7.14 we can observe the speedup that the Best dual-policy given by GeST achieves in comparison to the speedup that the baseline archives for Lbm. To identity the reasons behind that behavior we have to conduct a deep examination.

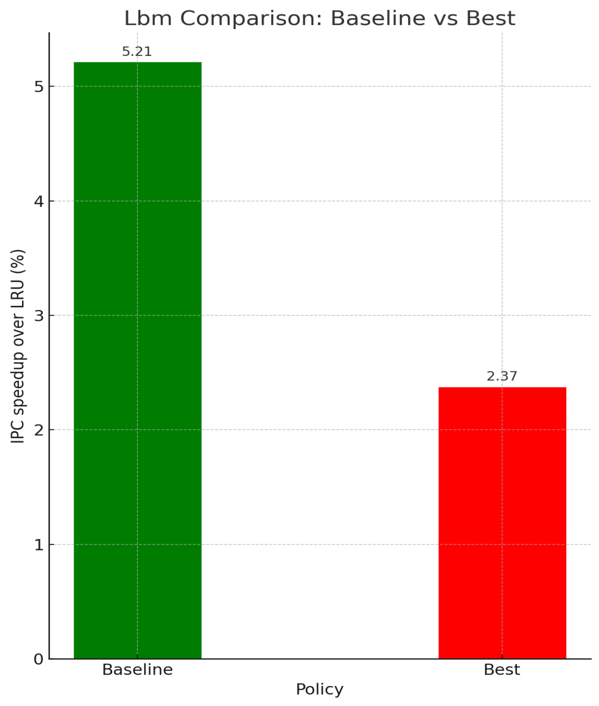


Figure 7.14 Baseline vs Best on IPC speedup over LRU for Lbm

Let's inspect the LLC snapshot for Lbm under both policies. In Figure 7.16 we can see a full LLC snapshot for both policies. We can observe that the Baseline demonstrates less LLC misses in total with the L2 cache prefetches, Writeback and RFO accesses being responsible for that behavior. Figure 7.17 shows a zoomed-in snapshot of the LLC. Yet, further examination has to be done as this alone does not explain the scale of the better performance of the Baseline.

