CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Implementation and Evaluation of Last-Level Cache Partitioning

Master's thesis in Computer science and engineering

Joel Blom Rydell
Felix Bråberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Implementation and Evaluation of Last-Level Cache Partitioning

Joel Blom Rydell
Felix Bråberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Implementation and Evaluation of Last-Level Cache Partitioning

Joel Blom Rydell, Felix Bråberg

Supervisor: Mehrzad Nejat, Madhavan Manivannan
Examiner: Per Stenström, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Implementation and Evaluation of Last-Level Cache Partitioning

Joel Blom Rydell, Felix Bråberg
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

With parallel applications executing on a multiprocessor comes the problem of interference due to resource sharing between processor cores. The contention of resources can lead to a lack of fairness between the cores. Furthermore, having a shared last-level cache introduces destructive interference in the form of cores evicting other cores' cache blocks. Cache partitioning is a method to solve these problems, which has both been extensively researched in academia and implemented in end-products in industry. In this thesis we will analyze the effects of cache partitioning in terms of performance and overhead. We implement six replacement algorithms and a cache design in RTL, all supporting way-based cache partitioning. These are then used in a test bench where execution is simulated by feeding the test bench memory traces extracted from the L2-accesses of benchmark programs from the SPEC-CPU-2006 benchmark suite.

From our results we conclude that way-based partitioning can easily be implemented with little overhead. We also show that in some cases, the increase of shared ways available to an application does not offset the effect of interference, which leads to worse performance of the application. From the aforementioned memory traces we generate reuse histograms to analyze the expected cache behaviour of the simulated benchmark programs. This gives us the ability to predict a good way to partition the cache, based on the running applications.

Keywords: Cache, cache partitioning, replacement algorithm.

Acknowledgements

We would like to thank our supervisors Mehrzad Nejat and Madhavan Manivannan for their support throughout this project. Without your help this thesis would not be possible.

Joel Blom Rydell & Felix Bråberg, Gothenburg, 2024-08-28

Contents

List of Figures

List of Tables

1

Introduction

With the introduction of chip multiprocessors (CMPs) comes the problem of interference due to resource sharing between processor cores. When cores contend for access to resources such as cache space or memory bus usage, performance issues arise. This is because, among other reasons, the contention of resources can lead to a lack of fairness between the cores. Furthermore, having a shared last-level cache introduces destructive interference in the form of cores evicting other cores' cache blocks. To solve these problems in the context of shared last-level caches, we can utilize cache partitioning [1], [2].

1.1 State of the Art

There are two primary problems which has to be solved to implement cache partitioning, which are the *allocation* and the *enforcement* problems. The allocation problem relates to how the cache should be partitioned for each application based on desired performance metrics. This problem has been the primary target for research. The enforcement problem is concerned with how a certain allocation should be enforced in hardware, and how the enforcement is implemented and guaranteed. A central part of the enforcement problem is the *replacement algorithm*, and how it can be used to enforce the allocated partitions. We will firstly look at what has been done in research in regards to the allocation problem. We will then present some solutions to the enforcement problem that has been implemented in industry and mention how the replacement algorithm is utilized in enforcement. After this we will go over some of the shortcomings that exist in the state of the art and how we think this thesis can help bridge this gap.

1.1.1 Cache Partitioning in Research

The first of the two most important mechanisms of cache partitioning is the question of how resources should be allocated for each application, which is mainly done with software management. For instance, the allocation policy might decide that application one gets to use 75% of the cache and application two gets to use 25% of the cache. The decision on how to partition the cache is an important task to maximize throughput, ensure quality of service (QoS), and guarantee fairness of the running applications. Therefore, the allocation policy has been subject to extensive research [3].

A cache can, broadly speaking, be divided into *sets* and *ways*. A cache consists of a number of sets and each set contains a number of ways where memory blocks are placed. These concepts are necessary to know since the cache can be allocated with different levels of granularity, which means how finely we can partition the cache for each application. The different levels of granularity are *set-based*, *way-based*, and *block-based* allocation granularity [2]. See section 2.3 for more information. Below are some examples of what has been done in research for different allocation policies.

Way-based is the coarsest of the allocation policies but also the most common allocation policy since it is relatively easy to implement. Way-based partitioning has been subject to most of the research done to allocation policies in research [2]. Besides being rather easy to implement, way-based partitioning does not require a cache flush when the allocation is reconfigured [4].

Set-based partitioning provides finer granularity than way-based allocation and works by allocating cache resources on a set-basis. A common method for doing this is called *cache coloring* [5].

Block-based partitioning offers the finest granularity of the three allocation policies but is also the hardest to effectively implement [2]. One proposed block-based partitioning technique is called *Vantage*, which offers block-granularity partitioning while maintaining high associativity and strong isolation between partitions [6].

1.1.2 Cache Partitioning in Industry

Cache partitioning has been widely designed and implemented onto system-on-chips (SoCs) by multiple companies. Today there exists technologies for cache partitioning in commercial SoC that enforces cache allocation based on the partitioning set by the user.

Advanced RISC Machines (ARM) has developed a framework called Memory System Resource Partitioning and Monitoring (MPAM) which allows the cache to be partitioned using identifiers. With this technology, applications are assigned an identifier and the cache resources are partitioned on a per-identifier basis [7].

Another example is a technology developed by Intel called Cache Allocation Technology (CAT) [1]. For a more detailed explanation of how CAT works, see section 2.3. We decided that Intel's CAT will be the main reference technology used for the work in this thesis.

Since Intel's CAT implements the enforcement policy on a way-based granularity, it utilizes the *replacement algorithm* to enforce the cache partitioning, and is therefore an integral part of the cache partitioning scheme. When data has to be removed from the cache and re-written to main memory, it is up to the *replacement algorithm* to decide what data should be removed. This algorithm is implemented directly in hardware. More information about replacement algorithms can be found in 2.2.

1.1.3 A Gap in the State of the Art

From what has been done both in research and industry, to the best of our knowledge, there has not been done any work to analyze the effects of cache partitioning in terms of performance and overhead. Specifically when it comes to overhead: the hardware overhead needed to implement support for partitioning in the cache design and replacement algorithms, and the extra latency needed to support partitioning.

1.2 Goals

The intended aim of this thesis is to look at the current state-of-the-art of shared last-level cache (LLC) partitioning and explore the effects of using different replacement algorithms. The algorithms will be implemented in hardware, to enforce the partitioning. The goal is to find how the choice of replacement algorithm for cache partitioning realized in hardware will impact the performance and what overheads it will incur.

Main Goals

The goals of the thesis are the following. Firstly, we want to analyze and make high-level designs of different replacement algorithms that partition the cache with inspiration from Intel's Cache Allocation Technology (CAT) system [1]. Then, we want to implement the designs designs in Register Transfer Level (RTL) for simulation. Lastly, we're going to evaluate the RTL-designs in terms of performance and overheads.

1.3 Contributions

In our work we have made an analysis of how partitioning of the LLC affects the execution of co-running applications with regards to performance. We analyze the implications of implementing support for way-based partitioning of the LLC in regards to added hardware and latency.

In RTL, we have implemented six replacement algorithms; *Binary Tree*, *DRRIP*, *LRU*, *NRU*, *Random*, and *Binary Tree Private*, which is a further development of the binary tree replacement algorithm. For details of how it works, see section 3.4.5. We also developed in RTL a simple cache design that all support way-based partitioning. We have also implemented test benches to test and run the replacement algorithms and cache design, both in a single and multi-application context.

We created an experimental framework that uses a modified version of the SniperSim computer architecture simulator [8]. The modified SniperSim is used to extract memory request traces of benchmark applications going from an L1-cache to an L2-cache. These memory traces are then used when performing RTL simulations of the cache design. We've also created a script to generate histograms representing the accumulated reuse distances of all addresses of the specific memory trace.

1.4 Summary of Outcomes

In this thesis we show that way-based partitioning can be implemented with very little overhead to storage and hardware. We also show that for larger caches, the effect of interference is larger than giving an application access to more shared ways. This shows that partitioning can increase performance of an application using cache partitioning.

1.5 Structure of the Report

The report is organized in the following manner: Chapter 1 introduces the problem statement and states the goals that this thesis will attempt to fulfill. Chapter 2 gives all the necessary background information that is required to properly grasp the concepts of the thesis. Then, in Chapter 3 we explain the design choices that were made and how the cache is structured. In Chapter 4 we state the methodology we used to gather results, such as the tools that were used in simulations and the size of the cache. Also we cover what evaluation metrics we are looking at in this thesis. Chapter 5 presents the results that was gathered and discusses the implications of said results. Lastly, chapter 6 concludes the work by summarizing the findings and discusses what potential future work can be done.

2

Theory & Background

This chapter will present the necessary theory and background to understand the problem that this thesis is trying to solve. In section 2.1 we explain how a typical cache hierarchy works and talk about the concept of set-associativity. Further, in 2.2 we talk about what a replacement policy is and go over the ones we have chosen to work with in this thesis. In section 2.2.5 we cover the concept behind cache thrashing and scans. Then, in section 2.3 we go over the idea behind cache partitioning. Lastly, in section 2.3 we explain the basics of Intel's *CAT* technology.

2.1 Cache Hierarchy

In modern computer systems the standby time for moving data far exceeds the instruction execution time. To bridge this gap in latency, a small and fast memory is placed near the processor to store the most frequently used data. See Figure 2.1 for an example of a typical cache hierarchy with multiple cores. This memory is called a *cache* memory and is always searched before going to main memory. Only if the requested data is not found in the cache, a request to the main memory is issued. This way, the cache memory decreases the number of long-latency accesses to the main memory and therefore reduces the overall standby time of the processor. Generally there is a trade-off between the size and speed of a cache memory, and therefore most computer systems have the cache system organized in a hierarchy with the smallest but fastest memory at the first level (also referred to as L1 cache) and the slowest but largest as the LLC. Memories with attributes in between these two are placed at an intermediate hierarchical level. All memory requests will first be checked in the L1 cache, and if the data is not present the L2 cache will be checked and so on until the requested data has been found.

Set-associative Caches

To determine how a specific memory block can be placed within the cache-memory, a *cache mapping* is used in order to assign a certain memory block to a cache line. Broadly speaking, there are three main types of cache mappings: direct-mapped, fully associative and set-associative. In a *direct-mapped* cache a block in the memory can only be assigned to a specific location in the cache, and no other assignment is possible. In a *fully associative* cache, any memory block can be assigned to any cache line in the cache. In the set-associative mapping the cache is divided into *sets*

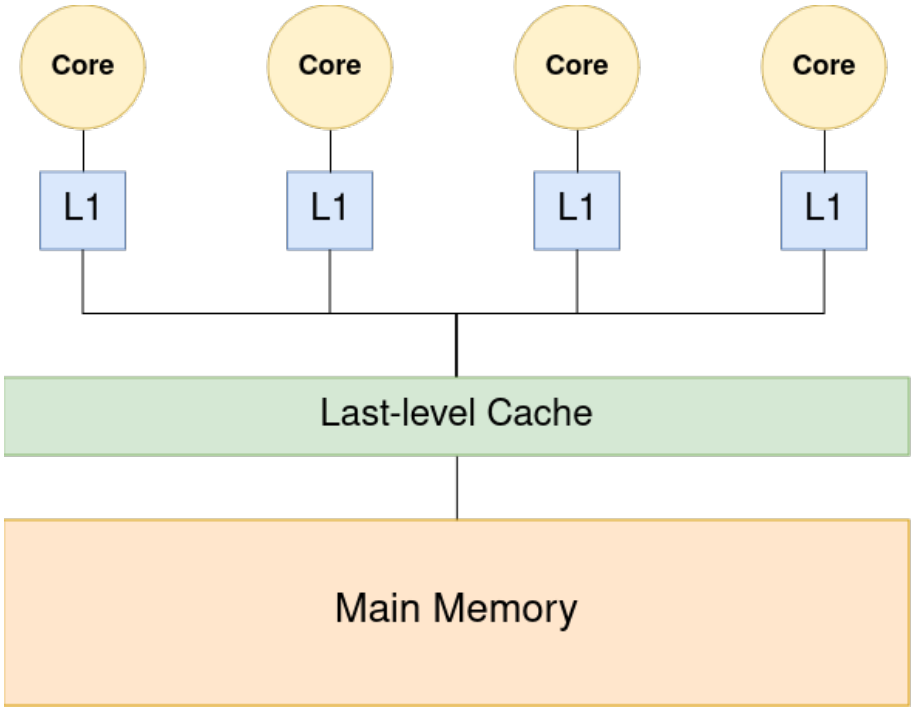

Figure 2.1: *A typical multi-core cache hierarchy.*

which in turn contains *ways* where the cache lines are stored. The number of ways in each set can differ but is usually in the range of two to eight [9].

A memory address is divided into three sections as seen in Figure 2.2. Each time a memory address is requested from the main memory, the address and neighboring addresses are fetched as one memory block. The *offset* part of the address is meant to uniquely identify each memory address within one memory block. A memory block will therefore contain 2^{offset_width} number of addresses. We usually refer to this memory block as a *cache line*.

In a set-associative cache, multiple memory blocks will exist in a cache set. Which cache set the memory block in question gets assigned into depends on the bits in the *index* part of the address. The number of sets in a cache will therefore be 2^{index_width}.

To be able to uniquely identify different memory blocks in a set, we use the *tag* part of the address. The tag is therefore used to see if there is an address in the set that matches the requested address. The tag part of the address gets assigned the remaining bits of the address after the index and offset bits have been determined. If, for example, we have an address width of 32 bits and 10 of those bits are for the index and five bits are for the offset, it means that the tag would get assigned the remaining 17 bits.

Equation 2.1 can be used to calculate the total cache size. If we, for example, have an 8-way set-associative cache, with 32 addresses per memory block and an index width of eight bits. The total cache size would be $cache\_size = 2^8 \times 8 \times 32 = 65\,536$ bytes or 64 KiB.

Figure 2.2: *The structure of a 32-bit cache address. Index specifies what set the memory block belongs to, the tag is used to uniquely identify the block within the set and the offset is used to specify the desired address and data within the block.*

$$cache\_size = 2^{index\_width} \times associativity \times cache\_line\_size \tag{2.1}$$

The way that a cache hit- and miss occurs in a set-associative cache is illustrated in Figure 2.3. The *index* part is used to get the tags and data of all the cache lines in a certain *set*. All tags are then compared in parallel with the desired *tag* taken from the address. If one tag matches the desired tag, there will be a cache *hit* and the corresponding data will be returned to the requester. If no tags match with the desired tag, a cache miss will occur and the data will have to be fetched from higher levels of cache or the main memory. After fetching the desired data, one of the data lines in the cache will have to be replaced with the new data. The choice of eviction candidate is made with the *replacement algorithm*. This is explained further in the following section.

2.2 Replacement Policy

Due to the limited capacity of a cache, lines sometimes have to be evicted and replaced with the desired line. This process of choosing an eviction candidate and replacing said candidate with another line is called *replacement*. The decision of which cache line to evict upon a miss is determined by the *replacement policy*. The simplest replacement policy is the *random* replacement policy which evicts a randomly chosen line. Different replacement policies exist with different performances such as least-recently-used (LRU), first-in-first-out (FIFO), pseudo-LRU, *etc* [10]. The ideal replacement policy is called *OPT* and uses information about the future memory accesses to determine the eviction candidate. This is useful because it can then evict cache lines that will be referenced furthest into the future. The OPT replacement policy is impossible to implement since we can't see into the future but it is useful for determining an upper bound on the performance that can be acquired from changing the replacement policy [9].

In the rest of this section we explain a few of the replacement algorithms in more detail. We cover TrueLRU which is a well known replacement algorithm and move on to a few pseudo-LRUs which are more practical and commonly used versions of LRU. We also explain random replacement which is the most simple algorithm with minimum overhead. Additionally, we look into a more advance algorithm proposed

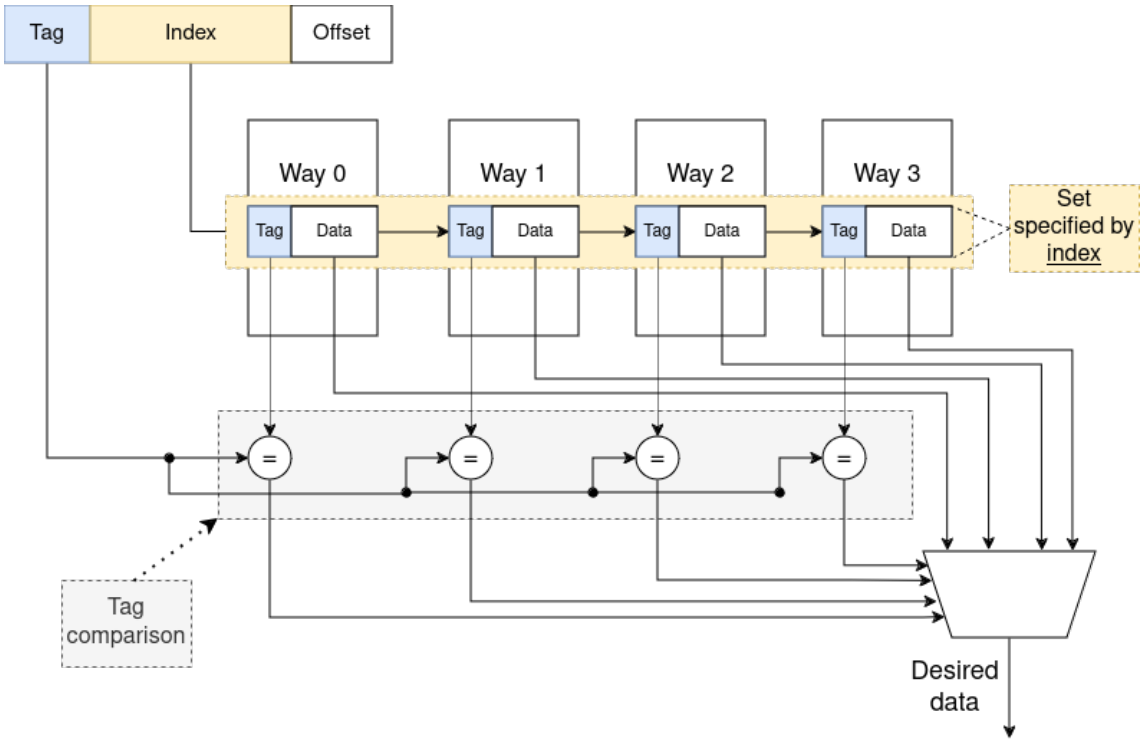

Figure 2.3: *An illustration of how a 4-way set-associative cache looks for a specific cache line in a cache set using the tag part of the memory address.*

for applications exhibiting thrashing and scan behavior (see section 2.2.5) called Re-reference Interval Prediction (RRIP).

2.2.1 Random

In the random replacement policy, the victim way is selected in a random or pseudo-random fashion. No past cache accesses need to be kept track of since the victim selection is not based on any access history. For this reason, random replacement is often cheaper in terms of hardware compared to other replacement algorithms [11].

2.2.2 TrueLRU

TrueLRU is one of the most common cache replacement policies. It keeps track of when a cache line in a set was accessed by associating a counter to each line in a set to keep track of where each line is in the access stack. For example, in a 4-way associative cache, 2 bits will be associated with each line, where `00` means that the line was most recently used and `11` means it is the least recently used line. When a cache line is accessed, its recency bits will be set to `00` and all lines between `00` and `11` will be increased. By doing this we can maintain the access stack of the accesses of the lines in a set. To do this, the replacement mechanism requires a counter for each cache line in a set for all sets in the cache. This means that the storage overhead for TrueLRU is $A(log_2 A)$ bits per set (where A is equal to the associativity) [12].

2.2.3 Not recently used

Not recently used (NRU) works by associating one bit per cache line that indicates if a specific cache line was recently used or not. When a cache line is inserted into the cache the NRU bit of that cache line is set to one, indicating that the line was recently used. When the cache set is full and all cache lines has its NRU-bit set to one, all lines in the set will have its NRU-bit flipped to zero. Upon a miss when a victim way should be decided, the first line found that has its NRU-bit set to zero will be evicted. The new cache line is inserted into the cache and has its NRU-bit set to one. The NRU-bit therefore indicates if a cache line was recently used and can be seen as if that specific cache line will be re-referenced again in the near future [12].

2.2.4 Binary tree

Another replacement policy is one using binary trees. Here, we utilize one binary tree per cache set with $A - 1$ (where A is equal to the associativity) nodes and find a replacement candidate by traversing the tree. The values inside the nodes will dictate whether we should descend into the upper or lower child of every node starting from the root node. The binary tree is always initiated with zero in each node of the tree, and when the tree is traversed, the value in the traversed node is inverted (a zero becomes one and vice versa). An example of a 4-way set associative cache using binary trees can be seen in Figure 2.4. If there is a *one* in the node, it indicates that the most recently used (MRU) cache line is found in the upper half of the tree and similarly if there is a *zero* the MRU is in the lower half tree. Thus, to find the LRU cache line we would want to go in the opposite direction than the nodes are pointing, which in the case of Figure 2.4 is down at the first level of the binary tree and then up [12].

2.2.5 Re-reference Interval Prediction

Re-reference interval prediction (RRIP) is a proposed replacement policy that tries to improve cache performance in cases where the applications experience bursts of references to scans or the working set of the application is larger than the cache. Applications that exhibit such behaviour perform poorly under LRU. A burst of references to non-temporal data is called a *scan* and can vary in length. The data referenced during a scan will not be referenced again in the near future but will still evict the working set from the cache, resulting in poor cache performance. RRIP tries to consolidate the working set within the cache by associating each cache line with a re-reference prediction value (RRPV) that consists of M bits[1]. The RRPV acts as a quantification of the predicted re-reference for the specified cache line. A low RRPV predicts a near re-reference while a high value predicts a distant re-reference. When a new cache line is inserted into the cache, the RRPV will be set to $2^M - 2$ for that cache line (i.e. the second-to-last place in the recency stack). The reasoning behind this is that all cache lines that have a distant re-reference interval will keep from polluting the cache. When a cache line is re-referenced from the

[1]From here on out, M will be assumed to equal two (M=2), since it was found to be the optimal value in terms of cache performance [3].

cache, the corresponding RRPV is set to zero. When an eviction from the cache is to be determined, the first found cache line with a RRPV of three ($2^M - 1$) is to be evicted. Since the RRPV of a cache line is statically determined upon a hit or miss, this version of RRIP is called *Static Re-reference interval prediction (SRRIP)* [3].

Cache Thrashing & Scans

Cache thrashing is a type of cyclic memory access pattern that arises when there is an access sequence of length k different addresses where k is larger than the number of available cache blocks in the cache. If this is the case, the working set of an application does not fit in the cache and the application will then start to experience cache misses since the data in its working is always being evicted [3].

A *scan* is a short burst of references to data that, after it is initially referenced, will not be re-referenced again in the near future of the execution of the application. Therefore a scan will pollute the cache with data that is known not to be re-referenced in the near future. Scans are common in real world applications [3].

2.2.6 Bimodal and Dynamic Re-reference Interval Prediction

SRIPP does not perform well when the re-reference interval is larger than the cache, and will result in thrashing and no cache hits. To overcome this, another variant of RRIP is *Bimodal Re-reference interval prediction (BRRIP)*. In this variant, most of the cache lines are given an RRPV of $2^M - 1$ when inserted into the cache and some will be inserted with an RRPV of $2^M - 2$. If an application exhibit non-thrashing access patterns, solely relying on BRRIP can cause degradation of cache performance. Ideally we would like to be able to dynamically switch between these two policies depending on the access patterns of the application, and that is the idea behind *Dynamic Re-reference interval prediction (DRRIP)*. DRRIP works by dedicating a few sets in the cache, called *Set Dueling Monitors* (SDMs), to statically use either the SRRIP or BRRIP algorithm. The size of an SDM is set to 32, since it has been found that 32 sets is sufficient to estimate the performance of the cache for the certain policy that the SDM should track [4]. These SDMs are then used in a technique called *set-dueling* to dynamically determine which policy to run on the follower sets in the cache [3]. The set dueling technique is covered in detail in the section below.

Set dueling

The idea of set dueling is to track the hit rate of two or more replacement policies and dynamically choose the policy with the best performance. There will be some number of dedicated sets called *set dueling monitors (SDMs)* that will use one of two replacement policies A or B. All SDMs share a saturating counter called *PSEL* that will be incremented when policy A has a cache hit and be decremented when policy B has a cache hit. The sets that are not SDMs will be *follower sets* that will use whichever replacement policy that performs best depending on the value of the

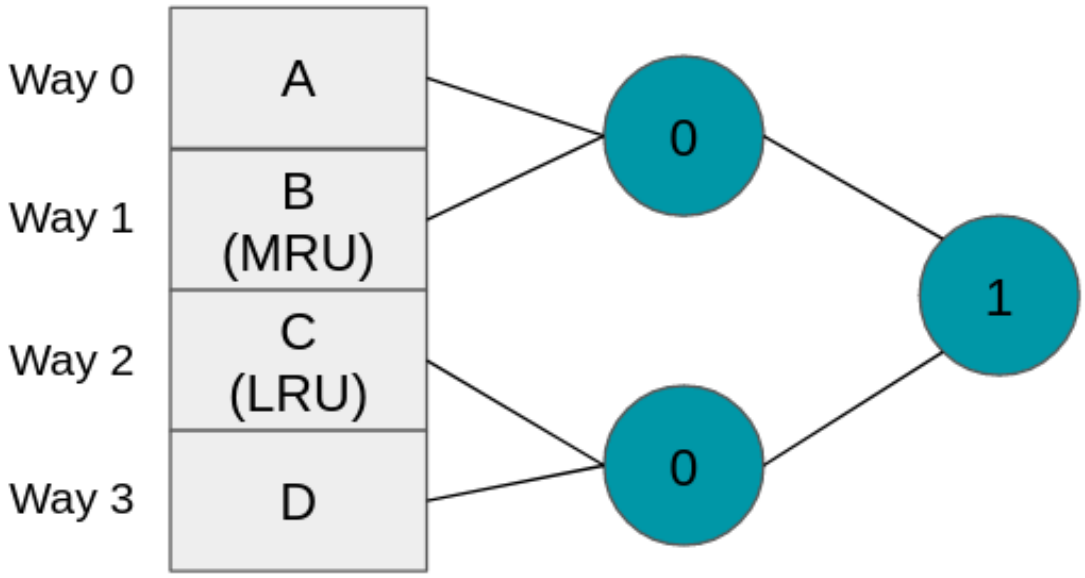

Figure 2.4: *An example cache set of a 4-way set associative cache using binary trees as replacement algorithm.*

most-significant bit of the *PSEL* counter. This is not the only variant of set dueling that exists. There are also thread-aware implementations of set dueling where the policy decision also takes the application into consideration [13], [14].

2.3 Cache Partitioning

Cache partitioning involves the allocation and segregation of the processor's cache memory into distinct logical regions or partitions, each serving a specific process or core. This innovative technique enables processors to improve QoS, fairness and performance due to the fact that interference is reduced between cores or processes [2].

