DIPLOMA PROJECT

Enhancing Genetic Search Efficiency for RRIP Cache Policies Across Diverse Workloads

MARKO ZAKO

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2025

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Enhancing Genetic Search Efficiency for RRIP Cache Policies Across Diverse Workloads

MARKO ZAKO

SUPERVISOR

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 2025

Acknowledgements

First and foremost, I want to thank my family for always being there for me. Their support, encouragement, and belief in me have meant everything throughout this journey. I wouldn't have made it this far without their constant presence and the sacrifices they made to give me the chance to study at a top university. I'll always be grateful for that.

Secondly, the support of my friends and classmates was what got me through these 4 years, their help and support always made me push through, even during the hardest days during this journey. I couldn't have asked for a better group to go through this experience with.

A special thank you to my supervisor, Mr. Yiannakis Sazeides, for his valuable guidance throughout the making of my thesis. His insight and advice were truly helpful and appreciated.

Lastly, I'd like to thank Ioannis Constantinou and Panteleimonas Chatzimiltis for always being willing to help whenever I ran into technical issues. Their support made a real difference.

Abstact

Computers are getting faster, and programs are getting more demanding. With the introduction of Amdahl's law and Moore's law, we can observe that computational power is hitting a plateau. One of the fields that support this plateau and can be improved on is the field of cache replacement policies. The most critical level in the hierarchy is the LLC, which can save us many cycles if the needed information is stored there.

There are some policies such as LRU (PLRU) that have overall good performance, but we observe that there is room for improvement. State-of-the-art policies such as DRRIP [2] improve on the performance of LRU, but are still far off Belady's optimal policy. One issue with most of these policies is that they do not have an adaptation technique to a workload's pattern.

This thesis is expanding on continued work that looks at how we can use genetic algorithms to improve cache replacement policies for better performance across different workloads. We study how the results change when the genetic algorithm is run using a single benchmark compared to using a mix of benchmarks. This helps us understand which parts of a policy really matter for performance. We then proceed on finding the important functions on the benchmark's best performing policies, to see how achievable it is to find common ground for a policy for more than 1 benchmark. Finally, we try to find the smallest number of policies needed to get good performance across all benchmarks and test.

There are some promising results on isolating the workload, showing a good improvement on the search of the algorithm. There was an importance observed on the insertion function of a benchmark, while demotion functions were not as important. The policies produced for one benchmark were all different from policies for other benchmarks, showing their differences in behavior.

These results suggest that a one-size-fits-all policy is insufficient, and that using a small set of tailored policies can better match workload diversity and improve overall cache efficiency.

Contents

Chapter 1

Introduction

1.1 Introduction

In the past decades, computers have evolved at a fast rate in terms of raw computing power, enabling unprecedented progress in nearly every field of science and technology. Faster processors, longer pipelines, more cores, and improved memory systems have all cumulatively enabled systems to become more powerful and efficient. In the meantime, however, while hardware has advanced, so have the demands put upon it, applications today manage increasingly large amounts of data and execute more complex logic, often with real-time or performance-sensitive constraints.

Two basic laws of computer architecture, Moore's Law and Amdahl's Law, have shaped performance progress expectations for decades. Both laws indicate a disturbing trend: historically, the pace of progress in general-purpose hardware is slowing. Physical limitations in transistor scaling, power constraints, and parallelization diminishing returns cause computational performance to plateau.

System designers must then concentrate on microarchitectural optimizations that make better use of what is available. One of the most impactful areas is the memory hierarchy,

and in particular, cache memory systems. Of the many levels of cache, the Last-Level Cache (LLC) is central. Positioned directly in front of primary memory in the hierarchy, the LLC is a large, shared buffer that can absorb costly memory accesses if managed properly. Retrieving data from the LLC is significantly faster than retrieving it from DRAM and can eliminate hundreds of CPU cycles per access.

The benefits of the LLC rely mostly on how well it can store useful information. One key parameter that influences cache performance is its cache replacement policy (CRP). A CRP decides what data to remove from the cache when new data is to be brought into a full set of caches. Figure 1.1 illustrates how different benchmarks suffer from suboptimal CRPs, showing increased miss rates and lower IPC compared to workloads where the cache policy aligns better with the access behavior.

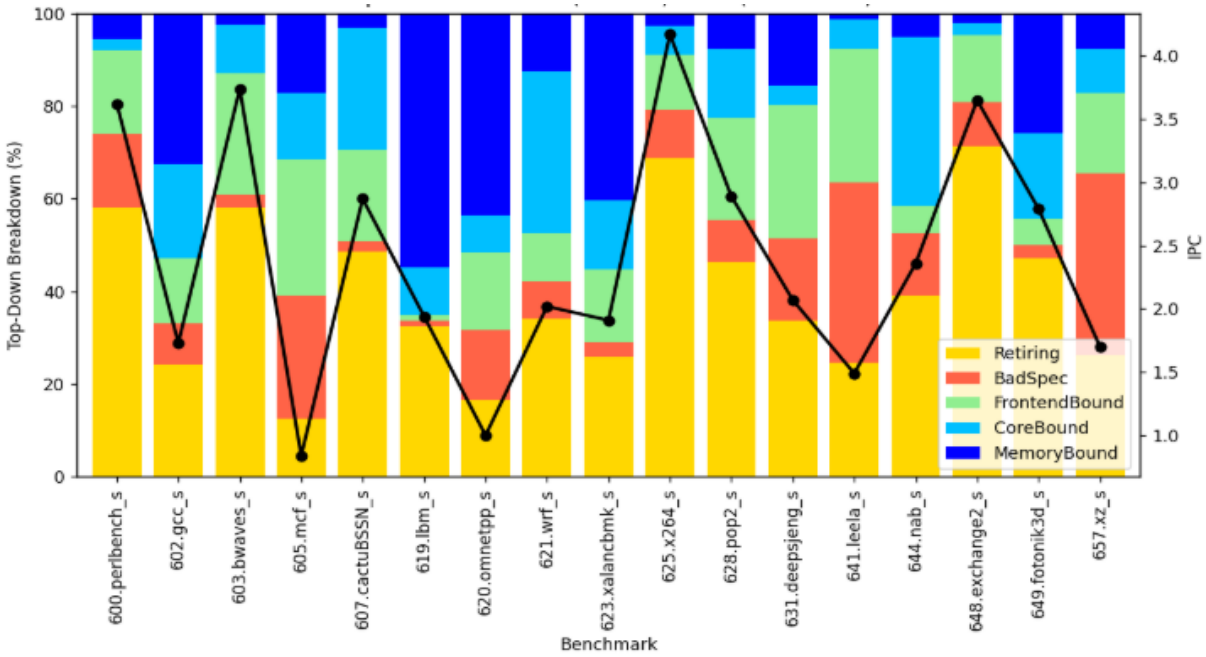

Figure 1.1: Breakdown of the Performance of SPEC CPU207 Workloads using top-down analysis

After looking at the benchmarks that recorded many cache misses (Figure 1.2), the results showed that what makes those benchmarks not perform so well was that they all had many LLC misses, demonstrating that having the optimal CRP for the LLC cache can really change its performance.

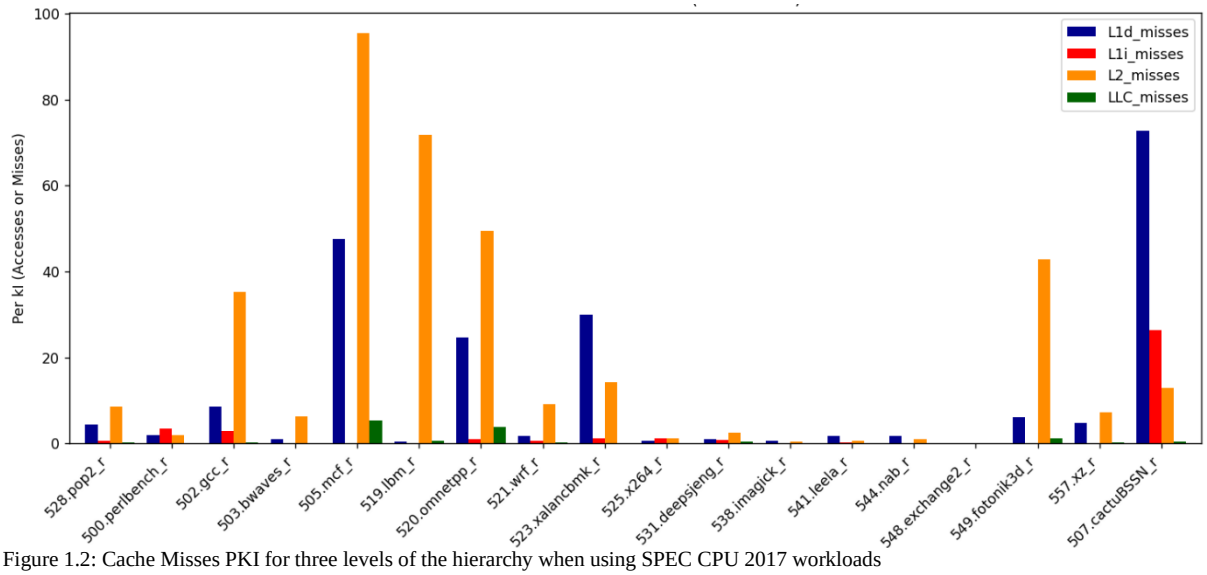

Figure 1.2: Cache Misses PKI for three levels of the hierarchy when using SPEC CPU 2017 workloads

Many systems use static cache replacement policies like Least Recently Used (LRU) and First-In-First-Out (FIFO). These policies are simple and easy to implement because they always follow the same fixed rules. However, they do not adjust to different workloads or changing access patterns, which means they can make poor decisions when the workload behaves differently. This has led to the development of adaptive policies, such as Dynamic Re-Reference Interval Prediction (DRRIP) [2], which attempt to dynamically switch between policies like SRRIP and BRRIP depending on runtime performance. While such solutions offer improvements, their adaptability is still limited to predefined options.

Another way of making the policies adaptive is using an optimization search, such as Genetic Algorithms (GAs), to find the optimal policy for a given workload [4]. Prior work [1] has shown that there are improvements over static policies like LRU and SRRIP, making GA search a relevant option to find the optimal policy.

In this work, genetic search is used to explore the design space of cache replacement policies by representing each policy as a set of functions, like where to insert a new data entry in the cache, or how to handle eviction. These decisions are different for all instruction types, like Loads, Request For Ownership, Writeback and Prefetching, to have a broader search space. This structured representation allows the genetic algorithm to systematically evolve and evaluate different combinations, effectively navigating the large policy space to discover those that yield the best performance for given workloads.

This thesis explores how the process of genetic search for cache replacement policies can be improved and extended to become more effective, flexible, and workload aware.

A central focus of this thesis is to explore the trade-off between two different optimization approaches: one where the cache replacement policy is tuned separately for each benchmark, and another where a single policy is chosen to perform well on average across all benchmarks. By comparing these two approaches, we can understand not just which method gives better performance but also learn more about the functions. Which are those functions that really made the difference for the policy to be optimal for its workload, and which functions might not matter as much. These insights can help guide the design of more flexible and effective cache replacement strategies in the future.

In addition, this thesis explores how many distinct policies are truly needed to achieve good performance across diverse workloads. Specifically, we investigate whether a small set of carefully chosen policies can cover the full spectrum of benchmarks, by selecting the right one for each case, rather than relying on a single general-purpose policy. These selected policies are later evaluated and compared against the single-best policies found for the same workload.

To answer these questions, this thesis combines the use of ChampSim, a state-of-the-art trace-driven simulator, and GeST, a genetic algorithm framework adapted to evolve and evaluate CRP configurations.

Throughout this thesis, several important observations emerged. First, we found that tuning policies for individual benchmarks consistently outperformed using a shared policy across all workloads. We also identified that insertion functions played a much more significant role in policy success than demotion functions. Another key finding was that starting the genetic search with a crossover point in the middle yields better results than a random crossover point. These insights support the idea that workload-aware and well-structured search strategies are essential for designing effective cache replacement policies.

1.2 Outline

This thesis is structured into seven chapters. Chapter 1 introduces the research topic, motivation, and goals. Chapter 2 provides the necessary background on modern CPUs,

memory hierarchies, cache structures, genetic algorithms, and simulation tools, ending with related work and motivation. Chapter 3 explains SRRIP and DRRIP [2] cache replacement policies, including their limitations. Chapter 4 describes the experimental frameworks and methodology, including, GeST [5], ChampSim [6], and the genome encoding process. Chapter 5 presents workload-based experiments and analyzes policy behaviors. Chapter 6 focuses on experiments and their results involving dual-policy configurations. Finally, Chapter 7 concludes the thesis and outlines directions for future work.

