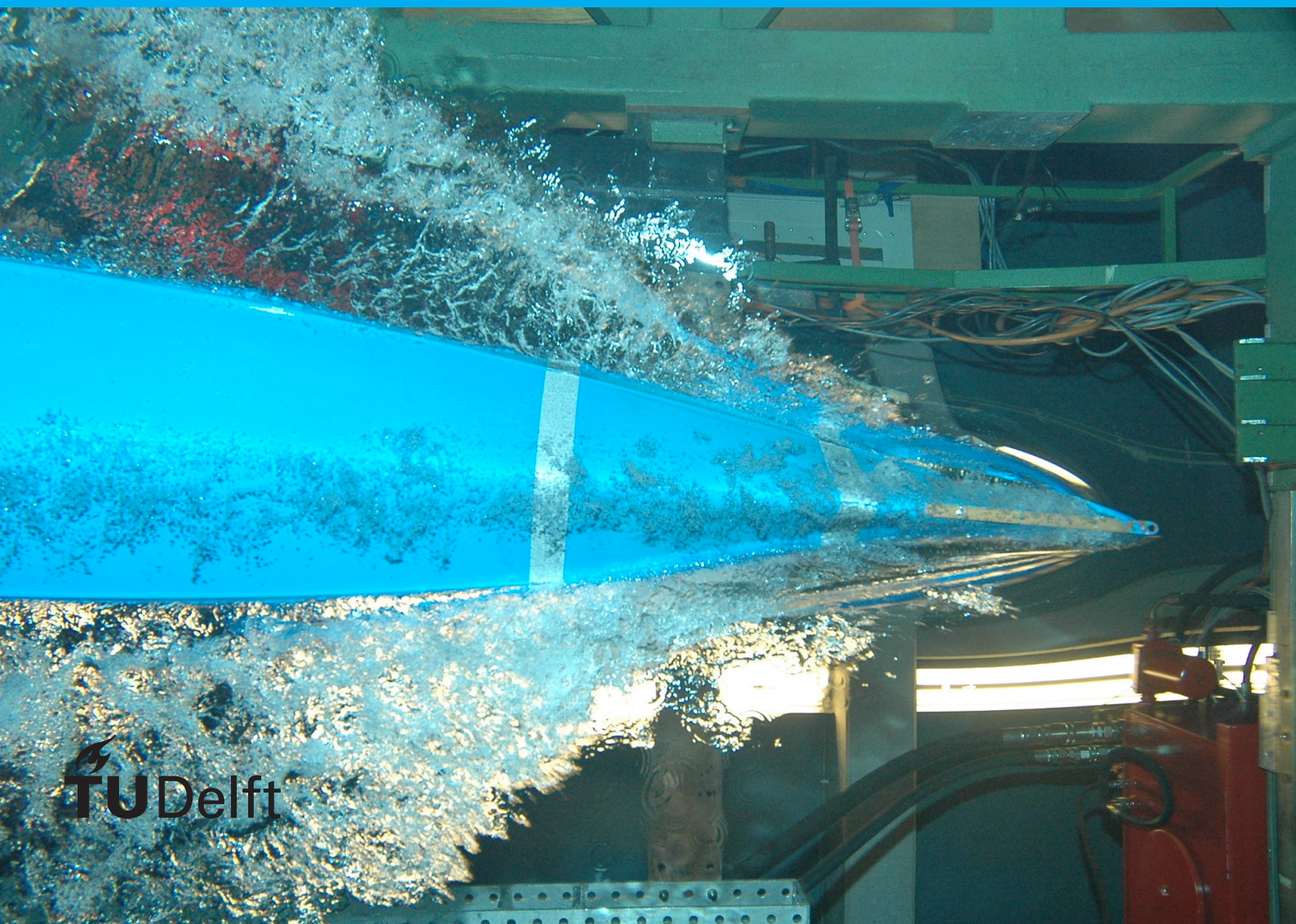
Dynamic cache configuration for the ρ-VEX platform

S.W. Vermaat

Computing Engineering department
Delft University of Technology

Q&CE-CE-MS-2021-02



Dynamic cache configuration for the ρ-VEX platform

by

S.W. Vermaat

to obtain the degree of Master of Science
for Computer Engineering
at the Delft University of Technology,
to be defended publicly on Wednesday June 16, 2021 at 14:00.

Student number: 1360906
Project duration: September 12, 2016 – June 16, 2021
Thesis committee: Dr.ir. J.S.S.M. Wong, TU Delft, supervisor
Dr.ir. T.G.R.M. van Leuken, TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The ρ-VEX is a processor designed at the Computer Engineering lab at TU Delft to be reconfigurable at runtime, resulting in a processor that can combine or separate instruction lanes according to the program requirements. The current cache for the ρ-VEX processor is direct mapped and always identical to the instruction group configuration. This is limited and not flexible, and a more flexible cache that can be reconfigured at runtime is desirable. This thesis introduces a more flexible cache, which is achieved by replacing the replacement policy with a more flexible variant, as well as adding an extra cache tree. The addition of the second cache tree allows for a more flexible cache size assignment, as either cache blocks of the small or the larger cache tree can be assigned to a specific instruction group. The assignments of the cache blocks can be reconfigured during runtime. The replacement policy is replaced by round robin and (pseudo) LRU, giving the required flexibility, as well as decreasing cache misses, which results in better overall performance for the ρ-VEX. Round Robin reduces the runtime when the application heavily uses the caches by 11.7%, but increases the runtime when the application has a low cache utilization. LRU always reduces the application runtime, and reduces the run times of cache heavy applications by about 13%.

Contents

1 Introduction

1.1. Context

Processors are utilized in almost all aspects of life controlling various and widely different embedded systems, e.g., from refrigerators to cars. Consequently, different applications pose different requirements for the design of processors or determine the choice for off-the-shelf processors. Requirements include, but are not limited to: performance, power consumption, and costs.

However, processors have a limited flexibility. A processor can have more computation power at the cost of a higher power consumption. Therefore, processors are selected such that the computation power is sufficient or slightly more than is required for the application. When applications have varying computational demands, choosing the right processor becomes difficult as either the timing constraints are not met or power efficiency is reduced. One solution of this problem is introducing a flexible processor that can change its computational power and power consumption according to program demands.

The ρ-VEX is a processor designed at the Computer Engineering lab at TU Delft to be reconfigurable at runtime, resulting in a processor that can combine or separate instruction lanes according to the program requirements. With this processor each application can be assigned one or more instruction lanes, giving each application the required processing power. If the application changes and requires less computational power, one or more instruction lanes can be assigned to other applications or turned off entirely for power saving.

As most processors, the ρ-VEX has a cache that holds frequently used main memory locations that can be accessed faster than the main memory. To limit the amount of cache access ports, each cacheblock should be assigned to only one application at a time. Therefore, the cache structure must be dynamic as the number of applications can change at runtime.

The unique design of the processor however imposes unique requirements of the design of the cache. All the cacheblocks have to be able to work together as well as separately. This not only complicates the connections of all the data lines but also makes it difficult to decide where data needs to be stored.

1.2. Problem statement and methodology

The current implementation of the cache for the ρ-VEX always has the same configuration as the instruction lanes. This ensures all the applications to have at least some cache space, but does not take into account which application requires more data storage. Making the cache capable of allocating more cache space for a specific application regardless of the core configuration would be a more efficient approach, as applications that do not need much data storage can give excess cache space to applications that need it. Furthermore, the cache blocks are always used as direct mapped, which results in frequently accessed memory being occasionally replaced. Therefore, the research question of this thesis is:

How to design a run-time adaptable cache organization for a dynamic multi-processor system?

To answer this question, the cache must first have a replacement policy that supports any kind of cache configuration, as direct mapped is only efficient on a power of two number of cache blocks. Furthermore, the unique cache interconnectivity of the ρ-VEX must be expanded to support configurations that do not have to be the same as the processor configuration. Regardless of the configuration of the cache, all the stored data must always be stored and returned without any error. The research question can be divided into these requirements:

1. The cache of the ρ-VEX must implement a better replacement policy that can support all configurations, as the current replacement policy only supports a power of 2 number of cache blocks.
2. The cache of the ρ-VEX must be able to dynamically reconfigure the cache configuration to allocate more cacheblocks for specified contexts as the current implementation only supports a cache configuration identical to the application configuration
3. The cache of the ρ-VEX must return the correct values to the ρ-VEX at all times during and after reconfigurations to maintain the integrity of the applications.

The following tasks will be performed to fulfill these requirements:

1. Investigate which replacement policies can be implemented on a run-time adaptable cache, and determine which performs best
2. Implement or expand a cache interconnectivity to support run-time reconfigurability
3. Verify the functionality of the cache by using conformance tests while changing the cache configuration

1.3. Overview

The outline of this thesis is as follows: In Chapter 2, the background and related work is presented. Chapter 3 explains the possible design choices and the architectural decisions. In Chapter 4, the implementation details are presented. Chapter 5 presents the testing methods, results and the discussion of these results. Chapter 6 summarizes the thesis, and the contributions and future work are presented.

2

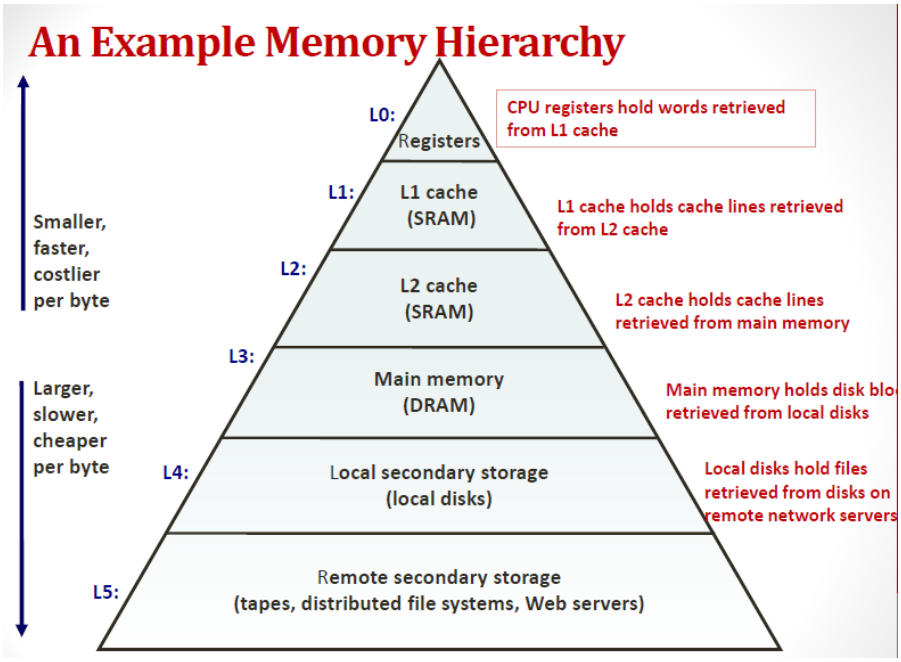
Background

In this chapter, the topics related to caches are discussed. Additionally, general power saving methods and specific for caches are explained, the general properties of Field Programmable Gate Arrays (FPGAs) are presented and the general structure of the ρ-VEX is discussed.

2.1. Caches

Every application can only execute if all the required data is fetched. However, when the application runs, new instructions and data need to be fetched, and has to stop if the data is not supplied quickly enough. At first the program is loaded from the storage device to the main memory. However, the speed of the main memory is significantly slower than the CPU and is unable to provide the necessary data quickly enough.