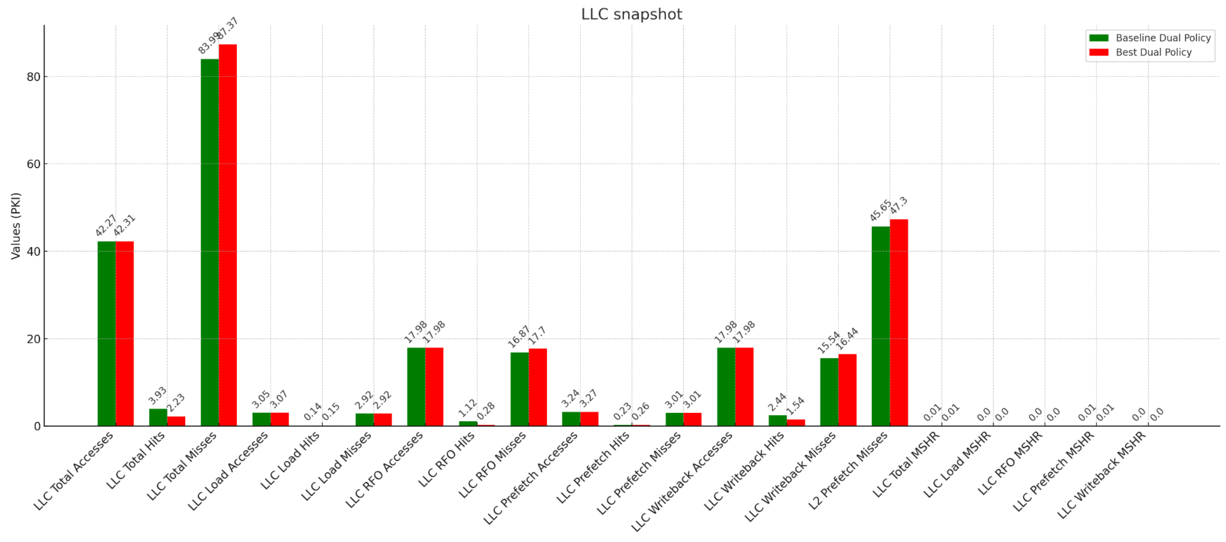


Figure 7.16 Full LLC snapshot

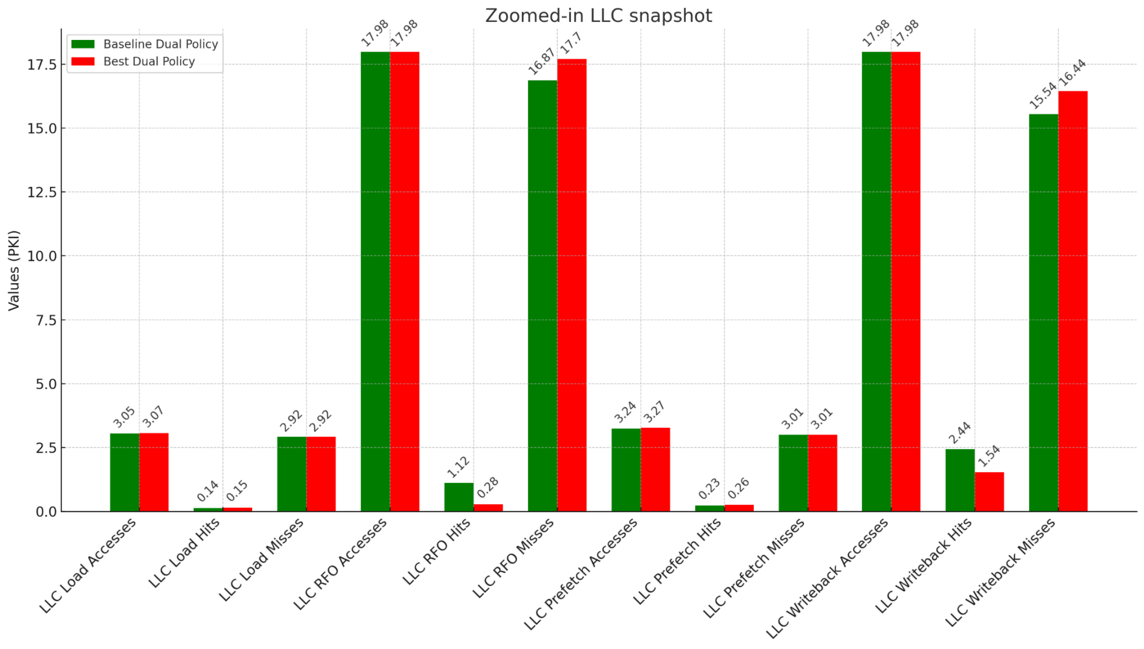


Figure 7.17 Zoomed-in LLC snapshot

Figures 7.18 and 7.19 show the RQ and WQ row buffer misses respectively. We can observe that while for WQ row buffer misses the two policies have similar performance, for the RQ row buffer misses the Baseline dual-policy outperforms the Best. Lower row buffer misses in the main memory means that the latency experienced for main memory requests is lower as well.