Chapter 2

Background

2.1 Modern CPUs

CPUs of today are the backbone of nearly all computing equipment, from a lone notebook to massive data centers. They have evolved a great deal to become highly sophisticated pieces of technology that can run billions of instructions per second with ease. The increase in performance has been made possible through numerous innovations in instruction pipelines, microarchitecture, and memory hierarchies.

When a program runs, the CPU cycles through a sequence of operations:

1.Fetch: The next instruction is fetched from memory by the CPU.

2.Decode: The instruction is decoded and translated into signals that the CPU can interpret.

3.Execute: The CPU performs the operation needed, such as a calculation or logical operation.

4.Memory Access: The CPU accesses memory, if needed, to write or read data.

5.Write Back: The result of the operation is written to a register or written back to memory.

To make this process faster, modern processors rely on deep pipelines and multiple execution units so that they can run several instructions simultaneously.

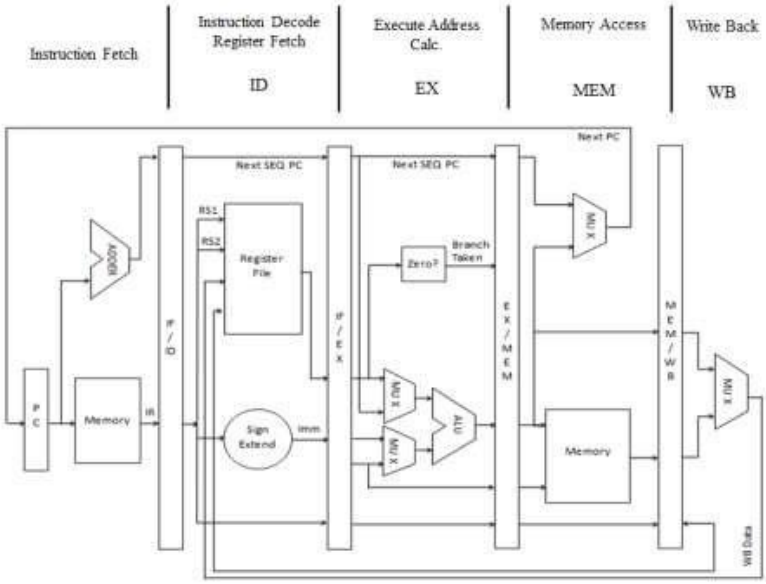

Figure 2.1: The CPU Pipeline and its 5 stages

The pipeline ensures that we in a cycle, all 5 stages are active, instead of waiting for each instruction to go through all 5 operations and then proceeding with the next instruction. Between each 2 stages there is a pipeline register. These are crucial for maintaining the separation and synchronization between each stage of the instruction cycle. They hold the output of one stage and pass it as input to the next stage during the next clock cycle.
A limitation of the pipeline is the dependency between subsequent instructions, which slows down the execution potential. Among the most significant developments here is out-of-order execution, a technique that enables the CPU to perform instructions as and when their operands are ready, so it executes instructions that do not depend on each other, rather than precisely in the order they appear in the program. This increases instruction-level parallelism and avoids wasting time waiting on slower operations.

To help with this, CPUs also exploit parallelism at different levels. From the execution of independent instructions simultaneously within one core, to the execution of multiple threads across many cores, modern CPUs are designed to do more in less time. Simultaneous multithreading (SMT), or hyper-threading, is a function that allows one

physical core to execute more than one instruction stream, or thread, at the same time. This enhances use of available resources and enables improved multitasking.

2.1.1 Moore's Law

For many years, Moore's Law was an important principle in the semiconductor industry. Moore's law states that the number of transistors on an integrated circuit would double approximately every two years. This growth of transistors would mean increased computational power, allowing CPUs to become faster, smaller, and more energy-efficient with each new generation. For decades, software performance often improved simply by waiting for hardware to catch up—new processors would deliver better results without having to make big changes to the code or the system architecture.
However, in recent years, Moore's Law has shown signs of slowing down. Transistors are now approaching atomic-scale dimensions, which is very expensive, providing minimal benefits, due to issues such as overheating manufacturing complexity. These issues make it harder to maintain the historical pace of improvement. As a result, so t here needs to be smarter system design to manage the hardware.

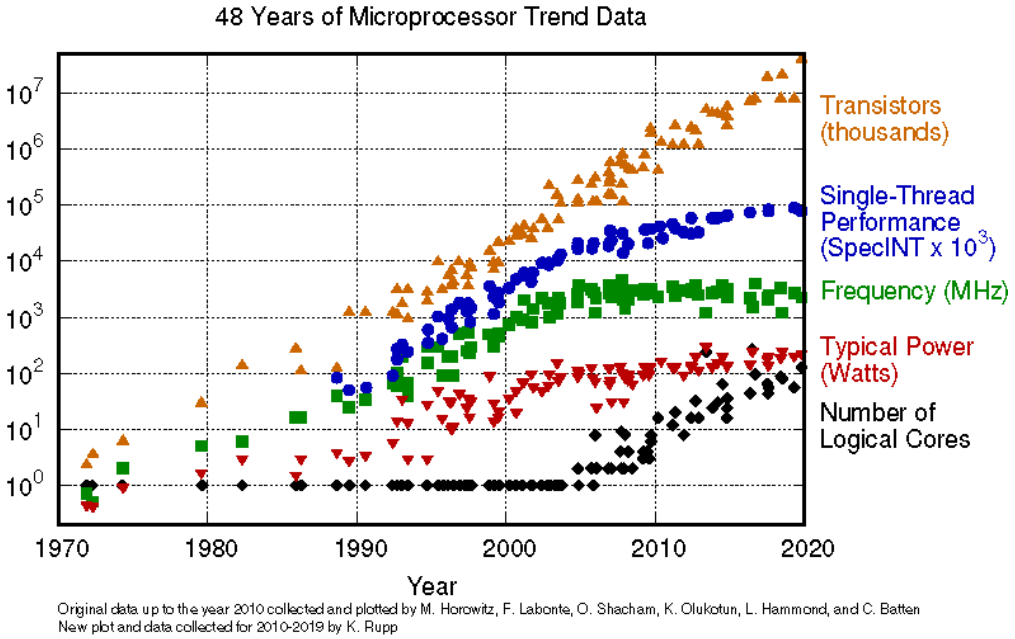

Figure 2.2: Moore's law through the years

From Figure 2.1.1 we can see that the single-thread performance and the clock frequency have found a bound, even with the transistor number increasing.

This slowdown has direct implications for system design. Rather than expecting hardware to solve performance bottlenecks, attention is now shifting toward microarchitectural and algorithmic optimizations. In this context, cache memory systems, especially the Last-Level Cache (LLC), play a crucial role. Efficient cache management helps bridge the performance gap between the CPU and main memory by keeping frequently accessed data closer to the processor.

The limitations imposed by the end of Moore's Law motivate the exploration of smarter cache replacement strategies. By using techniques like genetic algorithms to adapt and optimize replacement policies, the goal is to improve system performance through better use of existing hardware—rather than relying on future transistor scaling. This reflects a broader shift in computing, where intelligent resource management and data-aware algorithms are essential for extracting more performance from today's complex systems.

.1.2 Amdahl's Law

To break through the limitation of Moore's Law, parallel work on multiple cores became a growing trend. The reality is that even parallelism has limits. Amdahl's Law in parallel computing reminds us that program speedup is limited by the portions that can't be parallelized.

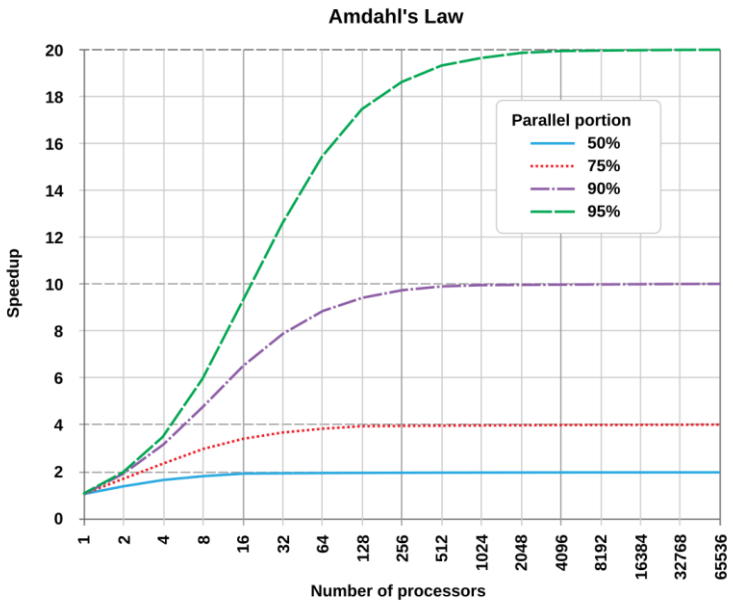

Figure 2.3: Amdahl's Law speedup limitation

No matter how many cores a processor has or how fast they run, there will always be bottlenecks, serial segments of code or system overheads, that cap overall performance.

This law reminds us of a valuable fact: it takes more than just adding hardware if the architecture and algorithms don't use it efficiently. This means that we must consider improving single-thread performance, so even if there are serial parts of a program, we optimize those periods with the right algorithm. One of the ways is to optimize cache behavior, i.e., the LLC, because it matters most to how many cycles an instruction will execute.

2.2 Memory Hierarchy

Modern computer systems utilize a hierarchical memory architecture designed to balance speed, capacity, and cost. As we move from the processor toward lower levels of the memory hierarchy, culminating in main memory and persistent storage, each level typically offers increased capacity but also higher access latency. This latency is measured in CPU cycles, and delays at lower levels can significantly impact overall system performance.

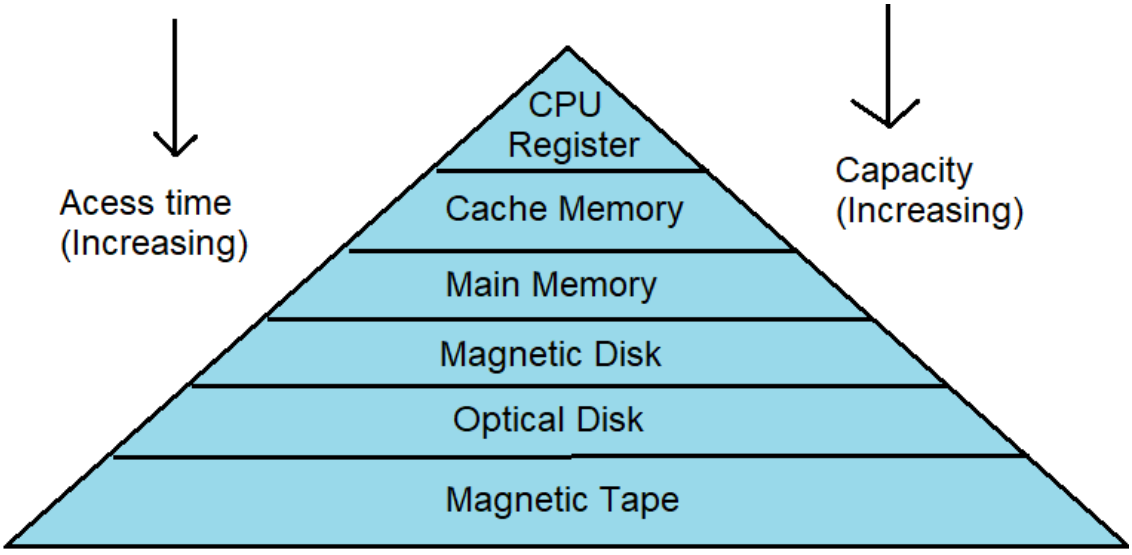

Figure 2.4: The memory Hierarchy

To mitigate the high cost of frequent main memory accesses, most modern processors incorporate a multi-level cache system, consisting of L1, L2, and Last-Level Cache (LLC). The L1 cache is the smallest but fastest, located closest to the CPU core. If data is not found in L1, the processor checks the L2 cache, which is larger but slower. Failing

that, it proceeds to the LLC, which is shared across cores and offers even more capacity at the cost of further increased latency.

This tiered caching strategy ensures that frequently accessed data and instructions are kept as close to the CPU as possible, reducing cache miss rates and significantly decreasing the number of costly main memory accesses. As a result, the memory hierarchy plays a critical role in maintaining high performance in modern computing systems.