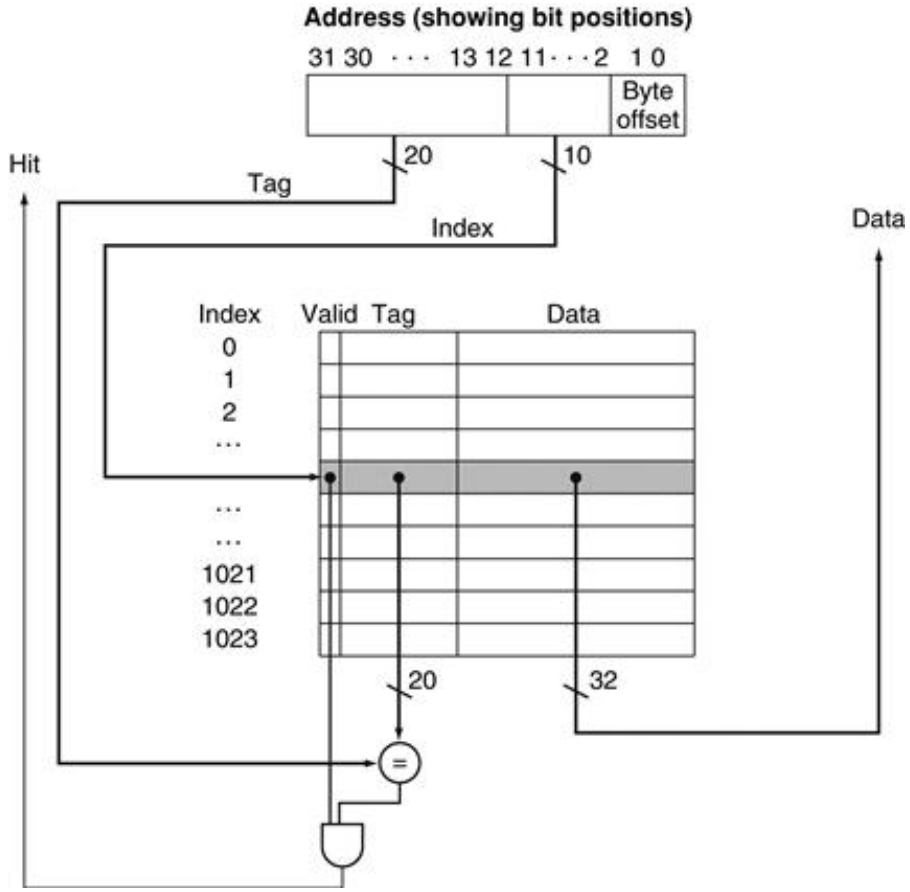
Figure 2.1: The memory hierarchy [1]



Therefore, CPUs have caches, which are memories that store the most frequently accessed memory locations close to the CPUs. They can be accessed quicker than the main memory, thereby reducing the memory latency compared to retrieving all the data from the main memory. The relation between the different types of memory is depicted in Figure 2.1. A cache usually consists of the stored

data, the tags to check if the correct data is in the cache, and additional bits that store information depending on the cache implementation. The additional information can be for instance if the current data is modified or shared, or information regarding replacement policies which will be explained in Section 2.3.

Figure 2.2: Dissection of the requested data access. In this figure, the number of sets is 1024, which requires $\log_2(1024) = 10$ bits.[6]



When data is requested, the cache checks if the data is stored in the cache as displayed in Figure 2.2. The last two bits can be used to only read a specific byte as the standard data width of processors is 32 bits. The next bits determine in which set the data could be stored. Using sets reduces the amount of places the data can be placed in the cache, and reduces checking the amount of locations. The remaining bits are called a tag, and when the requested and stored tags are identical the requested data is located. This also requires the valid bit in the cache to be set to true, as otherwise the stored data is invalid and cannot be used. When the tags are different or the valid to be false, other data is stored in that set and the cache needs to retrieve the data from a higher memory level. When the new data is fetched, it is also placed in the cache for quicker future accesses.

A cache can be differentiated in different ways: the amount of data it can store, how long retrieving data takes and which data is overwritten first. These ways impact the performance but also the other characteristics. For example, a bigger cache will need more time to access the stored data, therefore needing a longer time to retrieve the data. How these differences impacts the performance of the cache will be explained in the following Sections.

2.1.1. Cache sizes and resizing

Most modern CPUs use several levels of caches, where the smaller caches (level 1 cache) are very small but have a hit time[1] within a few cycles, and higher level caches (level 2 or 3 caches) take more cycles (10 to 100) to access. The lowest cache level is first checked when data is requested. In case the data is not found, the next higher level is searched, and in case of a miss, it is requested from the main memory. As a cache is storing frequently used data, there is a higher probability that the data stored in the cache is accessed again. This probability combined with a shorter access time will greatly reduce the memory access time. The cache must store enough data to decrease the overall memory accesses, since the cache must be fully searched first before accessing the main memory. This can be summed up in the following formula[5]:

$$\begin{aligned} &Average\ memory\text{-}access\ time = \\ &\quad Hit\ time * Hit\ rate + (Miss\ time * Miss\ rate) * Miss\_penalty \end{aligned} \tag{2.1}$$

The right size has to be implemented for the cache to improve the effectiveness of the cache, which depends on the data requirements of the applications running on the cores. The number of programs and data requirements of the programs can vary greatly, requiring an ideal cache to resize when necessary. Reducing the powered cache area has a large potential for energy saving. Because of this there are several studies done on this area, as described in Section 2.2.

In most caches parts can be turned off, but cannot be reassigned to other cores. In this thesis, a new cache structure will be introduced which can adjust and transfer parts of the cache between cores.

2.1.2. Cache associativity

The effectiveness of the cache size depends on where the new data is allowed to be placed within the cache, as replaced data needs to be refetched when needed. There are three different possible implementations:

- Direct-mapped
- Fully associative
- N-way cache

In a direct-mapped cache, all the memory locations can only be placed in one location in the cache. This method can create conflicts where multiple memory locations need to be placed in the same place. This method is depicted in Figure 2.2. The positive effect is that only one tag has to be compared per data access, making this method the fastest. In a fully-associative cache a certain memory location can be placed on any location within the cache, which prevents data being replaced while there are unused cache locations. A fully-associative cache has therefore a lower chance of expelling useful data, with the downside that every time data is requested the entire cache must be searched to ensure the data is not in the cache. In an N-way cache a certain memory location can be stored in multiple but limited locations, where N is the number of locations that a certain memory location can be stored in. A four way cache is displayed in Figure 2.3 .

Doubling the associativity effectively reduces the miss rate to an equal cache that is doubled in size when the associativity is low[5]. However, increasing the associativity means more tags need to be checked every access. This will increase both the power consumption and the hit time, which becomes unpractical for most memories with an associativity of 16 or higher.

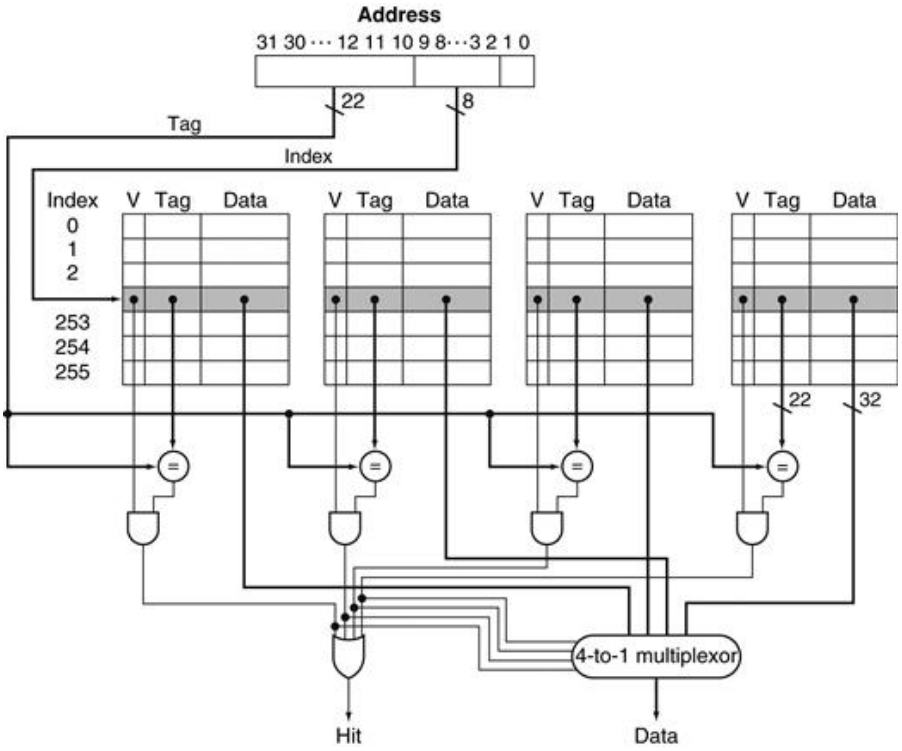
Normally, caches have a fixed number of sets or ways assigned per core. When parts of the cache are unused, one or multiple sets or ways can be deactivated as described in Section 2.2.3. However, all of these sets or ways are always assigned to the same core. This thesis proposes a new structure in which different ways can be assigned to different cores through hardware reconfiguration.

2.1.3. Write policy

There are two general methods of handling writes to the cache: write-through and write-back. In a write-through cache, every write is always forwarded to the higher memory levels, making sure the written

[1] The hit time is the time needed to retrieve data that is stored in the cache, while the miss time is the time required to retrieve the data when the data is not in the cache

Figure 2.3: Dissection of the requested data access for a 4-way cache. In this figure, the number of sets is four times smaller compared to Figure 2.2 as each set is four times bigger.[6].



data is updated everywhere. Write-through makes guaranteeing coherency easier as all the data copies are instantly updated. However, many writes to a local variable causes many bus accesses while the written data is only reused locally. Write-back caches only update the local data copy and changes the status bit signifying the data is modified. Only updating the local can save a significant amount of bus accesses, but requires extra mechanisms to keep the cache coherent which will be explained in Section 2.1.4.

2.1.4. Cache coherency

When multiple cores are accessing and writing the same data, it is very important that the most recent version of the data is used. Guaranteeing the correct data is used requires a special mechanism called cache coherency, which is most often done in two ways. One cache coherency mechanism is bus snooping, in which all the caches listen to the writes on the bus and updates its local value if the same location is detected. Bus snooping requires all the writes being put on the bus, making it trivial for a write-through cache but requiring multiple mechanisms in case of a write-back. A different cache coherency mechanism is directory based, meaning all the data is stored locally and only the status of every data location is stored globally. In case a core requires data that is stored on a different cache block, the global controller will request the data and forward it to the requesting core. This method significantly reduces bus accesses as locally used data is never broadcasted, but has a higher complexity as it requires multiple controllers for handling the data transfers between cores.

2.1.5. Conclusion

In this section, the general characteristics and working of a cache are presented. These essentials are needed to understand the current state of the cache of the ρ-VEX (which will be explained in Section 2.5) and how this thesis will improve it.

2.2. Power saving methods for memory

The memory of the processor takes a large portion of the die size. This means that when the entire cache of the processor is active, the memory has a high power consumption, making the memory the main target for saving power. In the following sections, the research for lowering the power consumption is presented.

2.2.1. Transistor level

A significant way of reducing the power for the cache is disabling the parts of the cache that are not used at this moment, requiring a way on the transistor level to disable data bits. One way of achieving this is using gated Vdd[4], which lowers the static power consumption of a transistor at the cost of not being able to store data, effectively setting the area of the cache that is not storing data to stand by mode. However, other methods need to be used to determine which part of the cache are needed, which will be explained in the next Sections.

2.2.2. Disabling individual data lines

One way of lowering the power is by disabling individual data lines. Since data is very unlikely to be referred to again after a certain interval[8] individual data lines can be turned off when a timer expires for that data line. This will lower the power usage while only discarding data that is likely to not be referenced again. This method can be expanded by discarding the data but keeping the tags[15], which can determine if an evicted data line would still be in the case without this policy. In the case evicted data is requested often, the cache can be signalled that a larger cache would generate less misses.

2.2.3. Disabling cache blocks

Another way of lowering the power is by determining if it is beneficial to disable one or more cacheblocks by either increasing or decreasing the cache size by increasing/decreasing the address space, or enabling/disabling ways in an associative cache. The former is called selective sets, the latter is called selective ways.