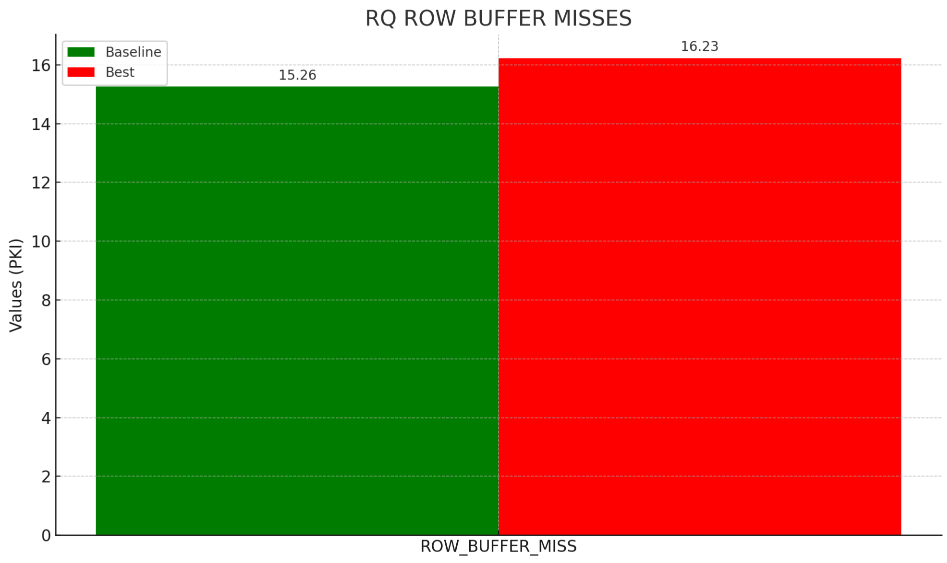


Figure 7.18 RQ row buffer misses

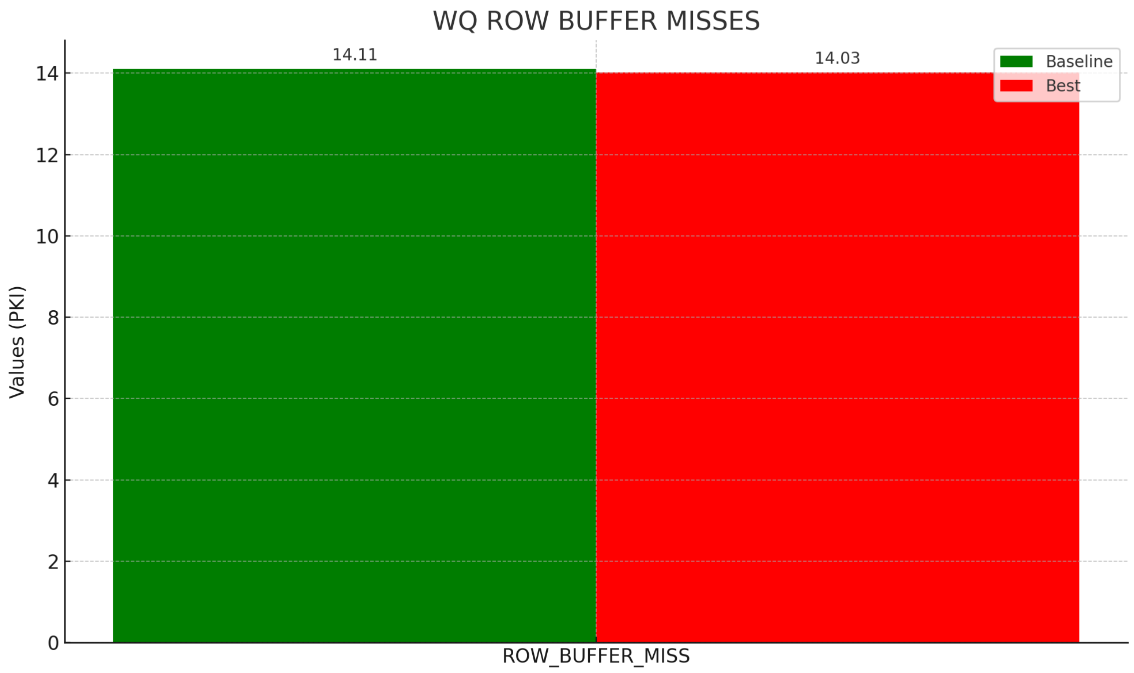


Figure 7.19 WQ row buffer misses

Figure 7.20 shows the average number of times that the two policies found the BUS of the DRAM congested. We can see that the Baseline again out performs the Best. When the DBUS is found congested less times means that the queuing delays will also be lower and thus the IPC will be higher. The higher times the DBUS was found congested by the Best validates the fact that the Best had more writeback misses and also its L2 cache experienced more prefetch misses, data that is shown in Figure 7.16.

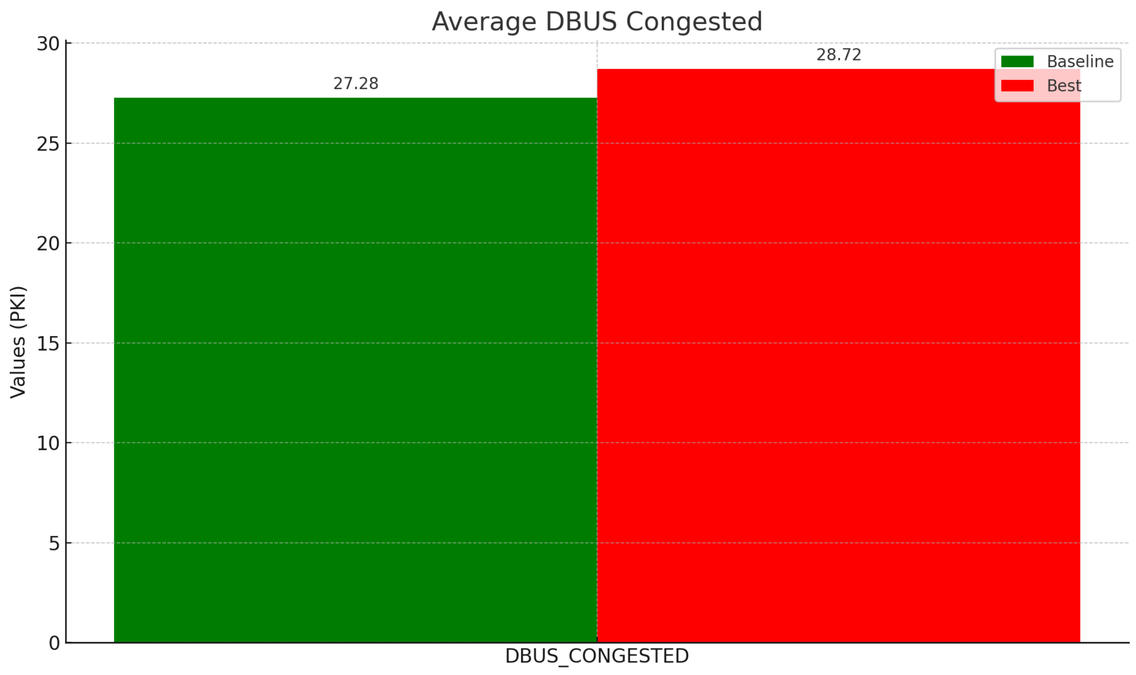


Figure 7.20 Average number of times the DBUS was found congested

Policy Comparison

In Table 7.1 we can see the parameters passed to the simulator by the Baseline and Best dual-policies for the Lbm. Next to each parameter we see the explanation for the values according to section 5.2.