2.2.1 Cache Memory

A typical cache is made up of fixed-length blocks known as cache lines, usually 64 bytes per line. Such lines are grouped into sets, and the number of cache lines per set determines the cache associativity. The general structure of the cache, the manner in which it stores data and retrieves data—strongly affects performance and ease of implementation.

There are three primary structures for the cache:

- Direct-Mapped Cache: Each memory block maps to a specific cache line using the lower bits of its address. It’s fast and simple but prone to conflict misses, where multiple blocks compete for the same line.

- Fully-Associative Cache: A block can be stored in any cache line, reducing conflict misses. However, this requires checking all tags on each access, increasing hardware complexity and access time. It's mainly used in small caches like the TLB.

- Set-Associative Cache: Common in modern systems, this structure divides the cache into sets, each containing multiple lines (ways). A block maps to a specific set but can occupy any line within it. It offers a good balance between performance and complexity, reducing conflict misses more effectively than direct-mapped caches without the overhead of fully-associative designs.

When a CPU reads from memory, the memory address is divided into three components:

- Offset: Determines the specific byte within a cache line.
- Index: Chooses the set in the cache to look for.
- Tag: To find out if the data being looked for is available in the chosen set.

The tag is used by the CPU to look up the correct set to determine whether the data is present, a cache hit or not, a cache miss. Each line has a valid and a dirty bit. A valid bit tells us if the data in there is meaningful or not, in most cases a valid bit of 0 indicates a cold cache. The dirty bit is used for write-back caches to indicate that the value has changed and when the block will be evicted, it needs to update the main memory with the new value. On a hit, the data is retrieved with minimal latency. On a miss, the block must be retrieved from a lower level in the hierarchy (e.g., L2, LLC, or main memory), which is much more costly in terms of latency. There are generally 3 types of cache misses:

- Cold Miss: At start of a program, there is an empty cache. Cold misses are mandatory misses. They cannot be avoided even with infinite memory.
- Capacity Miss: A miss when cache cannot store all the data required.
- Collision Miss: Even with enough space, there are misses because the entries go to the same cache line, bad usage of associativity (mostly direct mapped or 2-way associativity)

In the instance of a capacity cache miss, when a new block has to be brought into memory, the system must determine which block should be replaced. The choice of replacement is according to the cache replacement policy (section 2.3). How good a policy is impacts cache hit rates and system performance directly. A good policy works by attempting to estimate which of the blocks won't be needed soon, in turn retaining the highly temporally local blocks.

In the context of cache, not all memory accesses are the same. There are 4 key instruction types, Load, Request for Ownership (RFO), Prefetch, and Writeback. These types are distinct from others (such as control instructions or computation-focused

operations) because they directly interact with the memory hierarchy and influence what data enters or updates the cache. Load instructions fetch data from memory for read operations and are often performance-critical. RFOs are triggered during write operations that require exclusive access to a cache line, impacting both coherence and placement decisions. Prefetches are speculative loads issued in advance to reduce future latency; while they can improve performance, they also risk polluting the cache if mispredicted. Writebacks occur when dirty data is evicted and must be written to memory—these accesses do not typically bring data into the cache, but they still affect replacement logic.

The design and optimization of such policies, particularly at the Last-Level Cache (LLC), where misses are most expensive, is essential. By seeking adaptive and intelligent eviction policies, we aspire to improve data persistence in cache and reduce reliance on slow main memory access.

2.3 Cache Replacement Policies

Cache Replacement Policies (CRP) are algorithms that decide on which cache block to evict in a line, in the case of a cache miss. The target of CRP is to reduce memory access time, done in two ways:

- Increase hit rate
- Decrease latency

Improving one often comes at the expense of the other. For instance, a policy that aggressively retains frequently used blocks may increase the hit rate but also introduce delays in decision-making and complexity in replacement logic.

The optimal cache replacement policy is the one that can predict which block, out of all the blocks, will be requested the latest in future cache accesses. That’s known as Belady’s Optimal algorithm and is impossible to achieve, since nobody can see into the future. Modern cache replacement policies try to predict which block is best to be evicted, through various techniques.

A policy that is widely used even today is LRU (Least Recently Used). LRU tracks previous entries in the cache and in case where all blocks are full, removes the one that was accessed the longest time ago.

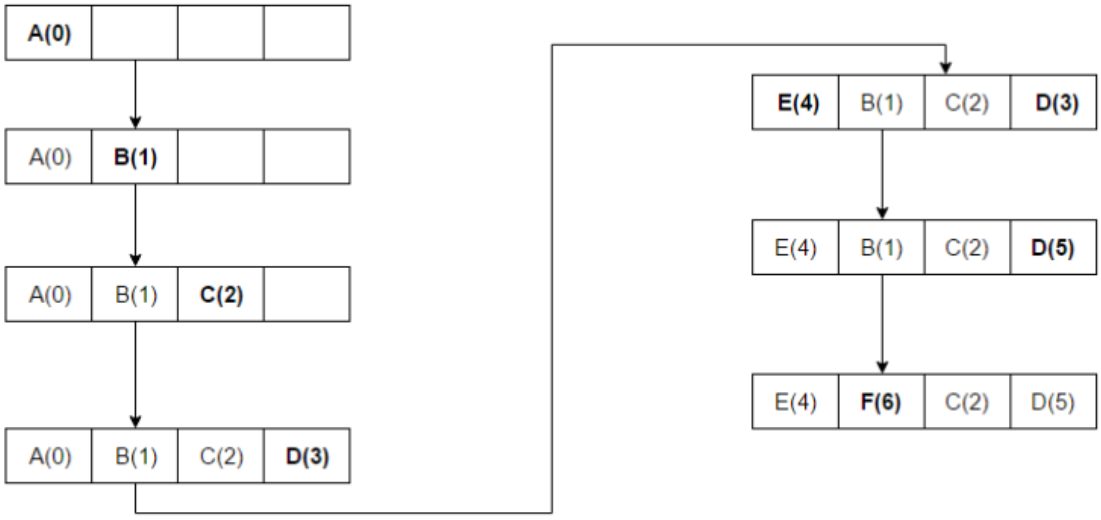

Figure 2.5: The LRU Policy

In the example above, we see that after each entry, each block has a number assigned to it, to keep track of when it was used. On a cache hit, all the numbers need to be updated, since the recency of each block changes, the one that was accessed becomes the most recently used element, and the other get an increment of +1 meaning they are least recently used than before. Computer systems use PLRU, which is a model that acts like LRU in most cases, but has significantly better hardware overhead. Traditional LRU needs $\log_2(N)$ bits per block, meaning $n\log_2(N)$, considering N-way associativity (for n=4 in the example above, each block requires 2 bits to track the position from 0 to 3, whereas PLRU only needs N-1 bits per set, making it more hardware efficient. This means that for a good cache replacement policy, hardware complexity is also a very important factor.

2.4 Genetic Algorithm

With so many cache replacement policies, a big search space is created in the attempt to find which policy suits a workload best. To solve this, we propose Genetic Algorithms (GAs). Genetic Algorithms are a class of search and optimization algorithms inspired by the natural process of selection and biological evolution. They belong to the broader family of evolutionary algorithms, which work best in solving complex problems with vast and ill-defined solution spaces where traditional optimization techniques are not adequate.

Essentially, genetic algorithms operate on a set of solution candidates, generally referred to as individuals or chromosomes. These individuals are all evaluated using a fitness function, where the fittest individuals (the ones with strongest genes) pass on to the next generation, while creating new individuals with traits from two of the fittest individuals. In Figure 2.4, we can see the workflow of a typical genetic algorithm.

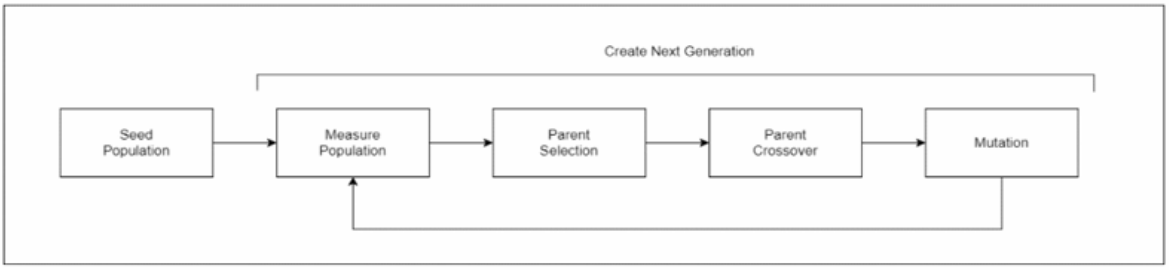

Figure 2.6: Genetic algorithm workflow

The steps that a genetic algorithm takes are explained as follows:

- Seed Population: In the start, we start with random individuals.
- Measure Population (Fitness Function): Every individual is evaluated through a fitness function that measures how well it performs towards the goal at hand. It is the natural analogue of survival of the fittest.
- Selection: Fitter organisms will be more likely to be selected for reproduction. Roulette wheel, tournament selection, and rank selection are the most widely used selection methods. In the experiments in this thesis, tournament selection is used.

- Crossover: Two chosen individuals are merged and parts of their genomes are exchanged to produce new individuals with traits from both parents. This generates diversity and allows beneficial traits to be passed on.
- Mutation: With a very small probability, random changes are applied to individual genes. Mutation maintains the genetic diversity of the population at the highest level and prevents converging too early to the suboptimal solution.
- Replacement: The new individuals are created, ready to be evaluated by the fitness algorithm. This process repeats until we reach the number of generations conFigured by the user.

Despite their versatility, applying genetic algorithms to cache replacement introduces certain challenges, most notably, how to accurately and efficiently encode a policy as a genome. The design of the fitness function and the representation of policy functions greatly influence the algorithm's success. Nevertheless, given the immense search space of potential policy configurations, GAs provide a promising foundation for discovering high-performing strategies tailored to specific workloads. This motivates our use of evolutionary techniques in this thesis (see Section 2.6.2) and leads to our proposed encoding framework in Section 4.3.2.

2.5 Microarchitectural Simulator

Microarchitectural Simulators give us the opportunity to emulate the behavior of a microarchitecture, by configuring many properties such as: memory size for all hierarchical levels, the branch prediction type and most importantly, the cache replacement policy to be used. This is very important for this thesis, since we are trying to run different experiments with a big variety of cache replacement policies, making it impossible to use real hardware. Microarchitectural Simulators are useful to test on what changes would be beneficial for a microarchitecture before applying them in practice. For the purpose of this thesis, ChampSim [6] (section 4.2) to run our simulations with different cache replacement policies, so we can evaluate each policy.

Chapter 3

SRRIP-DRRIP

3.1 SRRIP

On the talk of cache replacement policies, LRU works generally well, but it faces a big issue on some workloads, specifically on workloads with scans. Scans are data streams that are loaded/used only once. These types of workloads, when they are inserted into the cache, they get the MRU (Most recently used) position, even though they will never be accessed again, meaning that many entries that would be used later, are evicted, while cache stays polluted with such bad entries. An algorithm that would work well in this workload is MRU, which has the same logic as LRU, but instead of evicting the least recently used block, it evicts the most recently used block. This policy is scan-resistent, so it handles scans very well. The problem is that for any other workload, it has a very bad performance.

SRRIP [2] is a scan-resistant cache replacement policy that outperforms LRU, while being scan-resistant. SRRIP uses a counter of M bits for each block in the set, called Rereference Prediction Value (RRPV), which predicts if a cache block will be re-used. An RRPV value of 0 indicates that the block will be reused in the very near future, while a value of 2^M-1 (we consider M=2 for the examples below, since it performs the best) indicates that the block will be reused in the distant future, essentially saying that the block should be evicted from the cache. The SRRIP algorithm works as follows:

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.