There are numerous ways of implementing cache partitioning with different design decisions to be made. Such decisions should be made depending on requirements on design complexity, performance, scalability, *etc.* To give some examples, partitioning can have coarse-grained or fine-grained granularity, it can be way- or set-based and so on. Furthermore, different optimization objectives exist such as minimizing energy consumption, assuring fairness and QoS among others [2].

A cache partitioning technique typically consists of *(I)* a cache *allocation policy* which is an algorithm, usually implemented in software, that creates the assignments of cores or applications to cache partitions. Furthermore, a cache partitioning technique also needs *(II)* a cache *enforcement scheme* which is a mechanism, typically implemented in hardware, that actually enforces the assignments decided by the allocation policy. This idea is illustrated in Figure 2.5 [15].

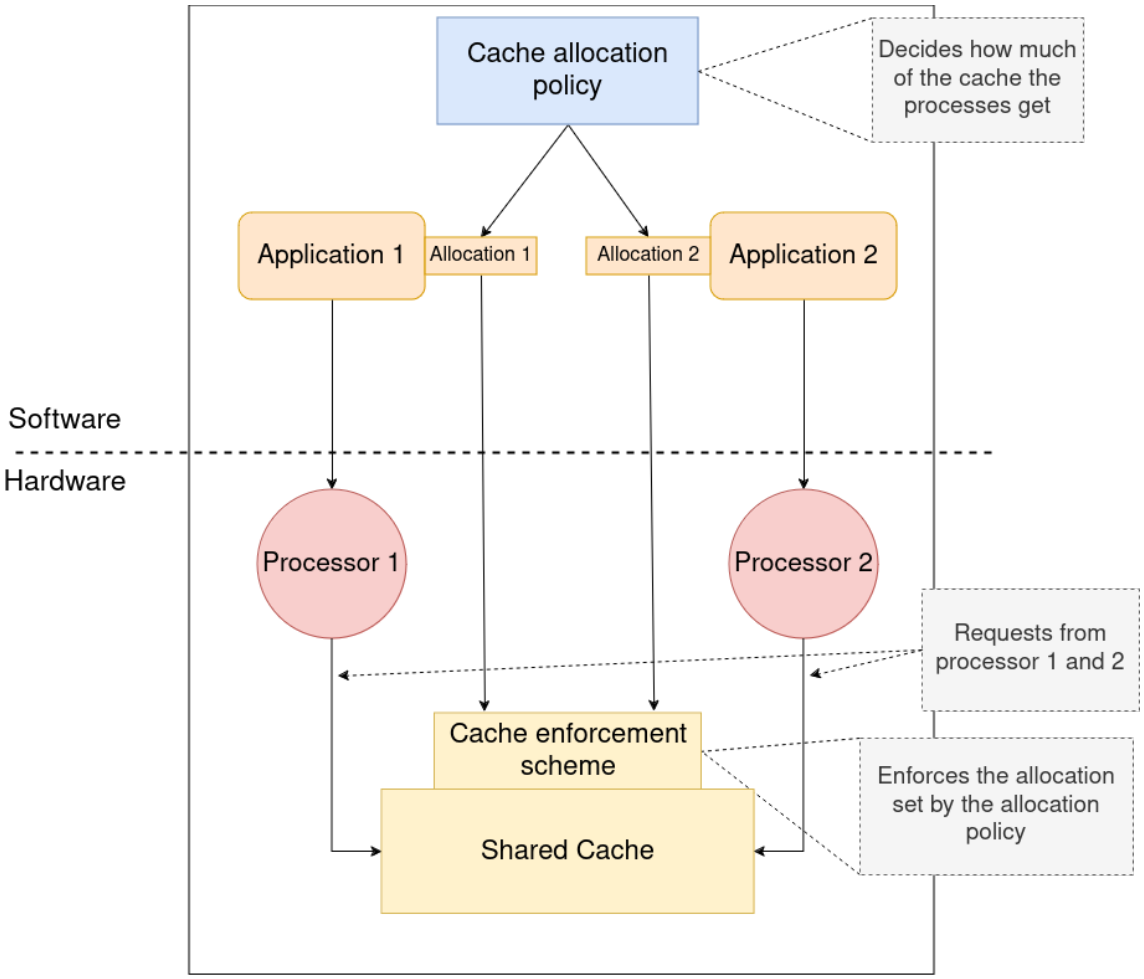

Figure 2.5: *An illustration of how the allocation and enforcement together make up the partitioning system.*

In the works by Wang et al. and Chiou et al. [15], [16], they describe that even if the allocation policy has assigned a definite amount of cache space to a particular application, it falls down to the enforcement scheme to actually guarantee that the application only uses the allocated cache space and not any other cache space, as illustrated in Figure 2.5. An ideal cache enforcement scheme should have the following properties:

1. *smooth resizing.*
2. *precise sizing.*
3. *high associativity.*
4. *no negative effect on clock speed*

Point *(1)* means that adjusting the size of a partition should incur little to no performance penalty, *(2)* allows the allocation algorithm to be able to precisely control the partition sizes that gets allocated, *(3)* means that the associativity of each partition should still be high even with many partitions in order for performance

to be maintained and finally *(4)* means that the enforcement should not decrease the clock frequency since the cache is on the critical path, a slow enforcement scheme would impact system performance.

There are typically two ways to enforce the cache allocation: placement-based and replacement-based partitioning. Placement-based cache partitioning works by enforcing where lines are placed in a cache, such as assigning certain ways to a core or thread. Replacement-based cache partitioning dynamically adjusts the cache line eviction rate of each core or thread to enforce partitioning [6].

Intel's CAT

Intel has a cache partitioning system called *cache allocation technology* (CAT) which is a way for the programmer to provide software guided hints in order to partition the last-level cache (LLC) according to the user's provided specification. The way it works is that you divide threads into *classes of service* (CLOS) which are then assigned to different partitions through a *capacity bit mask* (also called *partition bit mask*) (see Figure 2.6).

Dividing threads into classes of services and their respective bit mask is set by the programmer. There is a many-to-many mapping between threads and CLOSes. The capacity bit mask defines what fraction of the LLC a certain CLOS will get to use. The bit masks consist of either ones or zeros that indicate whether a certain thread, or collections of threads, have access to a specific way (or multiple ways). The bit masks are an abstraction, and are agnostic to the underlying implementation of the enforcement part of the partitioning scheme. The allocation of CLOSes to cache resources can either be disjunct or they can overlap (see Figure 2.7). An example of processors that have support for partitioning is the *Intel Xeon E5 v3* family of processors that use CAT [1].

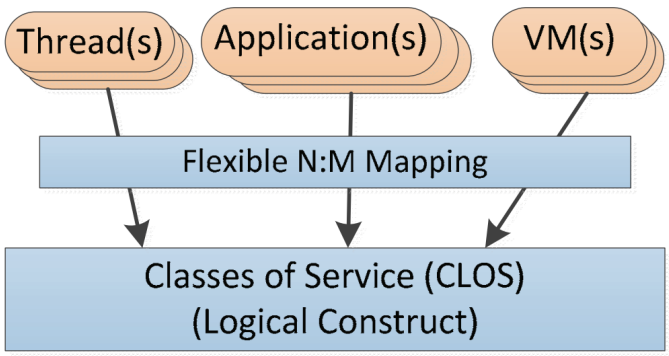

Figure 2.6: *Figure showing the N:M mapping of applications to CLOSes in Intel's CAT* [1]

.

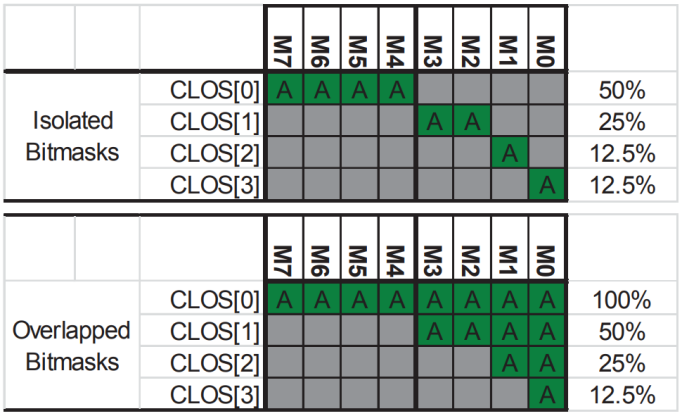

Figure 2.7: *Figure showing partition masks for each CLOS with disjunct partitions (upper image) and overlapping partitions (lower image) in Intel's CAT* [1]

.

3

Design

This chapter will present the designs that have been made through the course of the thesis, both for the top-level cache design and the individual components that are used to build the entire system. This also includes each of the replacement algorithm that were implemented for this thesis.

3.1 Cache Module

A diagram of the top-level cache module can be seen in Figure 3.1. The yellow component *replacement algorithm* is where implementations of the different replacement algorithms will be inserted. It is also here that the partitioning will be enforced. The cache design is intended to simulate the replacement and partitioning logic of a cache and not an entire functioning cache. The supporting components (the components not directly responsible for the replacement functionality) are also simplified to fit the scope and aim of this thesis.

3.1.1 Processor interface

The *processor interface* module serves as the entry point to the cache design. It is here that all read/write requests are made and also where the requested data is returned.

3.1.2 Directory interface

The *directory interface* module is responsible for reading and inserting tags into the cache directories. The point of this module is to be a common entry to the tags- and status bits directories.

3.1.3 Directory top

The *directory top* module is the module that contains the tag- and status bits directories. Its only purpose is to instantiate the directory modules and provide an interface for other modules to access the directories.

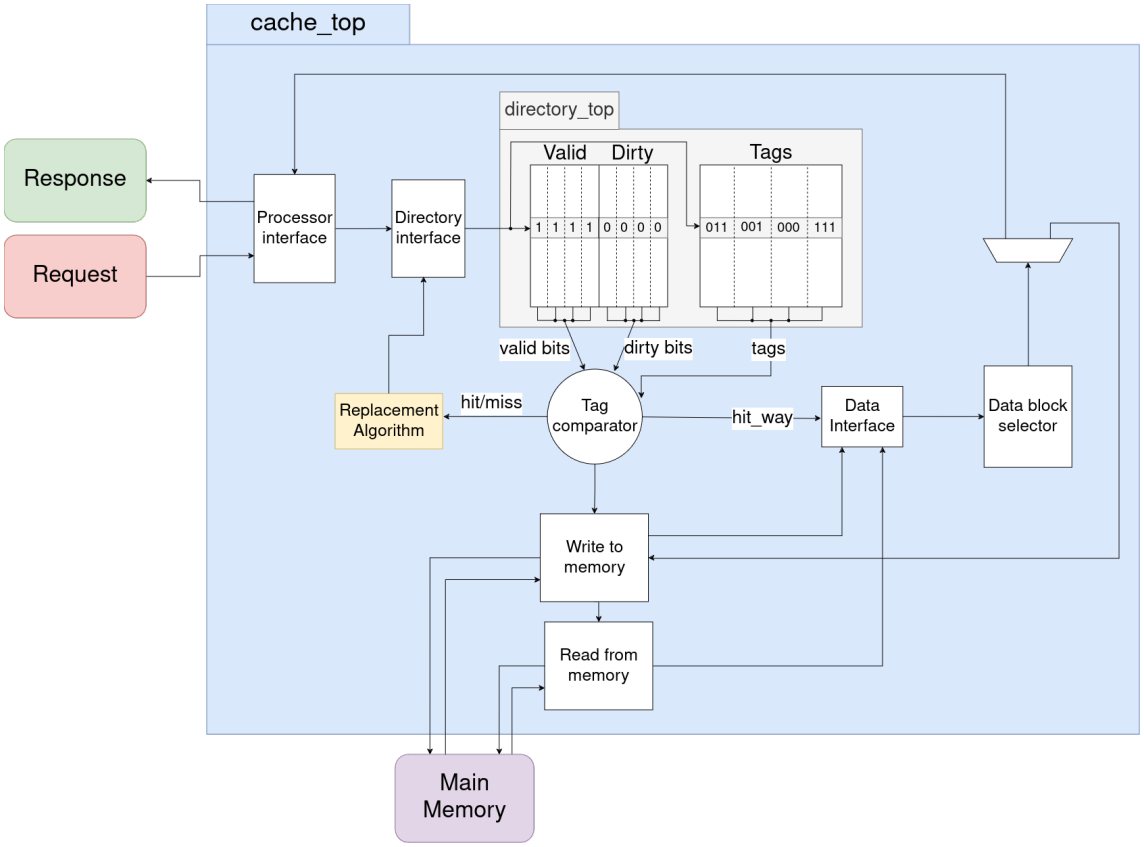

Figure 3.1: *The top level cache module.*

Tag Directory

The *tag directory* is implemented as a two-dimensional array of vectors where the number of rows are equal to the amount of sets in the cache and the amount of columns equal the associativity of the cache. Each vector has a bit-width equal to the decided tag bit-width of the architecture.

When a request is made to the cache, the tag directory will receive the index from the request address. The index will then be used to index the row of the cache. An output signal `tags_o` will be generated that will contain each tag stored at that index. The bit-width of `tags_o` will therefore be $associativity \times bit\_width_{tag}$.

Status Bits Directory

The *status bits directory* is implemented similarly as the tag directory. It uses two separate RAM banks, one for the `valid` bits and one for `dirty` bits. The bit-width for each of the RAM banks is the associativity since each bit in a register corresponds to a cache way. The number of rows in each bank is equal to the number of sets in the cache. The `valid` bits indicate whether or not a cache line contains valid data and the `dirty` bits indicate whether or not the data in the cache line has been modified and therefore needs to be written back to main memory upon an eviction. When resetting the cache, the dirty bits and valid bits are set to zero whilst the tags are not modified. This is because the valid bits will indicate that the tags are invalid so there is no need to change the contents of the tag directory.

3.1.4 Tag and valid comparator

The *tag and valid comparator* module is responsible for comparing a given tag with all the tags of a set as well as checking that the valid bit is set for the sought tag. The module is a combinatorial component. When you want to see if a tag is currently present in a set you feed the tag into the tag comparator as well as all the tags in a set. If the tag exists in the set, the miss output signal will be low, the hit output will be high and the hit way signal will indicate in what way the tag resides in the set. However, if the tag does not exist in the set the miss signal will be high and the hit signal will be low. The tag comparator also takes the partitioning mask into consideration so we only compare the desired tag to the tags that reside in the partition.

3.1.5 Data interface

The *data interface* module works much in the same way as the *directory interface* module in that it serves as an entry point to the *data block selector* module. It also is responsible for setting the dirty bit for the corresponding data in the correct set of the cache directory upon a write request.

3.1.6 Data block selector

Data block selector is the module which contains all the actual data of the cache implemented as a RAM module.

3.1.7 Replacement algorithm

The *replacement algorithm* module contains the desired replacement algorithm and connects it to the rest of the cache design. The replacement algorithms are chosen at design time and will be explained further in section 3.4 below.

3.1.8 Memory Interface

These modules are responsible for writing back dirty data and reading requested data from the main memory.

Write memory interface

When there is an eviction in the cache some data will have to be thrown out. The dirty bit of said data needs to be checked and if the bit is set the *write memory interface* module will make sure that the data that is dirty will be written back to main memory.

Read memory interface

The new desired data is read using the *read memory interface* module. This module together with the *write memory interface* module constitutes the interface from the cache design to the main memory.

3.2 Cache Design Operation

Now we explain how the cache operation works upon a request being made. A request to the design will have one of two types: either a read or a write request for a specific address. In the case of a write request there will also be some data that is meant to be written to the specified address. With this in mind, there are four different scenarios that can happen in the design namely: read hit, read miss, write hit and write miss.

On a request the address will go through the processor interface and the directory interface and eventually end up at the directory top module. Here it will be used to fetch the tags- and status bits. The tags will be compared to the desired tag in the tag comparator module which will either generate a hit or a miss. If it is a hit, the address goes to the data interface and data block selector. The desired data will be fetched and sent back to the processor interface in the case of a read request. If it is a write request, the specified data will be written to the data block selector as well as returned to the requester.

In the case of a miss, the replacement algorithm module will generate a victim way which will be sent to the directory interface. The directory interface will then read the old tags- and status bits and write the new tag on the victim way. The status bits are then sent to the write- and read memory interfaces and if the data that is being evicted is dirty it will be written back to main memory and the desired data will be fetched. Otherwise the desired data will only be fetched without writing back the old data. The desired data is then sent to the data interface and data block selector to be inserted. The data will then be sent to the processor interface where it will be returned to the processor interface and eventually returned to the requester.

3.3 Adding Support for Cache Partitioning

Adding support for partitioning in the cache design does not require any significant changes. The first difference is that the partitioning mask has to be passed with the request as well as the CLOS ID of the requester (see section 2.3 for details about CLOS ID). The partitioning mask will be used in the tag comparator so that it generates hits and misses according to the ways inside the CLOS's partition. Lastly, the partition mask and CLOS ID is used by the replacement module to only generate victim ways inside the assigned partition. See Figure 3.2 for an example of a bit mask.

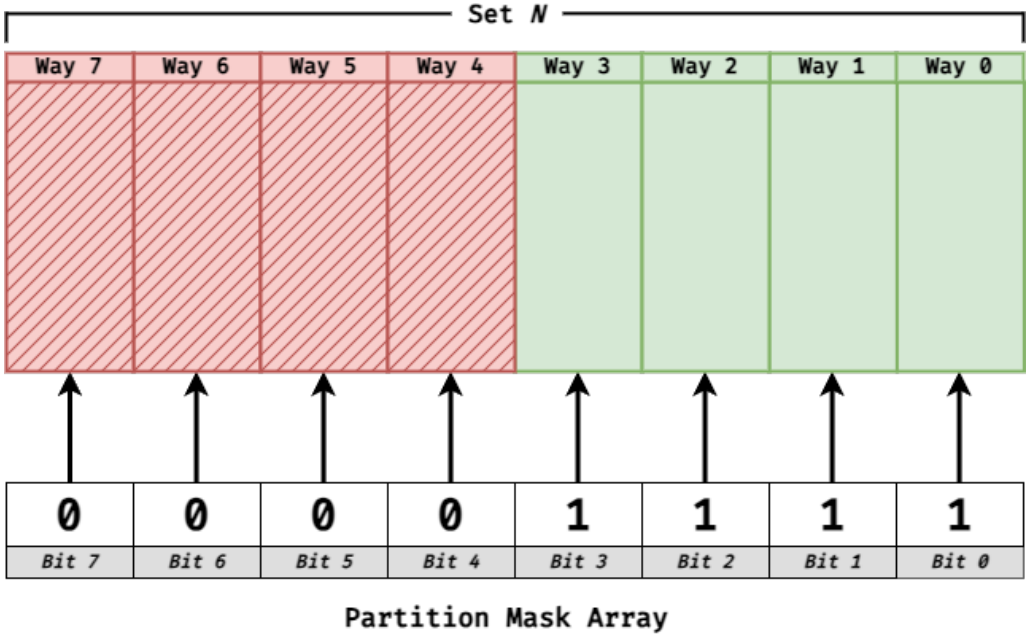

Figure 3.2: *An example of how the partitioning mask is used to mask off unavailable ways in a set N. If the bit* **b** *in the partitioning mask is set to zero, the respective way is masked off as unavailable.*

3.4 Replacement Algorithm Designs

This section will cover different designs of the replacement algorithms that will be the *replacement algorithm* component in Figure 3.1. Each algorithm has two main functions that make up the behavior of the algorithms. A *generate victim* function and an *update* function. The former specifies how a victim is generated and the latter describes how the state of the algorithm is updated.

3.4.1 Random

As has been said in section 2.2.1, this replacement algorithm picks a way in a set at random. This means that there is no need to keep track of any access history. The pseudo-code for the random replacement algorithm is implemented with a counter that is incremented every clock cycle. When a cache miss occurs, the counter will be sampled and the pseudo-random value will be converted into a valid way number based on the associativity as well as the partitioning bit mask. To implement partitioning with Random, only the ways within the current partition can be marked as a victim by the randomization engine.

It should be noted that this implementation has its flaws. If we assume that the interval between misses is constant and deterministic, it means that the sampled value which will ultimately become the victim way will always be the same as the counter will always be incremented with the same value. Following this logic, the randomness of the victim way generation will only be as random as the miss rate of the application. This is why the replacement algorithm is referred to as *pseudo-random* as opposed to *truly-random*. For the scope of this thesis, we consider this pseudo-randomness to be sufficient.

3.4.2 TrueLRU

The TrueLRU algorithm needs to keep track of the entire recency history of the cache lines in a set. To do this, each line needs to have a priority associated with it. As has been said in 2.2.2 each line in each set will have an additional $A(log_2 A)$ bits to keep complete recency information. Upon a cache line hit, all counters for the lines that have a higher priority than the line (and therefore a lower number in the counter) will be incremented by one. All the while the line that hit will have its number set to zero. This is illustrated in Table 3.1. In the case of a miss, the way with the highest counter will be selected as the victim for replacement. To implement partitioning with LRU, only the ways within the current partition will have their counters affected by a hit or a miss. When a miss occurs and a victim has to be found, only the ways in the current partition will be checked as candidates for eviction.

	Priority	
Lines	Before hit	After hit
A	1	2
B	4	4
C	0 (MRU)	1
D	7 (LRU)	7 (LRU)
E	6	6
F	2	3
G	3	0 (MRU)
H	5	5

Table 3.1: *A demonstration of how the recency stack gets updated on a cache hit using TrueLRU. Here, line* G *sees a cache hit and will therefore get the MRU position and update the cache lines that had a higher priority than itself.*

3.4.3 NRU

Since NRU is a psuedo-LRU algorithm, the design of NRU looks quite similar to LRU. Instead of associating $log_2(associativity)$ number of bits with each cache line, we only associate one bit per cache line, the idea being to indicate if the cache line is recently used or not. When a cache line gets a hit the used bit for that line is set to one. This continues until all bits in a set is set to one. In this case when we get a miss, all bits are reset to zero and we pick the first way in our partition as the victim. First here means the way with lowest index in our partition. This is a cheap way to implement pseudo-LRU behaviour since we keep track of what ways were recently accessed and therefore occupied at the cost of losing information about the recency of each way. To implement partitioning with NRU, only the ways within the current partition will have their bit affected by a hit. When a miss occurs and a victim has to be found, only the ways within the current partition will be affected by the reset functionality.

3.4.4 Binary Tree

To implement the data structure of a binary tree in hardware is not feasible, and so it was implemented as an array of registers, where each register contains the value of a node in the tree. The first index of the array corresponds to the root node of the tree. After this, each node starting from the top-down of a level (seen from a top view of the tree) corresponds to the next index in the array. By doing this mapping, a traversal from one node to one of its children can be calculated with:

$$\text{Node to upper child: } (2 * index) + 1$$
$$\text{Node to lower child: } (2 * index) + 2$$

This traversal will be done and calculated $log_2(associativity) - 1$ times since we always start the search process from the root-node with a known index. The tree is also updated when a traversal is done by inverting the bit in the node we are "leaving" to indicate a new path was taken to find the least recently used way. By

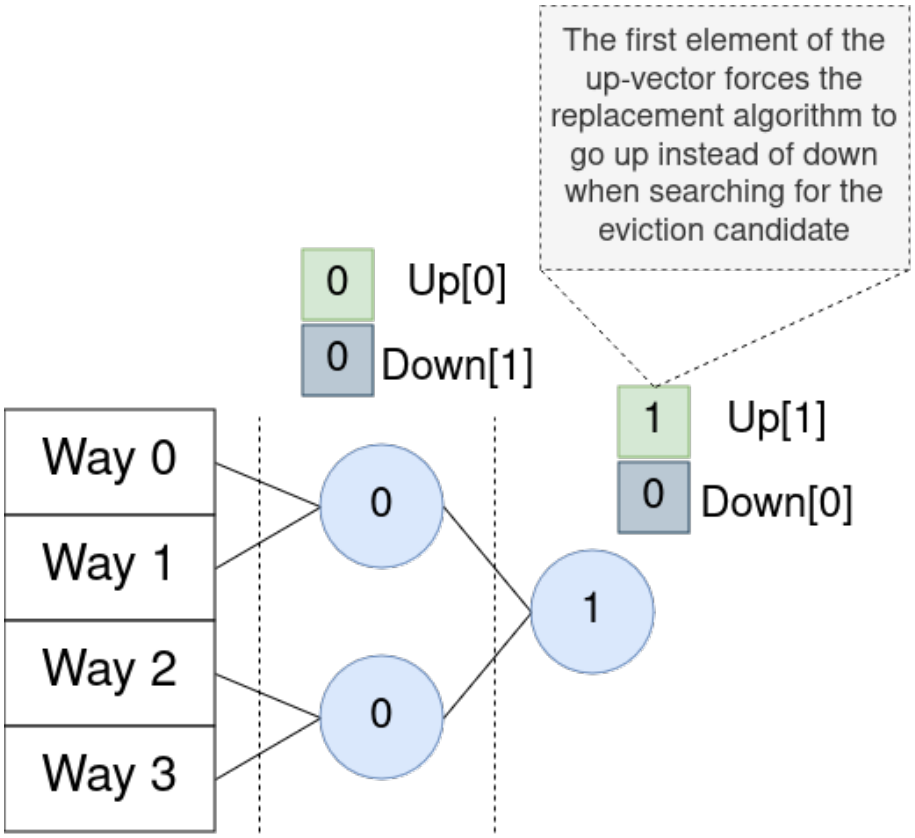

Figure 3.3: An illustration showing how the up-and down-vectors work. Here the binary tree replacement algorithm would traverse the lower half of the tree in the normal case. But since the first element of the up-vector is set to `one` the algorithm will go up instead of down to find he eviction candidate.

doing this we ensure that the tree is always updated with what way is the most recently used.

To incorporate partitioning with this algorithm we also implement two vectors called *up and down-vectors* respectively [12]. These vectors are used to enforce the traversals of the tree depending on the partitioning done. Therefore these vectors take precedence over the value in the nodes in the tree when we traverse it. If a traversal normally would go down from a specific node but the up-vector has its corresponding node-index set to one, the traversal would be forced to go up. The same principal applies for the down-vector. This also implies that for a specific index in the tree, both the up and down-vectors can not have their values set to one, which is how they are implemented in our solution. If both the up and down-vector for a specific node index is set to zero, we traverse the tree according to the tree node value. This is also the only case where we update the node values in the tree. See Figure 3.3.

3.4.5 Binary Tree Private

The *Binary Tree Private* replacement algorithm has the goal of removing the recency information interference that the one-bit binary tree solution has. In the binary tree solution where CLOSes share the recency information between them, the state of one partition can be affected by the traversal and altering of the binary tree state, depending on if the CLOSes has overlapping cache ways in their partitions. This can lead to a side channel between partitions and disruption of recency information.

To solve this issue, a modified version of the original binary tree algorithm is created. In this version, instead of having a single binary tree assigned to a set, there is a single binary tree assigned to each CLOS per set. For example, in a cache with 255 sets and two CLOSes, there would be 255×2 binary trees. In contrast to the original binary tree version where there would only be 255 binary tress (i.e. one tree for each set). Worth noting is that in the case of no overlapping partitions between CLOSes, binary tree private will work exactly the same as binary tree.