Parameter	Baseline for Lbm	Explanation (Baseline)	Best for Lbm	Explanation (Best)
L_DemClean	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.	3%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2. If RRIP = 0, set to 2.
L_DemDirty	3%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2. If RRIP = 0, set to 2.	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.
L_Insert	1%	New blocks get an RRIP value of 1, indicating high to moderate utility.	0%	New blocks get an RRIP value of 0, indicating high utility.
L_SPromClean	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.	2%	The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.
L_SPromDirty	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.
R_DemClean	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.	2%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2.
R_DemDirty	4%	If the maximum RRIP value is 0, all blocks get an RRIP value of 2. If the maximum RRIP value is 1, all blocks' RRIP values are increased by 1.	4%	If the maximum RRIP value is 0, all blocks get an RRIP value of 2. If the maximum RRIP value is 1, all blocks' RRIP values are increased by 1.
R_Insert	2%	New blocks get an RRIP value of 2, indicating moderate to low utility.	2%	New blocks get an RRIP value of 2, indicating moderate to low utility.
R_SPromClean	1%	The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.	1%	The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.
R_SPromDirty	1%	The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.	2%	The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.
P_DemClean	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.	0%	Increases the RRIP value of all blocks in the set until a block is evicted.
P_DemDirty	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.	3%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2. If RRIP = 0, set to 2.
P_Insert	1%	New blocks get an RRIP value of 1, indicating high to moderate utility.	3%	New blocks get an RRIP value of 3, indicating low utility.
P_SPromClean	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.	1%	The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.
P_SPromDirty	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.	2%	The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.
W_DemClean	2%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2.	3%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 2, but only for blocks with an RRIP value less than 2. If RRIP = 0, set to 2.
W_DemDirty	4%	If the maximum RRIP value is 0, all blocks get an RRIP value of 2. If the maximum RRIP value is 1, all blocks' RRIP values are increased by 1.	1%	Increases the RRIP value of blocks by 1 until the maximum RRIP value is greater than 1.
W_Insert	1%	New blocks get an RRIP value of 1, indicating high to moderate utility.	3%	New blocks get an RRIP value of 3, indicating low utility.
W_SPromClean	2%	The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.	0%	The accessed block's RRIP value is set to 0, indicating it has been recently used and has high utility.
W_SPromDirty	1%	The accessed block's RRIP value is set to 1, indicating it has been recently used and has high to moderate utility.	2%	The accessed block's RRIP value is set to 2, indicating it has been recently used and has moderate to low utility.

Table 7.1 The parameters passes to the simulator by each policy for the Lbm

As observed in Table 7.1, several parameters contribute to the Baseline dual-policy outperforming the Best policy identified by GeST. The first notable parameter is the insertion policy for prefetched blocks (P_Insert). The Baseline assumes that prefetched blocks will be referenced soon and assigns them a RRIP value indicating high to moderate utility. In contrast, the Best dual-policy assumes low utility for prefetched blocks, assigning them a RRIP value that makes them prone to premature eviction. This results in a higher miss rate and lower IPC for the Best policy. Consequently, the Baseline experiences fewer L2 cache prefetch misses (Figure 7.16). For load accesses (L_Insert), the Best policy assigns newly fetched blocks the highest utility RRIP value, which can lead to cache pollution if these blocks are not used soon. Conversely, the Baseline assigns these blocks a RRIP value indicating high to intermediate utility, which helps in evicting less useful blocks sooner and give enough time to the useful ones to prove their utility.

Regarding promotion, the Baseline promotes accessed blocks to lower RRIP values compared to the Best, thereby retaining frequently accessed blocks in the cache for longer periods. Specifically, for RFO accesses on dirty blocks (R_SPromDirty), the Baseline assigns the accessed blocks a high to intermediate utility value while the Best a intermediate to low. The fact that the Baseline keeps those blocks for longer periods on the cache makes it experience more RFO hits thus the behavior we observe in Figure 7.16. In the case of writeback accesses, the Best policy promotes clean blocks (W_SPromClean) to the highest utility RRIP value, whereas the Baseline uses a more conservative approach. This makes the Best policy susceptible to cache pollution by retaining less useful data for longer periods.

Lastly, in terms of demotion, the Baseline employs a more aggressive approach for clean blocks while keeping dirty blocks longer. This makes the Baseline experience less writeback misses since the blocks that are modified and need to be written back (dirty blocks) are more likely to be found in the cache contrary to the approach that the Best employs. This also explains the behavior observed in Figure 7.16 for the writeback misses.

7.2.2.4 Conclusions

This detailed examination highlights the critical factors impacting the performance of the Baseline and Best dual-policies. Despite the LLC snapshot appearing very similar between the two and not explaining the significant difference in IPC for Lbm, the DRAM statistics provide a clearer view of the reasons behind the observed behavior. Since main memory requests typically take longer to be served, encountering bus congestion or a higher number of misses has a substantial impact on IPC. Thus, policies that optimize these operations prove to be more efficient. Lastly, the analysis suggests that limiting our examination to cache statistics alone may not suffice to explain observed behaviors and distinguish between the effectiveness of two policies.

Chapter 8

Future work and Conclusions

8.1 Future work...……………………………………………………………………...53
8.2 Conclusions...……………………………………………………………………...54

8.1 Future work

The investigation carried out in this thesis into dual-policy cache replacement strategies using genetic algorithms opens several doors for further exploration. Although significant advancements were achieved, the dynamic and complex nature of cache management suggests that there is much more to be explored. Future studies could enhance and extend the methodologies employed here, potentially yielding more refined and innovative solutions.

One of the most promising areas for future work involves the implementation and testing of advanced crossover techniques discussed in this thesis (see section 4.2). The current study introduced concepts such as simple dual-policy crossovers and more complex ones. Future research should focus on practically implementing these theoretical crossover strategies to assess their effectiveness empirically. Moreover, by experimenting even more with hybrid models that combine different crossover techniques, researchers might uncover better configurations that offer a balance between diversity in the population and convergence speed towards effective solutions.

Following the same course of thinking, there is a lot to be investigated about the mutation as well. Mutation plays a critical role in maintaining genetic diversity within the population. Investigating dynamic mutation strategies that adapt based on the population's state or performance could lead to more effective evolutionary processes. For instance, mutations could be boosted during periods of population inaction to reintroduce variability, or they could be targeted more strategically, focusing on underperforming aspects of replacement policies. Such adaptive approaches could help in evolving more robust cache replacement strategies that are capable of coping with a variety of operational scenarios.