When using selective sets[13], the cache is doubled or halved in size per step. When doubling the cache, one more bit of the address is used to determine in which cache block the data will be placed, while halving the cache one extra bit is ignored. The tags must always be unique even after doubling of halving the cache, requiring always storing the maximal size the cache can have. A drawback of this method is that half of the useful data is disregarded when halving the cache size. Additionally, the cache size can only be doubled or halved, which only provide coarse-grained steps.

The cache size can also be changed by using selective ways[2], which adds or removes ways, increasing or decreasing the associativity. Since the number of sets does not change, the address and tag sizes are fixed. However, as mentioned in Section 2.1.2, an associativity of 16 or higher is unpractical, limiting how much bigger the cache can be increased using this method.

To achieve a bigger degree of resizing the cache, it is possible to combine both selective sets and selective ways[14]. By using selective sets the cache can be increased quickly, while selective ways can resize the cache with smaller steps. This method however also combines the negative aspects, namely the maximum size tags still need to be stored, and halving the size still disregards half of the useful data.

2.2.4. Dynamic cache structures

A different way to make a cache size more power efficient is by dynamically changing the cache structure. Most caches increases the cache associativity by adding or removing the cache data size as described in the previous Sections. However, certain programs benefit considerably more from a cache with a higher associativity compared to a cache with more sets. For these programs the same cache size would be more effectively used by reducing the number of sets and increasing the associativity. Therefore a cache that can detect these situations and change the cache is designed[10]. When the associativity is doubled, the size of each way is halved. Which associativity is chosen is based on the program statistics gathered during operation. However, this approach can not change the total data size of the cache.

2.2.5. Conclusion

In this section, multiple power saving methods for caches are presented. This allows considering which power saving method is most effective for the new cache structure of the ρ-VEX.

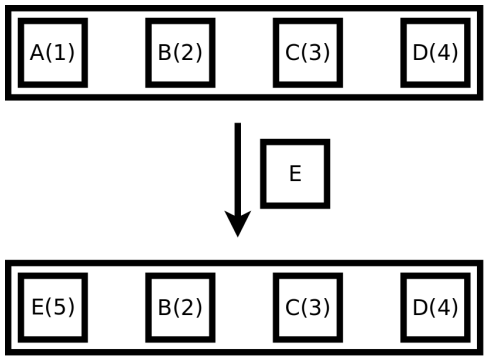
2.3. Replacement policies

In case a cache is not direct mapped, data from a specific memory location can be written to multiple locations. If all of these locations are already occupied, a decision needs to be made which one to replace. This is determined by the replacement policy. An ideal replacement policy replaces the data that will never be used again, whereas a poor one will replace data that will be used again, causing an extra cache miss. Therefore, a correct replacement policy is important for the effectiveness of the cache. There are multiple replacement policies which each have certain benefits and costs.

Random A random policy puts new data in arbitrary places. This requires no extra hardware, but it greatly depends if the best data is replaced or not, resulting in a relatively high miss rate.

FIFO A First In First Out (FIFO) policy keeps track of the order in which the data is placed, and replaces the data that is in the cache for the longest. A FIFO policy can be implemented by simply using a counter to specify which location is next. Programs usually do not use data that is not used for a long time[12]. However, this replacement policy does not take into account which data is used more often. This can result in often-used data being replaced by newer but more rarely used data. Therefore, the miss rate is still relatively high. In Figure 2.4 an example is given.

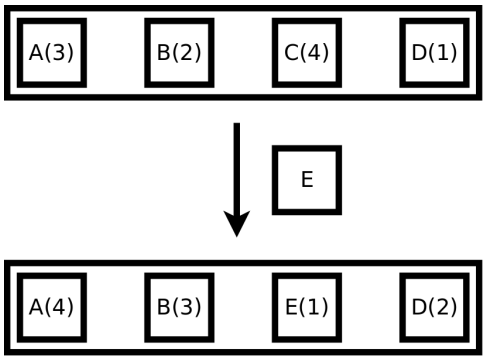
Figure 2.4: FIFO replacement policy example. In this example, data pieces A, B, C and D are stored, each stored with which order the data is placed in the cache (1 is first, 4 is last). When a new data piece E needs to be placed, the first placed item will be replaced, which in this example is A



LRU The Least Recently Used (LRU) policy keeps track of the order in which the data is used. When new data is placed in the cache, the data that is used the least recent will be replaced. This requires keeping track of the order of using all of the data locations that belong to the same group, which is a significant overhead as it requires storing the exact order. With this policy the most often used data usually stays in the cache while the less used data is replaced more often, resulting in a lower miss rate. In Figure 2.5 an example is given for the LRU replacement policy. When a new datablock needs to be placed, the block that was least recently used (the highest value) will be replaced. As this block is now the most recently used, the number of the other blocks are increased accordingly.

Pseudo-LRU A Pseudo-LRU policy also keeps track of the order in which the data is used, but only stores an approximation instead of the exact order. This will not guarantee that the least recently used will be replaced which can raise the miss rate, but requires less hardware overhead. One implementation of pseudo-LRU is only storing one bit per data word, where new data is placed on locations with a zero. Another pseudo-LRU implementation is storing $log(n)$ bits, essentially storing a binary tree that points to the least recently used. This replacement policy behaves the same as in Figure 2.5, with the

Figure 2.5: LRU replacement policy example. In this example, data pieces A, B, C and D are stored, each stored with the order in which the items are accessed (1 is the most recently used, 4 the least recently used). When a new data piece E needs to be placed, the element that is used least recently will be replaced, which in this example is C. The recently used values are adjusted when the new item is placed.

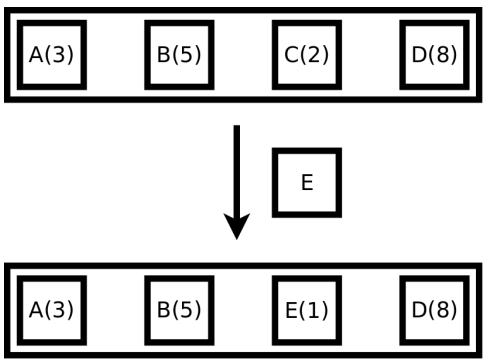


exception that not the exact order is stored and some of the blocks can have the same value. In such a case one of the highest numbers will be replaced.

Segmented LRU A segmented LRU cache is divided into two segments. A new data location is placed in one of the segments which functions as a normal LRU cache. If a data location already present in the cache is accessed, the location is moved to the other segment, which only holds other items that are not only placed but also accessed afterwards. This reduces the number of dead-on-fill blocks [9], keeping more potentially useful data. This would be similar as in Figure 2.5, with the addition of multiple separate blocks that only referenced blocks are placed.

LFU The Least Frequently Used (LFU) policy keeps track how many times all the data are used. When placing new data, the data that is used the least in general will be replaced. This requires keeping track of how often the data is used by using timers, which is also a significant overhead. As with LRU, the most often used data stays in the cache, while less used data is replaced. In Figure 2.6 an example is given for the LFU replacement policy. This policy needs an extra mechanism to expel the data that is used often but not any more (i.e., because a program has finished). Not having this mechanism will clutter the cache with obsolete data.

Figure 2.6: LFU replacement policy example. In this example, data pieces A, B, C and D are stored, each stored with the number of being accessed. When a new data piece E needs to be placed, the element that is used the least will be replaced, which in this example is C.



Decay A Decay policy is a modified LFU policy, where if the timer reaches a predetermined value the corresponding data is expelled immediately. This requires the same amount of hardware as LFU, but can give more information on what portion of the cache is filled with useful data.

Table 2.1: Replacement policies overview

	miss reduction	implementation
random	- -	++
FIFO	-	+
LRU	++	-
Pseudo-LRU	+	+
Segmented LRU	+	+

Conclusion Each of the replacement policies have different effects on hit rate and used area. A comparison of these effects are presented in Table 2.1. These replacement policies will be tested on an FPGA (which will be described in Section 2.4). The LFU and decay replacement policies require counters and timers for each data line, which are not feasible on an FPGA as these require many adders of which are only a limited amount available.

2.4. FPGAs

Field-Programmable Gate Arrays (FPGAs) are chips of which the functionality of the hardware and the interconnectivity can be programmed, allowing programming the hardware behavior of the chip. The clock speeds are lower compared to Application Specific Integrated Chips (ASICs), but there are no fabrication costs, making FPGAs ideal for prototyping purposes or low quantity uses.

FPGAs can reproduce the functions of logic gates by programmable Look Up Tables (LUTs), which when programmed correctly have the same truth table as the original hardware. The LUTs are bundled in Configurable Logic Blocks (CLB) which consists of multiple LUTs, a register and carry logic. These CLBs are then connected through the programmable routing network to reproduce the complete design. The bigger FPGAs have dedicated hardware to greatly improve certain hardware functions such as multipliers and Block RAMs (BRAM).

A hardware implementation on an FPGA is very different from an ASIC. First, on an FPGA the functions are translated into LUTs, whereas in ASICs the implementation can be tuned on a transistor level. Second, the FPGA interconnectivity is limited to the infrastructure, whereas in an ASIC extra routing layers can be added when needed. Because of these differences the area of a design can differ between an FPGA and ASIC implementation. However, a bigger design will be bigger in both implementations. Therefore, the size of a new design will be compared to the original by comparing the number of required LUTs.

As the ρ-VEX (which will be described in the next section) is used for research and not developed to be produced in large numbers, FPGAs are used to test and expand the design. As this thesis is expanding the cache of this project, the implementation is also done on an FPGA.

2.5. ρ-VEX

The ρ-VEX is a runtime reconfigurable processor designed at the Computer Engineering lab of TU Delft. The ρ-VEX is based on the VEX processor. This is a VLIW processor which processes syllables[2] which can be run simultaneously. The ρ-VEX processor is modified to become configurable, adjusting the size and computational power of the processor to the needs of the application. The ρ-VEX can be configured to increase the Instruction level parallelism (ILP) or thread-level parallelism (TLP). Combining all the resources increases the amount of instructions processed for one application, which is called increasing the ILP, whereas dividing the resources enables running multiple threads, which is called increasing the TLP. This concept was further expanded to allow runtime reconfiguration, making the processor able to adjust while running applications.

2.5.1. Instruction lanes and groups

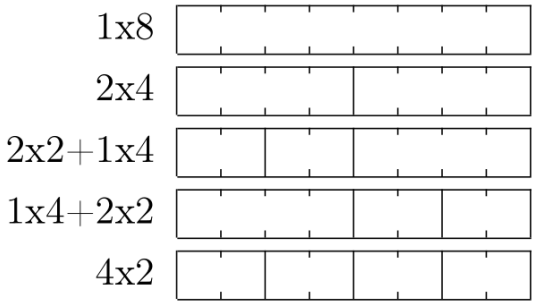
The ρ-VEX processes each syllable in an instruction lane. These lanes are grouped in pairs, which are called instruction groups. These groups can be divided or united at will. Assigning multiple groups to

[2] Syllables are individual instructions. A more complex function like multiplication can be separated into multiple syllables.

one application increases ILP, while separating all the groups increases TLP.

It is however not only possible to combine or separate all the instruction lane groups, it is also possible to combine just a few groups, or completely turn off one or multiple groups to preserve power. Not all possibilities of combining all the instruction groups is allowed because of implementation limitations. All the allowed configurations in case of an 8 instruction lane wide ρ-VEX processor is shown in Figure 2.7.