On Insertion:

1. Set RRPV value to 2.

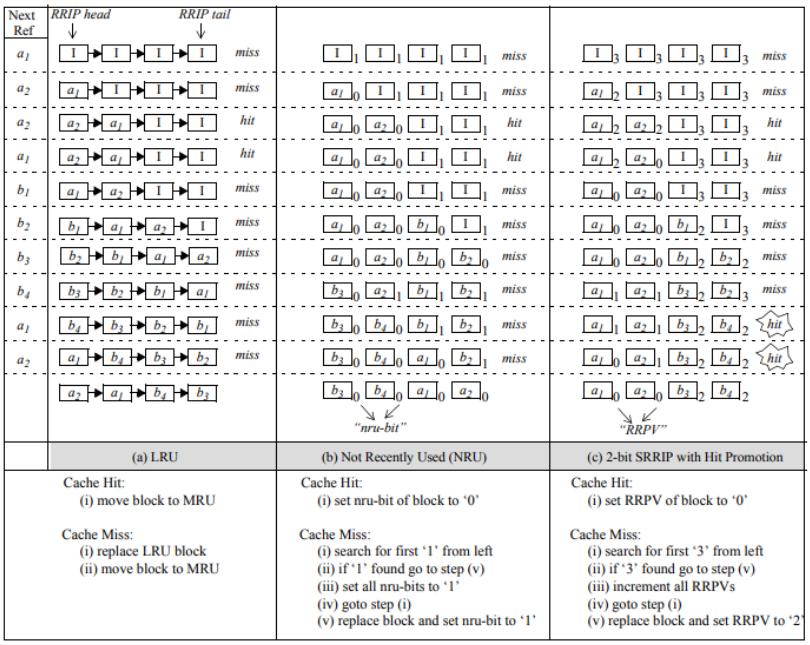

Figure 3.1: SRRIP compared to LRU and NRU

To better understand what makes SRRIP scan resistant and better than LRU, we consider the example shown in Figure 3.1. A steam of accesses on b1, b2, b3 and b4 are sent to the cache, and after there are normal accesses of frequent data like before the stream (a1 and a2). SRRIP is compared to LRU and NRU (which is SRRIP with M=1, showcasing how the M value is important to the policy being scan-resistant). We see that both LRU and MRU struggle to keep a1 and a2 inside the cache, but due to SRRIP inserting with RRPV 2 and by the way it handles eviction, we can see it keeps frequently accessed data, even after a sudden scan. It has to be noted that SRRIP works the same on all access types. In

later chapters we will see, that we can modify SRRIP to have a different behavior on these 4 different access types:

- Load: A read instruction that asks the CPU to get data from memory. If the data isn't in the cache, it causes a cache miss.
- RFO (Request For Ownership): Happens before a write. It asks for exclusive access to a cache line so the CPU can safely modify it.
- Prefetch: Tries to bring data into the cache before it is actually needed. It can help performance, but if the data isn't used, it wastes space.
- Writeback: Happens when changed data is removed from the cache and saved back to main memory. It doesn't fetch new data, but it still takes up a cache line and affects replacement decisions.

3.2 Set Dueling

While SRRIP performs well in workloads with scans, SRRIP still possesses a big issue. SRRIP performs bad with thrashing, which is a situation where constantly loading and evicting data entries of the working set, without keeping some of the working set data in the cache, caused when the working set is larger than cache. Also SRRIP is a static policy, which means that it does not have an adaptation system to the workload, it always performs the same steps, given all workloads. Set dueling [3] looks to fix this issue, by creating an environment where two policies can run at the same time, so the policy that best fits the current instruction sequence will be executed. Set dueling dedicates some sets (16 to 32) to each of the 2 policies, called the leader sets. So the leader sets will be always running one policy and evaluating it for the other sets (follower sets). We keep track of a counter called PSEL. PSEL is a saturating counter that keeps track of which of the two competing policies incurs fewer misses, one policy adds to the counter and the other subtracts if a miss occurs in the sets dedicated to it. The most significant bit of the counter determines the policy that performs better. Figure 3.2 illustrates how Set Dueling will work with LRU and BIP as competing policies.

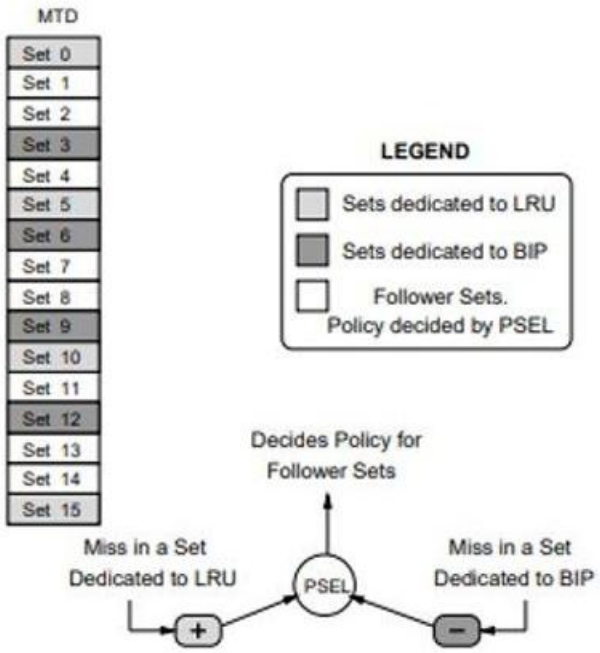

Figure 3.2: Set Dueling Mechanism on LRU and BIP

3.3 DRRIP

To make use of Set Dueling and provide a Thrash-resistant policy to help SRRIP with those workloads, BRRIP is proposed. BRRIP works by inserting with an RRPV value of 3 with a small probability of inserting with RRPV value 2. In thrashing scenarios, the working set is too large to fit in cache, and new data constantly replaces existing blocks. SRRIP might keep bringing in blocks that don't get reused, mistakenly giving them mid-term life and evicting more useful blocks .BRRIP, by contrast, treats most new blocks as likely trash, so it protects existing blocks longer—only admitting new entries cautiously. This results in fewer useful blocks being evicted prematurely, making BRRIP more resilient to cache pollution during thrashing.

Combining them together with set-dueling, we get DRRIP. DRRIP successfully combines the strengths of SRRIP and BRRIP by using Set Dueling to dynamically select the better-performing policy based on runtime behavior. This adaptability makes DRRIP effective across a wide range of workloads, balancing between scan-resistant and thrash-resistant behavior. However, despite its success, DRRIP remains a heuristic-based solution that

may struggle with workloads exhibiting more complex or mixed access patterns. As such, there is still an opportunity to improve cache replacement performance by developing more adaptive and learning-based techniques, this is explored in the next section.

3.4 Improving upon DRRIP

While DRRIP is a self-switching cache replacement policy between SRRIP and BRRIP, it still is stuck between those two specific policies, SRRIP and BRRIP. That is, DRRIP can never switch to another policy or even a custom version of SRRIP for a specific workload. For example, a workload might benefit from a policy that inserts into a different position or sells blocks differently on reuse. DRRIP can't do that because it's married to SRRIP and BRRIP.

In a recent study [4], the use of Genetic Algorithms (GA) to optimize tree-based Pseudo-LRU (PLRU) caches was studied. In that study, GA was used to discover enhanced insertion and promotion rules that improved performance on specific workloads. That showed that using an evolutionary approach like GA can be useful to discover new cache policies that capture the scope more than fixed policies.

Based on that idea, this thesis takes the use of genetic algorithms one step further, by refining how genetic search is executed and what it can tell us. Rather than simply searching for the best policy, we examine the performance of the search under different workload configurations, i.e., how results change when optimizing for a single benchmark vs. averaging across many benchmarks. This helps us identify whether workload structure affects what the GA learns. In addition, we study which components of a cache replacement policy contribute the most to good performance and which components contribute the least, knowledge that can be used to streamline or guide future designs. Finally, we study how many different policies are truly needed to deliver optimal performance for all workloads, and whether a small, well-designed subset can beat out a single general-purpose solution. These guidelines are designed to make the genetic search process not only more effective, but also more comprehensible and adapable to diversity in actual use.

Chapter 4

Frameworks and Methodology

4.1 HPC System

The experiments were carried out on a high-performance computing (HPC) platform located at the University of Cyprus. The platform is equipped with Intel® Xeon® Gold processors with a base frequency of 2.90 GHz. The platform has a dual-socket configuration with 32 physical cores in total, which is extremely appropriate for parallel computation. For the purpose of our research we will be running every experiment on a single core, while running many experiments across multiple cores, 1 per core. The operating system deployed is CentOS Linux, release 7.8.2003, providing a sTable and efficient platform upon which to run the high-level simulations required under this research. This powerful infrastructure was key to allowing the testing and evaluation of several cache replacement policies.

4.2 ChampSim

To evaluate candidate cache replacement policies in a controlled and realistic scenario, this thesis uses ChampSim [6], a trace-driven microarchitectural simulator specifically created for cutting-edge CPU and cache research. ChampSim is used throughout the academic and research community, serving as the testbed for a variety of large-scale competitions such as the 3rd Data Prefetching Championship (DPC3) and the 2nd Cache Replacement Championship (CRC2). Its extensible and modular design makes it especially well-suited to microarchitectural experimentation with functions such as branch predictors, prefetchers, and most importantly for this thesis, Last-Level Cache (LLC) replacement policies.

4.2.1 Cache Configuration

For our experiments we modeled a 1-core out-of-order processor with 3 levels of cache. The configuration used is described in Table 4.1. For the LLC replacement policy, we used a modular version of DRRIP replacement policy, configured to have the same policy switch between set-dueling, for running with single policies, and modified to set-duel between two policies for the last experiment. For all the simulations, we ran 100 million warm-up instructions, and 500 million execution instructions.

Parameter	Configuration
L1 I-Cache (Private)	32KB, 64B blocks, 8-way 8 MSHRs, 1 cycle latency LRU Replacement Policy
L1 D-Cache (Private	32KB, 64B blocks, 8-way 8 MSHRs, 4 cycles latency LRU Replacement Policy Next-Line Prefetcher
L2 Cache (Private)	256KB, 64B blocks, 8-way 16 MSHRs, 8 cycles latency

	LRU Replacement Policy IP-Based Stride Prefetcher
LLC Cache (Shared)	2MB per core, 64B Blocks, 16-way 32 MSHRs, 20 cycles latency Modular DRRIP-Based Replacement Policy
Branch Predictor	Perceptron

Table 4.1: ChampSim Cache Configuration

4.3 GeST

4.3.1 The framework

In this research, we utilize GeST [5] (Generator for Stress-Tests), which is a framework to automatically generate CPU stress tests using genetic algorithms. GeST is designed for the exploration of the effect of instruction-level workloads on processor performance and can target single microarchitectural structures such as functional units, the cache hierarchy, or the CPU pipeline to stress. GeST is modular by design and very flexible, and thus perfectly suitable for both cache behavior studies and replacement policy investigations.

The framework operates by evolving a population of candidate stress tests over a number of generations. Each individual in the population is a loop of assembly instructions, with the goal of placing high computational or memory pressure on the processor. The individuals are compiled and executed, and a fitness function, defined by the user, evaluates their performance in terms of instruction-per-cycle (IPC) or voltage fluctuation. In this dissertation, we modify the individual to be a cache replacement policy configuration, further explained in section 4.3.2, while changing the fitness function to be on speedup over a baseline LRU IPC. The evolution process follows standard genetic algorithm procedures: the fittest individuals are selected to reproduce the next generation, their instruction sequences are merged through crossover, and slight random changes are

introduced through mutation to ensure diversity and search space exploration. Figure 4.1 shows the overview of GeST.