Further, as mentioned in the beginning of the 5th chapter (section 5.1), the notion of starting with a first generation that includes some pre-selected, promising individuals, which has been previously tested, presents an intriguing avenue for future research. This strategy may help GeST converge earlier, thus exploring the search space more effectively and perhaps lead to better performance of the best individual.

An exciting avenue for future research involves extending the capabilities of the GeST framework to support individuals characterized by N-policy configurations, rather than being restricted to single or dual-policy formats. This could lead to more robust, adaptive systems that maintain optimal performance across a range of operational scenarios and potentially lead to superior performance by reaching closer to the optimal policy in the search space.

Lastly, evolving a population to identify the optimal replacement policy that maximizes performance for a specific benchmark (workload) is an intriguing area to explore. Unlike traditional approaches that seek a universal replacement policy performing adequately across various workloads, this method focuses on discovering multiple specialized replacement policies, each excelling in one specific benchmark. Consequently, these policies may not necessarily perform well on other benchmarks. This approach further specializes the strategy of cache block eviction, potentially leading to the development of a dynamic framework that adapts the replacement policy based on the current workload. This specialization could significantly enhance cache performance by tailoring policies to the specific access patterns and demands of different workloads, opening new avenues for cache replacement policy research.

8.2 Conclusions

Overall, the findings of this thesis underscore the potential of synergistic cache replacement policies to yield better performance than single-policy strategies. The best dual-policy configuration outperformed the best single-policy; however, it did not exhibit better performance than the baseline established in the initial experiment (section 7.2.1). This suggests that the evolution of dual-policy individuals might not be necessary to find the best-performing combination of two replacement policies. A more effective approach using GeST, coupled with a deeper analysis of the algorithm's evolution process, will reveal areas for refinement. Such improvements could potentially enable the evolution of dual-policy individuals to surpass the baseline dual-policy performance.

References

[1] Jaleel, A. et al. (2010) High performance cache replacement using re-reference interval prediction (RRIP). Available at:
http://www.jaleels.org/ajaleel/

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

[3] 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

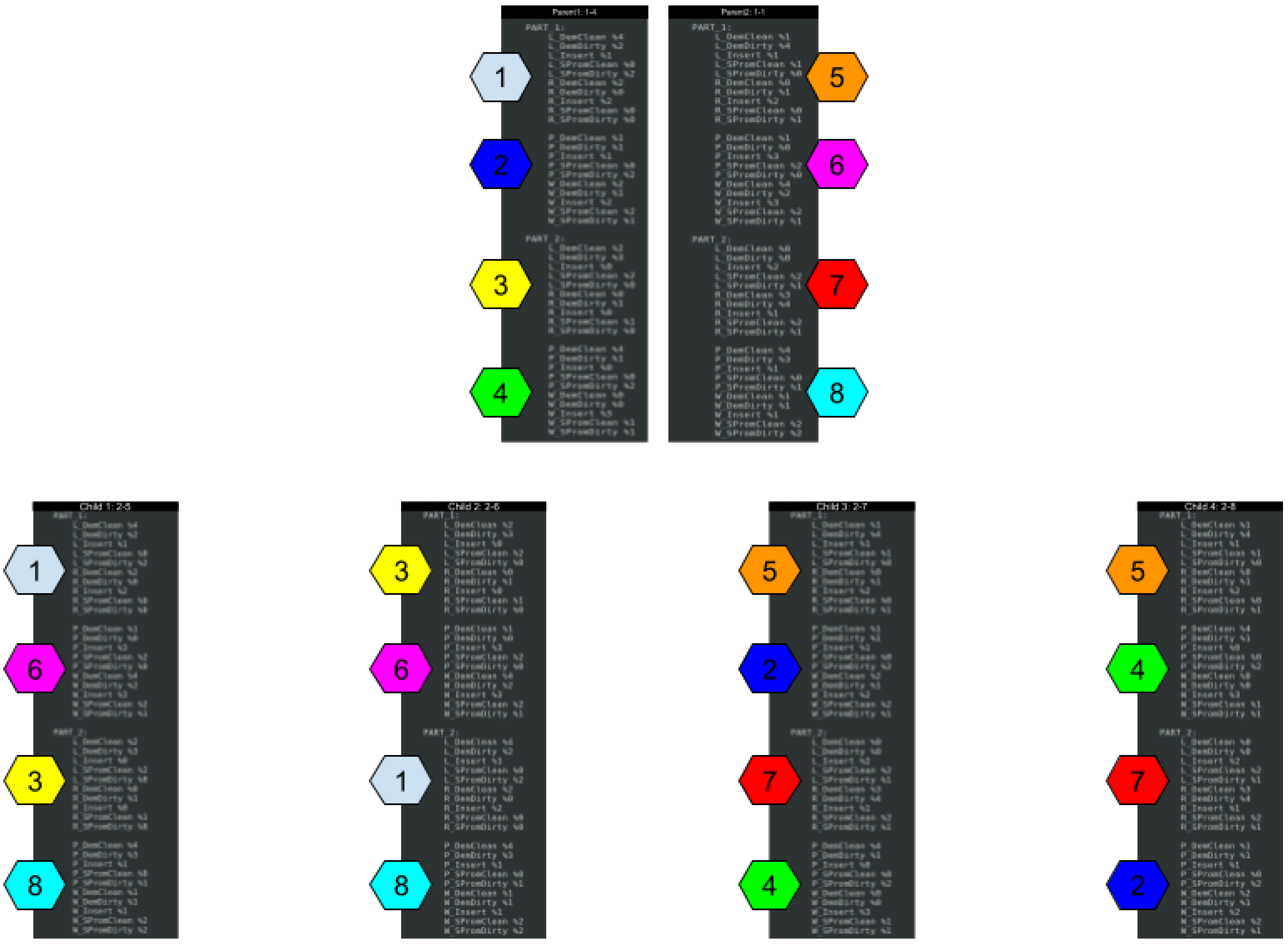
[4] Pseudo-LRU. Available at:
https://en.wikipedia.org/wiki/Pseudo-LRU