The theory behind this solution is that since each CLOS in the system will work with separate data sets, maintaining the recency information for each CLOS and its separate data set is key to assure the functionality of binary tree. If all CLOSes alters the same binary tree, the tree can only keep the recency information for the last running CLOS. Therefore, associating one tree for each CLOS, we can maintain coherency of the recency information in the system.

3.4.6 DRRIP

DRRIP[1] uses one RAM module where each entry contains all RRPV[2] values for a set of the cache. Therefore the bit width of each entry is $M * Associativity$. When a miss occurs, the entry for the specific set is fetched and searched for an appropriate victim. If no victim can be found, each way will have its RRPV value increased and the search is done again. Therefore the victim generation is implemented in a circular fashion, which in turn means that the time it takes to generate a victim is not constant. The worst case scenario is that all ways in our partition in the specified set has its RRPV value set to zero. Then the victim generation would take nine cycles versus where one or more of the ways in our partition has its RRPV value set to three, then it would take two cycles to generate a victim. When a hit occurs the entry for the specific set is fetched and the way that generated a hit will have its RRPV value set to zero. This operation is time constant and takes two cycles to perform.

Since DRRIP is bimodal with each policy having dedicated sets for each of them, there is some logic overhead for implementing the SDM[3]used for deciding during runtime what policy the follower sets should use. The SDM is implemented as a

[1]DRRIP = Dynamic Re-reference Interval Prediction, see 2.2.6
[2]RRPV = Re-Reference Prediction Value, see 2.2.5
[3]SDM = Set Dueling Monitor, see 2.2.6

2-bit saturating counter where a hit from the dedicated sets of SRRIP[4] and BRRIP[5] will increase and respectively decrease the counter. The uppermost bit will then be checked when deciding upon a policy to be used for the follower set. If the bit is one, SRRIP will be used and if the bit is zero BRRIP will be used.

Since BRRIP inserts new data with a chance of either *distant re-reference* or *future re-reference* as the initial RRPV value, there has to be some hardware that can create pseudo-randomness to make this decision. To achieve this a *linear feedback shift register* (LFSR) was implemented to generate a zero or a one each clock cycle with approximately 93% and 6% chance respectively. These values are then used to determine the RRPV value used when inserting the new data in the cache.

To implement partitioning with DRRIP, the functionality of the algorithm stays the same, but we use the mask to only alter and check the RRPV values from the ways in the current partition.

3.5 Area Overhead Analysis

This section will cover the area overhead analysis that was done for the various replacement algorithms. This analysis are the theoretical calculations that were done before results are generated from the actual design. We define *Area overhead* as the required storage overhead to implement the algorithms. In the following equations S is the number of sets in the cache, A is the associativity and K is the number of CLOSes.

3.5.1 Random

The storage overhead for the random replacement algorithm consists only of a single counter that is constantly incremented and occasionally sampled to generate a victim way. The size of the counter is $log_2(A)$ bits to be able to index every way in a set. A single counter is shared for all sets in the cache. This makes the random replacement algorithm the cheapest of all the replacement algorithms in terms of area overhead.

To add support for partitioning there is no need for additional storage overhead so it should be the same for both the partitioned and non-partitioned versions. It does require some logic for making sure the generated victim way falls within the allowed partition of the current CLOS.

$$\begin{aligned} Area_{random} &= log_2(A) \\ Area_{random\_partitioning} &= log_2(A) \end{aligned} \tag{3.1}$$

[4]SRRIP = Static Re-reference Interval Prediction, see 2.2.5
[5]BRRIP = Bynamic Re-reference Interval Prediction, see 2.2.6

3.5.2 TrueLRU

The TrueLRU replacement algorithm is typically the most expensive replacement algorithm to use in terms of area overhead. This is because it keeps track of the entire recency stack, in contrast to pseudo-LRU algorithms. The storage overhead for a typical TrueLRU can be seen in equation 3.2 below.

$$\begin{aligned} Area_{trueLRU} &= S \times A \times log_2 A \\ Area_{trueLRU\_partitioning} &= ...S \times A \times log_2 A \end{aligned} \tag{3.2}$$

Here there will, for each set in the cache, be a counter of size $log_2 A$ associated with each line. Therefore, each set will contain $A\, log_2 A$ bits of storage overhead and there are S sets in the cache.

Adding support for partitioning will not require any extra overhead since the same counters can be shared among different partitions (also not counting the partitioning bit mask that will be present in all replacement algorithms). However, if the assumption can be made that the partitions are non-overlapping there are ways to reduce the overhead, even compared to the non-partitioned TrueLRU algorithm. Starting from an 8-way set associative cache with two partitions there are in total seven ways to partition the cache. Namely: $\{\,(1,7),(2,6),(3,5),(4,4),(5,3),(6,2),(7,1)\,\}$ where a number in a tuple represents how many ways a partition will get out of the eight available ones. By doing area overhead calculations on all these possible partitions it is possible to derive the total area overhead which can be seen in Table 3.2. Here it is possible to see that $(4,4)$ is the best partitioning and it appears that just having any partitioning decreases the overhead compared to the unpartitioned version. It should be noted however that the partitioning needs to be dynamic and able to change. This makes it so that it is impossible to only use 16 bits of overhead even if we assume a $(4,4)$ partitioning since the partitioning might change to $(1,7)$ or even be turned off at some point in the future.

Partitioning	(1,7)	(2,6)	(3,5)	(4,4)	No partitioning
Area	21	20	21	16	24

Table 3.2: *The required area overhead (bits) for different partitions using TrueLRU.*

3.5.3 NRU

The NRU replacement algorithm is among the cheapest replacement algorithms to implement in terms of storage overhead. It only uses a single bit per way and will therefore have a total of A bits of storage overhead per set. As for the random algorithm, there is no need for additional storage overhead to add support for partitioning with NRU.

$$\begin{aligned} Area_{nru} &= S \times A \\ Area_{nru\_partitioning} &= S \times A \end{aligned} \tag{3.3}$$

3.5.4 Binary tree

The binary tree algorithm will associate a binary tree with each set of the cache. The nodes consist of one bit information that determines the path traversed to reach the LRU way or MRU, depending on how the one bit value is used to traverse the tree. Binary tree is area efficient since the amount of bits used to store the tree will relate to the associativity of the cache. When implementing the binary tree algorithm with partitioning, two additional vectors are needed that determine what path a certain CLOS has to take at each level. Two of these vectors will be used for each CLOS and the size of these two vectors will be the height of the tree. The area overhead for the two variants can be seen in equation 3.4.

$$\begin{aligned} Area_{bt} &= S \times (A-1) \\ Area_{bt\_part} &= S \times (A-1) + K \times 2 \times (A-1) \end{aligned} \tag{3.4}$$

3.5.5 Binary tree private

The binary tree private algorithm will require an entire binary tree for each CLOS in each set. The storage overhead will be as shown in equation 3.5. There is no non-partitioned version of this algorithm since without disjoint partitioning this algorithm would serve no purpose.

$$Area_{binary\_tree\_private} = K(S(A-1) + 2log_2 A) \tag{3.5}$$

3.5.6 DRRIP

The DRRIP algorithm effectively uses two algorithms in its implementation, the SRRIP and BRRIP algorithms. Because of the similar nature of the two algorithms, very little extra overhead is needed to implement both algorithms efficiently in hardware.

Both SRRIP and BRRIP can share the bits used for the RRPV values, so only one instance of an RRPV array has to be implemented. The size of each entry will be M, and each way need one entry. Since the success of both BRRIP and SRRIP has to be recorded in order to make a decision on which algorithm should be used by the follower sets, a mechanism is needed to perform these tasks. In our implementation, a 2-bit saturating counter was used. To implement BRRIP, some logic has to be implemented to determine what RRPV value should be used on a miss. In our implementation, this is done by a 4-bit logical feedback shift register (LFSR) to pseudo-randomly make this decision. No other logic overhead is needed to implement DRRIP with partitioning. The area overhead for DRRIP can be seen in equation 3.6.

$$Area_{drrip} = S \times M \times A + 6 \tag{3.6}$$

4

Experimental Methodology

This chapter will cover the experimental methodology verifying and evaluating the performance of the cache design.

4.1 The Cache Configuration

The cache design that has been created can be set up with different configurations (e.g. cache size and replacement policy). Different sizes of cache will be tested to see how it affects the miss rates of the replacement algorithms. The setup of the cache that we want to use can be seen in Table 4.1 below. The way that we vary the total cache size is by changing the index portion of the cache address to change the number of sets that are in the cache, all while keeping the offset constant and varying the tag portion as a consequence of varying the index.

Address width	48 bits
Cache line size	32 bytes
Associativity	8 ways
Total cache size	8 KiB up to 8192 KiB

Table 4.1: *Cache configuration: general configurations*

Another aspect of the cache configuration that will vary is the partitioning between different CLOSes. The replacement algorithms need to be tested using partitioning and so a set of allocations for different CLOSes will be used, see Table 4.2 below for some examples of partitioning masks. Also note that some of the example masks has overlapping ways between partitions.

Single CLOS	
11111111	
11110000	
00000011	
Multiple CLOSes	
11110000	00001111
11111000	00011111
11111100	00111111
11111110	01111111
11111111	11111111

Table 4.2: *Example bit masks for allocation configurations.* `1` *means the way is inside your partition and* `0` *means it is outside. E.g. the allocation* `11110000` *means that the CLOS in question gets assigned ways 7-4 in all sets.*

4.2 Verifying the Designs

The designs from the design- and preparation phase was implemented using RTL[1]. The designs were then implemented in SystemVerilog, which is a hardware description language. It was chosen because it offers high abstraction and flexibility. For synthesis and testing, Vivado was used.

In order to verify that our designs worked, a test bench was created. This test bench performed tests of five types of requests, namely: write hits, read hits, read misses with dirty data, read misses with clean data and write hits with dirty data. For each of these types, one hundred requests were performed with randomized data and addresses which would be five hundred randomized test in total. After each request, the internal state of the cache design was checked (status bits, data memory, etc.) to see if it matched the expectations.

There were different forms of test frameworks we could use to verify and evaluate our design. For example, a simple system that generates and emulates expected inputs and outputs for our implementation could be used to test the correctness and functionality of our system. Another method was to integrate our design into an open-source cache RTL-design in order to simulate how it interacts with an actual cache. We chose to implement a simple cache design ourselves since this gave us full control of the full system and therefore would be easier to use in our test bench.

[1]RTL = Register Transfer Level, an abstraction to describe behaviour and functionality of a circuit using high-level code

4.3 Evaluating the Designs

This section will cover how the evaluation process was done for the different replacement algorithms implemented for this thesis.

4.3.1 Benchmarks

We used the SPEC-CPU-2006 (Standard Performance Evaluation Corporation) floating-point benchmarks when simulating, which is a well known benchmark suite for single-threaded applications. The benchmark applications are easily available for the simulator we are using, which is described in section 4.3.3. Since this benchmark suite has a wide array of benchmark programs, 11 were used in our simulations, which were chosen at random, and is a satisfactory amount for this thesis. The applications used are listed in Table 4.3. Worth noting is that *GAMESS_1* and *GAMESS_2* are the same applications as *GAMESS*, but they use different input parameters. When running the benchmark applications, *PinPoints* of the specified applications were used. For more information, see 4.3.3.

Application	Description
Bwaves	Fluid dynamics
CactusADM	General relativity
Calculix	Structural mechanics
GAMESS	Quantum chemistry
GAMESS_1	Quantum chemistry
GAMESS_2	Quantum chemistry
GROMACS	Molecular dynamics
LBM	Fluid dynamics
Leslie3D	Fluid dynamics
NAMD	Molecular dynamics
POV-Ray	Image ray-tracing

Table 4.3: *The applications that were used in testing the replacement algorithms. These applications are from the SPEC-CPU-2006 floating-point application benchmark.*

4.3.2 Evaluation Metrics

The main metrics used for evaluating the performance of our designs are the *total number of misses* when running multi-application simulations and the *miss-rate* for single-application simulation. The reason for this separation is that for multi-application we can accurately compare quantitatively how the misses change, both separated and combined, for different partitions better when using the total number of misses. Since this separation is not needed when running single-application, the miss-rate metric is used instead.

Another metric for the performance of the replacement algorithms is the *latency* of the algorithms. In other words, how many cycles does it take to generate a victim

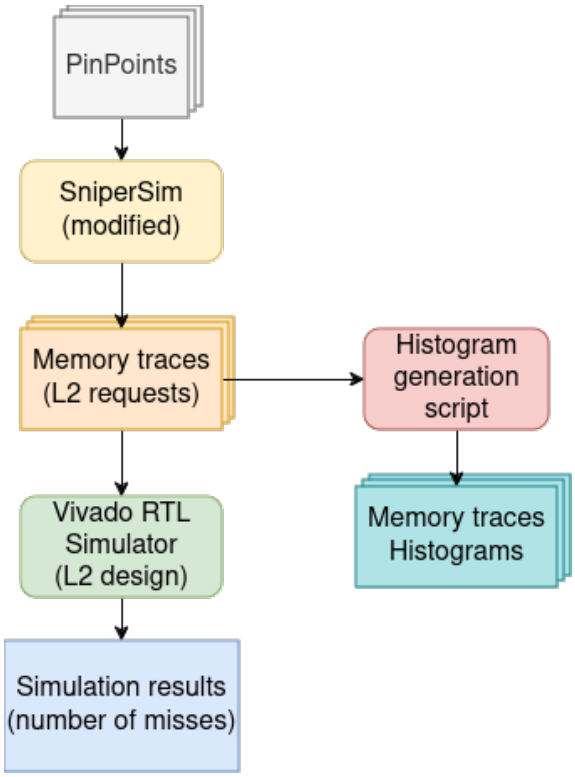

Figure 4.1: *An illustration of the flow from SimPoints to the evaluation metrics and histograms.*

way. In order to measure the overhead of our designs we used Vivado to see how many resources our designs used post-synthesis. The metrics we were interested in were mostly the number of lookup tables (LUTs) and the number of registers used.

4.3.3 Experimental Framework

An overview of the experimental framework can be seen in Figure 4.1 where we go from benchmark application *PinPoints* to histograms and simulation results.

PinPoints

Since it is not feasible to simulate the entire SPEC-CPU-2006 benchmarks in reasonable time, a technique called SimPoint was invented to select a small representative region of the application [17].

These smaller representative regions of applications were in the form of *PinPoints* [18]. The fraction of a benchmark program that a PinPoint represents differ between PinPoints. All the PinPoints have a *warm up* region followed by the *region of interest*, where each region consists of hundred-million and thirty-million instructions respectively. The warm up is supposed to fill the cache with data so that when execution reaches the region of interest, most of the cold misses are out of the way. When deciding which PinPoints to use for our thesis, we looked at all PinPoints for a specific benchmark program and chose the one that represented the largest fraction

of the whole program. All PinPoints used in this thesis were downloaded from the SniperSim website [8].

Sniper Sim

SniperSim is an open-source x86 simulator that is based on the *interval core model* and the *Graphite* simulation infrastructure [19]. It allows the user to perform timing simulations of workloads with over 100 cores at high speeds [8]. We used SniperSim to execute the PinPoints and extract the L2-accesses made in the PinPoints to generate the memory traces that were to be used in our simulation test bench. The memory trace will therefore be a list of L2-accesses that the application makes through its runtime.

In order to get the correct memory traces to use in RTL-simulation, we needed to filter out the L1-cache memory accesses made in the PinPoints since we want to simulate the L2-accesses made from the benchmark applications. This meant we needed to modify the source code of SniperSim. We started by changing the configuration for SniperSim to have a 32KB L1-cache. Furthermore, we added print-outs in the code to display the memory requests that go only to the L2-cache. This will ensure a more realistic memory trace since we filter out the requests that hit in the L1-cache. Lastly, we add a single print-out in SniperSim that indicate when a hundred million instructions have been executed. As mentioned in 4.3.3, this is due to the fact that the PinPoints we are using as benchmarks have a hundred-million-instruction long warm up-phase that will fill the cache hierarchy with data, followed by a thirty-million-instruction long region-of-interest where the actual benchmarking of the cache takes place. For this reason, we only want to count the misses that occur in this region-of-interest.

Memory Traces

To test the designs we've created a set of memory traces which were derived from actual benchmarks to show the differences between replacement algorithms in terms of miss rate. See section 4.3.3 under *Sniper Sim* for how the memory traces work.

Since the memory traces might contain overlapping addresses, they can not be executed in a multi-application context without making sure there is no address overlap. Therefore, one trace will have its address space shifted. The shift was done by calculating $max(\{\text{unique addresses in trace}\}) - min(\{\text{unique addresses in trace}\}) + C$, where C is some constant, in this case it is equal to 2048.

Each memory access is either a read- or a write request on a specified address. The memory traces were also used to generate histograms of the reuse distances of a trace, which will be discussed more in detail in the section below.

Histogram Generation

To better understand how the benchmark applications will interact with our cache design, we generated histograms plotting the reuse distances of all memory accesses in the memory trace. This will give us insight of how effective the programs might

utilize the cache. The reuse distance of an address is defined as the number of distinct accesses between two accesses to that same address [2]. As an example, the reuse distance of address **A** in the access stream {**A, B, C, D, D, C, A**} is **three**. Since only the tag and index part of an address is used to check for a hit or miss in the cache, the traces used for the histogram generation had to be modified so all addresses in the trace has the block-offset removed. This will give a more accurate reuse histogram of how the application will utilize the cache. This modification to the trace is handled in the script, and does not alter the original memory trace.

The generation of the histograms was accomplished by using a python script that analyzed a memory trace and yielded the reuse distances of all addresses in trace according to the definition. When it has calculated the reuse distance for an address, that distance is added to a list. When the reuse distance has been calculated for all addresses in the trace, that list is used by the script to plot the histogram where the x-axis is the span of reuse distances and the y-axis is the number of times a particular reuse distance occurs in the memory trace.

RTL Simulator

To simulate how the cache design works with the generated memory traces, Vivado's simulation tool was used [20]. To do this, an RTL simulation test bench was implemented that simulates memory requests done to the cache design from a text file containing memory traces. Since the traces used are generated from the L2 accesses (L1 misses) of the benchmarks, this effectively simulates the L2 accesses that are made from the benchmark applications used in this thesis. This is done by reading in the text file containing the addresses of the requests done.

Each address is then read line-by-line, parsed, and sent to the cache design. Each memory trace contains a warm up period where the hit or miss signal is not logged by the test bench. This is simply done by inserting a flag into the trace to mark that the warm up period has completed. When this flag has been reached in the trace by the test bench, the test bench starts to log the hit or miss signal of each following memory address. The test bench also supports multi-application execution by reading in two different traces and concurrently executing the traces and separating the results of each trace. Each application will execute 50 memory requests before switching to the other application, and each application will execute their own warm up period before the results from that application is logged. This gives us the potential to effectively analyze how the effects of multi-application execution affects the hit-rate of each application.

5

Results

This chapter covers and compiles the results generated throughout this thesis. It will cover both the RTL simulation performance results and the analysis of the latency.

5.1 Performance Evaluation

This section covers the results obtained from running tests using the specified SPEC benchmarks (see section 4.3.1). First we cover the histograms that have been generated. The histograms are mainly used for analyzing the performance results in the subsequent subsections. Then, we go through performances of the replacement algorithms without partitioning running a single application. After that, we present what happens if we then add partitioning running a single application. Then, we look at what happens if we have two applications that run in separate threads and partition them with no overlap. Lastly, we cover what happens if the partitions overlap.

5.1.1 Reuse Distance Histograms

Here we present the various reuse distance histograms for the benchmark applications that were tested. They were generated as explained in subsection 4.3.3. The histograms give a good overview of the reuse patterns of each benchmark program. The histogram figures can be seen in Figures 5.1 to 5.11. The x-axis represents the different reuse distances and the y-axis is the number of times that particular reuse distance exists in a trace.

So for example, in Figure 5.2 we see that there are more occurrences of small reuse distances (for example up to 10) than we see compared to Figure 5.1 where there are not as many bars in the same range. What is to be expected from a trace such as *CactusADM* is that it benefits more from using TrueLRU, since it keeps more detailed recency information about the application. Since *Bwaves* has comparatively longer reuse distances, it would not benefit as much from running TrueLRU.