Figure 2.7: 8 instruction lane wide ρ-vex configuration possibilities

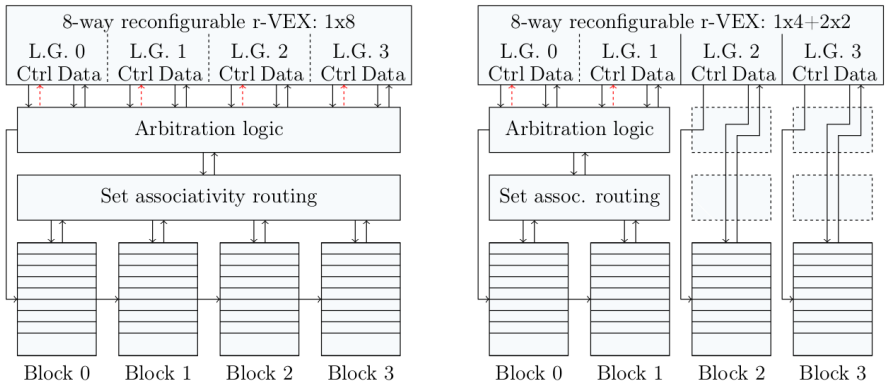


2.5.2. Cache structure

To greatly increase the processing speed of the ρ-VEX, a cache can be configured which is faster compared to the main memory and relieves the main memory bus. To make the implementation less complex, each application needs at least some space of cache. To ensure this, each instruction group is combined with a cacheblock that can not be separated. This means if more instruction groups are assigned to an application, it always has more cache size assigned to it as well.

When all of the instruction groups are combined, dedicated arbitration logic will decide which group has priority accessing the cache. Additionally, set associativity routing will decide which data is placed in which cache block. When instruction groups are detached, the arbitration logic and set associativity routing are separated as well. When an instruction group is running completely separate, the cache block is directly connected as no arbitration or associativity routing is needed. This is depicted in Figure 2.8.

Figure 2.8: ρ-VEX cache connectivity



(a) 1x8 mode. (b) 1x4 + 2x2 mode.

The cache is designed to be write-through, meaning all of the writes to the cache are also written directly to the main memory. Instruction groups can be reassigned to a different application, which also reassigns the cache associated with it. If the cache would be write-back, all of the locally stored

data must all be written instantly. This significantly slows down reassigning to a different application, especially on an FPGA as it needs to check every cache line to write local data to the main memory. To minimize the reconfiguration duration, the cache is made write-through.

Applications must be guaranteed to run properly. This means the correct data must be supplied when needed. This can become complicated if multiple applications are running using the same memory, and one application produces the data for another application. If the data is not properly updated, the application can use outdated data, making the application fail. To ensure this will not happen, a memory consistency mechanism must be implemented. The mechanism that the ρ-VEX uses is invalidating all the cache entries of the memory location that is written to. This is implemented by broadcasting an invalidation address.

The associativity of the cache of the ρ-VEX is direct mapped. As described in Section 2.1.2 the cache can reduce the miss rate when a higher associativity is used while keeping the same cache size. This thesis will implement a higher associativity cache. Furthermore, any cache that is not direct mapped needs a replacement policy, which will be investigated and implemented in this thesis. Additionally, caches will be separated from their instruction lane to allow assigning more cacheblocks to a different application.

2.5.3. Conclusion

In this section, the general structure of the ρ-VEX is presented. Additionally, the current state of the cache of the ρ-VEX is shown and possible improvements are discussed, which will be investigated and implemented in this thesis.

2.6. Conclusions

In this chapter, the background needed for the rest of this thesis is presented. In Section 2.1, the general information about caches are presented, with a focus on cache resizing and cache associativity. This information is needed to understand the design choices made in this thesis. In Section 2.2, the power saving methods for memory are listed and explained, which are used to consider the new design. In Section 2.3, all of the common replacement policies are explained, and the advantages and disadvantages of every policy is discussed. This information is useful as the current replacement policy needs to be replaced. In Section 2.4, the general workings of FPGAs are explained, on which the ρ-VEX is based. Finally, in Section 2.5, the general structure and workings of the ρ-VEX are explained with a focus on the cache, as this is where this thesis is based upon.

3

Design

In this chapter, the requirements of the current ρ-VEX design is presented, followed by the requirements of the new design. Additionally, multiple designs are proposed and evaluated. Finally, the different replacement policies are considered and evaluated for the ρ-VEX processor.

3.1. Current state

In the current version of the ρ-VEX processor the cache has a limited configurability. The amount of cache blocks is fixed at the number of instruction groups. This is decided to ensure each application is guaranteed to have at least some cache space. This results in the cache always having the same configuration as the instruction groups. Additionally, the size of the cache can not be changed at runtime. This means an application gets a certain amount of cache that can not be changed to increase the performance without changing the instruction group configuration.

The associativity of the current cache is direct mapped. This associativity has a mediocre performance as described in Section 2.3. Additionally, there is no control over where the data is placed. This prevents storing data in a specific cacheblock and then transferring to a different application, making it unable to effectively transfer data to another application.

The current state can be summed up as followed:

- Static cache size
- Fixed associativity
- No control over placement of data

3.2. Requirements

To address the research question, the new cache of the ρ-VEX processor needs to meet the following requirements:

- The cache of the ρ-VEX must be made run-time adaptable. This means the cacheblocks must be able to transfer between applications regardless of the instruction group configuration.
- The configuration must not take a significant large amount of cycles, as otherwise the penalty for reconfiguration is higher than the possible speedup.
- The cache must always return correct values, regardless if the cache is configuring or has configured or not.
- The new cacheblock structure must not significantly increase the critical path or increase the used area to limit the loss of clock speed or energy consumption respectively.

- To increase the flexibility of the possible cacheblock configurations, a new replacement policy is required that can efficiently handle more configurations. The new replacement policy must also give a better performance, i.e., reduce the amount of cache misses. The current replacement policy of the cache is direct mapped, which is only area efficient if there are a power of 2 number of ways.

To address these requirements, two aspects need to be changed. First, the cacheblock structure must be modified to allow the added configurability. Secondly, a new replacement policy is both needed and desired to improve performance. These two aspects will be examined in the following sections.

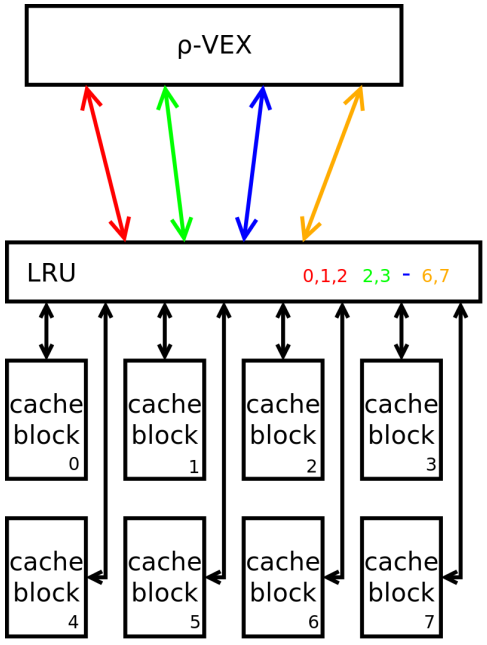
3.3. Cacheblock structure

The new cache block structure needs to be flexible enough to reassign other cacheblocks to applications. This practically means that more cacheblocks are required, and need to be able to be reassigned to the different applications. This section will present and discuss all the proposed design for the cacheblock structure.

3.3.1. Proposals

Design 1 is adding a cache managing unit between the ρ-VEX processor and the cache blocks as depicted in Figure 3.1. The added cache manager will be able to assign cache blocks when applications request more cache space. These cacheblocks can also be shared between applications when needed.

Figure 3.1: Proposed Design 1: using a cache manager to organize the cacheblocks.



Design 2 is adding the same cache manager as in the first proposal, but adding an additional small cache between the ρ-VEX and the manager, as depicted in Figure 3.2. This will guarantee that all the applications have at least some cache space. Most of the data accesses will be handled by the smaller cache, which allows the cache manager to handle bigger cache blocks at the expanse of slower access times.

Design 3 is not adding more cacheblocks to each application, but dynamically adjusting the cache block size for each application as depicted in Figure 3.3. This allows to assign more cache space to a certain application, but limits reassigning cache blocks to other applications.

Design 4 has all the cache blocks connected as in the current state. However, the network is doubled, and both of the networks can be separately configured as depicted in Figure 3.4. Additionally, the second network will have a configurable cache block size, such that assigning more cache space to an application is easier while maintaining at least some cache space to each application.

Figure 3.2: Proposed Design 2: using an extra cache level before the cache manager.

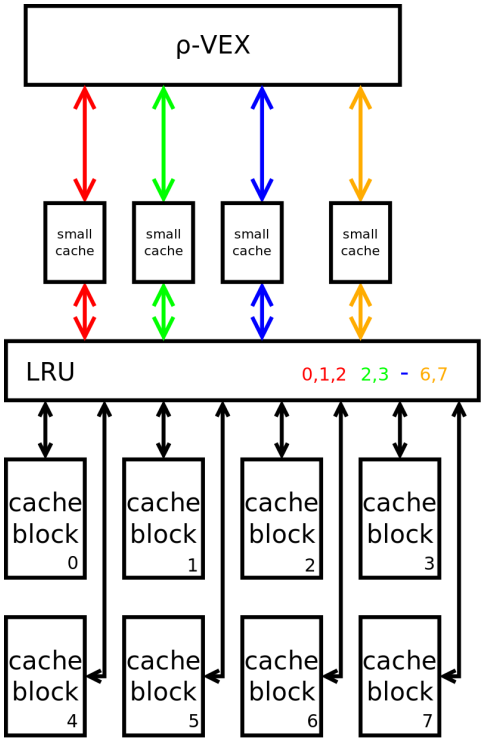
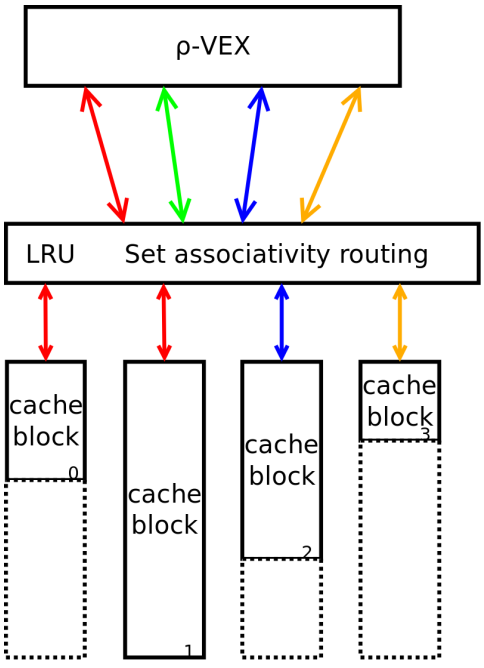


Figure 3.3: Proposed Design 3: unchanged number of cacheblocks, but adjustable cacheblock sizes.

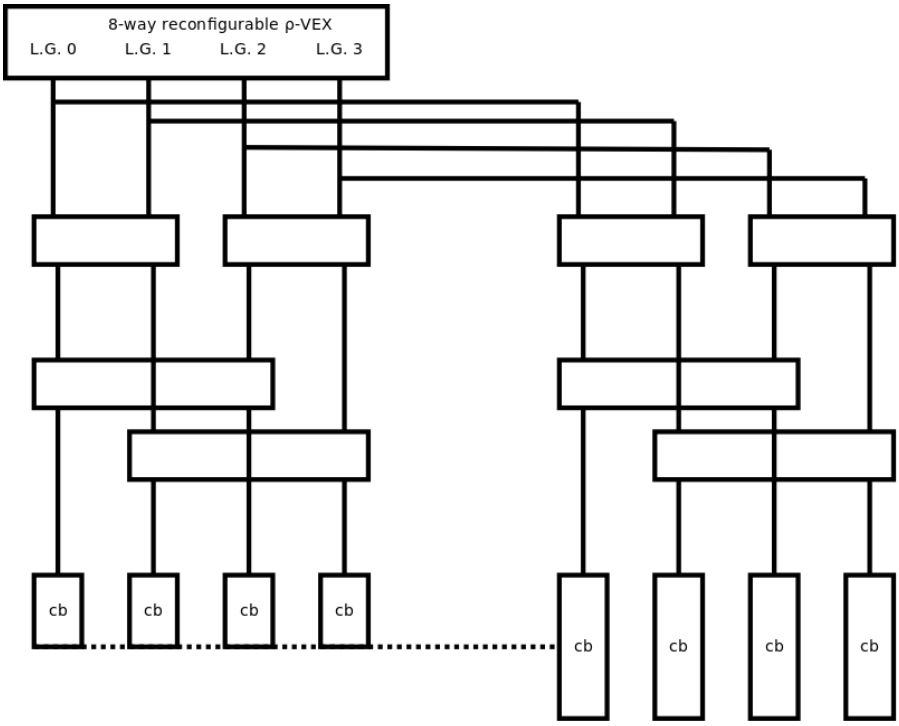


3.3.2. Comparison of proposals

The designs proposed in the previous section need to be qualitatively evaluated before a good decision can be made. The summary of this comparison is presented in Table 3.1.