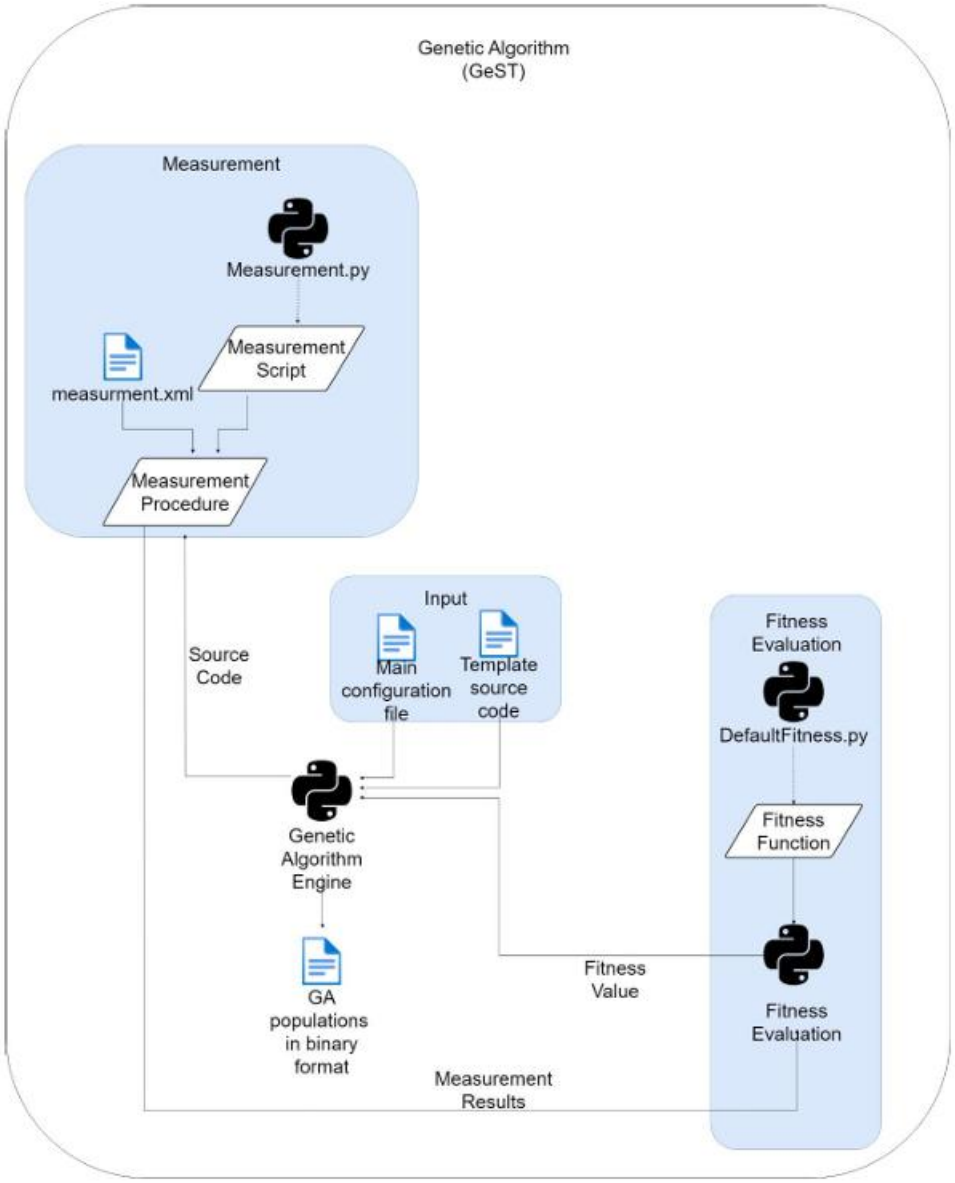

Figure 4.1: The GeST Framework

GeST gives complete control over these processes. Users can define the instruction set, loop structure, mutation rates, and even include user-defined fitness functions through a configuration file. These are implemented in separate files and can deal with single processor behaviors or with architectural aspects. This modularity enables researchers to adapt GeST to a wide range of use cases, e.g., the creation of test cases that resemble real-world workloads or that exercise particular known bottlenecks of the system. We will be

using this property of GeST to make our own configuration file, where we will be expressing cache replacement policy functions as the instructions.

Overall, GeST offers an efficient and flexible framework for developing insightful stress tests tailored to specific research goals. Its incorporation in this thesis enables a more comprehensive exploration of cache replacement policies and complements the validity of our simulation-based results.

4.3.2 Configuring policies as genomes

One of the trickiest challenges for this methodology is expressing a policy as a genome that can be processed by GeST. This includes questions as how to encode a CRP behavior, on eviction, insertion, promotion? We will be using the same logic as in prior work [1], which used SRRIP as a base and added different instruction types and functions for each instruction that determines how they handle promotion for clean and dirty blocks, Demotion for clean and dirty blocks, and insertion. Nikolas [1] proposed the 4 instruction types, mentioned in section 3.1.
For each instruction type, 5 functions are proposed:

- Insertion
- Clean Block Promotion
- Dirty Block Promotion
- Clean Block Demotion
- Dirty Block Demotion

Figures 4.2, 4.3 and 4.4 demonstrate how all this information was conFigured into GeST.

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

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

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

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

</instruction_types>
```
Figure 4.2: Configuring the instruction types in GeST

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

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

 <operand
id="SelfProm"
values="%0 %1 %2"
type="register"
toggle="False">
 </operand>
```
Figure 4.3: Configuring functions as GeST operands

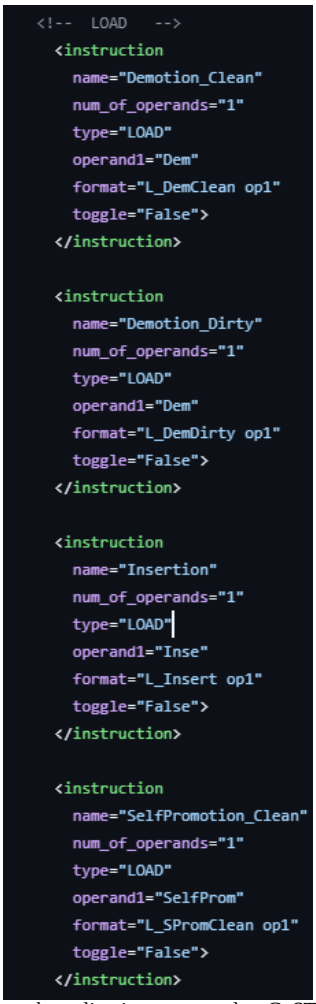

Figure 4.4: How the policy is represented as GeST instructions

Each function contains a value, Demotion taking values 0-4, Insertion taking values 0-3 and Promotion taking values 0-2. All these values represent a different mechanism of the policy that differs it from other ones. Tables 4.2, 4.3 and 4.4 describe the values that Demotion, Promotion and Insertion can take.

Demotion:

Value	Function
0	Default SRRIP Demotion
1	Asymmetric Demotion 1
2	Asymmetric Demotion 2
3	Asymmetric Demotion 3
4	Asymmetric Demotion 4

Table 4.2: Demotion function values

Asymmetric Demotions 1-4 were proposed in Nikolas' work. They are described as follows:

- Asymmetric Demotion 1:
    - Increase the RRPV of all blocks in the set by 1 until the maximum RRPV in the set is greater than 1.
- Asymmetric Demotion 2:
    - Increase the RRPV of blocks that have RRPV < 2 by 1, until the maximum RRPV in the set is greater than 2.
- Asymmetric Demotion 3:
    - Increase RRPV of blocks with RRPV < 2 by 1 until the maximum RRPV is greater than 2.
    - If the maximum RRPV is 0, set all blocks' RRPV to 2.
- Asymmetric Demotion 4:
    - If maximum RRPV is 0, set all blocks' RRPV to 2.
    - If maximum RRPV is 1, increase all blocks' RRPV by 1.

Promotion:

Value	Function
0	Assign RRPV = 0 to the promoted block
1	Assign RRPV = 1 to the promoted block
2	Assign RRPV = 2 to the promoted block

Table 4.3: Promotion function values

Insertion:

Value	Function
0	Assign RRPV = 0 to the inserted block
1	Assign RRPV = 1 to the inserted block
2	Assign RRPV = 2 to the inserted block
3	Assign RRPV = 3 to the inserted block

Table 4.4: Insertion function values

4.4 SPEC 2017 Benchmarks

Figures 1.1 and 1.2, fueling the motivation of this thesis, were implemented from a top-down analysis on the SPEC 2017 Benchmark Suite by Intel, which consists of 20 benchmarks of different workloads, an ideal selection for testing our model. For the experiments, we generated 20 traces on the benchmarks, and gave them as input to ChampSim, to test all the policies.

4.4.1 The benchmarks

Initially, we tested ChampSim on all 20 benchmarks. The list of the initial benchmarks can be seen in Table 4.5. We needed a baseline to compare to the policies generated by GeST in order to evaluate the fitness function on speedup (see section 4.5), so we used LRU as the baseline. When testing the 20 benchmarks on ChampSim, we thought that due to the diversity of the benchmarks, some of them would not necessarily be demanding to the cache, potentially weakening our results. We call these benchmarks ‘boring’, since they do not test the LLC, meaning that any policy in the LLC would yield the same results for all policies.

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

Table 4.5: The initial SPEC CPU 2017 traces used

We set a threshold on 0.2 LLC misses PKI to differentiate the boring benchmarks from the ‘exciting’ benchmarks. The results after simulating LRU in LLC to serve as a baseline we got the results showcased in Table 4.6:

Benchmark	**LLC Misses/KI**
Xalancbmk	31.071
Fotonik3d	24.713
Lbm	23.714
Parest	17.452
Omnetpp	14.972
Wrf	13.011
Roms	11.79
Mcf	10.887
cactuBSSN	2.636
Blender	1.989
Cam4	1.503
Gcc	1.186
x264	1.112
Xz	0.916
Bwaves	0.109
Imagick	0.044
Leela	0.037
Perlbench	0.009
Povray	0.004
Exchange	0.002

Table 4.6: The benchmarks sorted according to LLC misses PKI

We can clearly see a huge gap between Bwaves and Xz, where the threshold was also placed. That means that the last 6 rows are boring benchmarks, at least for the purposes of this thesis, and will be left out from further experiments.
Concluding, the final benchmark selection can be seen in Table 4.7.

Blender	Wrf
CactuBSSN	X264

Cam4	Xalancbmk
Fotonik3d	Xz
Gcc	
Lbm	
Mcf	
Omnetpp	
Parest	
Roms	

Table 4.7: The benchmarks sorted according to LLC misses PKI

4.5 Evaluation Methodology

The genetic algorithm works by first creating a set of random candidate policies. These are passed to an interface that runs simulations on each policy using the full set of benchmarks. RunSimFromGest.py takes each individual, and converts them into 3 16-bit masks, which ChampSim decodes into the policy he will be using on the simulations. After ChampSim finishes running all the traces, ipc_parser.py extracts the IPC from the ChampSim output files.

In this paper, something we did differently than Nikolas was using speedup in comparison to the baseline (LRU) as the fitness function, instead of pure IPC. Using speedup over the baseline as a metric allows us to measure the actual improvement a policy brings compared to a baseline, such as LRU, making it easier to compare across different workloads. In contrast, pure IPC values can be misleading, as some benchmarks naturally have higher or lower IPC due to their characteristics, regardless of the replacement policy used.

Next, the genetic algorithm uses the best-performing policies to create a new generation through selection, crossover, and mutation. This process is repeated over several generations, allowing the algorithm to gradually improve the quality of the policies it produces. Over time, it finds more effective cache replacement strategies that are better suited to different types of memory access.

Figure 4.5 shows how the genetic algorithm works together with ChampSim through the interface.

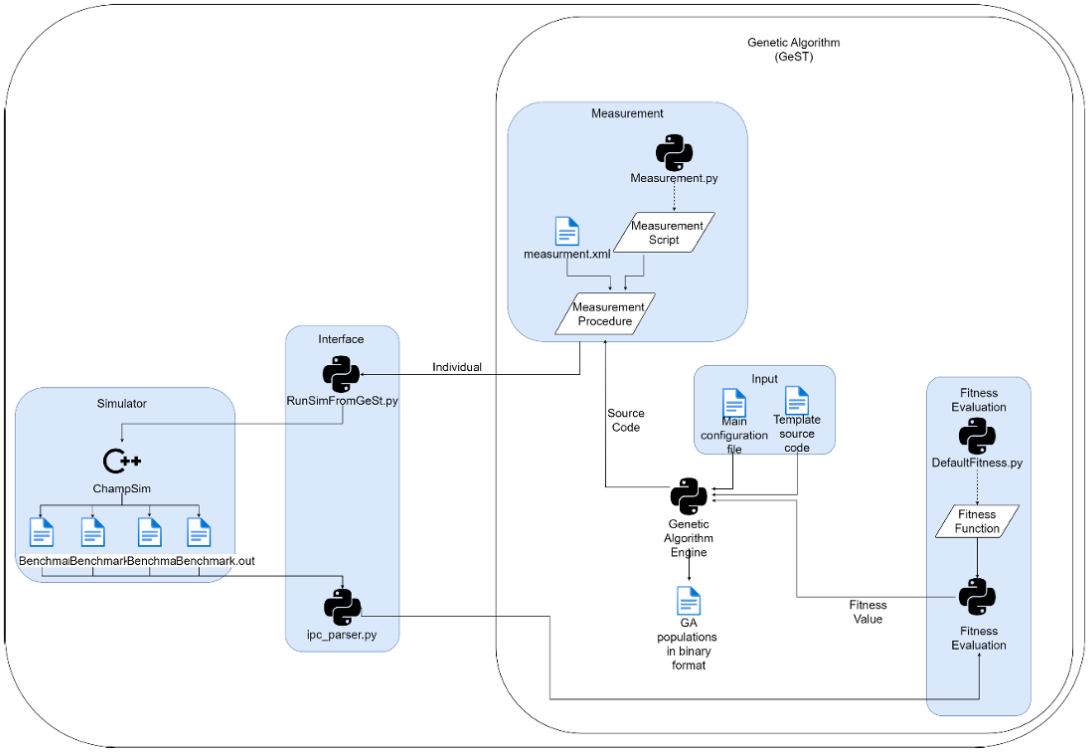

Figure 4.5: Using GeST together with ChampSim for evaluation

Chapter 5

Workload experiments

5.1 Isolated and averaged workloads

The first experiment we conducted aimed to test whether running the genetic framework on all benchmarks together was limiting its ability to converge on an optimal policy—at least for individual benchmarks—due to the differences between workloads. To explore this, we ran 15 simulations: one using all benchmarks together (averaged workload), and the remaining 14 using each benchmark in isolation as its own workload. One of our goals was to examine whether the averaged workload run could discover a strong policy for a single benchmark, but then discard it in later generations because it only performed well for that specific case. This would suggest that some policies are highly specialized and require specific configurations to perform optimally.