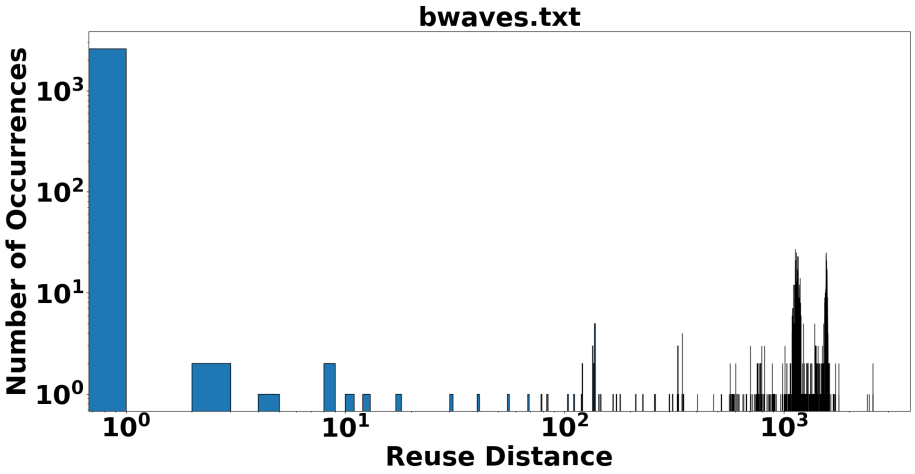

Figure 5.1: *Reuse distance histogram for the Bwaves benchmark.*

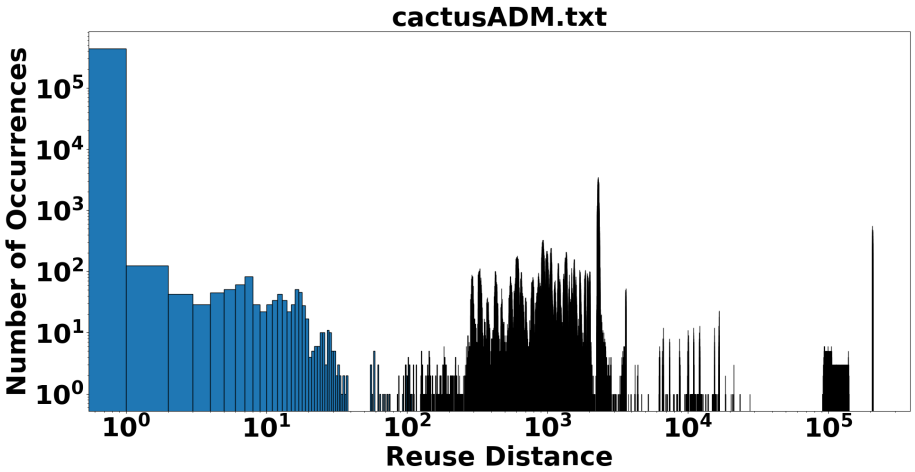

Figure 5.2: *Reuse distance histogram for the CactusADM benchmark.*

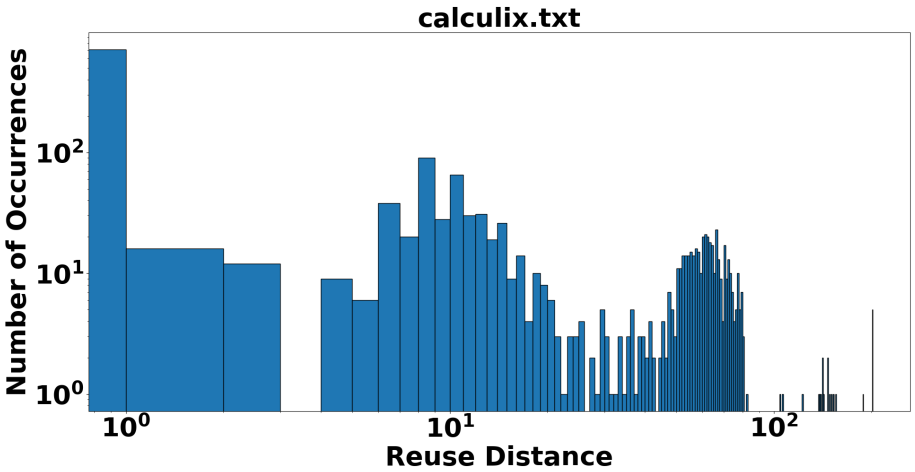

Figure 5.3: *Reuse distance histogram for the Calculix benchmark.*

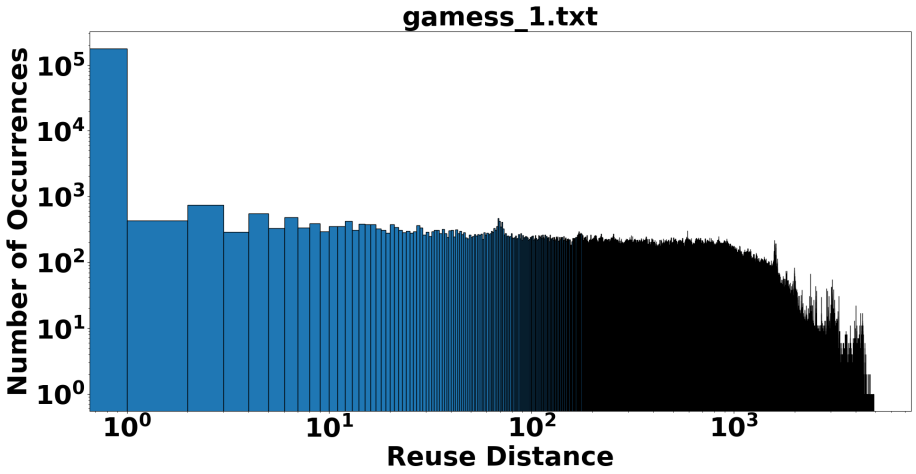

Figure 5.4: *Reuse distance histogram for the GAMESS_1 benchmark.*

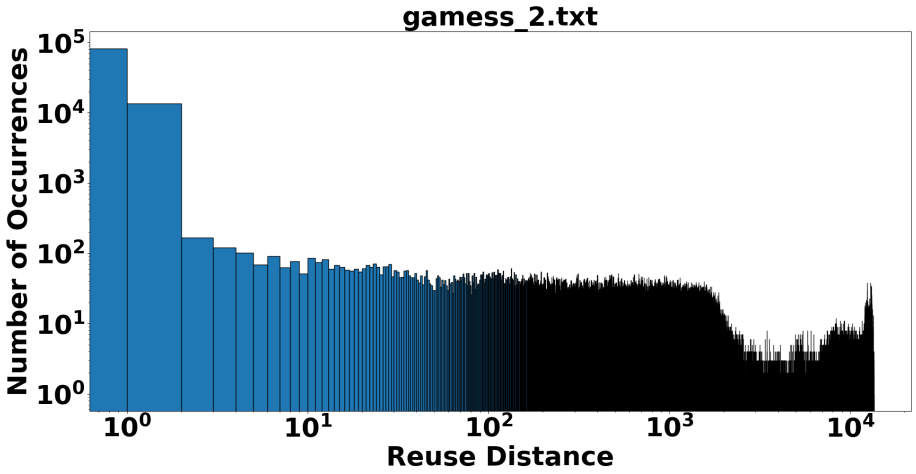

Figure 5.5: *Reuse distance histogram for the GAMESS_2 benchmark.*

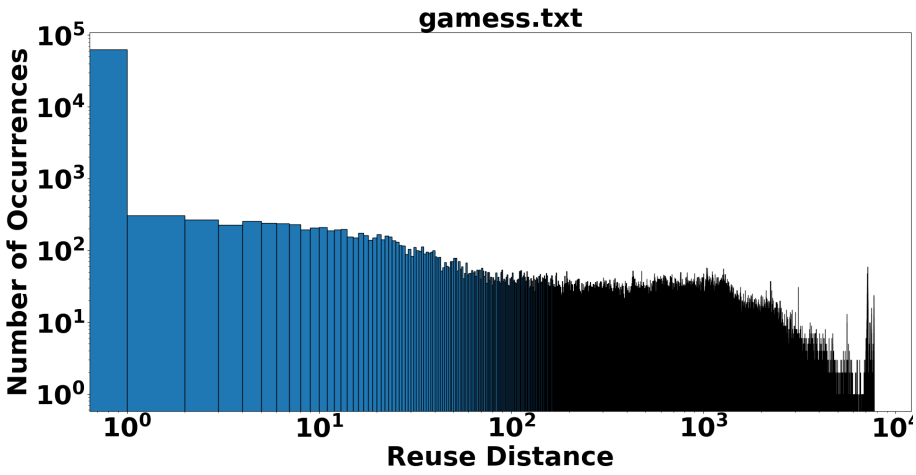

Figure 5.6: *Reuse distance histogram for the GAMESS benchmark.*

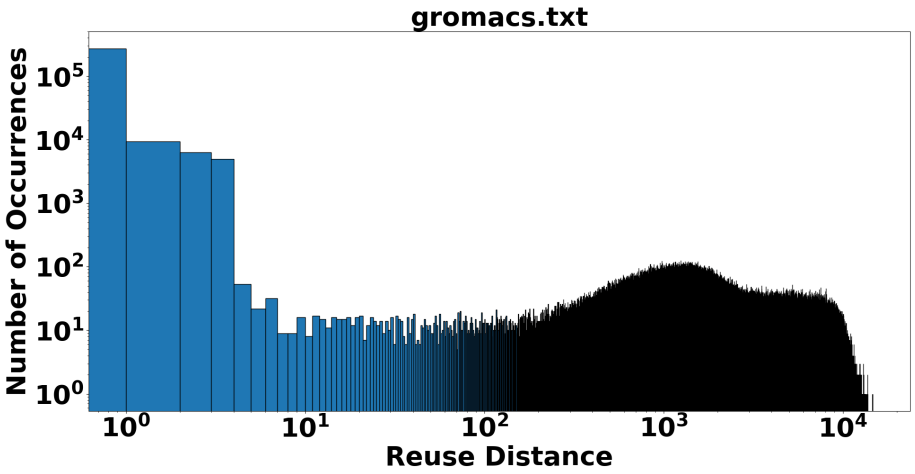

Figure 5.7: *Reuse distance histogram for the GROMACS benchmark.*

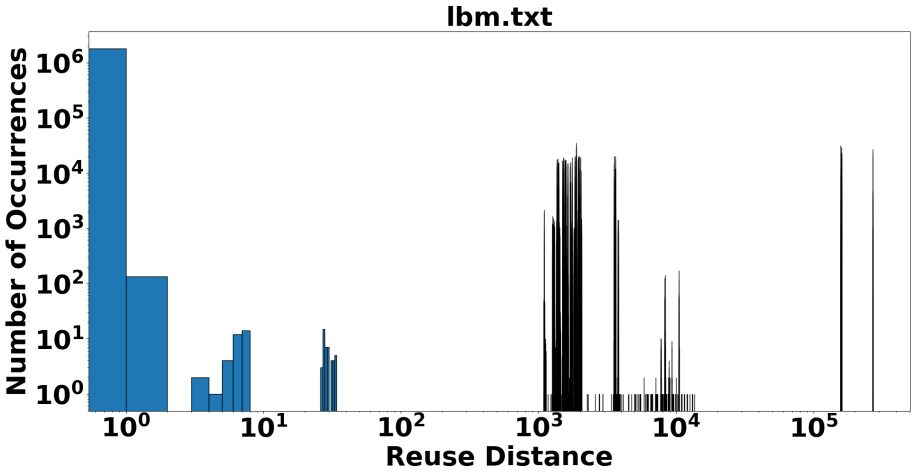

Figure 5.8: *Reuse distance histogram for the LBM benchmark.*

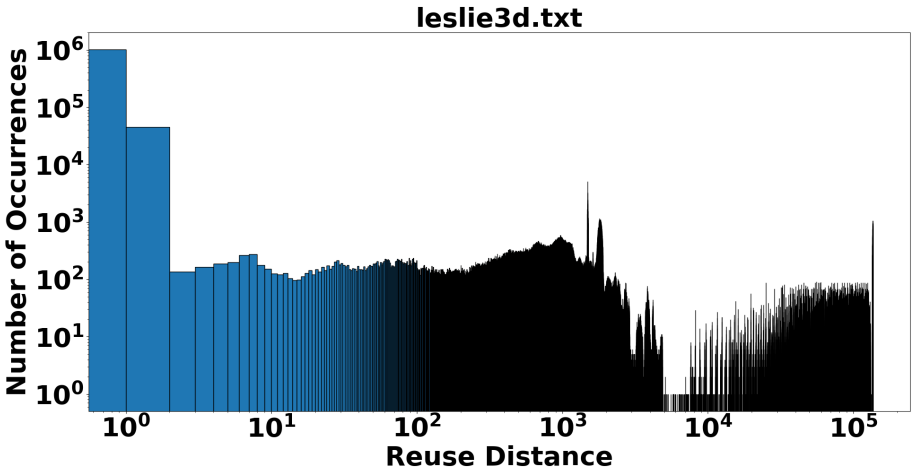

Figure 5.9: *Reuse distance histogram for the Leslie3d benchmark.*

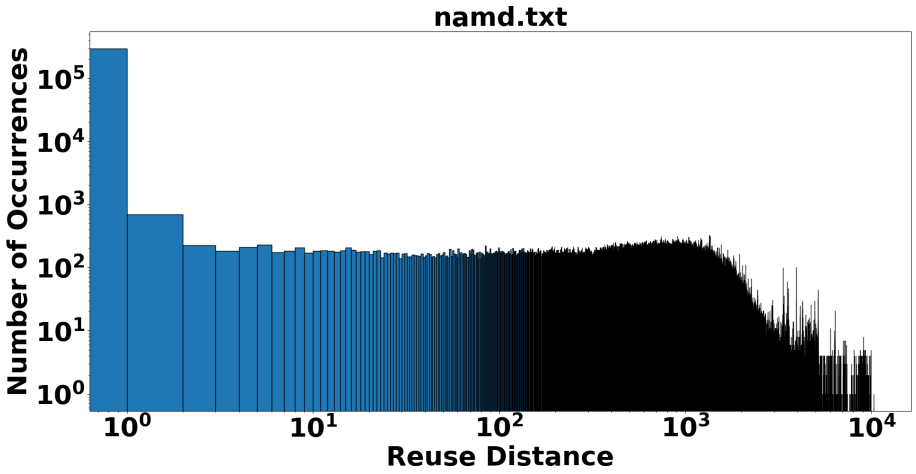

Figure 5.10: *Reuse distance histogram for the NAMD benchmark.*

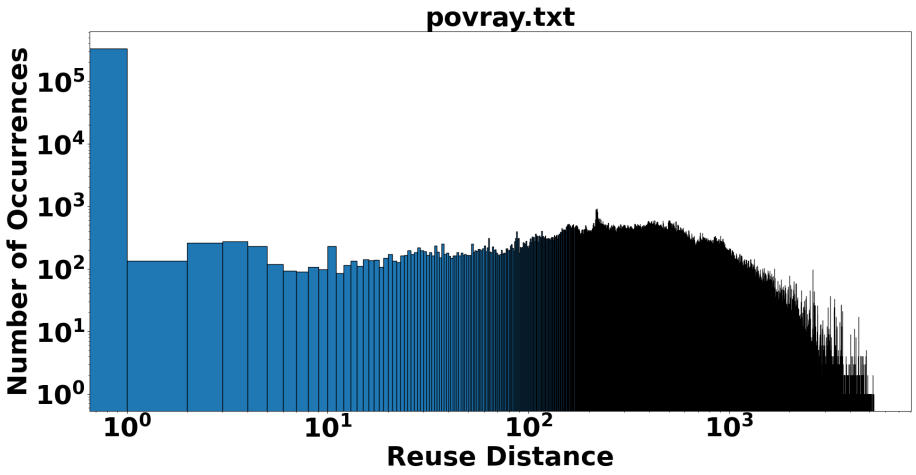

Figure 5.11: *Reuse distance histogram for the POV-Ray benchmark.*

5.1.2 Effects of the Replacement Algorithms on a Single Application Without Partitioning

The miss rates for the replacement algorithms running the application benchmarks with no partitioning can be seen in Figures 5.12 to 5.22. On the y-axis in the figures is the miss rates for each replacement algorithm running the trace and on the x-axis is the different cache sizes is increasing order. In all graphs we can clearly see that as the cache size increases, the miss rate decreases, which is to be expected. Another thing to note is that the miss rate remains unchanged after some cache size has been reached, after which the size of the cache does not affect the miss rate significantly. In the case where the cache size has reached a big enough size, mainly the cold misses remain which occur when the data is initially fetched from memory.

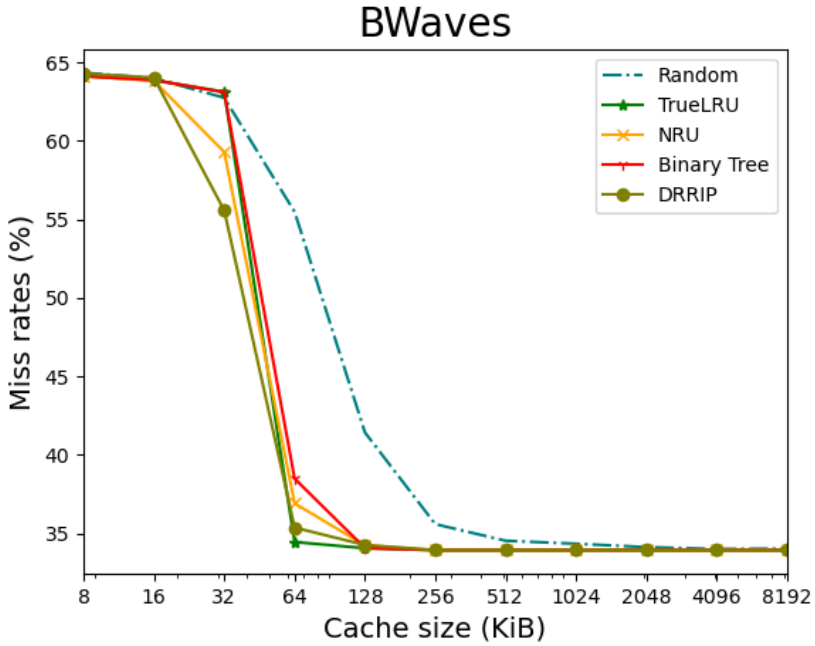

Figure 5.12: *The miss rate as a function of the cache size for different replacement algorithms with no partitioning running the Bwaves benchmark.*

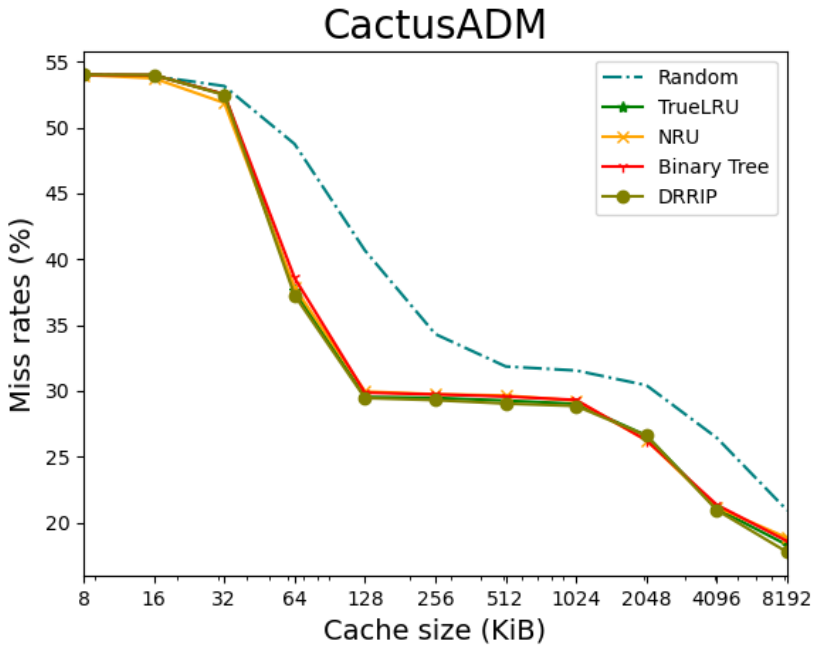

Figure 5.13: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the CactusADM benchmark.*

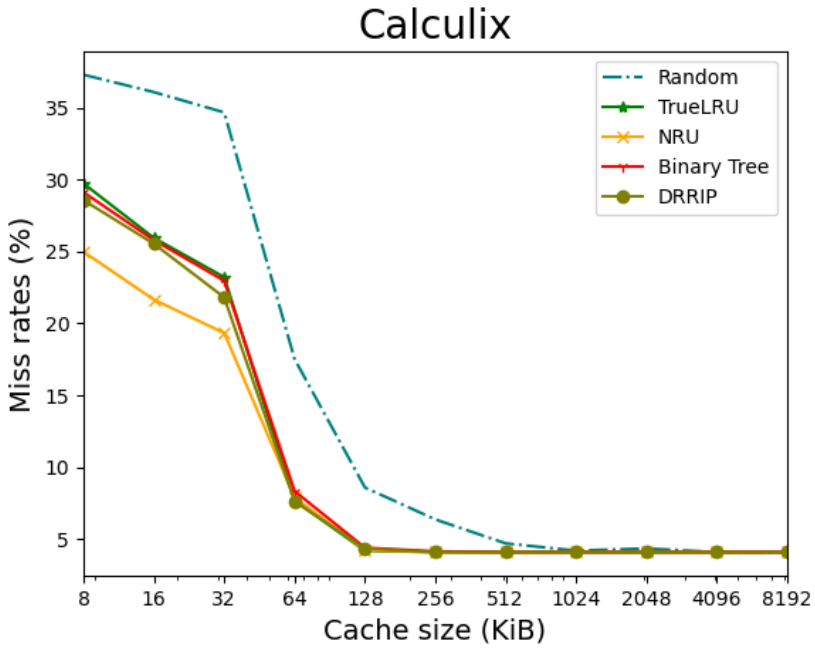

Figure 5.14: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the Calculix benchmark.*

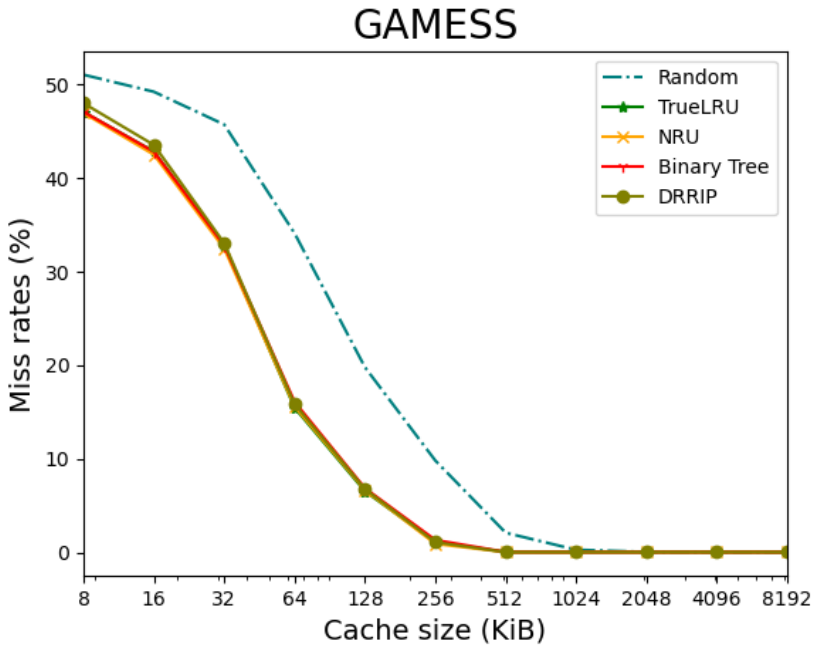

Figure 5.15: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the GAMESS benchmark.*

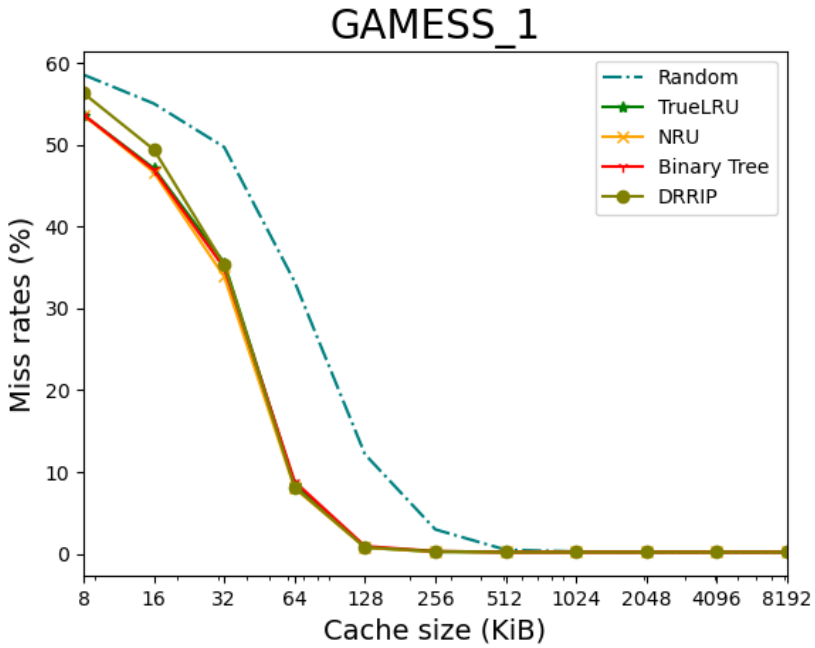

Figure 5.16: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the GAMESS_1 benchmark.*

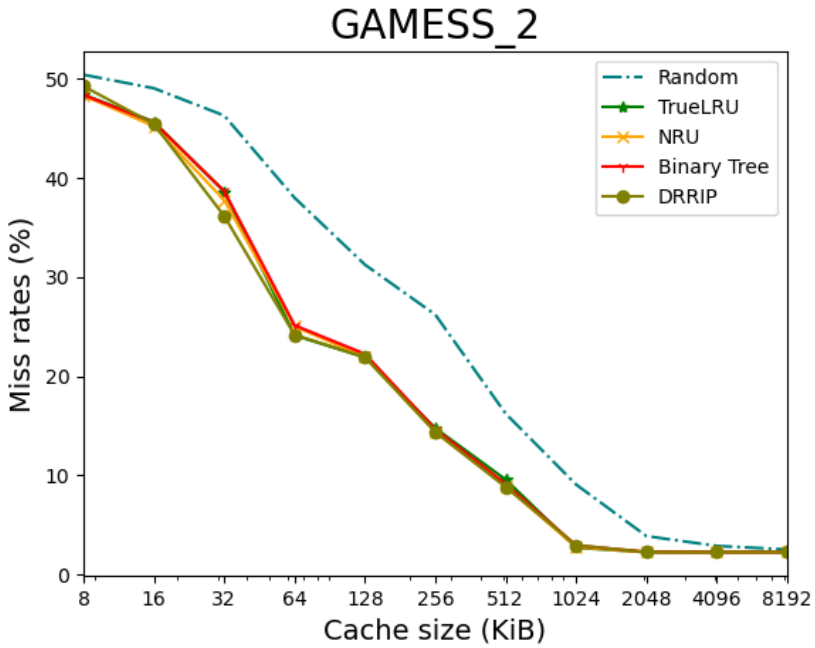

Figure 5.17: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the GAMESS_2 benchmark.*

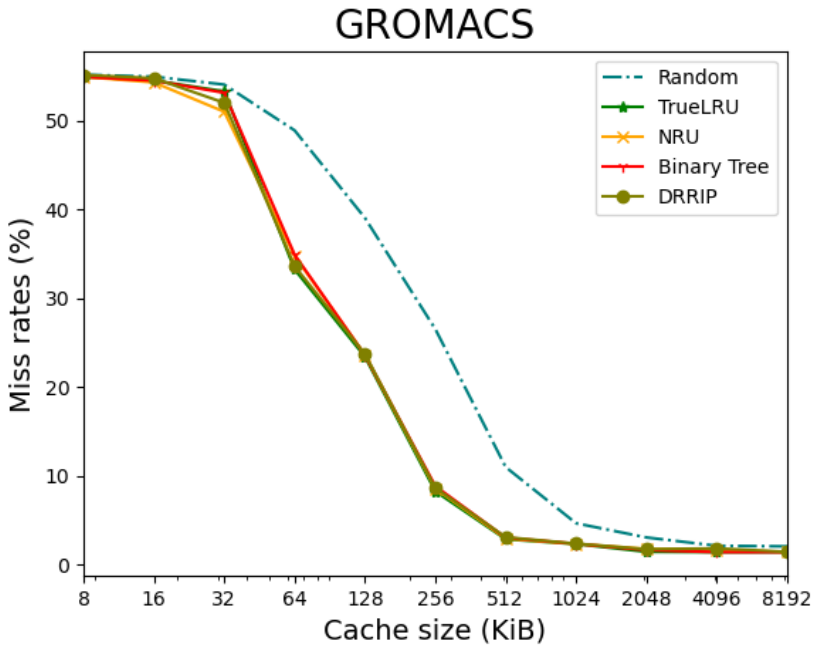

Figure 5.18: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the GROMACS benchmark.*

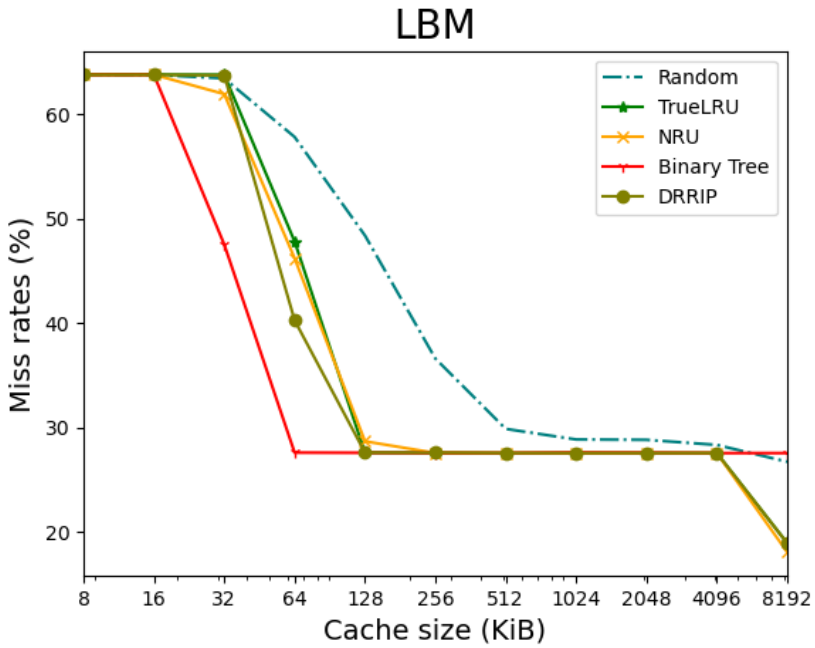

Figure 5.19: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the LBM benchmark.*

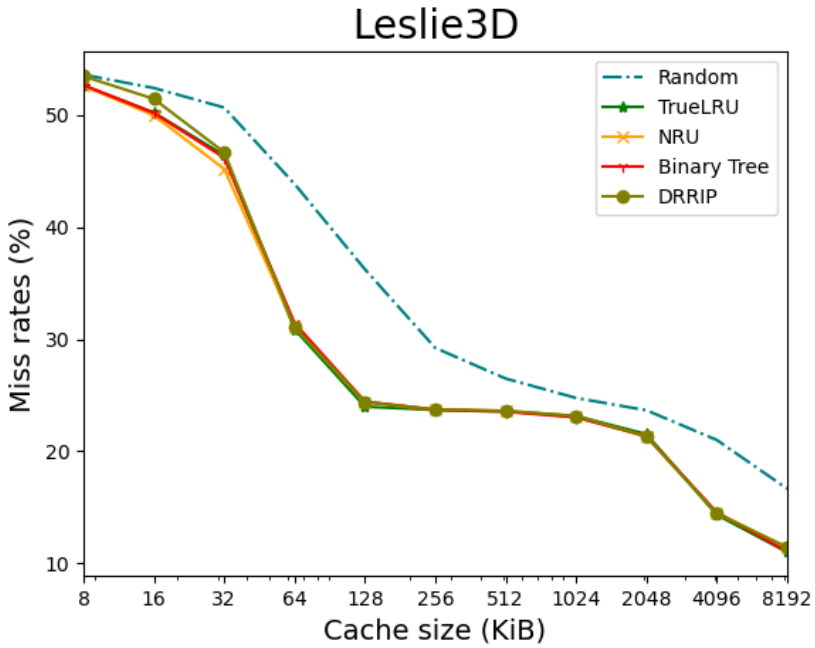

Figure 5.20: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the Leslie3D benchmark.*

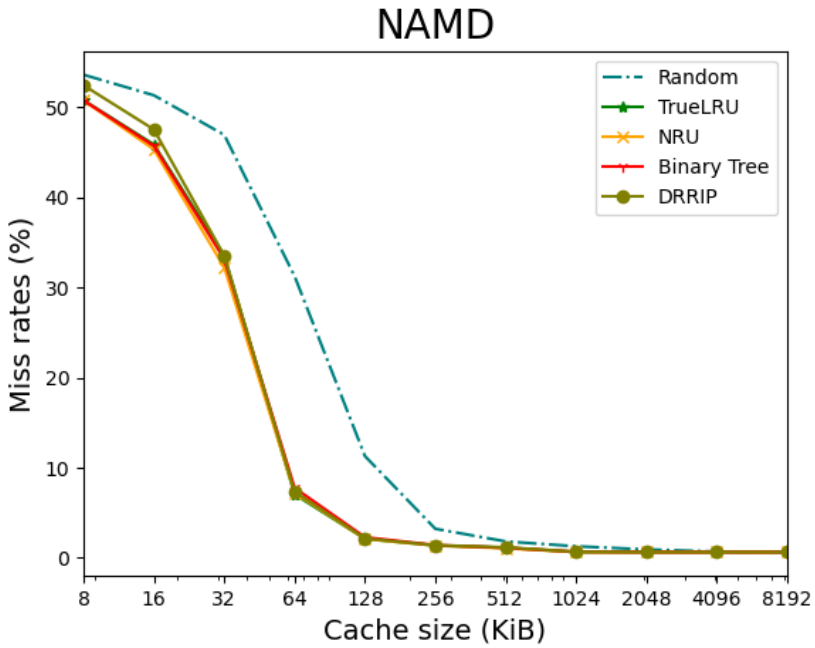

Figure 5.21: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the NAMD benchmark.*

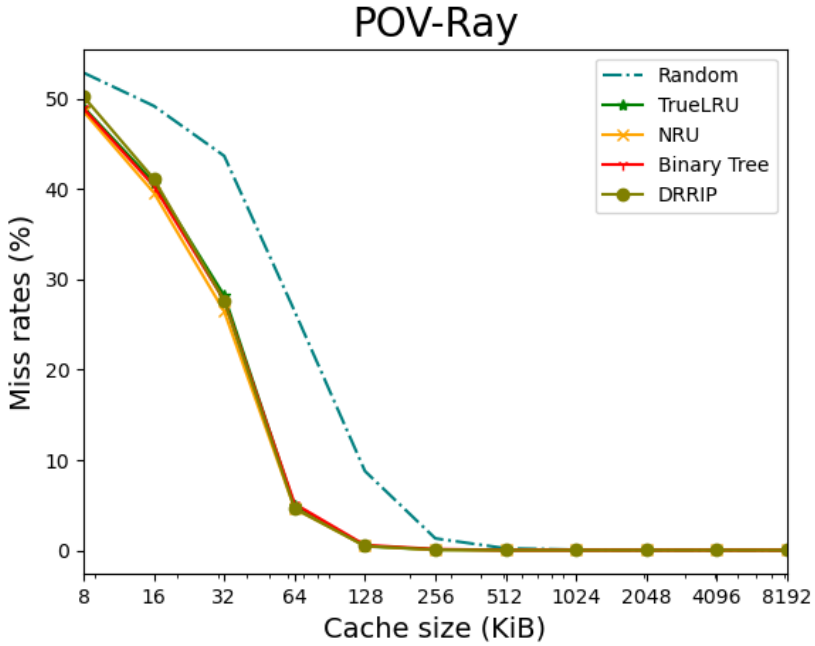

Figure 5.22: *The miss rate as a function of the cache size for all replacement algorithms with no partitioning running the POV-Ray benchmark.*

5.1.3 Effects of Partitioning on a Single Application

Now we compare the different replacement algorithms with different partition allocations. In Table 5.1 are listed the miss rates of the replacement algorithms running the CactusADM benchmark with three different partitioning allocations. We would expect the miss rates to increase as the cache size decreases regardless of partitioning mask. For example, if using no partitioning (i.e. partition mask = `11111111`) and a cache size of 4 KiB, then we would expect the miss rate to be similar to that of a cache using partitioning mask = `11110000` and a cache size of 8 KiB. This is because the total usable cache size would be similar in both cases.