[5] Nicolas Papaki: Genetic Search on RRIP Replacement Policies. Available at:
https://www.cs.ucy.ac.cy/index.php/education/thesisarchive

[6] Tournament selection. Available at:
http://en.wikipedia.org/wiki/Tournament_selection

Appendix A

Crossover from the experiment

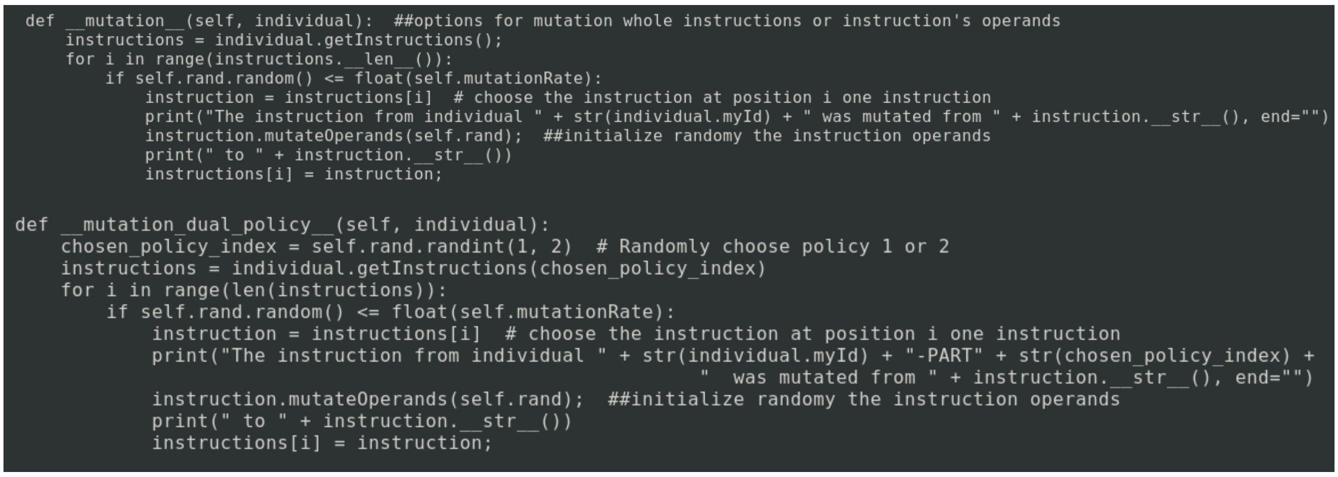


In the figure above, there were no mutations in any of the children which is something that is also shown in figure 6.5 of section 6.2