As a final objective of this experiment, we also compared the outcome of the averaged workload run against a manually designed policy created from prior work, to assess

whether the framework could discover an even better-performing solution. The manual policy configuration is shown in Table 5.1.

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

Table 5.1: Manually found SRRIP configuration from previous work

5.1.1 Comparing isolated workload to averaged out workload

The results of our first experiment can be seen in Figure 5.1. We first observe that, on average, the fittest individual of the averaged out workload beats the manual SRRIP, showcasing the benefits of genetic search. Secondly, we can see how the orange bar, being the personal best of the benchmark on the averaged out workload, is never kept as the fittest, shown by the orange bar being higher than the blue bar in all benchmarks. This is an indicator that the workloads want a very specific policy to perform well, such a policy that is clearly suboptimal on the other workloads, bringing the average speedup down, and consequently not being passed to later generations.

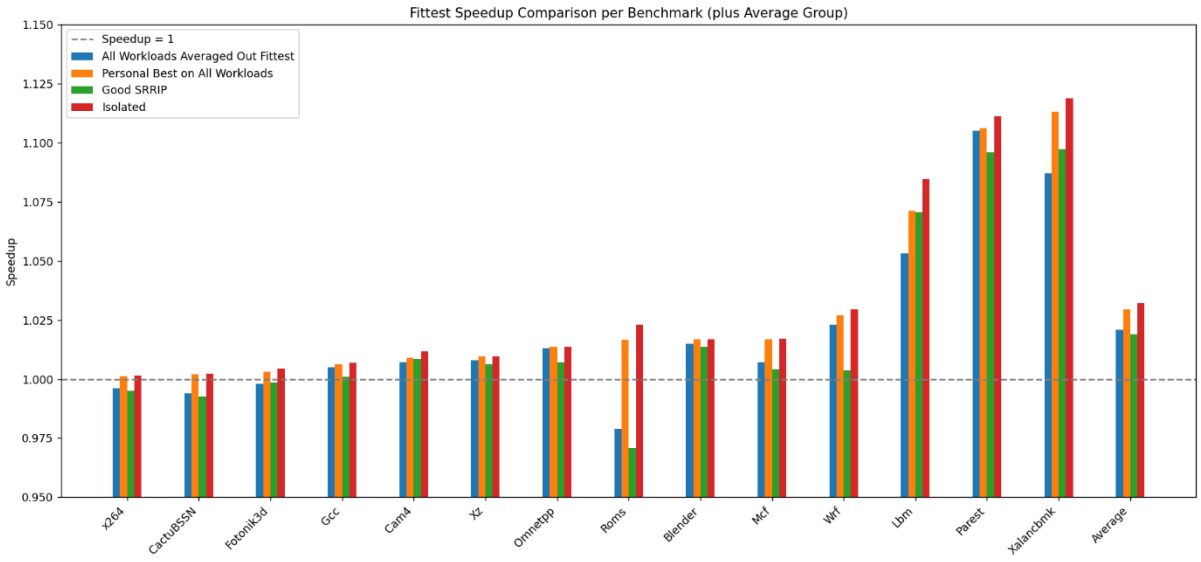

Figure 5.1: Results of first experiment, showcasing speedup over LRU

The final and most important observation is that the red bar, which corresponds to the fittest individual in the isolated workload simulation, dominated the blue bar in most cases, and has an edge over the orange bar in most cases, showing that the averaged out search limits the optimality of a workload, since that policy won't perform well on other benchmarks, it will not be passed to further generations. This result, also seen in the average plot, clearly shows that there is a benefit on running one benchmarks instead of averaging out many of them. The biggest difference observed is that of Lbm, having a clear edge over the other bars.

5.1.2 Similarity between policies

From section 5.1.1, we saw that when running one benchmark as the workload, the search converged to a better result than the best policy for that benchmark on the averaged out workload. The next thing we needed to check was, where do these policies differ from each other, in the same workload. So what functions were important for it to be of optimal speedup, and which functions were seen with different values in the fittest results? We separated the 99.9% speedups for each workload, that being a subset with all the policies that had the highest speedup value, and we got several policies per workload. Some of them were duplicates passed from generation to generation, so those had to be eliminated. We then needed a way to measure the logical ‘distance’ between two policies, to have a metric on how different they are, and to separate what function makes them different.

We came up with an algorithm, that when given all the 99.9% of the policies, will give us a statistical analysis of the functions that were and were not important, while also providing a comparison metric between two policies.

The algorithm works as follows:

1. Brute force all the combinations of 2 policies in the 99.9 percentile
(The algorithm runs unique combinations, so it doesn't run Policy A - Policy B and after Policy B - Policy A, that would hurt the statistics)
2. We keep track of the distance in an integer initialized to 0
    2.1. For DemotionClean and DemotionDirty:

If the configuration on both policies is the same number, counter remains the same.

If they are different, the difference between them does not matter, we just add 1 to the counter to resemble them being different.

2.2. For Insertion, PromotionClean and PromotionDirty:

We add to the counter the difference in absolute value (i.e. Suppose PromotionDirty %0 and PromotionDirty %2, |0-2| = 2 so we add 2 to the counter).

3. If the counter remains 0, that means the policies are identical. In that case, we remove one of the 2 policies from the list.

4. For those with counter > 0 , we keep track of which configurations were different and what was the difference

5. Repeat 1-4 until all policies left are unique.

6. Compute some statistics on all the 'differences' summed up (for all the duos), and present a distribution on which configurations were different across all comparisons

Consider this output:

Overall difference distribution:

L_DemotionClean: 5 (12.5%)

L_DemotionDirty: 8 (20.0%)

L_Insertion: 10 (25.0%)

L_PromotionClean: 7 (17.5%)

L_PromotionDirty: 10 (25.0%)

Here we have total 40 differences (for insertion and promotion if a difference would be 2, like example at 2.2, that would count as 2 'differences', although its just on one conf iguration pairing).

For each configuration we do /40 to get how many of those differences as a % were demotionclean etc.)

An example:

A:

L_DemClean %1

L_DemDirty %0

L_Insert %2

L_SPromClean %0

L_SPromDirty %0

B:

L_DemClean %0

L_DemDirty %4

L_Insert %3

L_SPromClean %2

L_SPromDirty %0

C:

L_DemClean %1

L_DemDirty %0

L_Insert %2

L_SPromClean %0

L_SPromDirty %0

A - B

For L_DemClean, different, counter +=1

For L_DemDirty, different, counter +=1

For L_Insert, counter+=|2-3|, so counter +=1

For L_SPromClean, counter+=|0-2|, so counter +=2

For L_SPromDirty, counter+=|0-0|, so counter +=0

Keep track that 1 difference from L_DemClean, 1 from L_DemDirty, 1 from L_Insert, 2 from L_SPromClean, 0 from L_SPromDirty

A - C

For L_DemClean, same, counter +=0

For L_DemDirty, same, counter +=0
For L_Insert, counter+=|2-2|, so counter +=0
For L_SPromClean, counter+=|0-0|, so counter +=0
For L_SPromDirty, counter+=|0-0|, so counter +=0

Keep track that 0 difference from L_DemClean, 0 from L_DemDirty, 0 from L_Insert, 0 from L_SPromClean, 0 from L_SPromDirty

B - C

For L_DemClean, different, counter +=1
For L_DemDirty, different, counter +=1
For L_Insert, counter+=|2-3|, so counter +=1
For L_SPromClean, counter+=|0-2|, so counter +=2
For L_SPromDirty, counter+=|0-0|, so counter +=0

Keep track that 1 difference from L_DemClean, 1 from L_DemDirty, 1 from L_Insert, 2 from L_SPromClean, 0 from L_SPromDirty

We check for duos that had 0 differences overall (Identical policies)
Finds A-C, deletes C from list

So in second iteration we only have:

A:
L_DemClean %1
L_DemDirty %0
L_Insert %2
L_SPromClean %0
L_SPromDirty %0

B:

L_DemClean %0

L_DemDirty %4

L_Insert %3

L_SPromClean %2

L_SPromDirty %0

A - B

For L_DemClean, different, counter +=1

For L_DemDirty, different, counter +=1

For L_Insert, counter+=|2-3|, so counter +=1

For L_SPromClean, counter+=|0-2|, so counter +=2

For L_SPromDirty, counter+=|0-0|, so counter +=0

Keep track that 1 difference from L_DemClean, 1 from L_DemDirty, 1 from L_Insert, 2 from L_SPromClean, 0 from L_SPromDirty

We check for duos that have 0 differences overall, none found.

Now we compute the statistics, total we have 1 + 1 + 1 + 2 + 0 = 5 'differences'. (if there were other duos we would add the differences to the total, so if we had another duo that tracked 1 difference from L_DemClean, 1 from L_DemDirty, 1 from L_Insert, 2 from L_SPromClean, 1 from L_SPromDirty, total would be 11).

For each configuration type, we count how many times they appeared and divide with total (* 100 to display as a percentage).

So:

For L_DemClean, 1/5 = 20%

For L_DemDirty, 1/5 = 20%

For L_Insert, 1/5 = 20%

For L_SPromClean, 2/5 = 40%

For L_SPromDirty, 0/5 = 0%

(With the other duo we considered before: 1 difference from L_DemClean, 1 from L_DemDirty, 1 from L_Insert, 2 from L_SPromClean, 1 from L_SPromDirty

For L_DemClean, 2/11 = 18.18%
For L_DemDirty, 2/11 = 18.18%
For L_Insert, 2/11 = 18.18%
For L_SPromClean, 4/11 = 36.36%
For L_SPromDirty, 1/11 = 9%)

After running this algorithm on the different isolated benchmark workloads, we got some interesting results. The results can be seen in Figure 5.2.

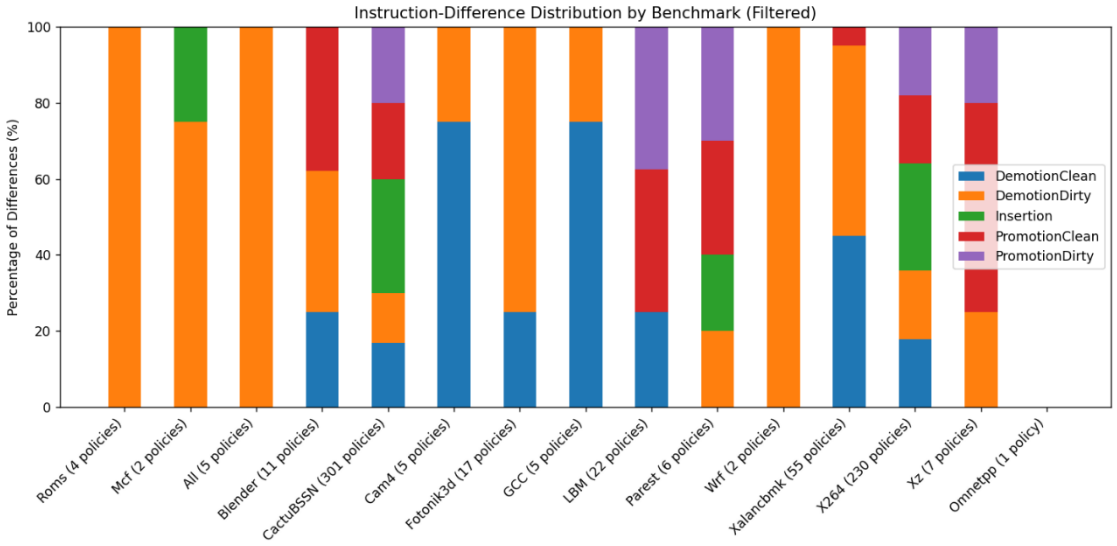

Figure 5.2: The difference distribution between functions on 99.9 percentile of policies.