As we see in Table 5.1 for example, the miss rates are not exactly the same but relatively close. This could be due to the fact that the misses from a partitioned cache of size 8 KiB and the misses in a non-partitioned cache of size 4 KiB do not occur for the same reason. In the former case, the misses are caused by a lack of associativity while the misses caused in the latter case are due to a lack of cache size. The miss rates illustrated in graphs can be seen in Figures 5.23 to 5.33. The figures show the miss rate curves for all benchmark applications running the *TrueLRU* replacement algorithm with three different partition allocations. The allocations can be seen in Table 4.2 under *Single CLOS*. We can see that, as we allocate less and less of the cache for the applications, the miss rate curve is shifted to the right. This shows that the partitioning works, since for example a cache of size 64 KiB with half the ways allocated (partition bit mask= `11110000`) performs comparatively to a cache of size 32 KiB with all the ways allocated (partition bit mask= `11111111` as seen in Figure 5.26.

CactusADM												
`Allocation = 11111111`												
Cache size (KiB)	8192	4096	2048	1024	512	256	128	64	32	16	8	Average
Random	200	252	291	301	304	328	388	466	508	515	516	370
LRU	175	201	254	277	280	281	282	359	502	516	516	331
NRU	180	202	251	279	283	284	286	362	496	513	516	332
Binary Tree	177	204	250	280	282	284	285	368	502	516	516	333
DRRIP	170	200	254	276	277	280	281	356	501	516	516	330
`Allocation = 11110000`												
Cache size (KiB)	8192	4096	2048	1024	512	256	128	64	32	16	8	Average
Random	246	294	311	314	318	372	464	507	516	516	516	398
LRU	192	243	288	296	298	299	380	492	516	516	516	367
NRU	190	236	286	297	298	303	364	482	516	516	516	364
Binary Tree	192	243	288	296	298	300	383	492	516	516	516	367
DRRIP	190	239	286	295	296	297	381	487	516	516	516	365
`Allocation = 00000011`												
Cache size (KiB)	8192	4096	2048	1024	512	256	128	64	32	16	8	Average
Random	316	318	322	325	334	460	503	516	516	516	516	422
LRU	256	298	314	314	316	330	476	512	516	516	516	397
NRU	253	295	312	312	314	328	474	510	516	516	516	395
Binary Tree	256	298	314	314	316	330	476	512	516	516	516	397
DRRIP	254	297	313	313	315	373	475	509	516	516	516	400

Table 5.1: *Different miss rates for the CactusADM benchmark running different partition allocations. All miss rate units are in* 10^3 *(e.g. 200 means 200 000 misses in the table). Colors indicate how results can be compared to each other in regards to cache size.*

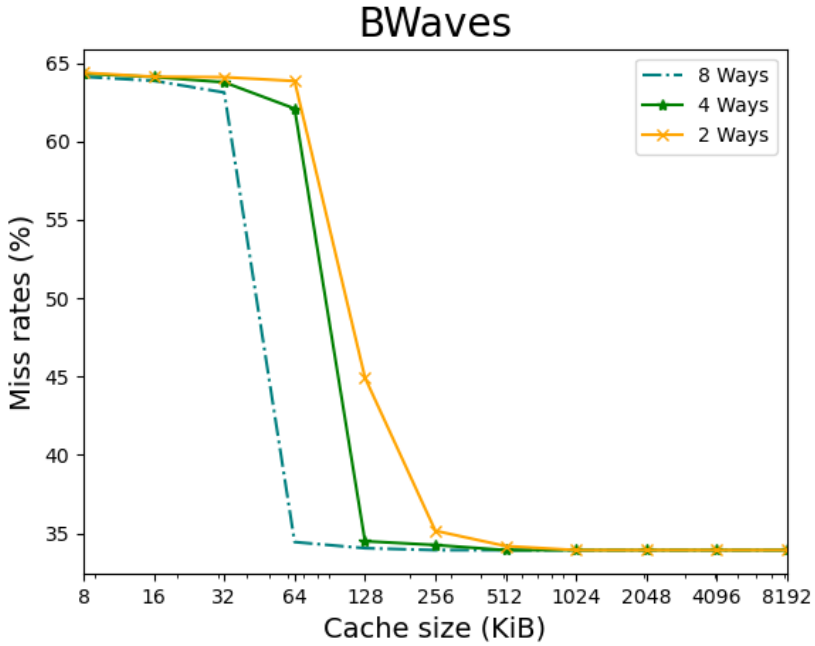

Figure 5.23: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the Bwaves benchmark.*

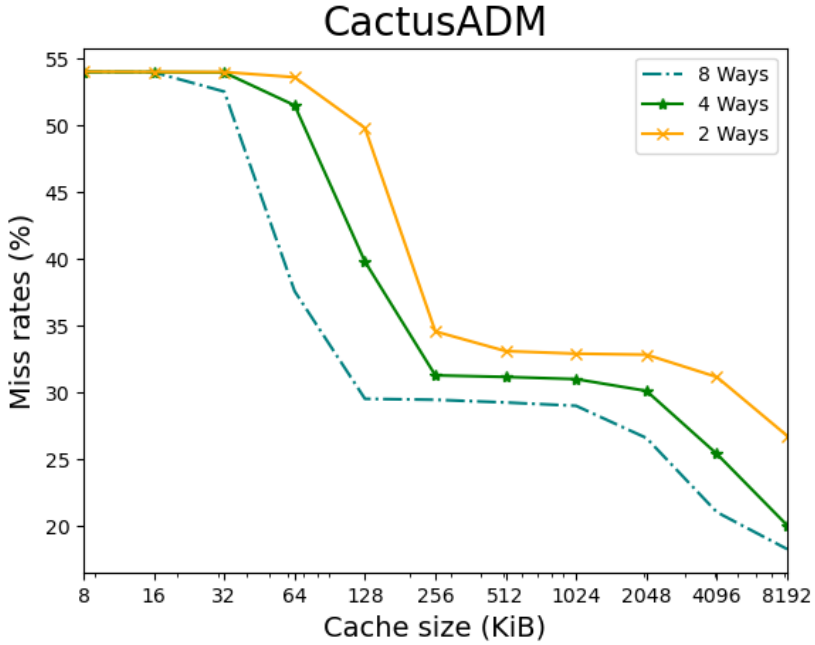

Figure 5.24: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the CactusADM benchmark.*

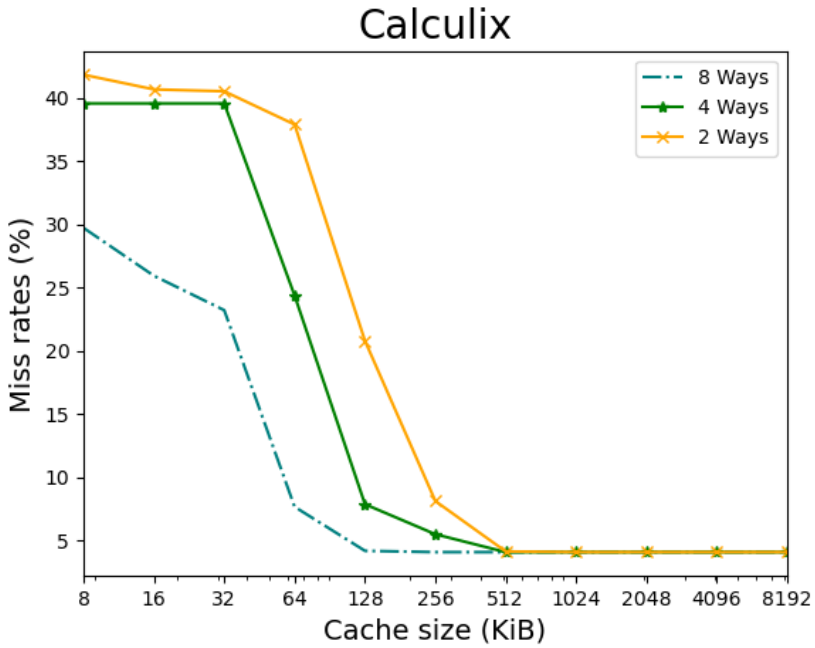

Figure 5.25: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the Calculix benchmark.*

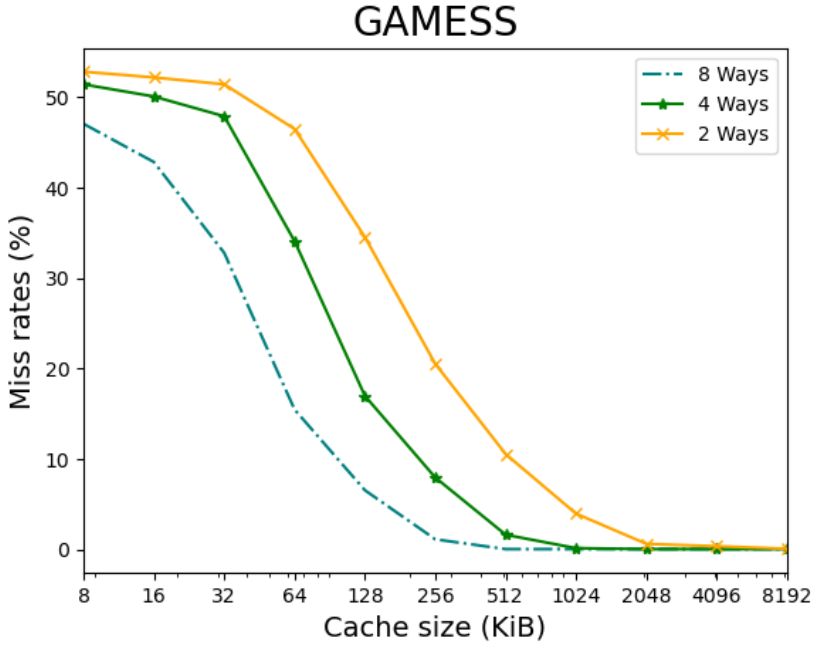

Figure 5.26: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the GAMESS benchmark.*

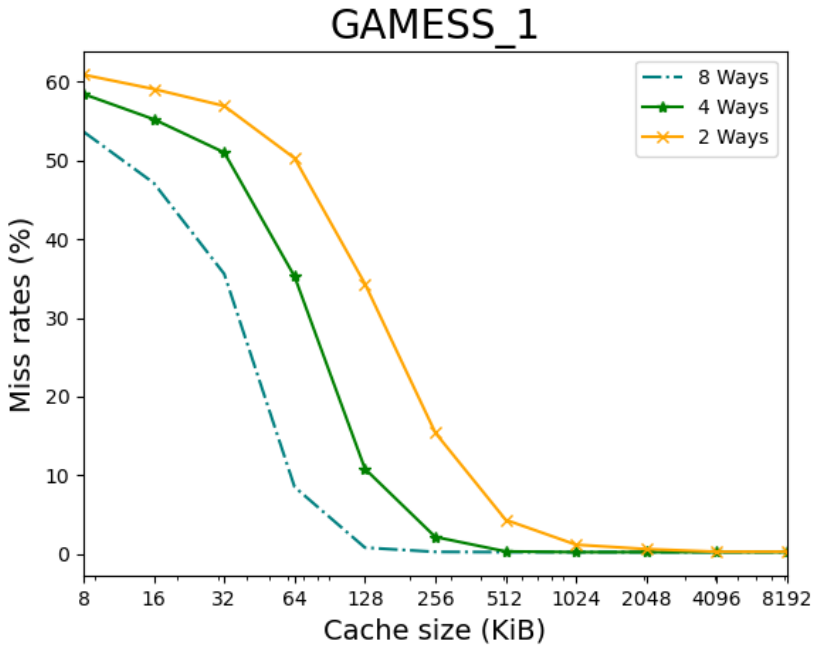

Figure 5.27: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the GAMESS_1 benchmark.*

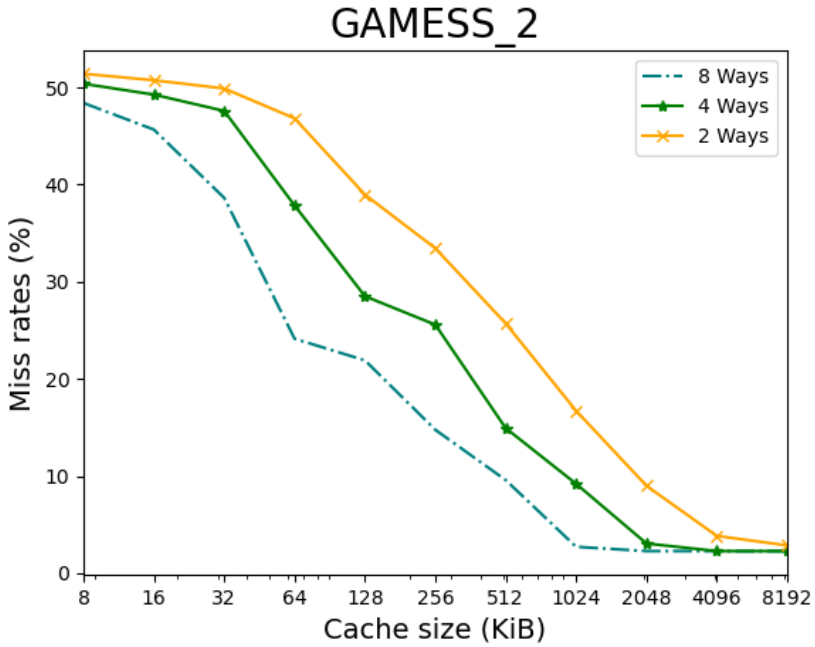

Figure 5.28: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the GAMESS_2 benchmark.*

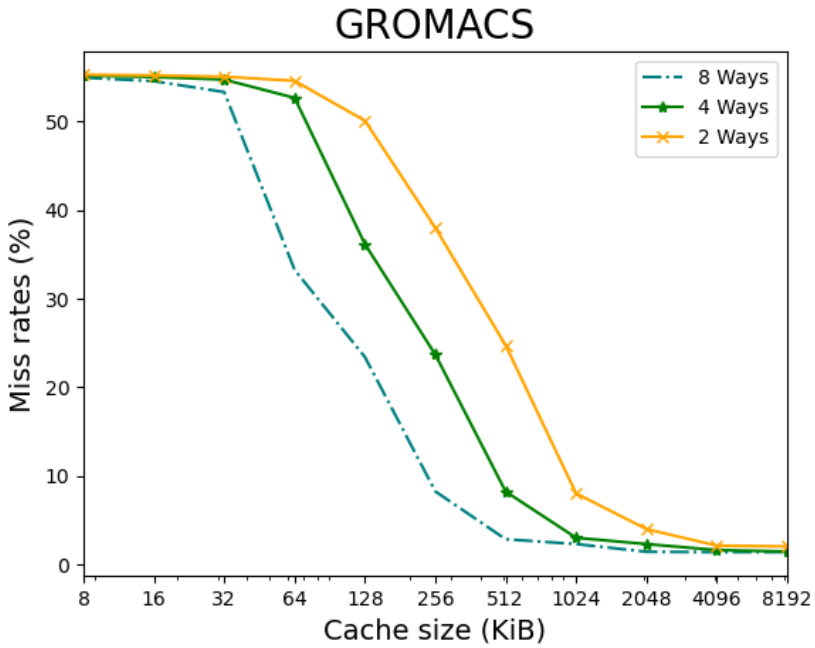

Figure 5.29: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the GROMACS benchmark.*

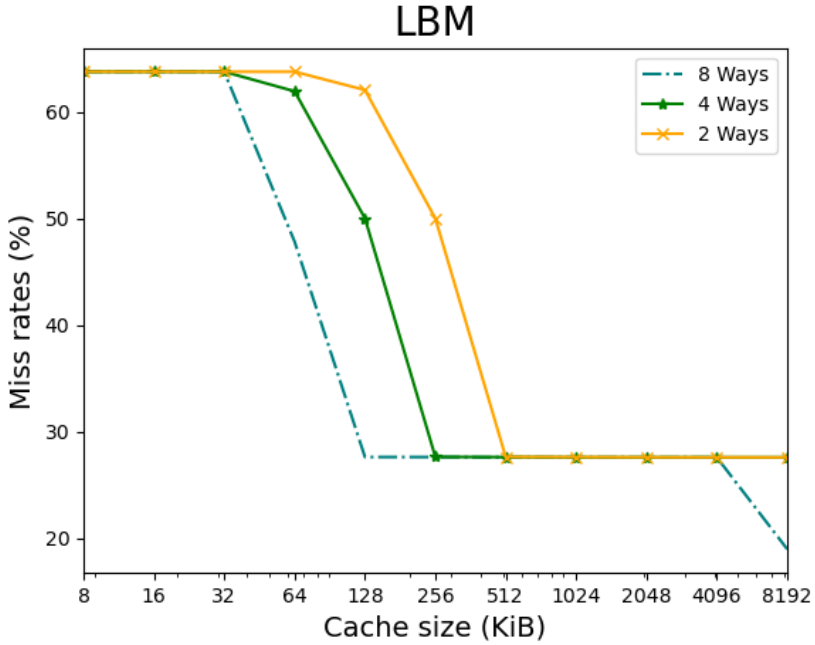

Figure 5.30: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the LBM benchmark.*

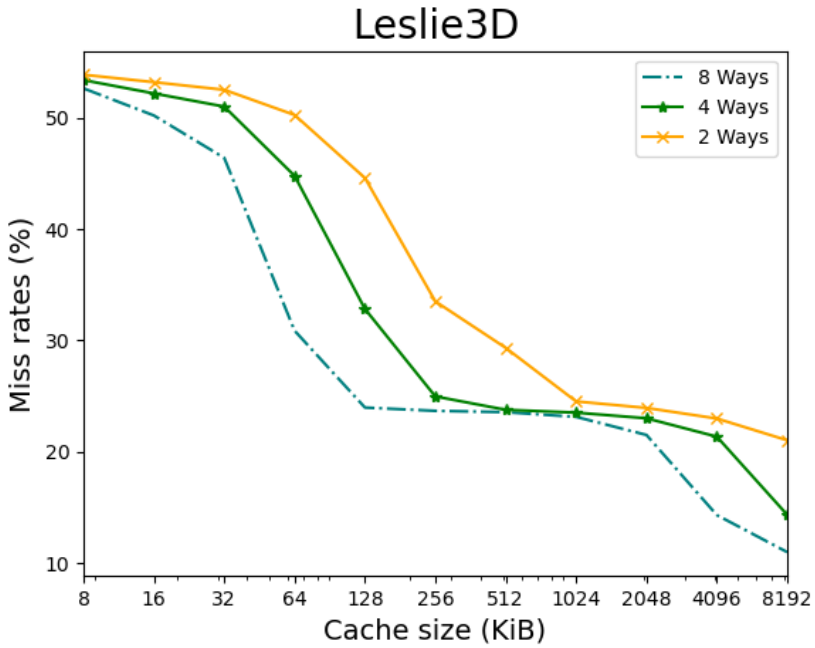

Figure 5.31: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the Leslie3D benchmark.*

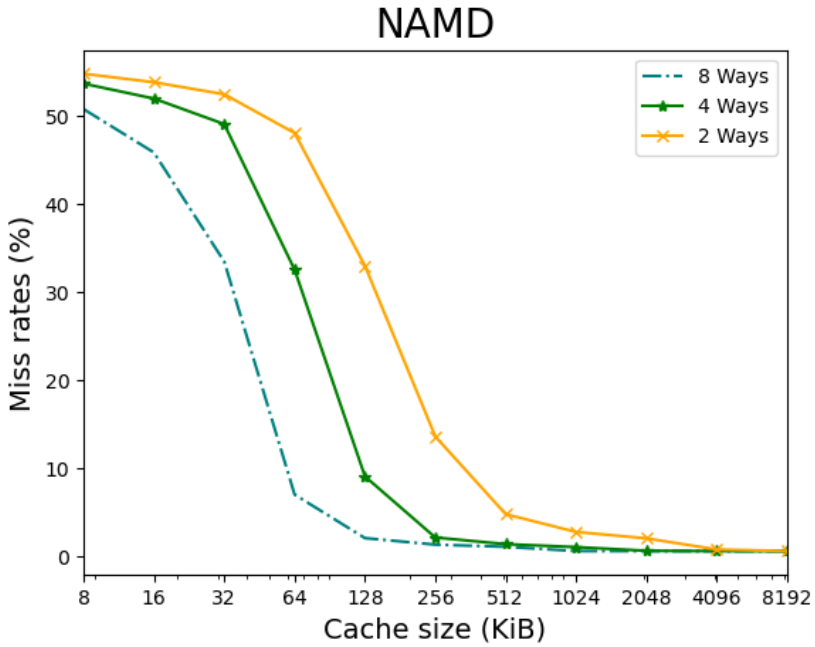

Figure 5.32: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the NAMD benchmark.*

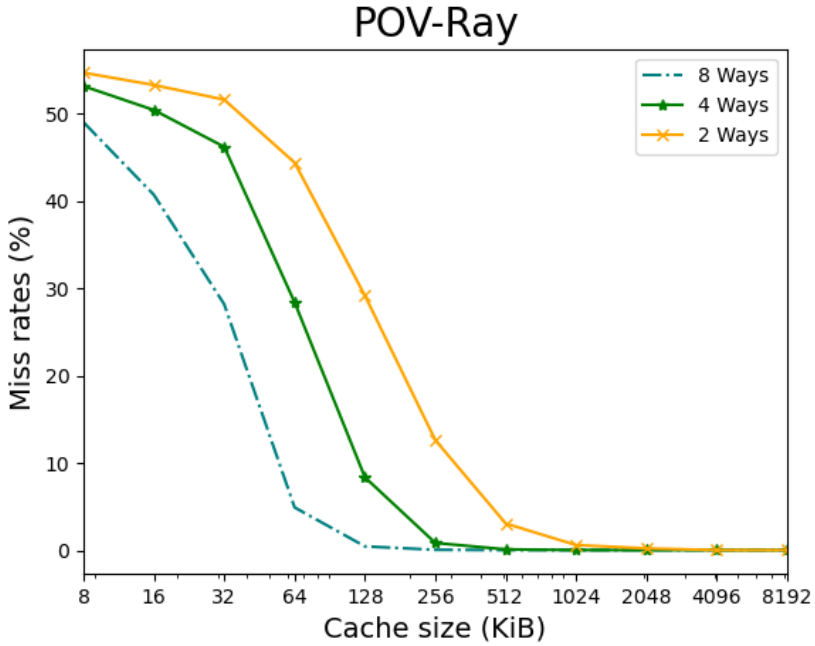

Figure 5.33: *The miss rate as a function of the cache size for the* TrueLRU *replacement algorithm with eight cache ways(bit mask = `11111111`), four cache ways (bit mask = `11110000`) and with two cache ways (bit mask = `00000011`) available in the partition running the POV-Ray benchmark.*

5.1.4 Effects of Partitioning on Multiple Applications with Different Ratios of Disjunct Partitions

This sections presents the results gathered from the tests of running two applications with disjunct partitioning and varying the ratio of how much each partition gets. It varies from 20/80, 50/50 and a 80/20 split between the partitions. For testing how partitioning affects the performance in a multi-application context, the test bench executed memory traces running in pairs to simulate the execution of multiple applications.

The three trace pairs chosen were *GAMESS* & *GAMESS_2*, *NAMD* & *GROMACS*, and *NAMD* & *POV-Ray*. These trace pairs were chosen because they were pairwise similar in the amount of L2 requests they executed. This creates an interesting competitive scenario between the two applications. The amount of L2 requests executed excluding warm up for each application pair can be seen in Table 5.2. For the multi-application simulation, *GAMESS_2, GROMACS,* and *POV-Ray* had their memory space shifted, as according to section 4.3.3 under *Memory Traces*.