Appendix B

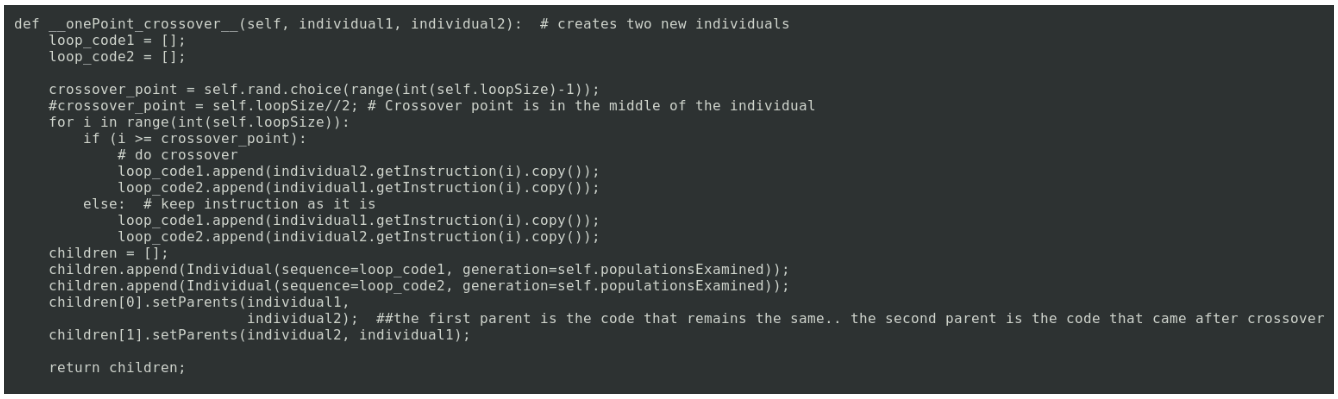
Modifications code

Mutation

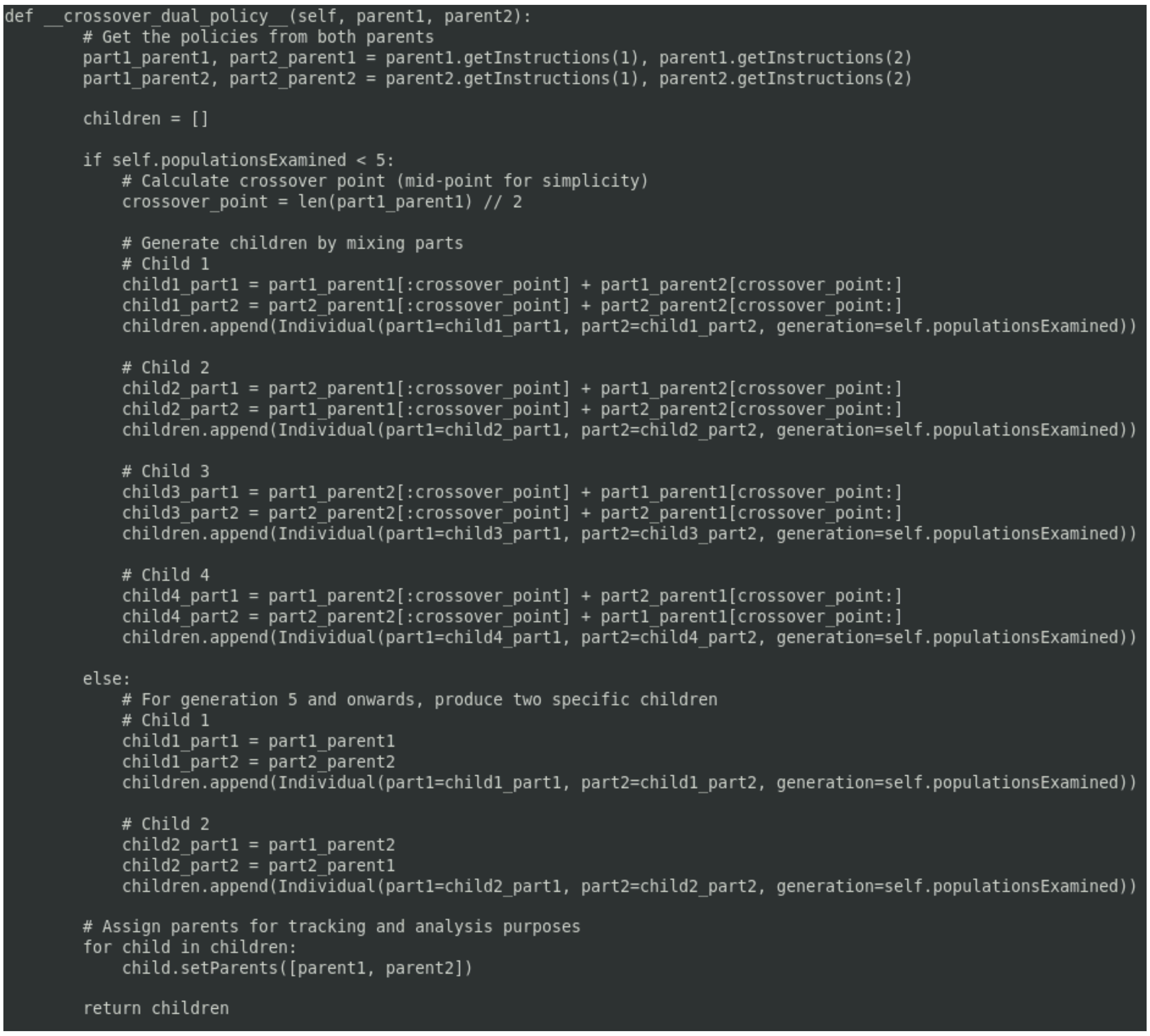


```
 def __mutation__(self, individual):  ##options for mutation whole instructions or instruction's operands
     instructions = individual.getInstructions();
     for i in range(instructions.__len__()):
         if self.rand.random() <= float(self.mutationRate):
             instruction = instructions[i]  # choose the instruction at position i one instruction
             print("The instruction from individual " + str(individual.myId) + " was mutated from " + instruction.__str__(), end="")
             instruction.mutateOperands(self.rand);  ##initialize randomy the instruction operands
             print(" to " + instruction.__str__())
             instructions[i] = instruction;

def __mutation_dual_policy__(self, individual):
    chosen_policy_index = self.rand.randint(1, 2)  # Randomly choose policy 1 or 2
    instructions = individual.getInstructions(chosen_policy_index)
    for i in range(len(instructions)):
        if self.rand.random() <= float(self.mutationRate):
            instruction = instructions[i]  # choose the instruction at position i one instruction
            print("The instruction from individual " + str(individual.myId) + "-PART" + str(chosen_policy_index) +
                                                          "  was mutated from " + instruction.__str__(), end="")
            instruction.mutateOperands(self.rand);  ##initialize randomy the instruction operands
            print(" to " + instruction.__str__())
            instructions[i] = instruction;
```