Design 1 has a very flexible cache block structure, where every cacheblock configuration can be made. This will also allow easy sharing of cache blocks in case applications need to share data. One

Figure 3.4: Proposed Design 4: the original cacheblock structure is used, but duplicated. The second cacheblock tree has larger cacheblocks of which the size can be configured. Additionally, the cacheblock organization can now be set by the software, as opposed to always following the instruction group configuration.



problem with design 1 is the complicated replacement policy as a cacheblock can be used by multiple applications, requesting more cacheblocks can impede the performance of other cache blocks needing applications. Another problem is the high connectivity as all possibilities have to be possible.

Design 2 has the same cache flexibility as design 1, but will act as a level 2 cache with a separate level 1 cache. This will allow for a slower access time which gives the connectivity more possibilities. The downside is that it has to act as a level 2 cache, meaning it has to be considerably larger for it to be worthwhile, requiring either a very large level 2 cache or a very small level 1 cache. The problem with a very small level 1 cache is that many misses will occur, resulting in an increased overall access time.

Design 3 has every cacheblock completely separate, preventing any interference with other applications and reducing the amount of interconnectivity. The downside is that this design does not prevent the problems of normally resizing the cache, resulting in a varying index length and losing data.

Design 4 is a combination of the other designs, combining the strengths of the other designs while reducing the drawbacks. This design has a limited increase in the the interconnectivity of the cache, and allows changing the amount of cacheblocks per application. The downside is that only a limited amount of configurations are possible because of the used network.

3.3.3. Chosen design

One final design needs to be selected as implementing and testing multiple designs is not time efficient. As described in Section 2.4, the FPGA has limited possibilities for interconnectivity. After trying the possibilities of expanding the current design, more connectivity greatly increases the critical path. In Design 1 and 2 the connectivity greatly increases, and will become a limiting factor for the designs. Therefore a design is required that limits the increase in interconnectivity, and Design 1 and 2 are no longer feasible. Design 3 does not increase interconnectivity in any way, but does lose data when resizing. This results in retrieving data again, lowering performance. Additionally, cacheblocks can not be reassigned to other application. Design 4 slightly increases interconnectivity, but prevents the problems of losing data when resizing. Additionally, Design 4 is able to transfer cacheblocks to other applications. Because of these advantages, Design 4 is chosen as the final design.

Table 3.1: Pros and cons of designs

	pros	cons
Design 1	Cache block amounts per context are flexible, easily shared	complicated replacement policy, interference, high connectivity
Design 2	Separate level caches, L2 can take longer as it is used less	L2 needs to be significantly larger than L1, L1 has to be worthwhile
Design 3	No interference	Resizing changes indexing, losing data
Design 4	Reduced interconnectivity compared to other designs	Not all possible configurations possible

3.3.4. Discussion

The chosen design allows for more flexibility, and could be used by other platforms for extra possible configuration abilities. Reassigning cache blocks can be particularly useful for other data processing platforms that can adept to different applications. For example, the cache block reconfiguration can be used to let the main processor use the cacheblocks of a data processing core when the latter is idle, increasing the possible cache space for the main processor.

This design introduces extra connections, introducing additional critical paths, meaning this design is not for platforms with a high clock speed, e.g. general purpose processors. Additionally, ASICs and other single purpose circuits do not get any benefit with this new design. The main benefit of the new design is increased flexibility, which is useless for a single purpose circuit, as a static design is sufficient.

3.4. Replacement policies

The newly proposed design allows additional cache block configurations, including assigning an odd number of cache blocks to a certain application. An odd number of cache blocks makes direct mapped inefficient, as using a part of the address is not sufficient anymore. Therefore, a more efficient replacement policy has to be selected and implemented.

3.4.1. Common replacement policies comparison

At first the most common replacement policies are listed and evaluated. The common replacement policies are presented in Section 2.3, and are summed up as followed:

- Random
- FIFO (First In, First Out)
- LRU (Least Recently Used)
- LFU (Least Frequently Used)
- Decay

These replacement policies will be first evaluated in general, then evaluated specifically for the situation of the ρ-VEX, and finally the best policies are selected to be implemented.

The comparison of the common replacement policies is presented in Table 2.1.

3.4.2. Comparison of replacement policies specific for ρ-VEX

The comparison of the common replacement policies specific for the ρ-VEX cache is presented in Table 3.2.

- **The Random policy** places the data on a random place in the cache. This can be implemented with simple hardware as only one random number generator is needed. However, the performance is poor as useful data can be pushed out which then needs to be retrieved.

Table 3.2: Pros and cons of replacement policies for ρ-VEX

	description	pros	cons
Random	Random	Simple hardware	Low performance
FIFO	Evict oldest data	Simple hardware	Mediocre performance
LRU	Evict data that was least recently referenced	Good performance	Needs to be stored (BRAM)
LFU	Evict data that was referenced the least	Most referenced is more likely to be referenced again	Needs to be stored (BRAM), needs to evict older data
Decay	Evict data that was not accessed for certain time	Transitional decaying	Updating counters of all lines is complex, disabling one line does not save power

- **The FIFO policy** evicts the oldest data first, effectively rotating the data in the cache. This can also be implemented with simple hardware as one counter per data line is sufficient. The performance is better compared to random, but is still reasonably low as often used data is equally treated compared to unused data.
- **The LRU policy** evicts the data that has the longest time since it was previously used. This policy makes sure that often used data remains in the cache as those are referenced often. This method however needs to store the order in which each of the items are accessed, which in on an FPGA needs to be stored in an BRAM.
- **The LFU policy** counts how often each data is used, and evicts the data that is referenced the least. This improves performance as often referenced data is more likely to be used again and remains in the cache. This means that every item needs to store how often the data is used, which also needs to be stored in BRAMs. Additionally, at some point older data needs to be evicted, as otherwise the most referenced item of a previous application remains in the cache. Solving this requires an extra mechanism.
- **The Decay policy** keeps all the stored data for a limited time only in the cache, and evicts them when the limited time has passed without being used. This will automatically remove unused data, making clearing the cache shorter. This however does mean that every data line needs to have some kind of timer, which on an FPGA is very costly. Additionally, disabling the data lines does not save power as the rest of the BRAM is still used.

3.4.3. Selection of replacement policies

The new replacement policy must not have an excessive amount of area overhead, similar to the direct mapped policy with the new cacheblock combinations. Additionally, the new replacement policy must have an adequate performance, i.e. have a low miss rate.

A selection has to be made of feasible replacement policies to be implemented. The arguments for and against every policy are:

- **The Random replacement policy** can be implemented easily, but has a low performance. Therefore, other replacement policies are preferred. FIFO has a simple implementation as well, and has a better performance, making this a replacement policy to consider.
- **The FIFO replacement policy** needs to store which element is accessed last. This will require extra BRAM space, which is available. The rest of the implementation is simple and requires low area overhead, although the performance is mediocre.
- **The LRU replacement policy** needs to store the order in which the data is accessed, which is a slight area overhead. Additionally, the full LRU implementation is costly with more than 4 ways.

The new cacheblock structure allows a maximum of 8 cacheblocks for an application, meaning a smaller version needs to be implemented. Therefore a Pseudo-LRU policy needs to be used instead. This policy will give a good performance while the area overhead is reasonable.

- **The LFU and Decay replacement policies** will require a large amount of area as timers are very costly on an FPGA. Additionally, LFU needs a mechanism that requires to evict older data, which is likely to be solved using timers as well. Therefore, these two replacement policies are not area effective replacement policies for the FPGA implementation of the ρ-vex.

The two replacement policies that have at least a decent performance and not have excessive overhead are the FIFO and LRU policies. These two implementations will be implemented, but require a method to implement them. Both of the policies are implemented as follows:

- **Round Robin**: this method fills all the available places sequentially. When the last place is used, it starts overwriting the places from the first to last. This effectively means that the first placed item will be replaced first, making this an implementation of the FIFO replacement policy.
- **Pseudo LRU**: In a full LRU replacement policy the entire order of accessing is stored. As this is costly with more than 4 ways, an approximation is needed. The simplest form uses 1 bit per data block. This bit is set to 1 when the block is read from or written to. When all of the blocks have a 1, all the values are reset to 0 except for the read or write that initiated the last 1. This results in having an approximation of order of accesses. When multiple bits are used, more information can be stored. For example, one bit can be used for reads and the other for writes. As a read means the data is reused, the read value will have more weight, and the written only data will be replaced first.

In conclusion, the remaining two replacement policies to be implemented are:

- Round Robin (FIFO)
- Pseudo LRU with variable bits

3.4.4. Discussion

These chosen replacement policies can be used to freely redistribute the cacheblocks among processors. These new possibilities can be used in other systems, and enables new structures as discussed in Section 3.3.4.

Currently both the original direct mapped and the new proposed replacement policies are still write-through. This will eliminate possible reconfiguration delays, as the data in the cache can not be lost, since all the data is directly written to the main memory. Changing the replacement policy to write-back can decrease the required bandwidth, which in turn can increases overall performance, but dirty data needs to be written back to memory before switching, giving a possible additional reconfiguration delay. If a reconfiguration is known in advance, e.g. a data processing unit is starting up, the dirty data could be flushed to the main memory before the reconfiguration starts, reducing the reconfiguration delay.

3.5. Conclusions

In this chapter, the new structure and replacement policy are determined. First in Section 3.1, the current state of the cache is determined. In Section 3.2 the requirements of the new cache are determined. These requirements require a new cacheblock structure and a different replacement policy. In Section 3.3 three designs are proposed to fulfill the requirements, and the advantages and disadvantages are discussed. Subsequently, multiple designs are combined into a final design. The implementation of this design will be explained in the next chapter.

In Section 3.4, all the common replacement policies are listed and discussed for this particular design. The Random policy was not selected because of bad performance, and the LFU and Decay policies are not selected because of a problematic implementation on an FPGA. The selected replacement policies are the FIFO and LRU replacement policies. FIFO will be implemented using the Round Robin method, while LRU will be implemented using a Pseudo LRU method that supports any amount of cache blocks.

4

Implementation

In the previous chapter, the new cache requirements are given and several designs are proposed and compared. Eventually, Design 4 was chosen as described in Section 3.3.3. In this new design a second cacheblock tree is added. In order to make this design work, the signals of the two cacheblock trees need to be combined. Connecting these two cacheblock trees raises several problems, which will be described in Section 4.1. This includes expanding the configuration controller to facilitate these modifications.

As a consequence of adding an extra cacheblock tree, the replacement policy needs to be replaced as described in Section 3.4.3. The implementations of the chosen two replacement policies are FIFO and Pseudo LRU. The implementation details of these replacement policies will be described in Section 4.2.

4.1. Data trees

The two cacheblock trees of the new design need to be connected to double the possible amount of cacheblocks. This introduces complications that need to be addressed, which will be discussed in this section. The modifications of the cacheblock structure is explained in Section 4.1.1. Afterwards, the modifications of the configuration controller to facilitate the cacheblock structure modifications are explained in Section 4.1.2.