Figures 5.34 to 5.42 contains stacked bar graphs showing the total number of misses for two applications, where the y-axis shows the total number of misses of the two applications and the x-axis shows the different ratios of disjunct partitioning (e.g. 20% would be two ways out of the eight available). The two different colors represents the misses for each application and each bar represents each replacement

Application Pairs	Application 1	Application 2
GAMESS & GAMESS_2	139,721	201,807
NAMD & GROMACS	679,378	652,304
NAMD & POV-Ray	679,378	761,129

Table 5.2: *Table containing the amount of L2 requests executed for each application pair excluding warm up. Application 1 refers to the first named application in each pair.*

algorithm used. We can see in Figure 5.40 to 5.42 that the total number of misses stays the same over different disjunct partitions. If we look at the histograms for NAMD and POV-Ray (Figure 5.10 and Figure 5.11), we can see that their reuse distance look very similar for the shorter reuse distances, which would mean that they utilize the cache in a similar manner. We can therefore conclude that, in this case, the partitioning between these two applications does not drastically affect their performance.

Discussion of Results

When it comes to the results gathered from using different ratios of disjunct partitions (Section 5.1.4) we can see that in Figures 5.34 to 5.36 the total number of misses remain relatively unchanged as we vary the ratios of disjunct partitions. This would make sense if both applications (GAMESS and GAMESS_2) react similarly to gaining and losing cache ways.

However, we can see when comparing *NAMD* and *GROMACS* (see Figures 5.37 to 5.39) that it is apparent that NAMD sees more benefit from getting more cache ways than GROMACS does. If we compare NAMD's and GROMACS' memory trace histograms (Figures 5.10 and 5.7) we see that NAMD's curve is more uniform than GROMACS'. This might indicate that as we give NAMD more cache ways it can use those ways more efficiently than GROMACS since there are fewer occurrences of memory accesses with a reuse distance larger than 10 if compared to NAMD which is a bit higher. If we also take a look at the histograms of GAMESS and GAMESS_2, we see that they are more similar to each other than NAMD and GROMACS are. Also, comparing *NAMD* and *POV-ray* in Figures 5.40 to 5.42 shows that both *POV-ray* and *NAMD* see significant benefit from added cache ways. This is to be expected as their histograms (Figures 5.10 to 5.11) look similar.

The conclusions that can be drawn from this is that it is best to split the cache 50/50 to yield best overall performance in terms of number of misses in these example scenarios. This can be seen as the total number of misses is the least with the 50/50 partition split in Figures 5.34 to 5.42. This is just an example of when the target metric is to reduce the overall number of misses across the applications, and the target metric can differ along with the optimal allocation used. An example might be to ensure quality of service for a specific high priority application.

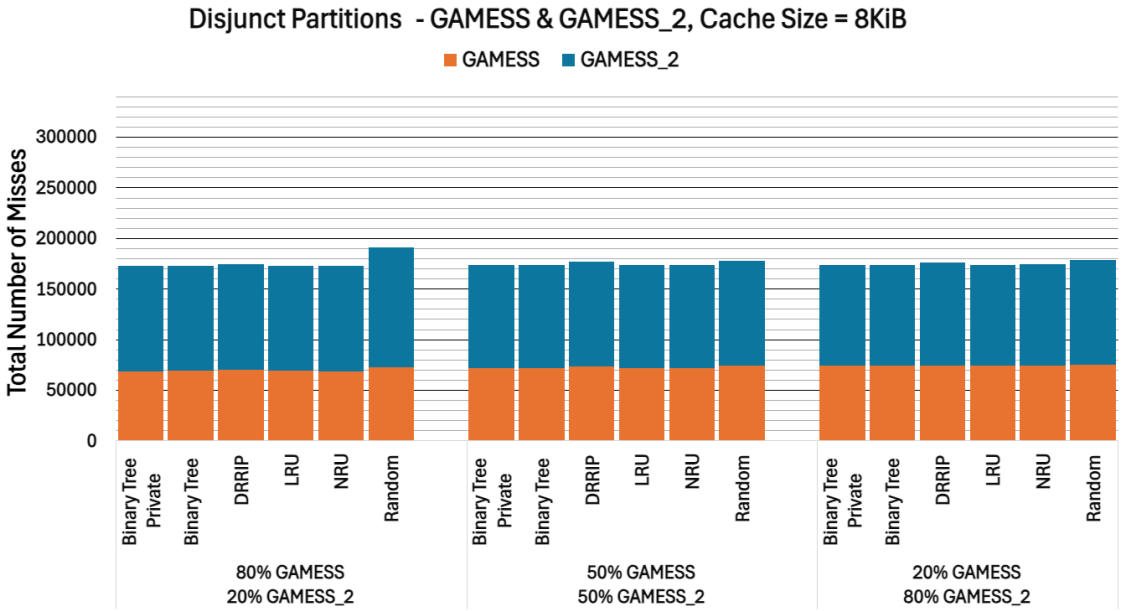

Figure 5.34: *The number of misses of two applications running the* GAMESS *and* GAMESS_2 *benchmarks for a 8 KiB cache.*

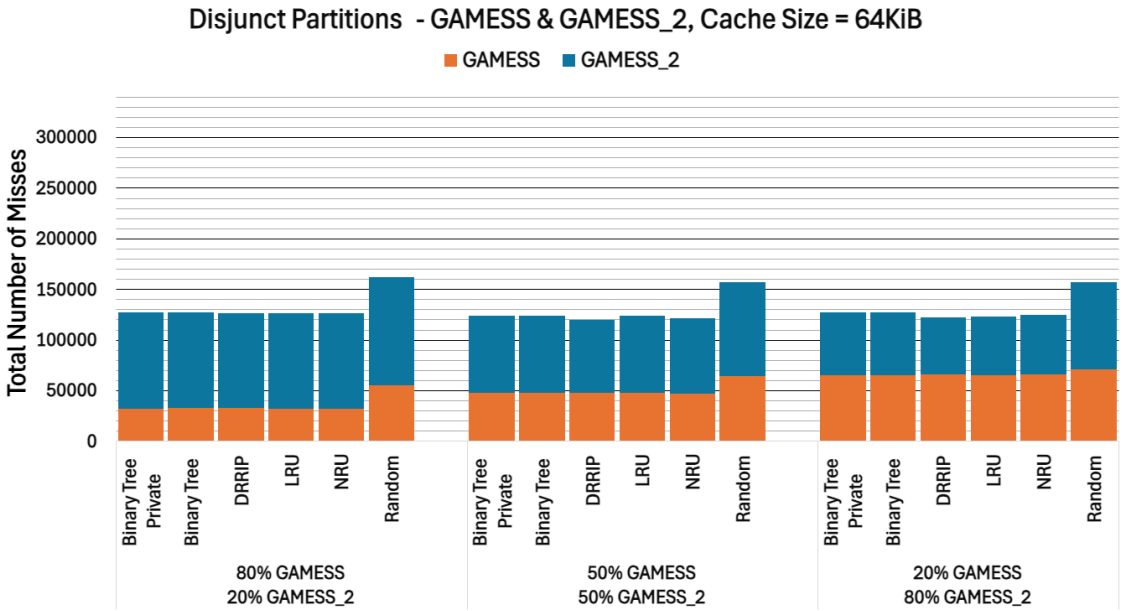

Figure 5.35: *The number of misses of two applications running the* GAMESS *and* GAMESS_2 *benchmarks for a 64 KiB cache.*

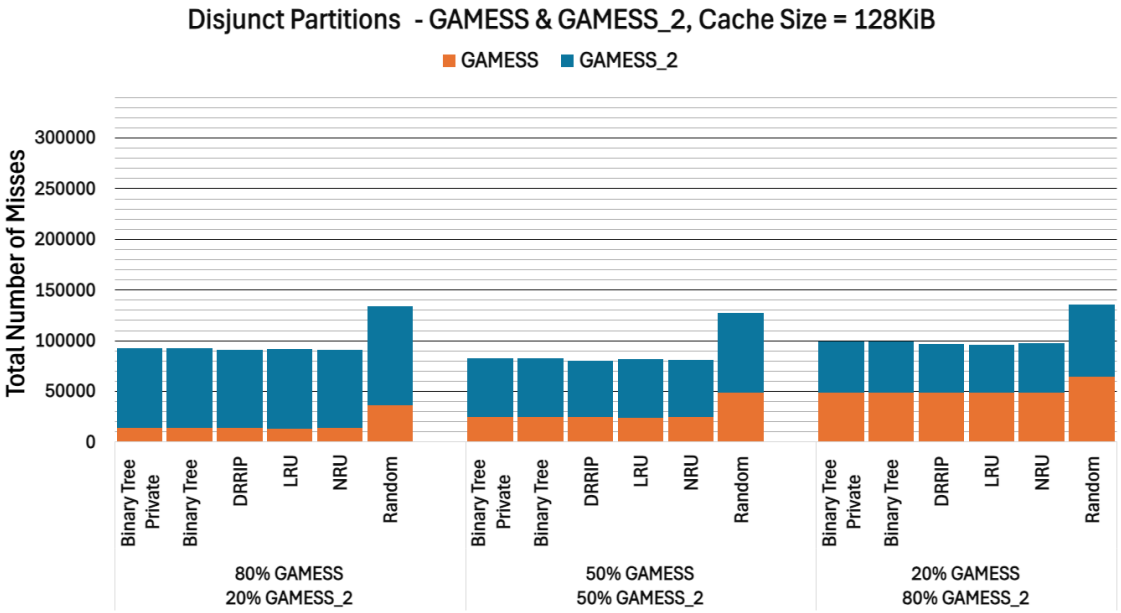

Figure 5.36: *The number of misses of two applications running the* GAMESS *and* GAMESS_2 *benchmarks for a 128 KiB cache.*

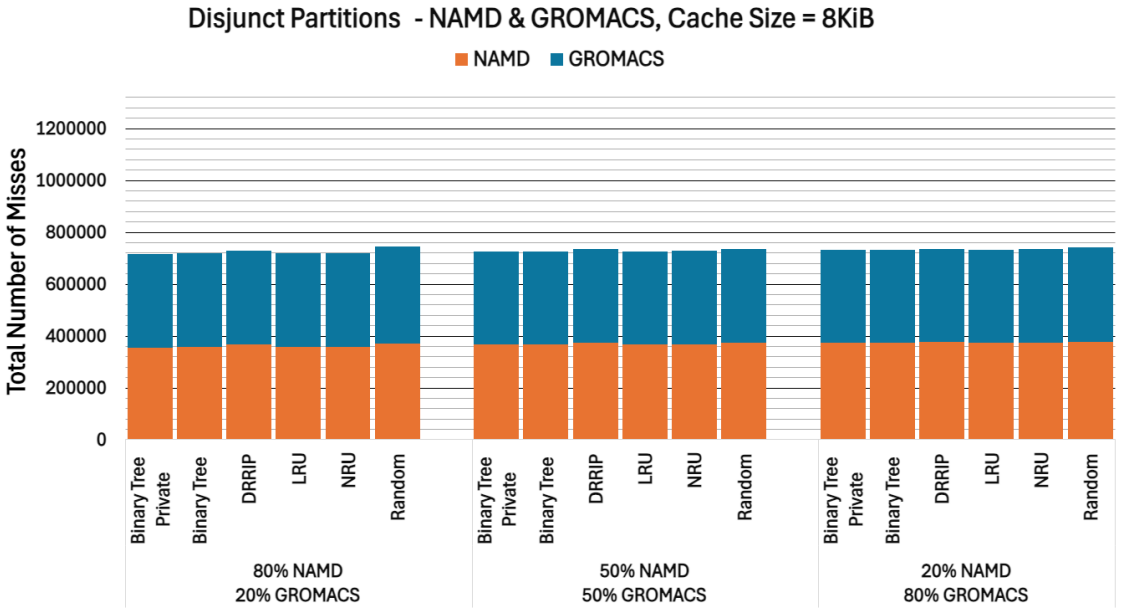

Figure 5.37: *The number of misses of two applications running the* NAMD *and* GROMACS *benchmarks for a 8 KiB cache.*

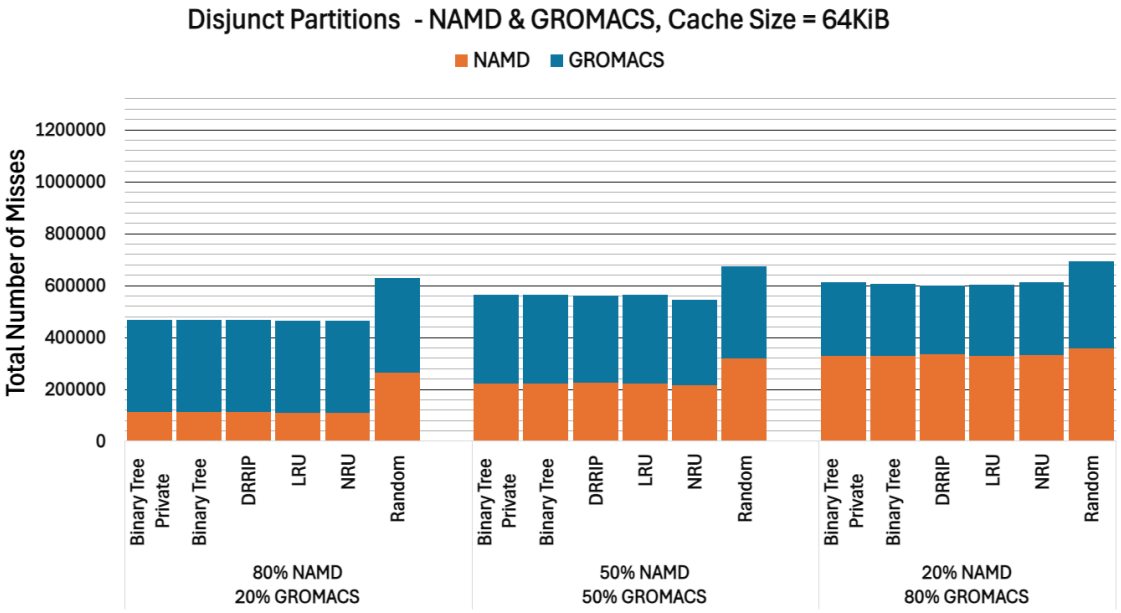

Figure 5.38: *The number of misses of two applications running the* NAMD *and* GROMACS *benchmarks for a 64 KiB cache.*

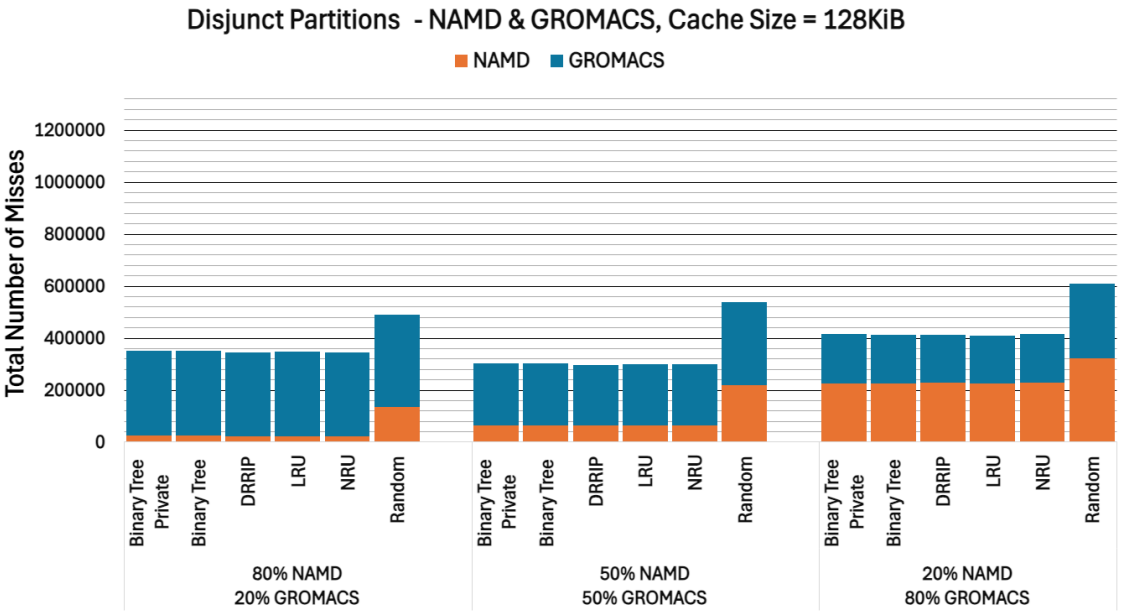

Figure 5.39: *The number of misses of two applications running the* NAMD *and* GROMACS *benchmarks for a 128 KiB cache.*

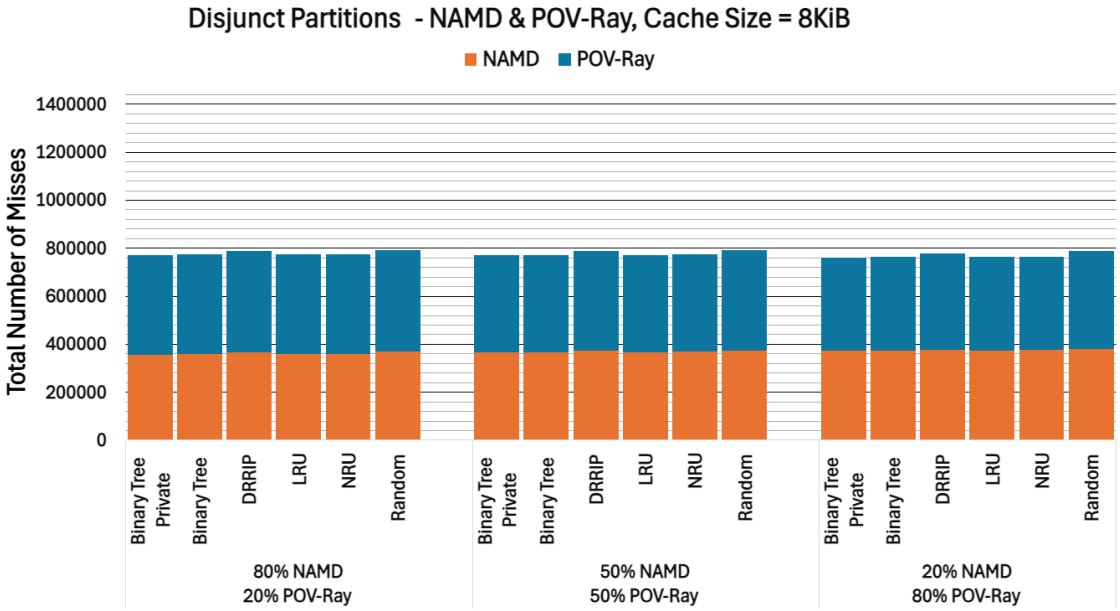

Figure 5.40: *The number of misses of two applications running the* NAMD *and* POV-Ray *benchmarks for a 8 KiB cache.*

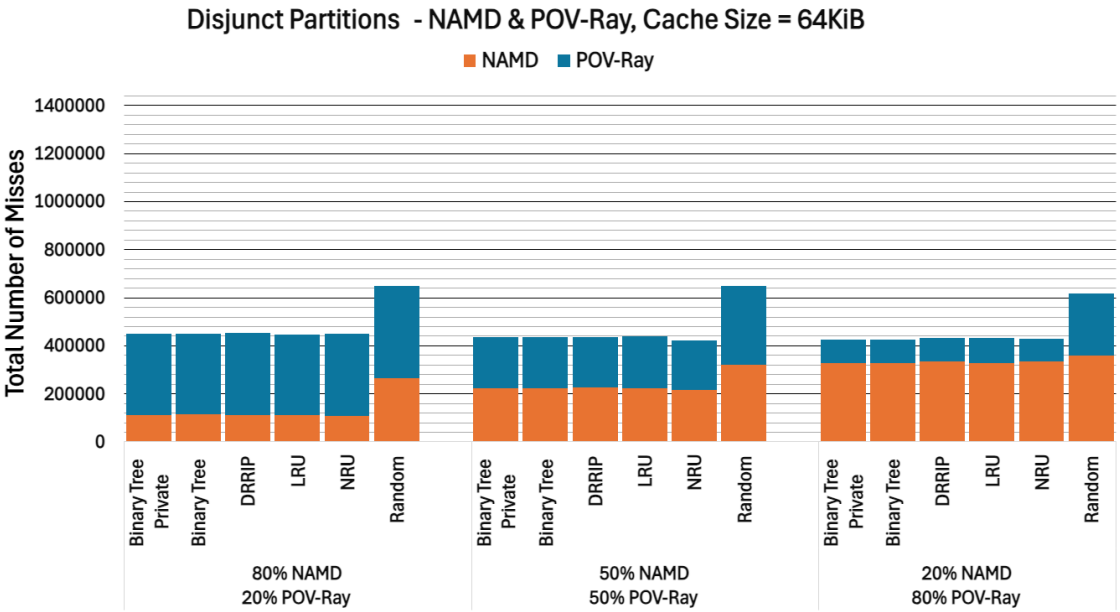

Figure 5.41: *The number of misses of two applications running the* NAMD *and* POV-Ray *benchmarks for a 64 KiB cache.*

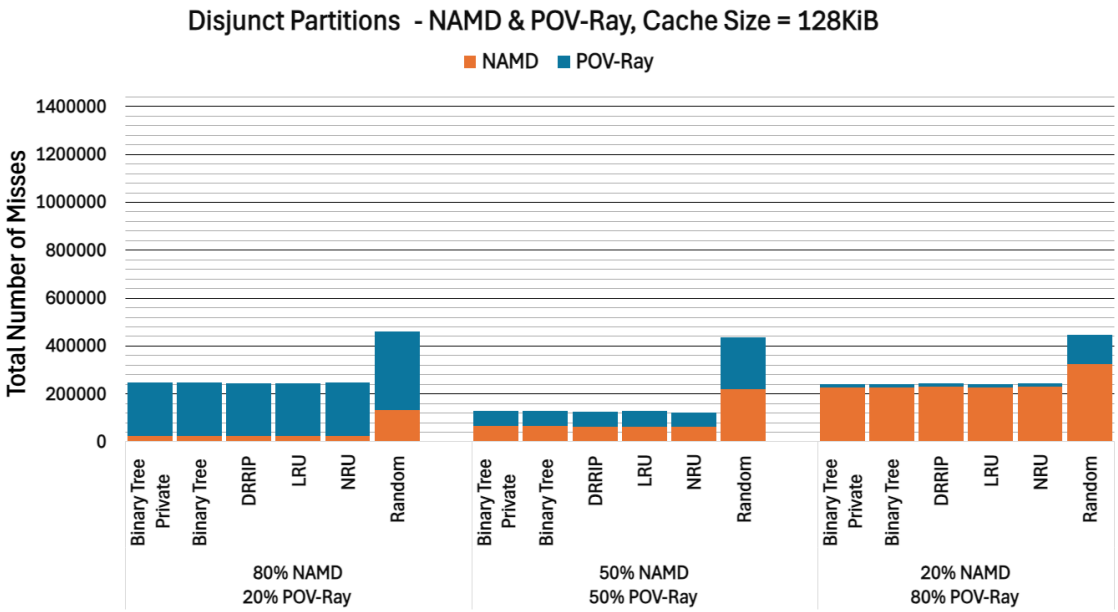

Figure 5.42: *The number of misses of two applications running the* NAMD *and* POV-Ray *benchmarks for a 128 KiB cache.*

5.1.5 Effects of Partitioning on Multiple Applications with Overlapping Partitions

Previously we showed the results where each application had an isolated partition of the cache. We will now present the results of overlapping each applications' partition. For these tests, the same application pairs as described in Section 5.1.4 were used. Instead of having disjunct partitions between the pairs, we varied the partitioning from a completely disjunct allocation to a fully overlapped allocation.

The Y-axis shows the total number of misses and the X-axis shows the different levels of overlapping partitions. For example, a 6/8 overlapping partition means that six of the eight ways in the cache are shared between the application pairs. The results from these tests can be found in Figure 5.43 to Figure 5.60.

Discussion of Results

After testing how the algorithms perform in a multi-threaded scenario for the *GAMESS* and *GAMESS_2* benchmarks, it became clear that the applications performed better when they had access to more ways in the cache, even though they had to compete over shared cache space. Looking at Figures 5.43 to 5.48, the number of misses for the GAMESS application remain relatively the same as the overlap increases while the misses for GAMESS_2 decrease significantly.

If we look at the tests for *NAMD* and *GROMACS* (see Figures 5.49 to 5.54) we instead see for some of the replacement algorithms the number of misses increase slightly as the overlap increases. Furthermore, looking at the tests for *NAMD* and *POV-ray* (see Figures 5.55 to 5.60) the trend seems to be that the misses decrease as the overlap increases. Therefore, whether or not the number of misses increase or decrease with the overlap seems to be depending on which applications are paired with each other. Also it appears to be depending on the size of the application's working set as can be seen in Figure 5.52 where only the 128 KiB cache sees an increase of misses as the overlap increases. This might be due to the fact that the extra ways of an already big cache might not outweigh the interference effect of having more overlap.