We see that most benchmarks have a difference that corresponds only to DemotionClean and DemotionDirty function values, demonstrating that these two functions can change, and still the benchmarks will have optimal speedup, while the other function values are critical to its optimality. There are some interesting cases like CactuBSSN and X264, having differences in all functions. Those two benchmarks are interesting, by seeing the number of policies in the 99.9 percentile, there would be a thought that they are boring benchmarks, since the subset in 99.9 percentile is massive. This is not the case for those benchmarks, as they are examples where most policies will work the same on them, but some policies can perform very badly on them, causing a large decrease in speedup. We

can see from section 4.4.1, that these 2 policies have sufficient cache activity to not be considered boring. For the other benchmarks, we can see that in some of them PromotionDirty and PromotionClean can be of different values, but the most important function by far is that of insertion.

5.2 Finding the ideal policy pairs

We saw from section 5.1.2 that most function values are quite important to the optimality of a policy, meaning that finding a single policy that would produce optimal results for all the benchmarks is a tough challenge. However, if we could produce a pair or several pairs of policies with different functionalities, there could be a possibility that this pairing between the policies could produce optimal results for all the workloads. In this section, we try to answer this question, by trying to find these policies and putting them to the test.

5.2.1 Similarity between workloads

As a first step, we ran an algorithm that checked all the policies on the 99.9 percentile of each workload (like in section 5.1.2), but instead of computing a distance, it checked for equality of the policies. The thought process we used was that by having a policy that appears in multiple isolated workloads, it means that there is a pattern that can yield optimal results for multiple policies. The results can be seen in Figure 5.3.

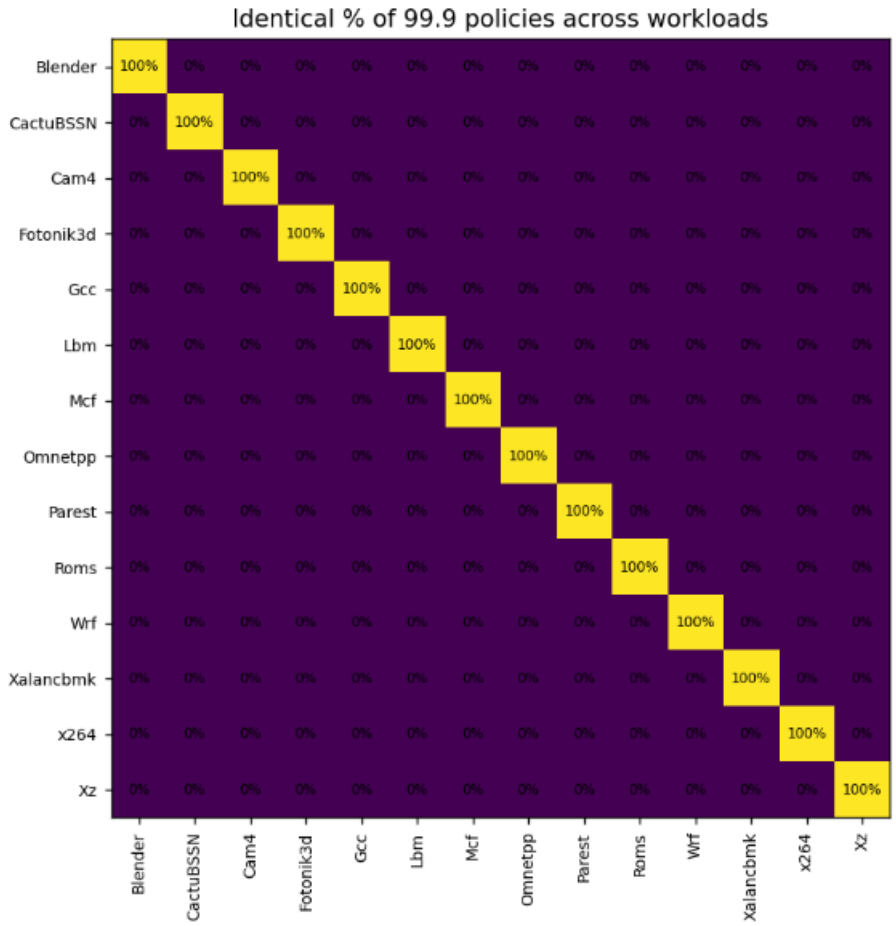

Figure 5.3: Similarity % between all pairings of isolated workloads

Even by including the benchmarks CactuBSSN and X264, which we originally thought would pollute the results due to its high policy count in the 99.9 percentile, we see that there are no identical policies between any combination of workloads, besides obviously them compared to themselves (the diagonal). From this result, we understand that the policies are very specific on their policies, and those policies are quite unique compared to the optimal policies of the other workloads. This poses quite a challenge to find good pairings.

5.2.2 Minimum policies needed to cover all workloads

Although from the experiment of section 5.2.1 we saw that there was no pattern between the 99.9 percentile policies of two benchmark workloads, we implemented an algorithm that would take the popular function values from each function, and see how ‘popular’ they are amongst the 14 workloads, and build some policies using the most popular function values, in a way that, if we use n-way set-dueling between the n policies, we would obtain optimal results.

In this context, a single policy is trying to cover many benchmarks, by possessing function values that were included in many of the workloads, from that we try to construct the minimum amount of these policies. For a function value to be considered popular for a benchmark, it must be included in at least 1 of the subset's fittest policies.
The algorithm is explained as follows:

1. We take for each workload, the policies

For each workload, we load every config file (99.9% percentile, meaning the fittest score) and pull out its 20 function settings (like L_Insertion %2, R_DemotionClean %4, etc.).

Result: for each workload and each function (e.g. L_Insertion), we build a small set of "function values I saw in at least one of the workload's fittest policies"

2. Flipping the Perspective: Function-Value → Which Workloads?
Instead of "which values does the workload like?", we ask "for a given function setting—say L_Insertion=2—in which workloads did that ever show up?"

So we basically check in which benchmarks this function value appeared at least in 1 fittest policy (meaning that it is valid to use for the policy to be fittest for that workload)

3. Turning Lists into 14-Bit Masks
Now imagine a 14-slot attendance chart (one slot per workload). For each function-value pair (F=v), we build a bitmask:

(i.e. L_Insertion=2)
Blender CactuBSSN Cam4 … x264 Xz
1 0 1 … 1 0
(meaning L_Insertion=2 was found in at least 1 of the fittest policies of Blender, Cam4, x264, but never found in CactuBSSN and Xz, suggesting that this value for L_Insertion does not provide good results for CactuBSSN and Xz, but good results for the other 3).

– A “1” means “yes, I saw this function value in at least 1 of the fittest policies of this workload.”
– A “0” means “no, I never saw this function value in any of the fittest policies of this workload.”

Mask example for L_Insertion:

val=2 → 10100001110101

val=1 → 01001010000010

val=3 → 00000100001000

val=0 → 00010000000000

4. Covering All Workloads with the Fewest Settings

The goal: pick as few values of L_Insertion as possible so that every one of the 14 workloads sees at least one “1.” In bit terms, we want to OR our chosen masks and end up with 11111111111111.

We use a simple greedy:

1. Pick the value whose mask covers the most yet-uncovered workloads.
2. Mark those workloads as covered.
3. Remove that value from future consideration.
4. Repeat until all 14 are covered (or we run out of values).

5. Example: L_Insertion for the benchmark 'Blender'

We had these masks (Table 5.2):

Value	Mask	Popcount
2	10100001110101	7

1	01001010000010	4
3	00000100001000	2
0	00010000000000	1

Table 5.2: Example of the masks for each function value and its popularity count

(Popcount = in how many workloads L_Insertion %Value appears in a 99.9% configuration) (number of 1's in the mask)

1. Step A: Start uncovered = all 14 workloads.
   The biggest mask is val=2 (covers 7).
   Choose 2, mark those 7 as covered.
2. Step B: Now 7 remain.
   Among the remaining masks, val=1 covers 4 of the uncovered.
   The Mask of using val=2 and val=1 becomes:
   11101011110111
3. Step C: 3 still uncovered.
   val=3 covers 2 of them—pick 3.
   The Mask of using val=2 and val=1 and val=3 becomes:
   11101111111111
   Now 1 workload left.

4. Step D: Only one workload still hasn't seen L_Insertion, and that's exactly the one where val=0 showed up—so we pick 0.

After picking the 4 values, we have 11111111111111, so we cover all workloads on achieving the fittest result (for this one function)

Result: we needed 4 different L_Insertion values (2, 1, 3, 0) to guarantee every workload had at least one fittest match.

6. Repeat for All 20 Functions

We do the same thing for each of the 20 configuration types (L_DemotionClean, R_PromotionDirty, etc.). Some functions only need 3 distinct values to cover all 14 workloads, others need 4, etc.

7. Take the most popular function value for all 20 configuration types, and create a policy, the same with 2nd most popular and so on, until all are needed values are covered.

We tested this algorithm on our results and produced the following results (Table 5.3). The x values in the Table mean (Don't care). There are 4 policies that the algorithm produced. The 4th policy can be considered redundant, due to it only needing a unique function value from R_DemotionDiry, which was concluded in section 5.1.2 as a function that is not so important to the policy. We will be taking these policies to the test in section 5.5.

Feature	Policy 1	Policy 2	Policy 3	Policy 4
L_DemotionClean	0	2	4	×
L_DemotionDirty	0	1	2	×
L_Insertion	1	2	3	×
L_PromotionClean	2	1	×	×
L_PromotionDirty	0	1	2	×
R_DemotionClean	1	0	2	×
R_DemotionDirty	1	2	0	4
R_Insertion	3	2	×	×
R_PromotionClean	2	1	×	×
R_PromotionDirty	0	1	×	×
W_DemotionClean	4	2	0	×
W_DemotionDirty	4	2	0	×
W_Insertion	0	2	1	×
W_PromotionClean	1	2	×	×
W_PromotionDirty	0	1	×	×
P_DemotionClean	4	0	1	×
P_DemotionDirty	1	0	3	×
P_Insertion	2	3	0	×
P_PromotionClean	2	0	×	×
P_PromotionDirty	2	0	×	×

Table 5.3: The results after running the algorithm

5.3 Finding an ideal crossover point

When initially conducting the experiments, we had a thought that maybe the searching of the genetic algorithm could be helped by a randomized crossover point, something that was done differently in prior work. In prior work, a crossover point in the middle of the instruction sequence was used, but we proposed a different searching style, to see if the genetic algorithm would converge to an even better value. We ran the framework once with a middle crossover point, and the other with a random crossover point, and got the results presented in Figure 5.4. The results showed that many benchmarks converged to the same value, but a few benchmarks had a different outcome.

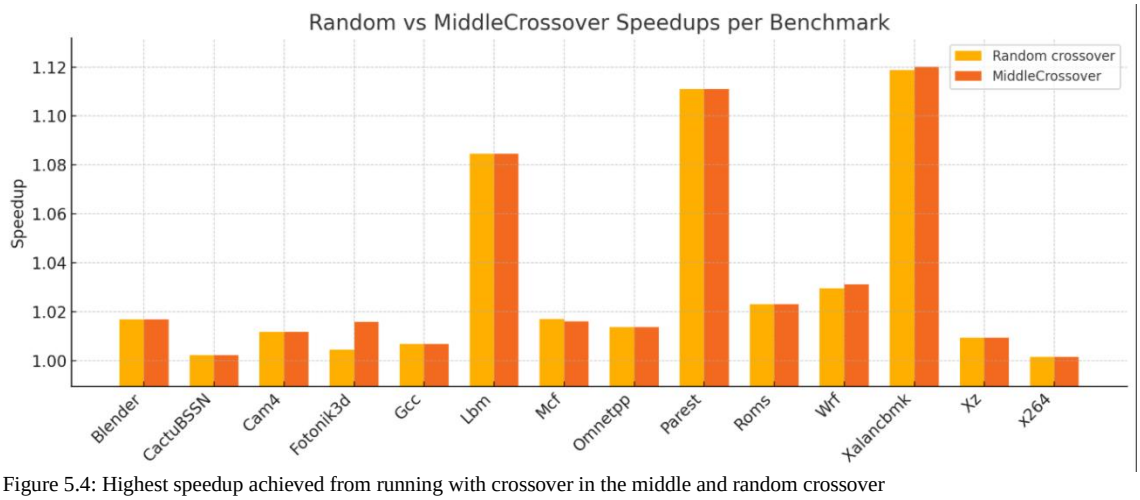

Figure 5.4: Highest speedup achieved from running with crossover in the middle and random crossover

As seen in Figure 5.4, most benchmarks converge to the same value, but if we observe benchmarks Fotonik3d, Wrf and Xalancbmk, there is a difference between the two runs. In most cases, the method with crossover point set in the middle performed better than a random crossover, especially for Fotonik3d, which had an increase of 1% in the difference of speedups. These results solidify that the edge the crossover in the middle had was not pure luck of the search, but we can see that the randomness of the individuals of the first generation has a big impact on where the GA converges, as seen in Mcf, where the random crossover point outperforms the middle crossover point. As the experiments were done with random policies in the first generation, this emphasizes the importance of the policies that are randomly generated in the first generation, giving us the idea that maybe if we used a manual set of individuals for the first generation, we can maximize the convergence of the genetic search.