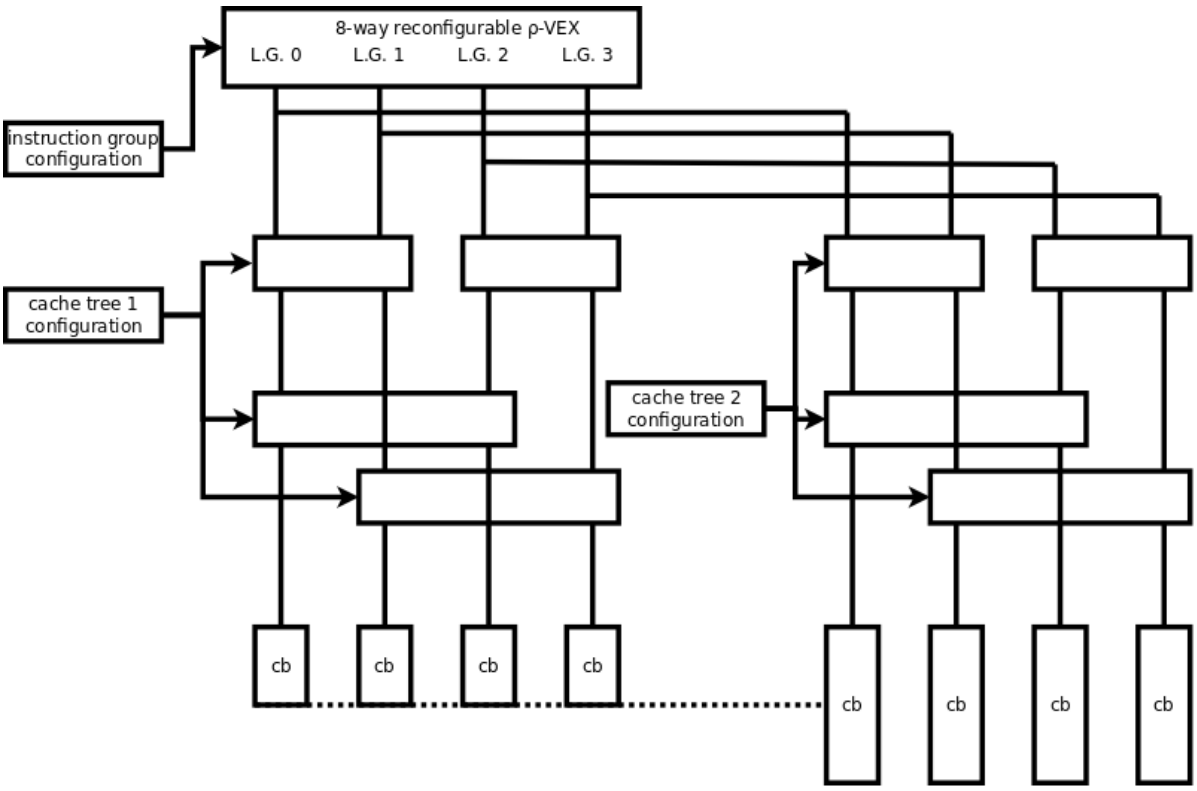
4.1.1. Cacheblock configurations and connections

The cache trees do not need to be identical to the instruction lane groups in the new configuration. Additionally, the configurations of the two cache trees can become different from each other. Therefore, each of the trees need to be given a separate configuration. This concept is depicted in Figure 4.1. This requires extending the configuration controller, which is described in Section 4.1.2. The configuration controller gives the current configuration to the cache, ensuring the configuration is never outdated.

Each cache tree will both propagate data and control signals. When both cache trees are connected, these data and signals need to be combined. In the old design, these signals follow the same pattern as the instruction groups, making combining the signals relatively easy. However, the new design does not have to follow the instruction group configuration, which makes combining the signals more difficult. Combining certain signals is easy. One example is the error signal, which is always propagated to all the cacheblocks and instruction groups. The data is always broadcasted from the instruction groups to the cacheblocks and the other way around.

The most problematic signal to combine is the miss signal. This signal not only tells the instruction-group to stall and wait for information, it also signals the cacheblocks to request the data from the higher memory. Simply reusing the existing path is not possible, as all the miss signals from both the cache trees need to be combined before a final result is available, and that result needs to be forwarded to both the instruction groups and the cache trees. As traversing both trees is too slow, a direct approach is needed for these particular signals. The solution with the lowest critical path was combining and comparing the signals per context. This is depicted in Figure 4.2. In this figure all the cacheblocks that have the same color are grouped together with the instruction group. In this figure the leftmost

Figure 4.1: The three needed configurations that need to be controlled in the new cache structure.



cacheblock of cache tree 1 and the entirety of cache tree 2 is assigned to the same context as the leftmost instruction group. These cacheblocks combine their signals in order to give one combined signal to the instruction group. Additionally, all the signals given by the instruction groups are given to all the cacheblocks of the same context. This method is only used for the signals that need to both combine and broadcast all the signals.

The ρ-VEX processor can be configured to have more or less instruction lanes, and the programming code is written for providing this flexibility. The cache needs to be written in the same style to maintain the flexibility. This means the cache must be able to synthesize with different cache sizes. This includes the cacheblock sizes, as well as if one or two cache trees are requested. This will be achieved using generate loops. These generate the user-specified amount of cacheblocks and cache trees, as well as the requested cacheblock sizes. The parameters to set the sizes and the number of cache trees that need to be generated are taken from the highest level parameter list, making the parameters easy to find.

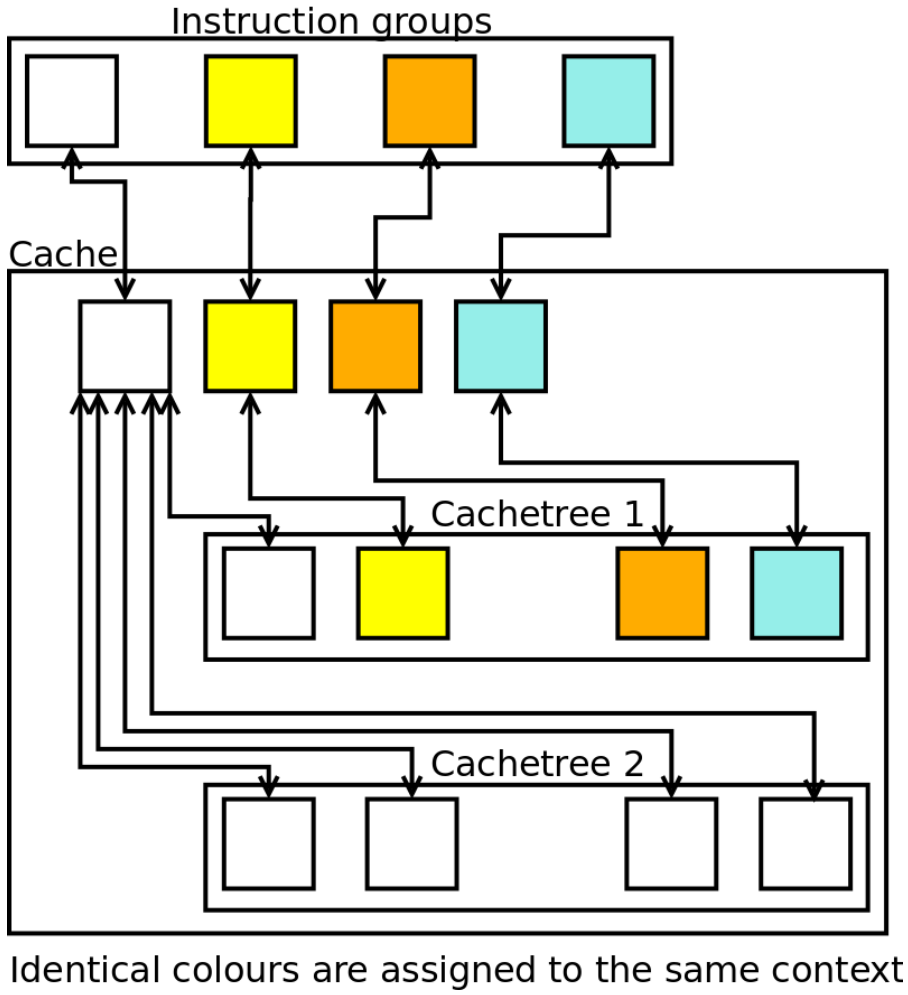
The entire design is depicted in Figure 4.3. The cache blocks are connected to a bus that handles the read and write requests to and from the main memory. The red blocks and connections are related to the replacement policy, of which the general structure is depicted in Figure 4.2. The implementation details of both of the replacement policies will be explained in Section 4.2.

4.1.2. Configuration controller

The modified cache trees need separate configurations as described in the previous section. To assure this is possible, the configuration controller needs to be extended. The configuration of the instruction groups and the cache trees needs to be separately calculated, as well as separately stored. This is depicted in Figure 4.1.

The configuration controller needs to check if the configuration that is requested is correct. This requires the configuration controller to check which cacheblock belongs to which context. A cacheblock will be turned off when it is not assigned to any context. This results in a list in which cacheblocks are grouped together, and this list is checked with a predetermined list to see if the cache tree can handle

Figure 4.2: Overview of the new cache tree connections.



this configuration.

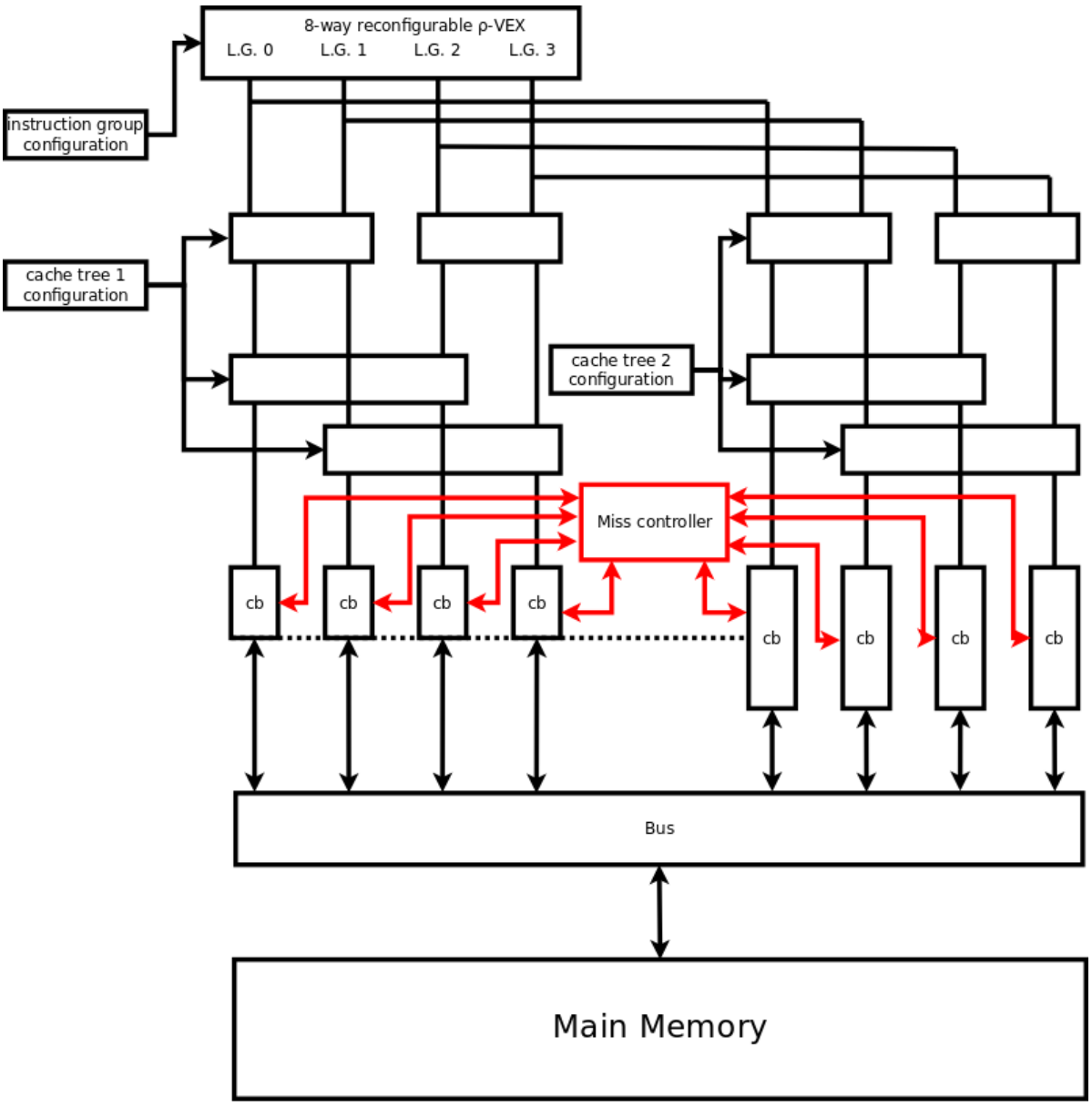
The instruction groups can be configured separately, so a different command for configuring the cache is needed. The configuration of the two cache trees can be combined into one command, and both the instruction groups and cache tree reconfiguration can be issued at the same time to prevent a redundant configuration delay. This can be achieved by first sending all the requests, which are stored in a separate control registers. Subsequently, the reconfiguration can be requested, at which point the control registers are compared to the current configuration. If the requested configuration is identical to the current configuration, the reconfiguration is skipped. If the new configuration differs from the current configuration, both the instruction groups and the cache trees are configured according to the new configuration.