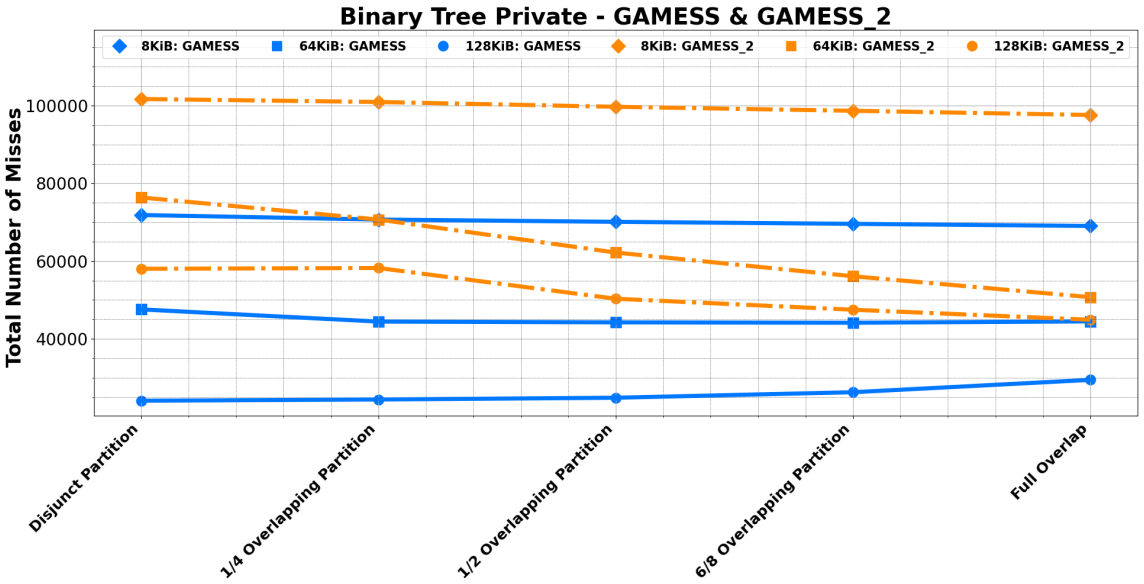

Figure 5.43: *Misses for GAMESS and GAMESS_2 running Binary Tree Private*

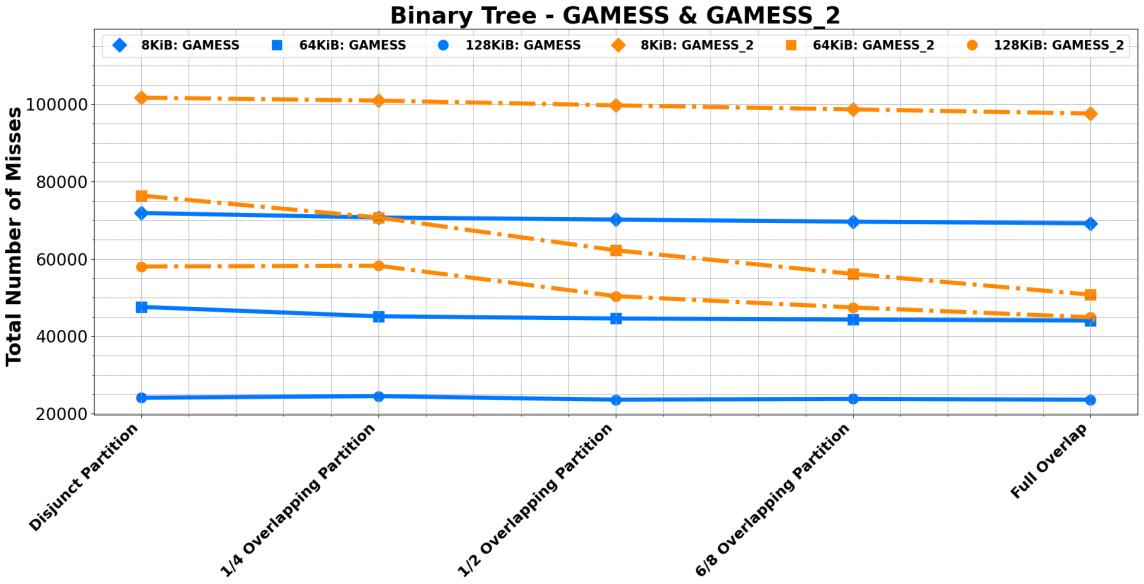

Figure 5.44: *Misses for GAMESS and GAMESS_2 running Binary Tree*

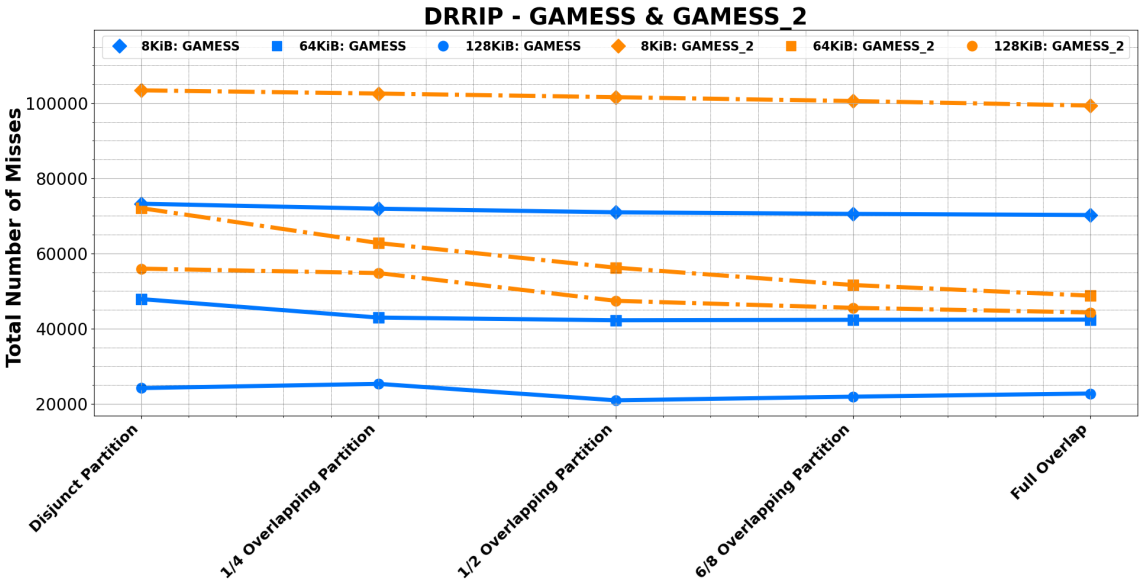

Figure 5.45: *Misses for GAMESS and GAMESS_2 running DRRIP*

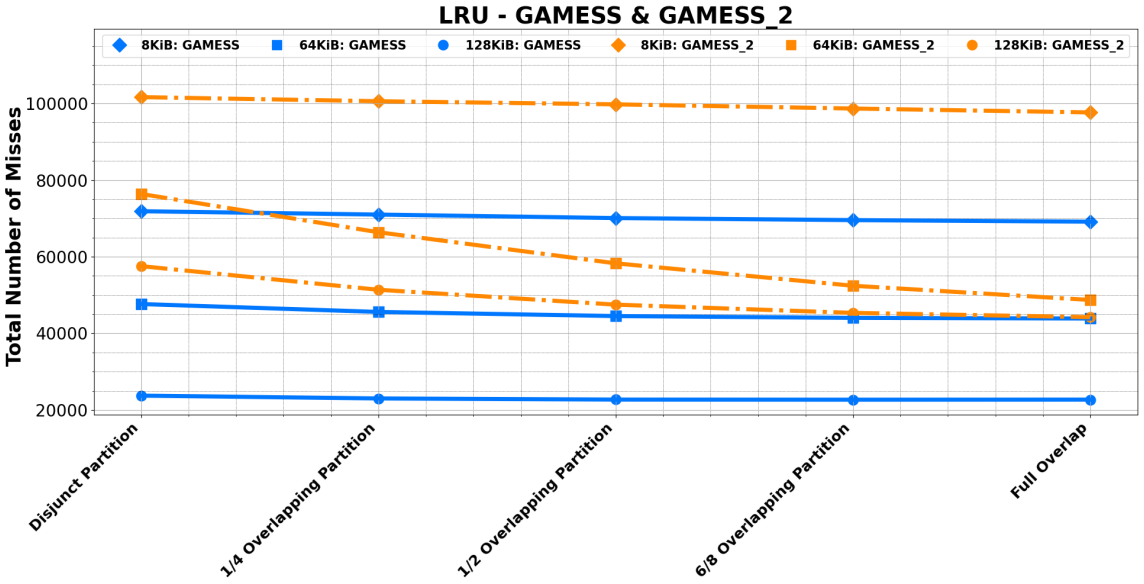

Figure 5.46: *Misses for GAMESS and GAMESS_2 running LRU*

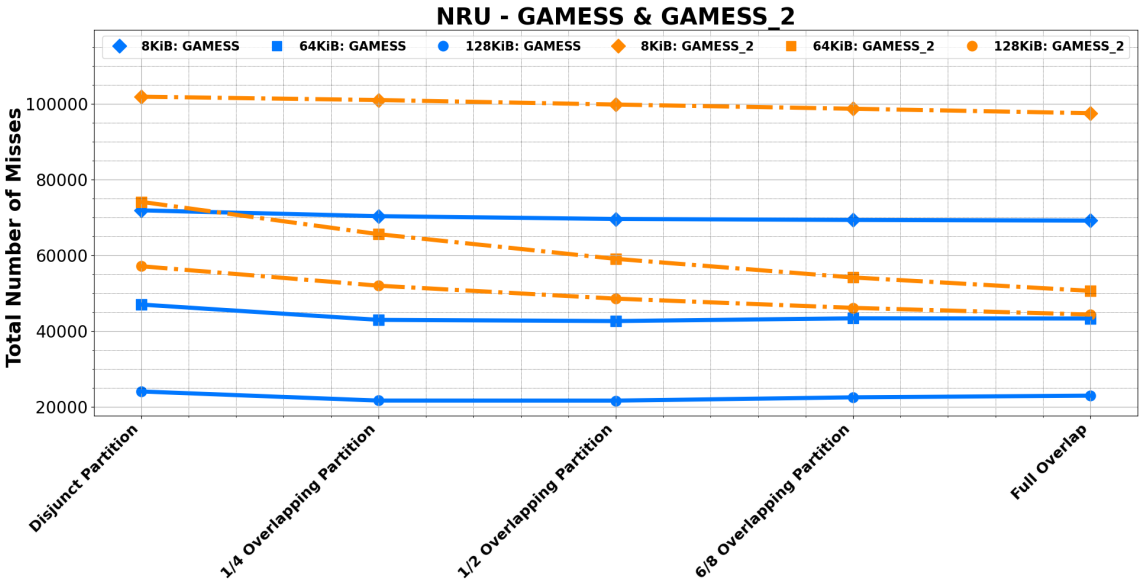

Figure 5.47: *Misses for GAMESS and GAMESS_2 running NRU*

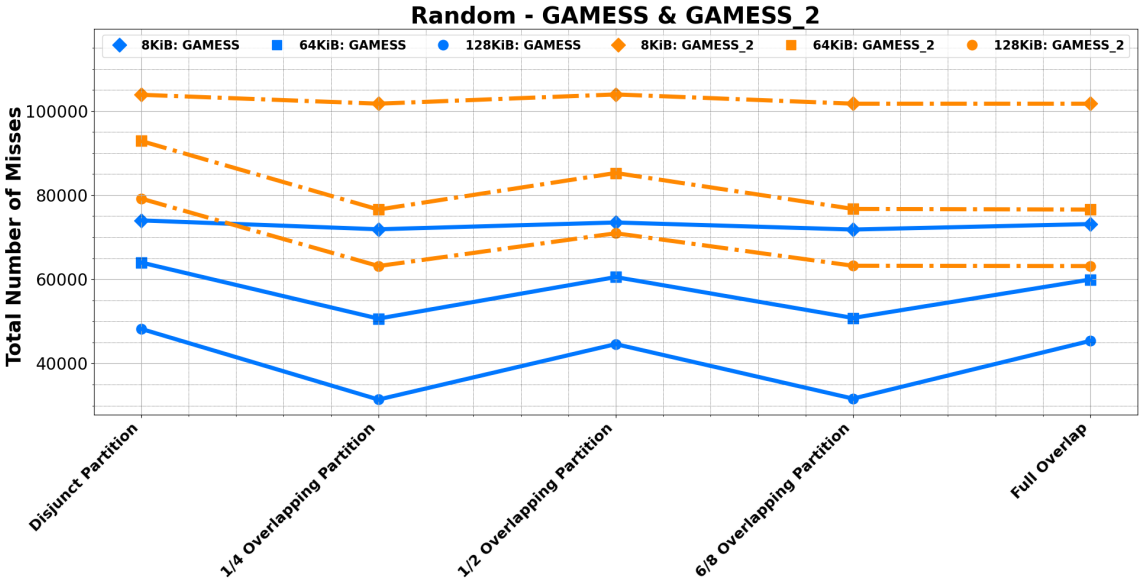

Figure 5.48: *Misses for GAMESS and GAMESS_2 running Random*

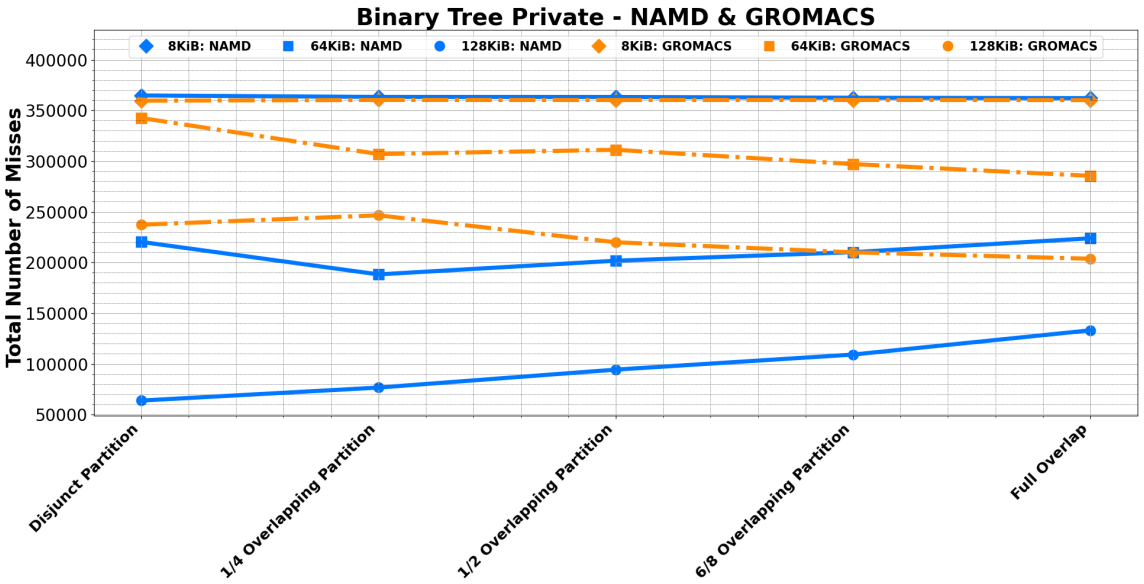

Figure 5.49: *Misses for NAMD and GROMACS running Binary Tree Private*

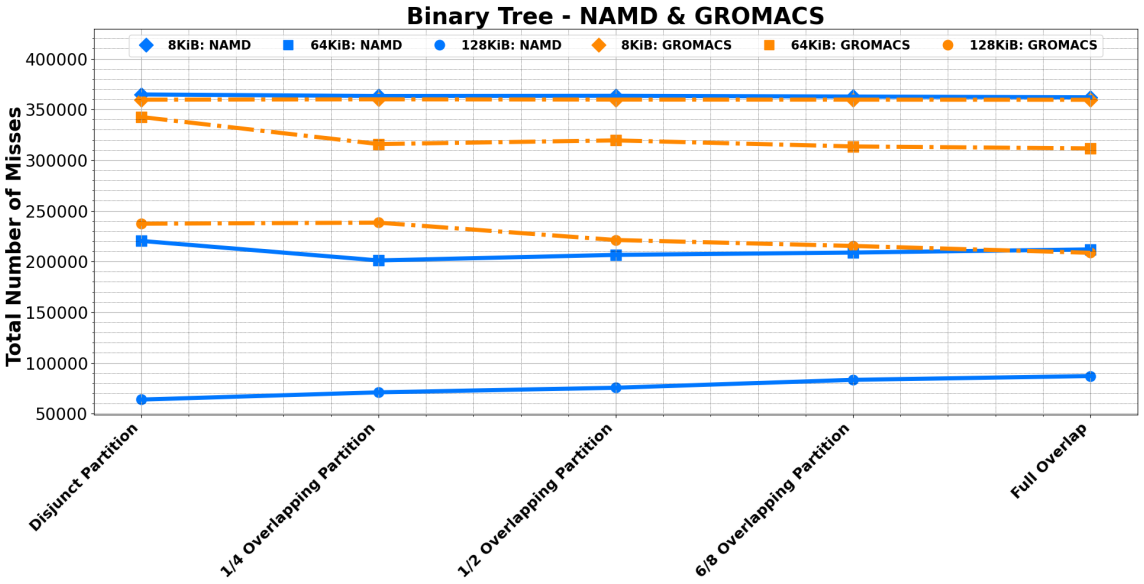

Figure 5.50: *Misses for NAMD and GROMACS running Binary Tree*

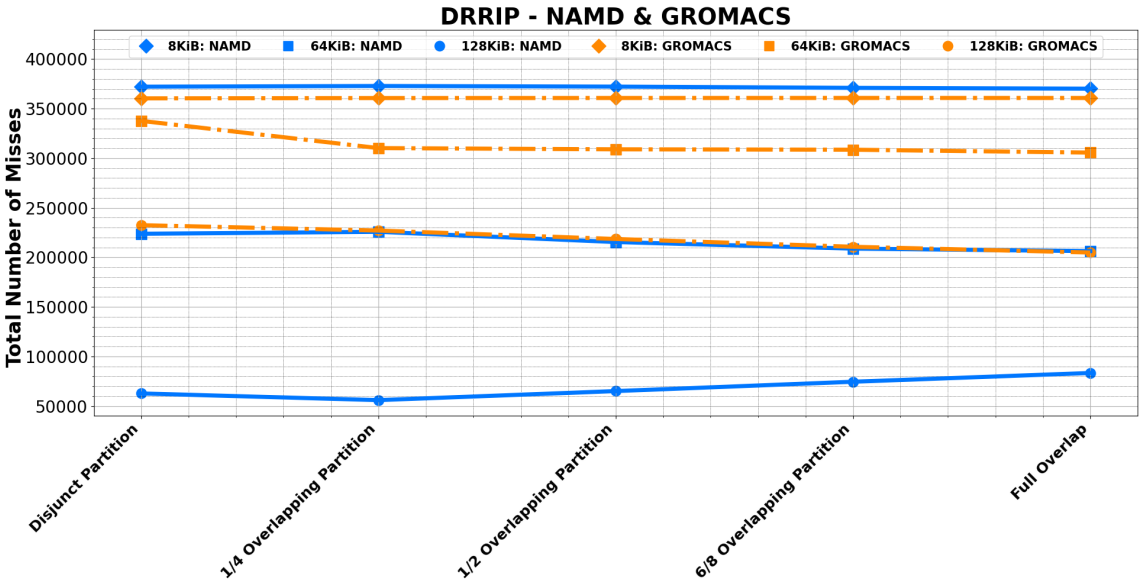

Figure 5.51: *Misses for NAMD and GROMACS running DRRIP*

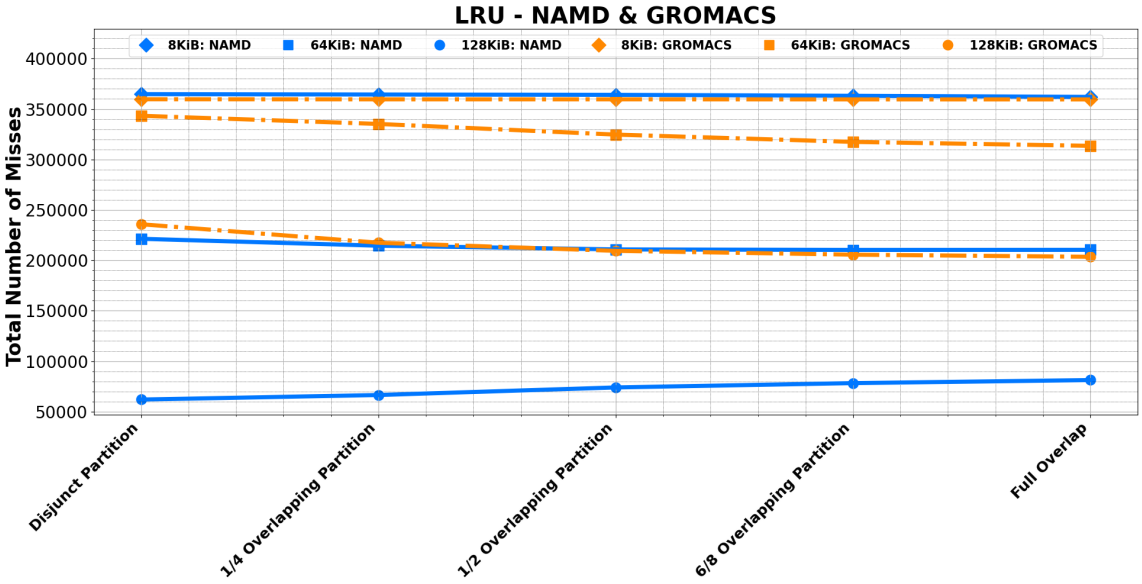

Figure 5.52: *Misses for NAMD and GROMACS running LRU*

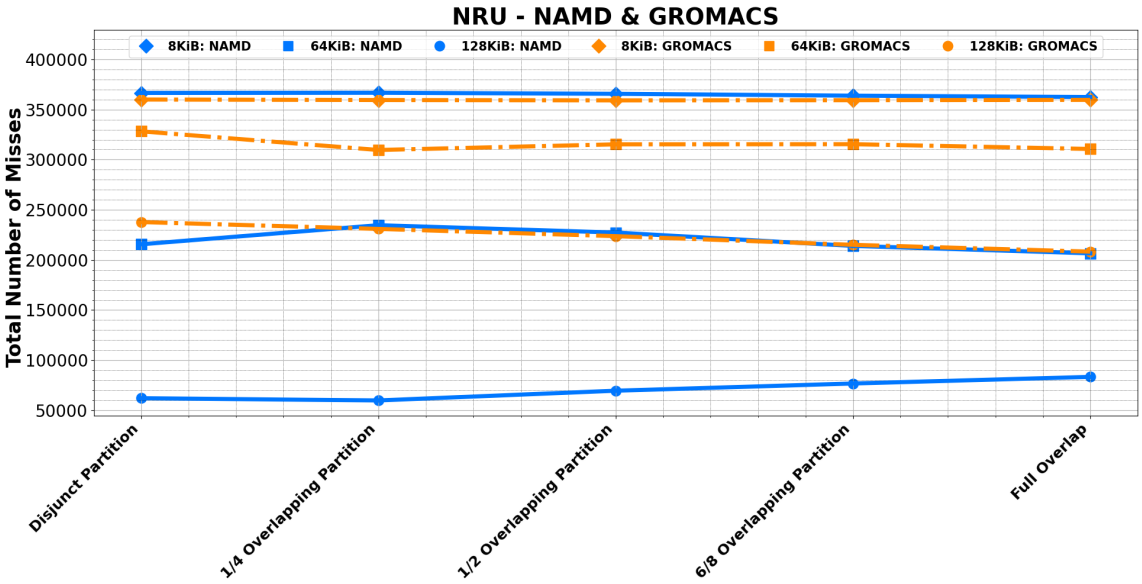

Figure 5.53: *Misses for NAMD and GROMACS running NRU*

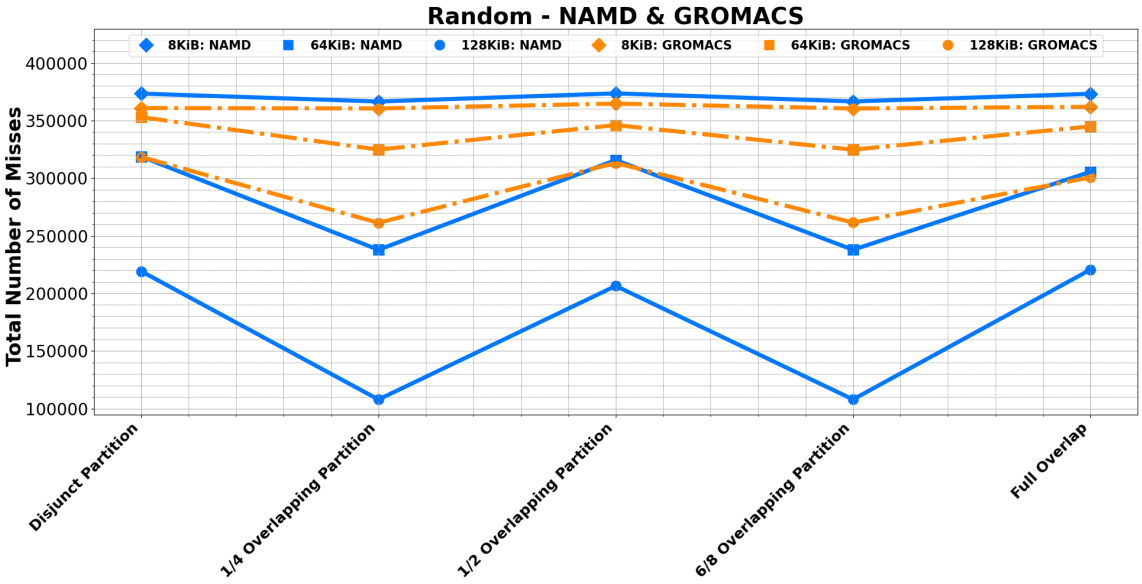

Figure 5.54: *Misses for NAMD and GROMACS running Random*

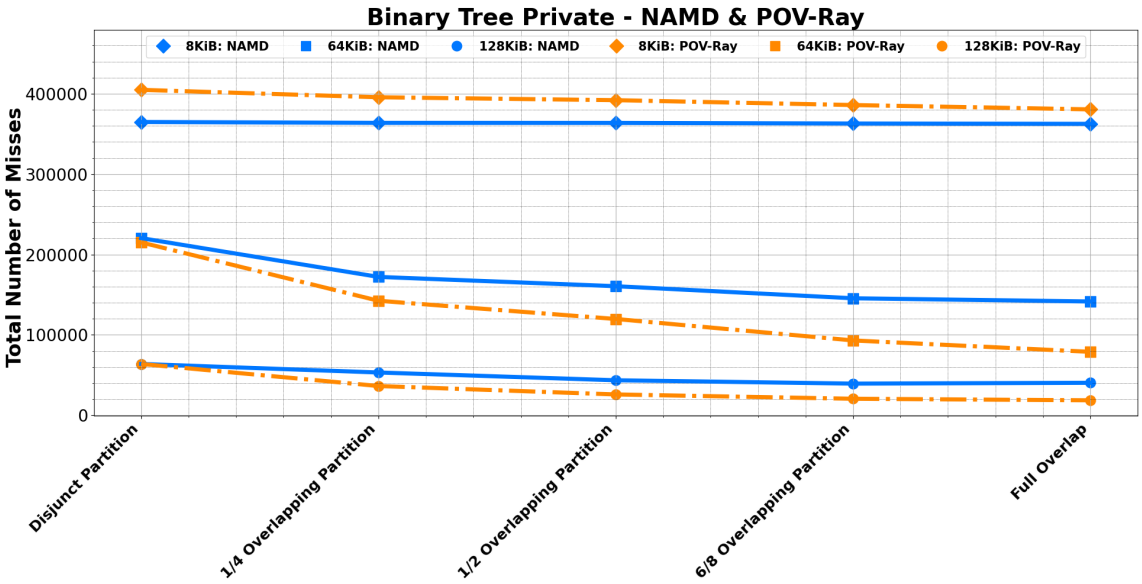

Figure 5.55: *Misses for NAMD and POV-Ray running Binary Tree Private*

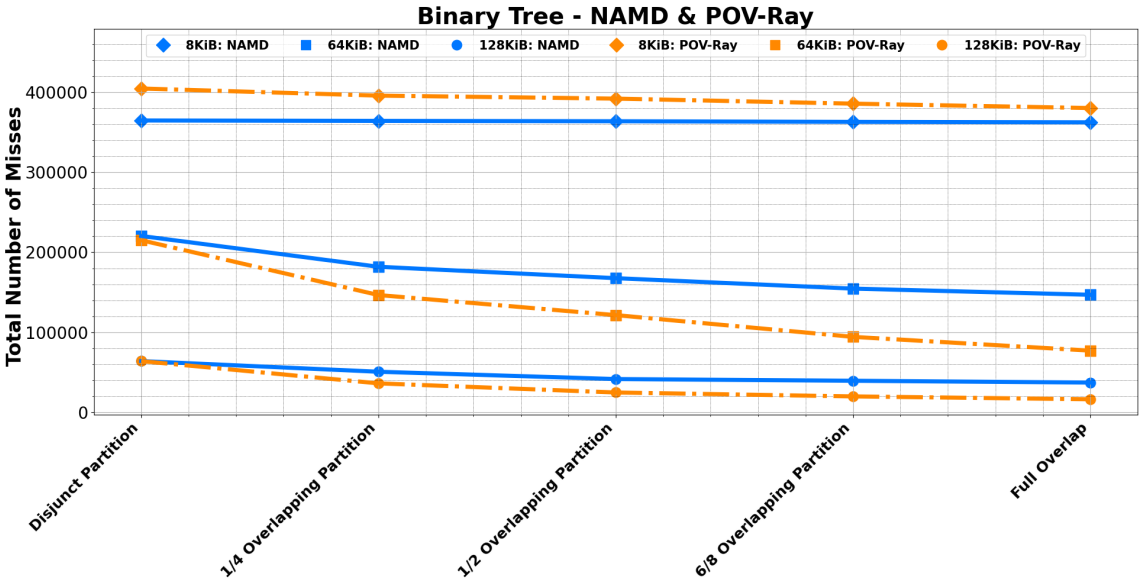

Figure 5.56: *Misses for NAMD and POV-Ray running Binary Tree*

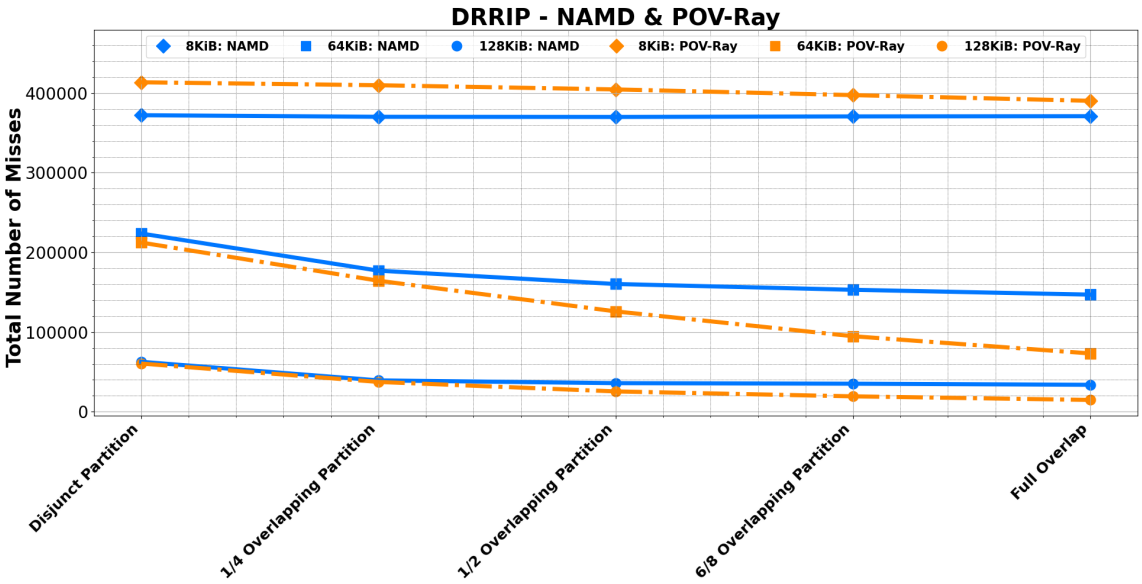

Figure 5.57: *Misses for NAMD and POV-Ray running DRRIP*

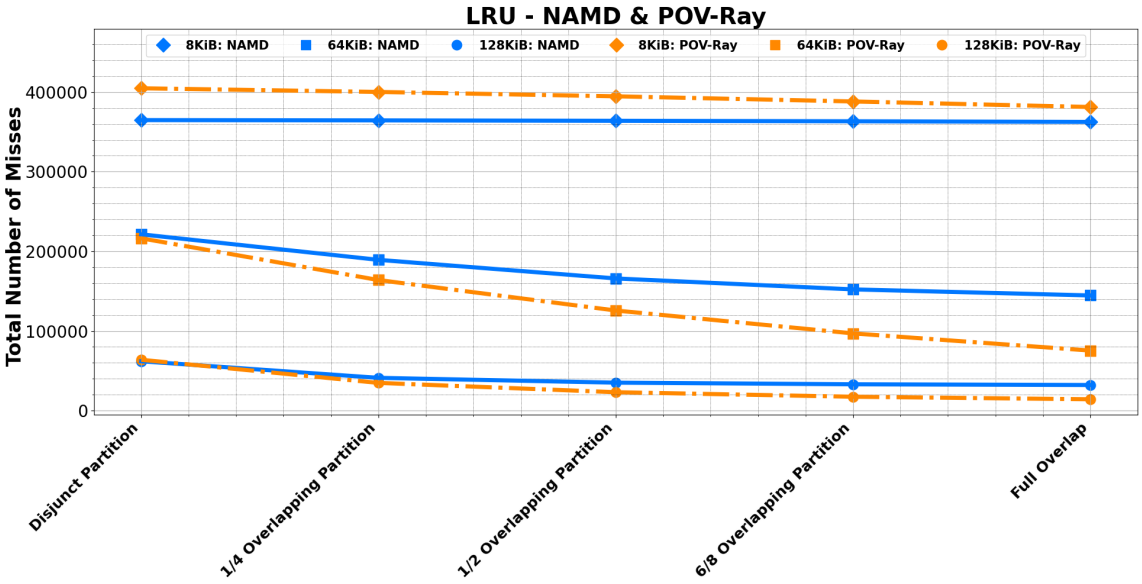

Figure 5.58: *Misses for NAMD and POV-Ray running LRU*

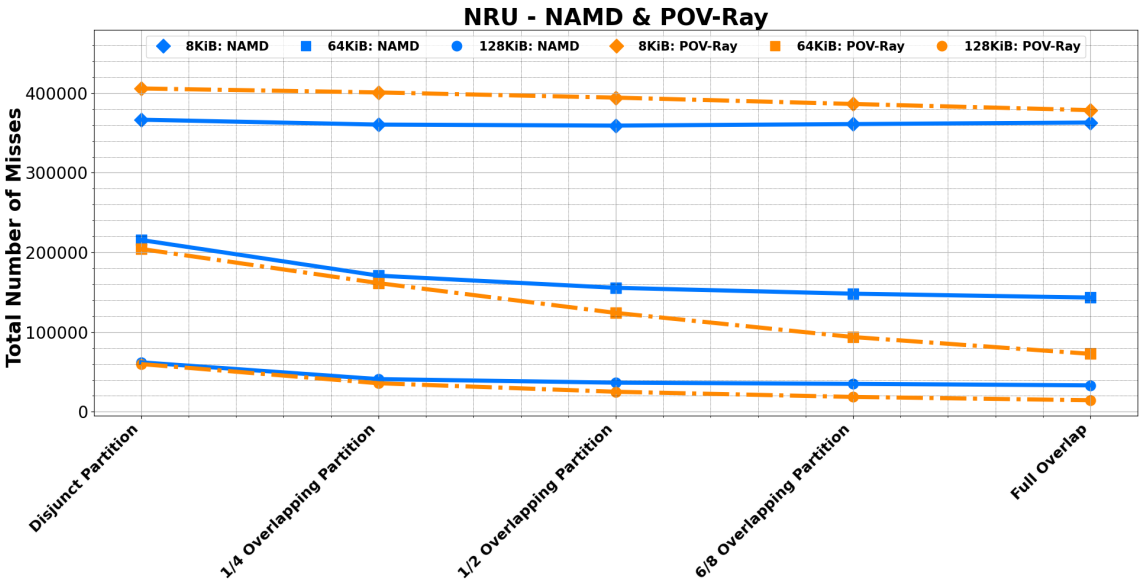

Figure 5.59: *Misses for NAMD and POV-Ray running NRU*

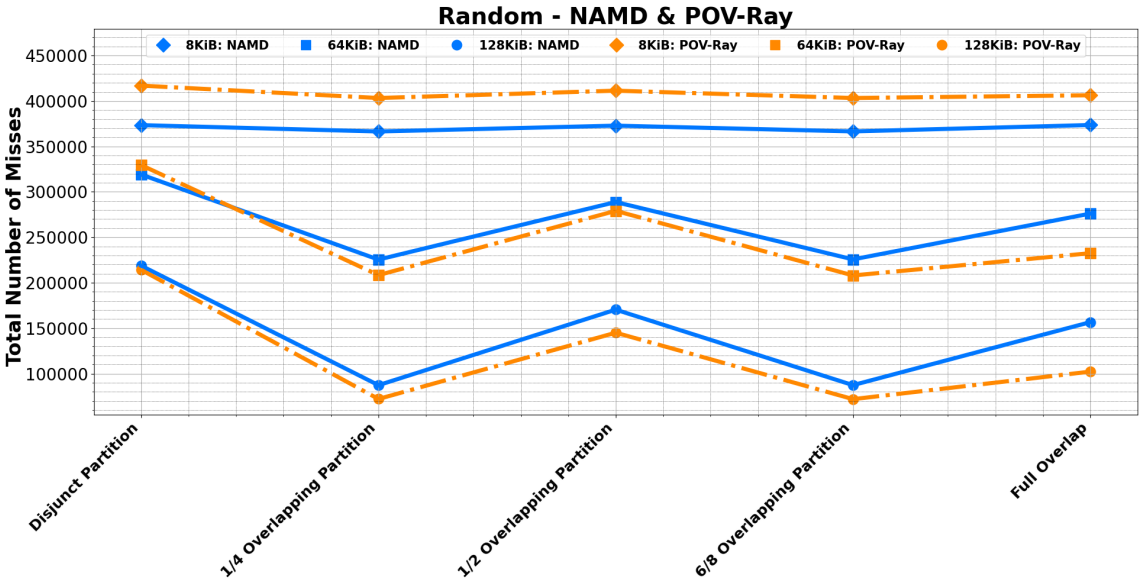

Figure 5.60: *Misses for NAMD and POV-Ray running Random*

8 KiB Cache Size					
	Disjunct	1/4 overlap	1/2 overlap	6/8 overlap	Full overlap
NAMD	364793	363374	363316	362557	362158
GROMACS	359671	360360	360304	360346	360240
64 KiB Cache Size					
NAMD	220366	188316	201824	210248	223912
GROMACS	342570	307018	311220	297124	285535
128 KiB Cache Size					
NAMD	63871	76683	94391	109241	133140
GROMACS	237353	246578	220004	210108	203755

Table 5.3: *The misses of NAMD and GROMACS benchmarks running Binary Tree Private in an 8, 64 and 128 KiB cache.*

8 KiB Cache Size					
	Disjunct	1/4 overlap	1/2 overlap	6/8 overlap	Full overlap
NAMD	364793	363290	363490	362741	362049
GROMACS	359671	360017	359763	359696	359565
64 KiB Cache Size					
NAMD	220366	201112	206574	208867	211994
GROMACS	342570	315900	319521	313548	311604
128 KiB Cache Size					
NAMD	63871	70997	75520	83322	87067
GROMACS	237353	238378	221257	215403	208638

Table 5.4: *The misses of NAMD and GROMACS benchmarks running Binary Tree in an 8, 64 and 128 KiB cache.*

8 KiB Cache Size					
	Disjunct	1/4 overlap	1/2 overlap	6/8 overlap	Full overlap
GAMESS	71858	70686	70104	69581	69042
GAMESS_2	101702	100934	99697	98671	97590
64 KiB Cache Size					
GAMESS	47573	44440	44241	44141	44489
GAMESS_2	76368	70692	62198	56101	50726
128 KiB Cache Size					
GAMESS	24075	24395	24843	26269	29441
GAMESS_2	58021	58220	50314	47488	44912

Table 5.5: *The misses of GAMESS and GAMESS_2 benchmarks running Binary Tree Private in an 8, 64 and 128 KiB cache.*

8 KiB Cache Size					
	Disjunct	1/4 overlap	1/2 overlap	6/8 overlap	Full overlap
GAMESS	71858	70700	70160	69610	69193
GAMESS_2	101702	100934	99697	98662	97590
64 KiB Cache Size					
GAMESS	47573	45140	44572	44294	44043
GAMESS_2	76368	70687	62209	56096	50732
128 KiB Cache Size					
GAMESS	24075	24484	23572	23767	23561
GAMESS_2	58021	58222	50355	47428	44909

Table 5.6: *The misses of GAMESS and GAMESS_2 benchmarks running Binary Tree in an 8, 64 and 128 KiB cache.*

5.2 Latencies of the Replacement Algorithms

The latencies presented in this section are specific to the implementations used in this thesis. We exclude any latency not relevant for finding a victim upon a cache miss or handling updates upon a hit. This means that we do not consider the extra clock cycle for fetching the recency information from RAM. Binary tree and binary tree private will be combined in the same section since their functionality are virtually the same and use the same amount of clock cycles to find a victim. A table with the latencies of the replacement algorithms can be seen in Table 5.7.

Random

Random is the simplest algorithm when it comes to latency. A counter is increased by one each clock cycle. When a victim has to be decided, the counter is sampled and the partition mask is used to enforce that the sampled value is within the partition, and this can be done in one clock cycle. Since this is the only operation done by random, the total number of cycles needed is one.

TrueLRU

For LRU, we fetch all counter values for an entire set from RAM. Then generating a victim way can be done in one clock cycle. Also, performing the update of the counters in case of a cache miss also takes one clock cycle. Both these actions are possible due to the fact that they are implemented in RTL as a combinatorial block, which means that they are not clocked. Since they produce an output as soon as the counter values are available to the replacement algorithm logic from RAM, we can get that output in one clock cycle. This is also true for the remaining algorithms except DRRIP.

NRU

The NRU is a very simple algorithm in nature and does not require any complex operations to generate a victim way. Upon a miss, if our partition is full we set all bits in our partition to zero, otherwise we directly pick the first way in our partition that is set to zero as the victim way. This functionality is fast and easy to implement in hardware and can be done in one clock cycle.

Binary Tree and Binary Tree Private

For both these algorithms the latency needed to find a victim is one clock cycle. Depending on the associativity of the cache, the amount of traversals that has to be made in the tree is always static according to $log_2(associativity)$ and the steps taken in the tree depends on the values of each node. For instance if the associativity is eight, the amount of steps taken in the tree will be $log_2(8) = 3$.

DRRIP

This algorithm is the most dependent when it comes to the number of cycles needed to find a victim. Since the table containing the RRPV values has to be updated if a victim could not be instantly found, the algorithm needs additional clock cycles to update the table and redo the check once more to find the victim. The number of cycles needed by the algorithm are therefore depending on the values in the table when a victim has to be found. Therefore a fixed amount of cycles that the algorithm needs can not be set, but the guaranteed worst case will always be $2 * M$, (Once again, we consider M to equal two) since if the RRPV is zero, we need to check the tables and increment the values by one and redo the checks to find the victim.

Replacement Algorithms	Hit update	Victim generation
Random	N/A	1
TrueLRU	1	1
NRU	1	1
Binary Tree	1	1
DRRIP	1	$2 * M$
Binary Tree Private	1	1

Table 5.7: *The latencies for the hit and victim generation functions of the replacement algorithms.*

5.3 Overheads

We present the various area overheads for the implementations of the replacement algorithm designs.

5.3.1 Resources

The design was synthesized using *Vivado* for the part *xc7k70tfbv676-1* with different cache sizes and the various replacement algorithms to form a table summarizing the required resources (registers and LUTs) to use the algorithms. See Table 5.8 and 5.9 below. We can see that *Random* is constant no matter the cache size. *TrueLRU* seems to increase as the cache size increases and peaks at a cache size of 1028 KiB. The *NRU* and *DRRIP* algorithms increase as the cache size increases. Lastly, *Binary Tree* and *Binary Tree Private* initially gets smaller as the cache design increases but then grows in size.

Replacement \ Cache size (KiB)	4096	1028	256	64	16
Random	45	45	45	45	45
TrueLRU	246	2866	243	202	191
NRU	4269	1136	125	120	91
Binary Tree	3643	955	88	82	87
DRRIP	7663	1994	142	128	125
Binary tree private	26712	6731	101	103	174

Table 5.8: *The slice LUTs utilized by the various replacement algorithms.*

Replacement \ Cache size (KiB)	4096	1028	256	64	16
Random	7	7	7	7	7
TrueLRU	69	91	65	63	61
NRU	45	43	33	31	37
Binary Tree	48	46	37	35	40
DRRIP	76	74	56	51	49
Binary tree private	2105	647	269	235	257

Table 5.9: *The slice registers utilized by the various replacement algorithms.*

5.4 Comparing Binary Tree Private vs. Binary Tree

During this thesis, the question was raised about interference when two or more CLOSes have overlapping partitions, specifically for the binary tree replacement algorithm. To counter this, a modified version of binary tree was conceived where instead of all CLOSes sharing a binary tree for each set in the cache, every CLOS has its own binary tree for every set in the cache. The idea behind this was that the recency information for each CLOS will not be overwritten by another CLOS, and that each CLOS will have a better hit-rate because no recency information will be lost during a context switch.

When comparing Figure 5.43 with 5.44 (or Tables 5.5 to 5.6) and Figure 5.55 with 5.56 to 5.4, it is clear that the number of misses for *binary tree private* saw no significant change in performance when compared to *binary tree*. Although when comparing Figure 5.49 with Figure 5.50 (or Tables 5.3 to 5.4) we can see that *binary tree private* suffers more from interference when the two applications share ways between them compared to the *binary tree* replacement algorithm. The most likely reason for this is that even though every CLOS has its own recency information, the data that it keeps track of has been evicted by another CLOS during a context switch, so the information that the CLOS has kept track of has been evicted.

Another reason might be because the *binary tree private* penalizes the other running CLOS since each CLOS evicts only based on its own recency information. For this reason it might result in more misses when compared to using the more fair *binary tree* algorithm. Since *binary tree private* works the same as *binary tree* when there is no overlap between CLOSes, the performance between the two algorithms are exactly the same. Taking this into consideration and when you also consider the extra amount of overhead (both logical and physical) the modified binary tree was an interesting idea but in practice, it is not a reasonable replacement algorithm to implement and use.

5.5 Other Beneficiary Aspects of Partitioning

Apart from the performance aspects of cache partitioning which this thesis has mainly focused on, there are other benefits to using cache partitioning. Some of these benefits will be discussed here.