Crossover

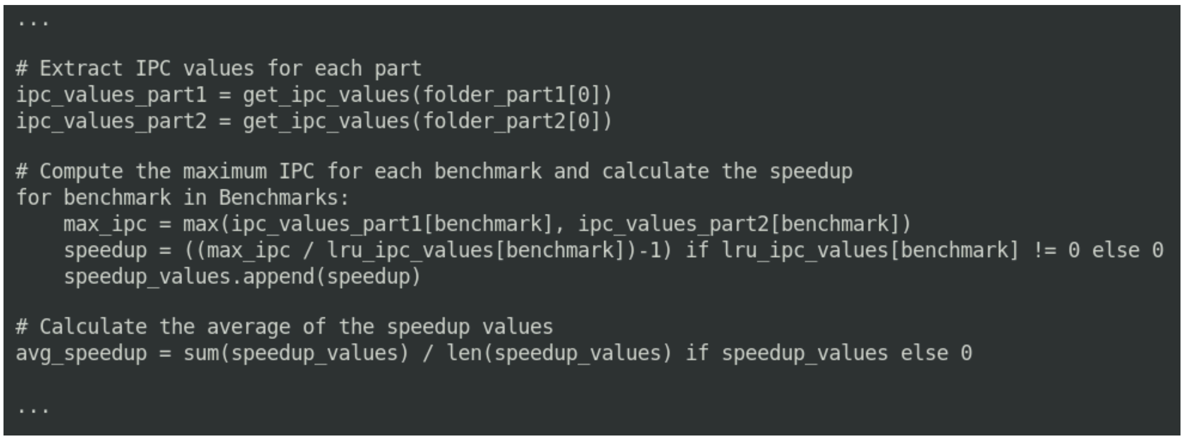


```
def __onePoint_crossover__(self, individual1, individual2):  # creates two new individuals
    loop_code1 = [];
    loop_code2 = [];

    crossover_point = self.rand.choice(range(int(self.loopSize)-1));
    #crossover_point = self.loopSize//2; # Crossover point is in the middle of the individual
    for i in range(int(self.loopSize)):
        if (i >= crossover_point):
            # do crossover
            loop_code1.append(individual2.getInstruction(i).copy());
            loop_code2.append(individual1.getInstruction(i).copy());
        else:  # keep instruction as it is
            loop_code1.append(individual1.getInstruction(i).copy());
            loop_code2.append(individual2.getInstruction(i).copy());
    children = [];
    children.append(Individual(sequence=loop_code1, generation=self.populationsExamined));
    children.append(Individual(sequence=loop_code2, generation=self.populationsExamined));
    children[0].setParents(individual1,
                           individual2);  ##the first parent is the code that remains the same.. the second parent is the code that came after crossover
    children[1].setParents(individual2, individual1);

    return children;
```



```python
def __crossover_dual_policy__(self, parent1, parent2):
        # Get the policies from both parents
        part1_parent1, part2_parent1 = parent1.getInstructions(1), parent1.getInstructions(2)
        part1_parent2, part2_parent2 = parent2.getInstructions(1), parent2.getInstructions(2)

        children = []

        if self.populationsExamined < 5:
            # Calculate crossover point (mid-point for simplicity)
            crossover_point = len(part1_parent1) // 2

            # Generate children by mixing parts
            # Child 1
            child1_part1 = part1_parent1[:crossover_point] + part1_parent2[crossover_point:]
            child1_part2 = part2_parent1[:crossover_point] + part2_parent2[crossover_point:]
            children.append(Individual(part1=child1_part1, part2=child1_part2, generation=self.populationsExamined))

            # Child 2
            child2_part1 = part2_parent1[:crossover_point] + part1_parent2[crossover_point:]
            child2_part2 = part1_parent1[:crossover_point] + part2_parent2[crossover_point:]
            children.append(Individual(part1=child2_part1, part2=child2_part2, generation=self.populationsExamined))

            # Child 3
            child3_part1 = part1_parent2[:crossover_point] + part1_parent1[crossover_point:]
            child3_part2 = part2_parent2[:crossover_point] + part2_parent1[crossover_point:]
            children.append(Individual(part1=child3_part1, part2=child3_part2, generation=self.populationsExamined))

            # Child 4
            child4_part1 = part1_parent2[:crossover_point] + part2_parent1[crossover_point:]
            child4_part2 = part2_parent2[:crossover_point] + part1_parent1[crossover_point:]
            children.append(Individual(part1=child4_part1, part2=child4_part2, generation=self.populationsExamined))

        else:
            # For generation 5 and onwards, produce two specific children
            # Child 1
            child1_part1 = part1_parent1
            child1_part2 = part2_parent2
            children.append(Individual(part1=child1_part1, part2=child1_part2, generation=self.populationsExamined))

            # Child 2
            child2_part1 = part1_parent2
            child2_part2 = part2_parent1
            children.append(Individual(part1=child2_part1, part2=child2_part2, generation=self.populationsExamined))

        # Assign parents for tracking and analysis purposes
        for child in children:
            child.setParents([parent1, parent2])

        return children
```

Fitness Function



```python
...

# Extract IPC values for each part
ipc_values_part1 = get_ipc_values(folder_part1[0])
ipc_values_part2 = get_ipc_values(folder_part2[0])

# Compute the maximum IPC for each benchmark and calculate the speedup
for benchmark in Benchmarks:
    max_ipc = max(ipc_values_part1[benchmark], ipc_values_part2[benchmark])
    speedup = ((max_ipc / lru_ipc_values[benchmark])-1) if lru_ipc_values[benchmark] != 0 else 0
    speedup_values.append(speedup)

# Calculate the average of the speedup values
avg_speedup = sum(speedup_values) / len(speedup_values) if speedup_values else 0

...
```