4.2. Replacement policies

As described in the previous chapter, two replacement policies will be implemented. These two replacement policies are mutually exclusive and can not be used simultaneously. Only one replacement policy implementation can be synthesized at a time to limit the area overhead and reduce the synthesizing time. A parameter can be set to determine which replacement policy will be implemented. The two replacement policies are implemented as followed:

- **The FIFO replacement policy** will be implemented by using the Round Robin method as described in Section 3.4.3. This implementation needs a counter per cache group. The counter determines in which cache location the data is written, and afterwards the counter is increased

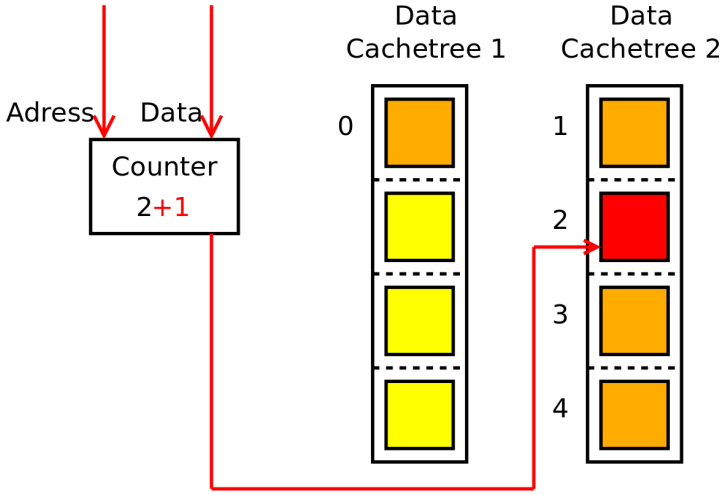
Figure 4.3: Overview of the new cache design.



by one as depicted in Figure 4.4. The yellow cacheblocks are assigned to other instruction groups, and the orange cacheblocks are part of the same instruction group with the wrong tag. The values next to the red cacheblocks show the numbering that serve as labels. In this figure, the data is written to the cacheblock labeled 2 as the counter has that value. The counter is reset after the last cacheblock is written to. If the counter is higher than the total amount of cache locations because of a reconfiguration, the counter is reset and the data is written to the first location. The maximum amount of the cache locations is determined by adding the total amount of locations per context. This directly determines the maximum value, making it clear when the counter needs to reset.

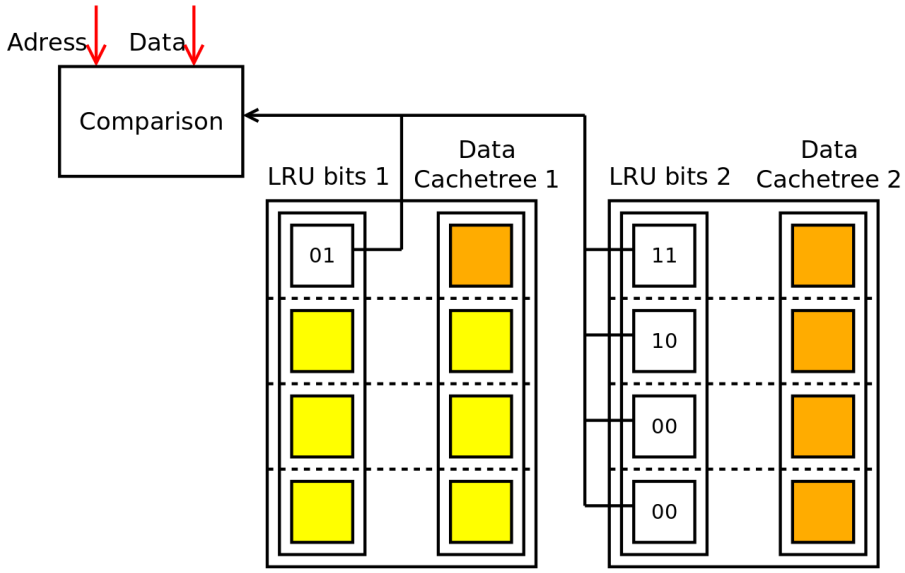
- **The LRU replacement policy** will be implemented by using a pseudo LRU implementation as described in Section 3.4.3. This implementation will store one or two bits per data word. The number of bits used per word can be set by using a parameter when synthesizing the platform. If one bit per word is used, the implementation works as followed. If data needs to be written to the cache, the first location that has a 0 will be used to store the data, and that location bit will be

Figure 4.4: Example of the implementation of the FIFO replacement policy in the form of Round Robin.



set to 1. When one of the locations is successfully read from, the location bit will also be set to 1. When the data is written to the last 0 location, reset all the other locations to set to 0. When using 2 bits, a read make the left bit 1, and a write sets the right bit. When the last 0 is written to, shift all the other values one to the right. The lowest bit value is written to if new data needs to be placed, as depicted in Figures 4.5 and 4.6. The yellow cacheblocks are assigned to other instruction groups, and the orange cacheblocks are part of the same instruction group with the wrong tag. The values in the white cacheblocks represent if the cacheblock is read from (left bit) or written to (right bit). If the last 00 value is written to, all the values are shifted one bit to the right.

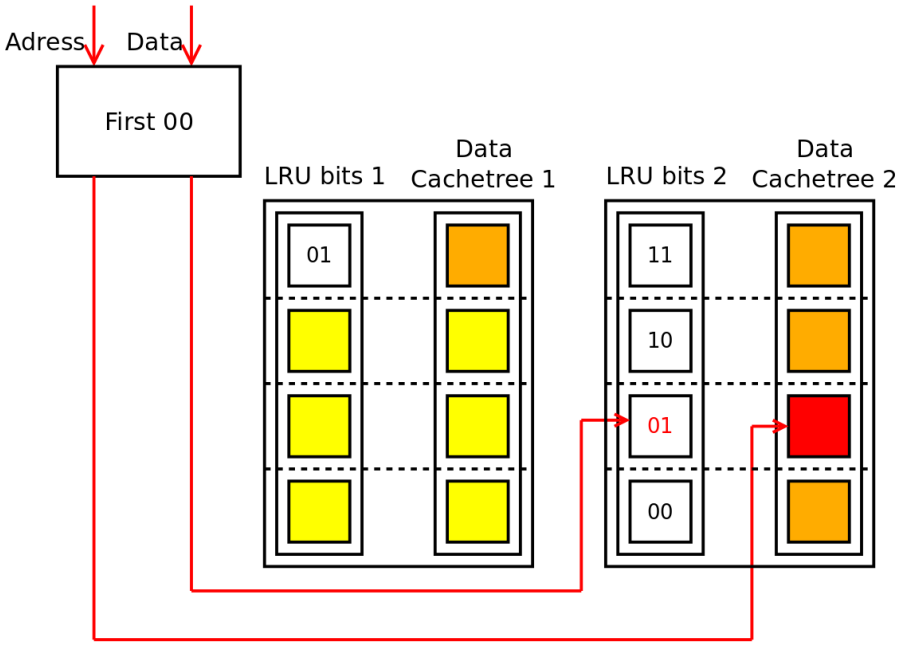
Figure 4.5: Determining the lowest LRU value, as part of the implementation of the LRU replacement policy in the form of a two bit wide Pseudo-LRU.



4.3. Conclusion

In this chapter, the implementation of the new structure is presented. First in Section 4.1 the structure of the cache trees is further explained. The problems that stem from combining the two trees are pre-

Figure 4.6: Writing to the lowest LRU value and updating the LRU bits, as part of the implementation of the LRU replacement policy in the form of a two bit wide Pseudo-LRU.



sented, and the solutions for these problems are given. Additionally, the extensions of the configuration controller are explained to facilitate the changes to the cache trees. In Section 4.2, the implementations of the replacement policies are presented and explained. These implementations are the Round Robin and the pseudo LRU methods. After the implementation, the design needs to be tested, which will be done in the following chapter.

5 Verification

In this chapter, the implementation of both the replacement policies and the new cache structure as described in the previous chapter need to be tested. First, the benchmark that will be used for comparing the replacement policies and designs will be explained. Following are the tests of the replacement policies in a simulator written in C. Additionally, the design is implemented on the FPGA, and the execution times are compared for the different designs. Finally, the discussion of the results is presented.

5.1. Benchmarks

The new design must be compared to the original design to show and measure the speed improvement. The programs that are made available for this project are part of three benchmarks: the MiBench benchmark suite, the SPEC benchmark suite and the Polybench benchmark.

- Mibench is a free, commercially representative embedded benchmark suite [3]. Several characteristics distinguish the representative embedded programs from the existing SPEC benchmarks including instruction distribution, memory behavior, and available parallelism. The embedded benchmarks, called Mibench, are freely available to all researchers. This benchmark suite makes extensive use of both the caches, so cache improvements should be shown in the results.
- The Standard Performance Evaluation Corporation (SPEC) is a nonprofit consortium whose members include hardware vendors, software vendors, universities, customers, and consultants. [7]. SPEC's mission is to develop technically credible and objective component- and system-level benchmarks for multiple operating systems and environments, including high-performance numeric computing, Web servers, and graphical subsystems. This benchmark suite uses everything, including both the caches, so cache improvements should be shown in the results.
- Polybench is a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.)[11]. Polybench is developed at the Ohio state university. This benchmark suite mostly uses numerical computations with static control flow, resulting in very little instruction and data caches usage. Polybench will be used to measure if the cache improvements will negatively impact programs which do not regularly use the caches.

To compare the speedup of the new cache structure, first the program is run on the platform using the old configuration and replacement policy. Next the program is run again using the new cache structure, but with the same configuration as the instruction groups. Finally the programs are run using the extra cache tree, which results in expanded cache space. This will determine the final possible speedup with the same available cache storage.

5.2. Replacement policies

Modifying the simulator to accurately measure the number of cache misses takes less effort compared to implementing the new replacement policies on the FPGA. Only a few replacement policies will be

implemented on the FPGA to limit the implementation time. Therefore, a selection of the replacement policies will be made based on the simulator results. Based on the results, only the best replacement policies will be implemented on the FPGA in the next section.

5.2.1. Testing methods

The replacement policies will be tested in a simulator. This simulator is written in C, and simulates the entire core. This includes the configuration and interactions of the instruction groups, the instructions being processed by the cores, and the fetching and using of instructions and data for the applications. The simulator keeps track for correct application behavior, therefore a wrongly stored or fetched data or instruction from the cache will give incorrect values.

The data and instructions in the caches are accurately tracked, including the tags and validation bits. The simulator can also keep track of the number of misses per cache. These values are the main comparison for the effectiveness of each of the replacement policies.