Demonstrating the better convergence of the middle crossover configuration, we can observe the convergence of Fotonik3d benchmark with both random and middle crossover point in figure 5.5.

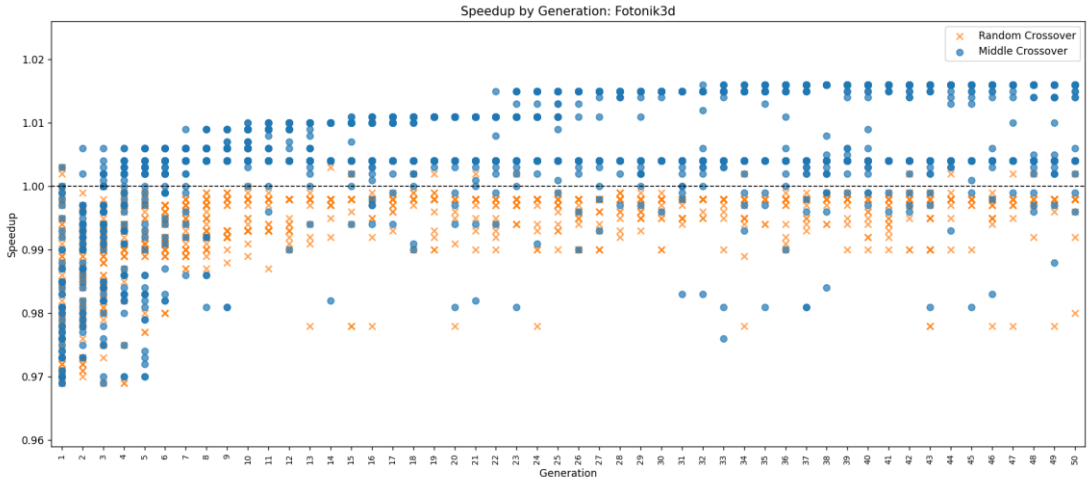

Figure 5.5: The speedup scores per generation, with random crossover and middle crossover

The main reason behind this improved search is because a middle crossover keeps the values of a single instruction type organized, meaning that all functions of load instructions will be grouped together, while a random crossover can separate the functions of the same instruction type, which can damage the behavior on that instruction type.

5.4 Impact of Initial Population on Convergence

Following up to the findings in Section 5.3, where we determined that genetic search is influenced by structural properties such as crossover placement, we conducted another experiment to evaluate how the initial population influences convergence. That is, we wanted to see whether using a manually selected group of strong individuals as the initial generation would guide the genetic algorithm towards better solutions, as compared to having an entirely randomly created initial population.

To achieve this, we created an initial population with 3 of the many fittest policies for each benchmark isolated workload. The reason we chose this, was to see if the genetic algorithm would converge to a better result and find the function combinations on its own, that would give good results for many benchmarks as the workload. We were not looking for optimal results for all the benchmarks, but just for a way to see if a more 'targeted' start would help the genetic search. We compared it to a typical randomized initial population.

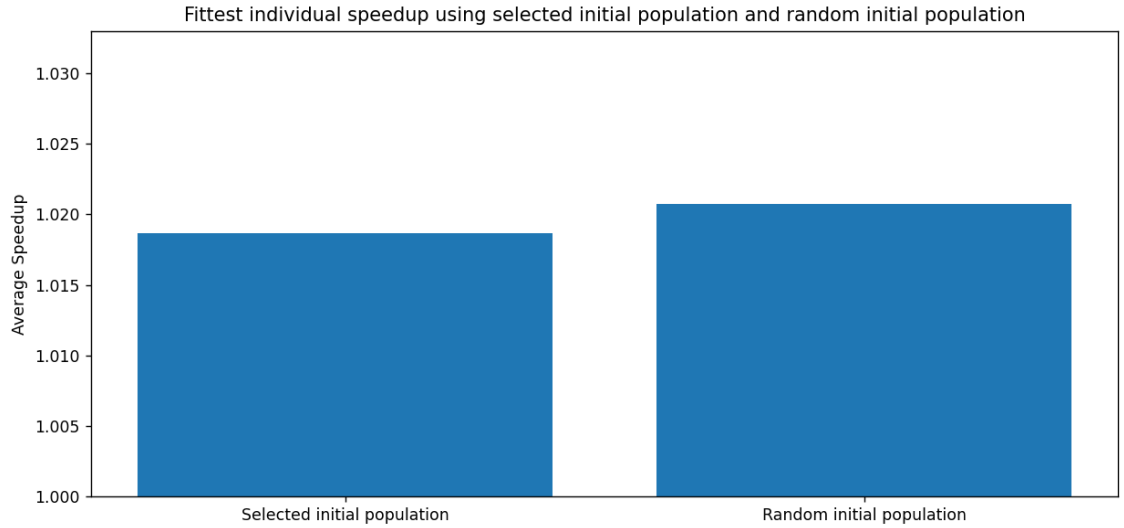

Figure 5.6: Fittest individual when running with selected initial population and with random initial population

Figure 5.6 shows the outcome of comparison. It shows the best average speedup achieved by each approach. The one with the initial population that was started with the selected population did worse than the random start, while noting that all individuals of the 1st generation had an average speedup score less than 1.

To observe these results compared to results from previous experiments, we added the fittest individual from this experiment and the personal best per benchmark on that simulation in figure 5.1. The results are displayed in figure 5.7. Note that for the personal best we did not consider the 1st generation, as the 1st generation policies are several isolated best policies per benchmark, to see the trends with these added bars.

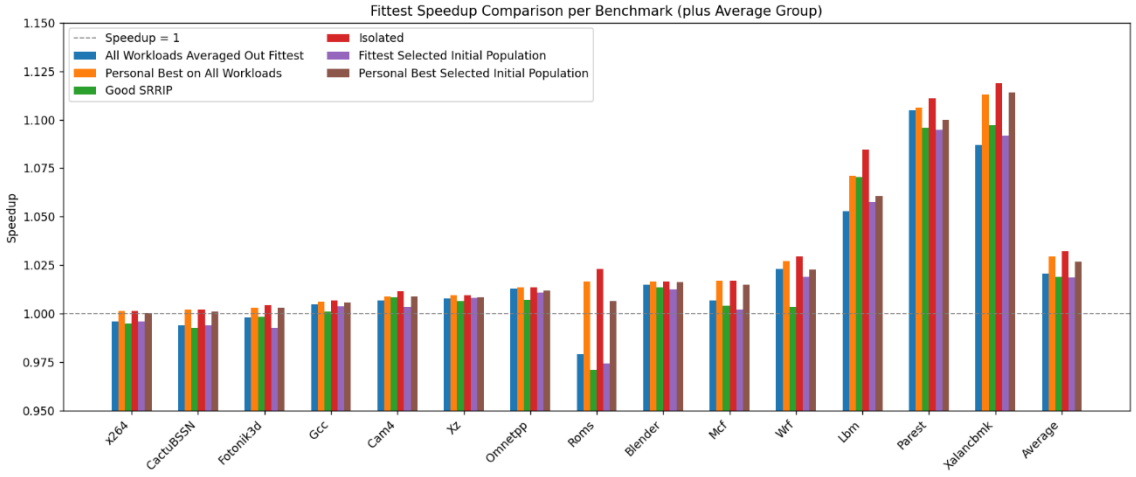

Figure 5.7: First experiment side-to-side with results from selected initial population

We can see that the personal bests all drop below the isolated personal best, meaning that it compromises to a 'worse' policy for that benchmark, to obtain higher scores in the other benchmarks. We also see that the personal bests without the 1st generation are a little worse than the ones found in the random initial population policies, showing that that also stays consistent with the results of the fittest policy of random against selected initial population.

This experiment contradicts the initial hypothesis, showing that starting with a carefully selected or partially optimized set of policies can hinder the search process. The convergence of the genetic algorithm was significantly limited by the specificity and bias introduced by these pre-selected policies. Rather than allowing the algorithm to broadly explore the solution space, the targeted initialization narrowed the search too early, preventing discovery of policies that generalize well across benchmarks. In contrast, a fully random initial population enabled greater diversity, which led to better results by uncovering policies that perform well across a wider range of workloads.

5.5 Testing the policies together

Taking the policies produced from section 5.2.2, we put them to the test to see if those policies really improve compared to their singular best counterpart. We used set-dueling between policy 1 and policy 2. This would answer our initial question on whether the popularity of a function was sufficient to construct policies that would cover the needs for all benchmarks. In Table 5.5, we display the results of the two policies used with

set-dueling and compare it to the fittest policy found for the averaged out workload, also seen on section 5.1.1.

Benchmark	Speedup of two popular policies	Fittest result of averaged out workload
Blender	1.0077	1.015
CactuBSSN	0.9929	0.994
Cam4	1.0046	1.007
Fotonik3d	0.9894	0.998
Gcc	1.0019	1.005
Lbm	0.9991	1.053
Mcf	0.9945	1.007
Omnetpp	1.0046	1.013
Parest	1.1823	1.105
Roms	0.9568	0.979
Wrf	1.0073	1.023
Xalancbmk	1.0933	1.087
x264	0.9944	0.996
Xz	1.0052	1.008
Average	1.0111	1.020

Table 5.4: Comparison of the two ‘popular’ policies with the fittest result found on previous experiments

We can see that the policies did not improve the speedup on average, which tells us the popular policy approach did not work and did not find good policies. It found the good functions, but it did not focus on function combinations of the same instruction (like in section 5.3). A method proposed for future work would be to enhance the algorithm to work for feature combinations of an instruction type, instead of a function on its own.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Through the experiments conducted, we learned that cache policy optimization on individual benchmarks would always result in optimal performance as against the use of a composite workload. This aligns with the hypothesis that workload-specific tuning is required for best performance. Second, through the analysis of functional differences among top policies, we found that insertion behavior contributed the most toward policy success, while demotion settings contributed less.

We also attempted to explore whether a few policies were sufficient to accommodate all the workloads with set-dueling. While the synthesized "popular" policies reflected most frequent function values, the overall outcome was not better than the best single policy achieved for the workload averaged out. This proved that a good policy is not only a matter of possessing the proper individual functions but also of how the functions are combined—particularly in the same instruction type.

We also proved that even a small implementation decision such as the crossover point in genetic recombination could affect the search outcome. A middle-point crossover preserved instruction type structure better, and as a consequence, in certain cases, it led to superior results. These findings confirm that convergence and quality in searching rely heavily on both initial policy creation and the application of genetic operations.

Overall, our results vindicate the idea that one-size-fits-all is not going to work. The challenge that comes with that, is to find the method to find the policies that together can produce near-optimal results for many benchmarks as the workload.

6.2 Future Work

While this thesis has managed to explore the design space of cache replacement policies with genetic algorithms, there are several areas left to investigate. One such area is the creation of more instruction-type-conscious policies. Our results indicated that how functions like insertion and promotion are categorized within the same instruction type has a tendency to affect performance more than their respective values. It may be typed in the genetic representation to incorporate these aggregations and help the algorithm evolve better-organized and optimized policies.

Another key area is the possibility of dynamic switching between policies. Rather than having a fixed policy or a hardcoded pair, upcoming work can investigate mechanisms that can infer runtime workload changes and switch to the most appropriate pre-evolved policy. This will allow the system to remain responsive and adaptive to access pattern variations.

Another important factor is using the importance of each feature to limit the search space. By removing or not considering non-important features, the search space of the GA will be severely smaller, giving it more opportunity to converge to better results.

Finally, while trace-based simulation with ChampSim was utilized within this research, further work would be to deploy and test these developed policies upon actual hardware. This would ascertain whether goodness observed in simulation is carried over

to real systems, keeping in mind real-world behavior such as pipeline stalls, OS noise, and hardware-level interactions that are not easy to model with simulation.

References

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

[2] A. Jaleel, A. et al. (2010) High performance cache replacement using re-reference interval prediction (RRIP). Available at: https://doi.org/10.1145/1815961.1815971

[3] Moinuddin K. Qureshi The University of Texas at Austin et al. (2007) Adaptive insertion policies for high performance caching, ACM SIGARCH Computer Architecture News. Available at: https://doi.org/10.1145/1454115.1454145

[4] Jiménez, D. (2017) Insertion and Promotion for Tree-Based PseudoLRU Last Level Caches. Available at:
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7847633

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

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