When no partitioning of the cache is implemented, it is effectively up to the replacement algorithm to decide how much of the cache each application has access to at any given time. This in turn means that we can not guarantee fairness between multiple applications. By using partitioning, we can take control from the replacement algorithm and give it to the user instead. This gives us the ability to guarantee the fairness of each application, since the user can set the portion of the cache that each application has access to without compromising the functionality of the replacement algorithm.

There is also the aspect of increased security that cache partitioning introduces to a computer system. If we isolate applications by giving disjunct partitions of the cache, we can mitigate timing based attacks on applications since no application can interfere with another application's cache ways. This in turn will greatly reduce the risk of timing-based attacks on applications, for example the Meltdown vulnerability [21].

When an application executes with a disjunct partition, there is no ambiguity of how another running application might affect its hit-rate. This means we can better predict the performance of the application, which is an important aspect for real-time applications where the knowing the execution time is crucial to the real-time system.

6

Conclusion

In this chapter we conclude the results that the thesis has generated and discuss future directions.

6.1 Conclusion

Our goal in this thesis was to implement a cache design with support for partitioning and evaluate the trade-offs between the performance gained and the extra resources required. We extracted memory traces from the representative regions of some of the SPEC-CPU-2006 benchmarks and simulated them in our cache design. The number of misses were collected for different scenarios: a single application with- and without partitioning, multiple applications with overlapping partitions and multiple applications with different ratios of disjunct partitioning. We also test the performance of a modified version of the *Binary Tree* algorithm called *Binary Tree Private*.

The results for having non-overlapping partitions (see section 5.1.4) indicate that some applications see more benefit from having more cache ways than other applications. In the same way, some applications do not get as penalized as others for losing cache ways. This means that if you specifically want one application to have more performance, than it might be an option to partition the cache accordingly. Overall the best performance seems to be had from partitioning 50/50 for the workload scenarios that we investigated.

The results from having overlapping partitions (see section 5.1.5) tells us that for some applications the benefit of reducing interference might not outweigh the drawback of removing ways from the application. For some pairs of applications it might yield less misses (for example Figure 5.56), but in general it looks to be roughly unchanged from having no partitioning.

Regarding the *Binary Tree Private* algorithm, it does not appear to yield less misses compared to the normal *Binary Tree* algorithm. It might be due to the fact that, even though each CLOS keeps track of its own recency information, the data itself might have been evicted by the other CLOS. This would then not yield any less misses than the normal binary tree algorithm. Also, it might be due to the fact that when each CLOS evicts a cache line, it only considers its own recency data. This should penalize the other CLOS which, in turn, could increase the overall miss rate.

In the case where there is no overlapping partitions, both algorithms work exactly the same and there would be no point of using the *Binary Tree Private* algorithm.

In conclusion, partitioning does not appear to yield any significant performance gain in terms of reducing cache misses for the test cases in this thesis at least. One reason for this could potentially be because we only paired applications that had similar histograms. Also not all applications in the SPEC 2006 benchmark suite were used. However, there are other aspects that might be of more importance than performance when considering different use cases. For example security, QoS, determinism and fairness. See section 5.5 for more details.

6.2 Future Works

During the thesis we delved deep into how we could possibly generate full traces of benchmark applications that also had all memory accesses filtered through a L1 cache to better simulate the accesses made to a LLC that we were out to test. In the future, the use of more benchmark suites like SPEC2017-benchmarks. This could potentially give a more thorough picture of how the algorithms perform in an actual computer system. Further testing could include increasing the workload, for example running four applications in multi-application context instead of two. Also testing more configurations of the cache, like increasing the associativity, could also yield more interesting results.

Another aspect to be improved upon in the future would be to improve the randomness of the random replacement algorithm. As mentioned in section 3.4.1 the implementation of the random replacement policy is flawed in that it might generate the same victim way, or the same series of victim ways, if many misses occur at once or if the interval between misses happens at constant intervals. This is due to the fact that the replacement algorithm uses the system clock to generate a victim way and since it always takes the same number of cycles to handle a cache miss the random replacement algorithm will typically generate the same victim ways upon each eviction. For this reason, improving the randomness of this replacement algorithm would be needed.

One aspect of improvement is to implement the partition-supported replacement algorithms into an existing fully-designed open-source cache pipeline, instead of the simple cache design used in this thesis. As of now, the cache design contains a few FIFOs between some of the modules. The idea was to have FIFO modules to make it easier to transfer data from one module of the design to the next, as well as potentially balancing out the pipeline so it can be clocked faster. However it might've been better to make the design without FIFOs to start with and then see if it is necessary to include them in the end as the latency increases with every added extra stage to the pipeline. Also, some of the modules in the design (e.g. the directory- and data interfaces) can likely be omitted and the functionality they have can be implemented in the data memory and directory themselves. This could potentially save some cycles.

On the topic of pipelines, the design right now only supports one memory request

at one time. It would be interesting to explore how partitioning would be affected if multiple different requests populate the pipeline at one time. Furthermore, it would be interesting to see if it is possible to assign different replacement algorithms depending on the CLOS that is making the request. Effectively running multiple replacement algorithms for different CLOSes. This would likely result in more overhead but could potentially yield a better miss rate if the replacement algorithm is tailored to the type of work that a CLOS is performing.

Bibliography

[1] A. Herdrich, E. Verplanke, P. Autee, *et al.*, "Cache qos: From concept to reality in the intelő xeonő processor e5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 657–668. DOI: 10.1109/HPCA.2016.7446102.

[2] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, May 2017, ISSN: 0360-0300. DOI: 10.1145/3062394. [Online]. Available: https://doi.org/10.1145/3062394.

[3] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: Association for Computing Machinery, 2010, pp. 60–71, ISBN: 9781450300537. DOI: 10.1145/1815961.1815971. [Online]. Available: https://doi.org/10.1145/1815961.1815971.

[4] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006, pp. 423–432. DOI: 10.1109/MICRO.2006.49.

[5] S. Mittal, Y. Cao, and Z. Zhang, "Master: A multicore cache energy-saving technique using dynamic cache reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 8, pp. 1653–1665, 2014. DOI: 10.1109/TVLSI.2013.2278289.

[6] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 57–68.

[7] ARM, *Arm memory system resource partitioning and monitoring (mpam) system component specification*, Apr. 2024. [Online]. Available: https://developer.arm.com/documentation/ihi0099/latest/.

[8] snipersim, *The sniper multi-core simulator*, 2023. [Online]. Available: https://snipersim.org/w/The_Sniper_Multi-Core_Simulator.

[9] M. Dubois, M. Annavaram, and P. Stenström, "Parallel computer organization and design," in Cambridge university press, 2012, ch. 4, pp. 198–203.

[10] P. Panda, G. Patil, and B. Raveendran, "A survey on replacement strategies in cache memory for embedded systems," in *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, 2016, pp. 12–17. DOI: 10.1109/DISCOVER.2016.7806218.

[11] Q. Javaid, A. Zafar, M. Awais, and M. A. Shah, "Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques," *Mehran University Research Journal of Engineering and Technology*, vol. 36, no. 4, pp. 831–840, Oct. 2017. [Online]. Available: https://hal.science/hal-01700364.

[12] K. Kdzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-lru replacement policies," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470352.

[13] A. Jain and C. Lin, *Cache Replacement Policies*. Springer Cham, 2019.

[14] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer, "Adaptive insertion policies for managing shared caches," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 208–219.

[15] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 356–367. DOI: 10.1109/MICRO.2014.46.

[16] D. Chiou *et al.*, "Extending the reach of microprocessors: Column and curious caching," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[17] G. Hamerly, E. Perelman, J. Lau, and B. Calder, *Simpoint 3.0: Faster and more flexible program phase analysis*, https://cseweb.ucsd.edu/~calder/papers/JILP-05-SimPoint3.pdf, [Accessed 15-08-2024], Sep. 2005.

[18] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel ő itanium ő programs with dynamic instrumentation," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 81–92. DOI: 10.1109/MICRO.2004.28.

[19] J. E. Miller, H. Kasture, G. Kurian, *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416635.

[20] AMD, *Amd vivado - amd vivado design suite*, 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[21] N. Abu-Ghazaleh, D. Ponomarev, and D. Evtyushkin, *How the spectre and meltdown hacks really worked - ieee spectrum*, en, Feb. 2019. [Online]. Available: https://spectrum.ieee.org/how-the-spectre-and-meltdown-hacks-really-worked.