The explanations and selection of the best replacement policies for this system are discussed in Section 3.4.3. The replacement policies that will be tested relative to the current direct mapped policy are:

- Round Robin
- LRU with 1 bit
- LRU with 2 bits, 1 for read and 1 for write
- LRU with 3 bits, 2 for read and 1 for write
- LRU with 3 bits, 1 for read and 2 for write

The programs will first be run on the simulator on the current direct mapped platform. The number of cache misses of both the instruction- and data cache act as the reference for the other replacement policies, showing the performance relative to direct mapped. The first part of the results show the results of the replacement policies with the exact cache structure as the old design. The second part of the results show the effect of adding the extra cache tree. In this part the original direct mapped design has double the original cache space to compensate for the added cache blocks for the other replacement policies.

5.2.2. Results

The first part of the results show the results of the replacement policies with the exact cache structure as the old design. The second part of the results show the effect of adding the extra cache tree.

The results are separated for the data cache and instruction cache. There is a graph for each benchmark that displays the relative amount of cache misses per individual benchmark in Appendix A. A table of averages is provided for each aspect in this section.

Original cache sizes

First the benchmarks are run with the original cache size and cacheblocks configuration. This will ensure that only the effect of the replacement policies will be tested, as the rest of the system is completely identical.

Data cache Results of the Mibench simulations are in Figure A.1, and the averages are in Table 5.1. Round Robin has a reduction of 10%, but LRU has a reduction of 23%. Results of the SPEC simulations are in Figure A.2, and the averages are in Table 5.2. Round robin has a reduction of 23%, but LRU has a reduction of 25%. Results of the Polybench simulations are in Figure A.3, and the averages are in Table 5.3. All values are about the same, although Round Robin and both 3 bit LRU versions have a negative impact.

Table 5.1: Average relative misses of the entire Mibench benchmark for the data cache with the original size

	reduction of misses in %
Round Robin	10.34 %
LRU 1-bit	22.97 %
LRU 2-bit	22.95 %
LRU 3-bit (2R, 1W)	22.74 %
LRU 3-bit (1R, 2W)	22.69 %

Table 5.2: Average relative misses of the entire SPEC benchmark for the data cache with the original size

	reduction of misses in %
Round Robin	22.23 %
LRU 1-bit	23.38 %
LRU 2-bit	24.54 %
LRU 3-bit (2R, 1W)	24.68 %
LRU 3-bit (1R, 2W)	24.68 %

Table 5.3: Average relative misses of the entire Polybench benchmark for the data cache with the original size

	reduction of misses in %
Round Robin	-5.00 %
LRU 1-bit	1.02 %
LRU 2-bit	0.39 %
LRU 3-bit (2R, 1W)	-0.54 %
LRU 3-bit (1R, 2W)	-0.54 %

Table 5.4: Average relative misses of the entire Mibench benchmark for the instruction cache with the original size

	reduction of misses in %
Round Robin	75.24 %
LRU 1-bit	77.51 %
LRU 2-bit	77.37 %
LRU 3-bit (2R, 1W)	77.22 %
LRU 3-bit (1R, 2W)	77.22 %

Table 5.5: Average relative misses of the entire SPEC benchmark for the instruction cache with the original size

	reduction of misses in %
Round Robin	84.52 %
LRU 1-bit	85.20 %
LRU 2-bit	85.01 %
LRU 3-bit (2R, 1W)	85.00 %
LRU 3-bit (1R, 2W)	85.00 %

Instruction cache Results of the Mibench simulations are in Figure A.4, and the averages are in Table 5.4. Round Robin has a reduction of 75%, but LRU has a reduction of 77%. Results of the SPEC simulations are in Figure A.5, and the averages are in Table 5.5. Round Robin has a reduction of 84.5%, but LRU has a slightly higher reduction of 85%. Results of the Polybench simulations are in Figure A.6, and the averages are in Table 5.6. All values are about the same, although Round Robin has a negative impact.

Added cache tree

In these tests the extra cache trees are added, which doubles the amount of cacheblocks. Adding extra cacheblocks increases the total available cache space, lowering the miss rate. In this section, the effect of the cache trees will be tested, meaning the total available cache space should be equal. In order to ensure the equal amount of cache space, the extra added cacheblocks are of equal size of the original cache tree. The original direct mapped design will have double the cache block size to compensate for the extra cache space added by the extra added cacheblocks.

Data cache Results of the Mibench simulations are in Figure A.7, and the averages are in Table 5.7. Round Robin has a reduction of 7%, but LRU has a reduction of 15.5%. Results of the SPEC simulations are in Figure A.8, and the averages are in Table 5.8. Round Robin has a reduction of 23.8%, but LRU has a reduction of 26%. Results of the Polybench simulations are in Figure A.9, and the averages are in Table 5.9. The Round Robin policy negatively impacts the number of misses, while all the LRU versions reduce the number of misses, most around 6.5%.

Table 5.6: Average relative misses of the entire Polybench benchmark for the instruction cache with the original size

	reduction of misses in %
Round Robin	-1.37 %
LRU 1-bit	0.03 %
LRU 2-bit	0.03 %
LRU 3-bit (2R, 1W)	0.03 %
LRU 3-bit (1R, 2W)	0.03 %

Table 5.7: Average relative misses of the entire Mibench benchmark for the data cache with the increased size

	reduction of misses in %
Round Robin	7.13 %
LRU 1-bit	15.56 %
LRU 2-bit	15.48 %
LRU 3-bit (2R, 1W)	15.18 %
LRU 3-bit (1R, 2W)	15.50 %

Table 5.8: Average relative misses of the entire SPEC benchmark for the data cache with the increased size

	reduction of misses in %
Round Robin	23.80 %
LRU 1-bit	26.26 %
LRU 2-bit	26.31 %
LRU 3-bit (2R, 1W)	25.96 %
LRU 3-bit (1R, 2W)	26.26 %

Table 5.9: Average relative misses of the entire Polybench benchmark for the data cache with the increased size

	reduction of misses in %
Round Robin	-1.49 %
LRU 1-bit	6.88 %
LRU 2-bit	6.26 %
LRU 3-bit (2R, 1W)	4.34 %
LRU 3-bit (1R, 2W)	7.05 %

Table 5.10: Average relative misses of the entire Mibench benchmark for the instruction cache with the increased size

	reduction of misses in %
Round Robin	76.35 %
LRU 1-bit	77.85 %
LRU 2-bit	77.85 %
LRU 3-bit (2R, 1W)	77.85 %
LRU 3-bit (1R, 2W)	77.85 %

Table 5.11: Average relative misses of the entire SPEC benchmark for the instruction cache with the increased size

	reduction of misses in %
Round Robin	86.18 %
LRU 1-bit	89.46 %
LRU 2-bit	89.46 %
LRU 3-bit (2R, 1W)	89.43 %
LRU 3-bit (1R, 2W)	89.42 %

Instruction cache Results of the Mibench simulations are in Figure A.10, and the averages are in Table 5.10. Round Robin has a reduction of 76%, but LRU has a reduction of 77.9%. Results of the SPEC simulations are in Figure A.11, and the averages are in Table 5.11. Round Robin has a reduction of 86%, but LRU has a slightly higher reduction of 89.4%. Results of the Polybench simulations are in Figure A.12, and the averages are in Table 5.12. All values are about the same, although Round Robin negatively impacts the number of misses.

5.2.3. Discussion

First the data cache benchmarks are discussed. Following is the discussion for the instruction cache. Afterwards is the comparison of the effects of the newly added cache tree. The discussion will be concluded afterwards.

Original cache tree
In these results the cacheblock structure is left completely unchanged, only the replacement policies are implemented and tested.

Data cache The data cache Mibench misses are reduced for Round Robin by 10% and for LRU by 22%. This shows that both policies reduce the number of cache misses, although LRU has a more than double reduction. The reduction of data cache misses for the SPEC benchmarks are more similar for both policies, namely 22.2% for Round Robin and 23.4% and 24.7% for LRU. In both cases the LRU policies give better results, although using more bits seem to have little effect. The Polybench benchmark shows an increase of misses for Round Robin, indicating that the number of misses is

Table 5.12: Average relative misses of the entire Polybench benchmark for the instruction cache with the increased size

	reduction of misses in %
Round Robin	-0.41 %
LRU 1-bit	0.03 %
LRU 2-bit	0.03 %
LRU 3-bit (2R, 1W)	0.03 %
LRU 3-bit (1R, 2W)	0.03 %

increased for programs with little cache usage. For the LRU policies a small reduction is achieved for all benchmarks.

Instruction cache The instruction cache Mibench misses are reduced for both Round Robin and LRU by 75%. This shows a sizable reduction of misses relative to direct mapped. The SPEC benchmarks show a reduction of 85% instruction cache misses. The Polybench benchmark show very small changes compared with direct mapped, although the number of misses are very low and likely mostly start up misses, which are unavoidable. Round Robin does slightly increase the number of instruction cache misses, while the LRU replacement policies slightly reduce the number of misses.

Added cache tree
The extra added cache tree adds extra cacheblocks that can be separately configured. In these tests the performance of the cache tree is compared with equal total cache sizes. This tests if the replacement policies still give benefits regardless of configuration.

The cache misses are still reduced compared to the original direct mapped replacement policy. The reduction is somewhat lower, which can be explained by the increased total cache size, as some programs has reached the optimal cache space and replacement policies have less effect.

The main purpose of the added cache tree is extra flexibility of allocating cacheblocks for different instruction groups. The miss rate reduction is similar when the extra cacheblocks are added, showing that the replacement policy can handle the maximum amount of cacheblocks.

Combined
Both Round Robin and LRU show a significant reduction of cache misses. Round Robin reduces the misses when the cache is used extensively, but increases the misses when the cache is used rarely. Round Robin is relatively easy to implement and will still be selected for the FPGA implementation. LRU show a higher increase in cache miss reductions compared to Round Robin. The performance of the different versions of LRU policies vary between the benchmarks. Only the 1 and 2 bits implementations never increase the number of misses. Because of this, only the 1 and 2 bits versions of the LRU will be implemented on the FPGA.

5.3. Implementation on the FPGA

In this section the implementation of the new cache implementation will be compared and discussed. First the validation of the design will be discussed. Afterwards, the size and performance of the new design is presented. Finally, the hardware implementation will be discussed.

5.3.1. Verification of hardware implementation

The hardware implementation needs to be tested before the performance can be compared. The two main methods of verification are checking correct application behavior, and checking if the exact amount of expected cache accesses are made.

The benchmarks compare the results with the expected values. The results of the benchmarks can be wrongly calculated if the used values are incorrect. The larger benchmarks use billions of cache accesses in different manners, diminishing the chance a faulty stored value is missed. Therefore, the benchmarks will report if values are stored and retrieved incorrectly.

The simulator is tested and verified to show the exact timings of the hardware implementations, including the caches. The implementation stores and checks correctly all the indexes of the cache entries. If done correctly, the hardware should show the exact same amount of cache misses and clock cycles as the simulator. The hardware shows a faulty implementation if there is a difference between the number of cache misses or the number of clock cycles for the simulator and the hardware.

All the benchmarks are run on the hardware while staying alert of the two previously mentioned results. All the benchmarks show the results are exactly as expected, showing all the used values are correctly stored and retrieved. Additionally, both the number of cache misses and the number of clock cycles of the hardware are identical to the simulator. This shows the hardware implementation behaves exactly as predicted by the simulator, which is the correct behavior. In turn, this also gives extra validation to the simulator, as both completely changing the replacement policy and separating the cacheblocks from their instruction groups have not occurred and tested before.

Table 5.13: Number of LUTs in the implementation with the same amount of cache space.

	number of LUTs	number of RAM blocks
Direct mapped (current design)	55,482	228
Round Robin	60,957	228
LRU 1-bit	59,060	236
LRU 2-bit	60,178	236

Table 5.14: Average reduction of runtime in % per benchmark suite relative to current direct mapped design with the original size

execution time	Mibench	SPEC	Polybench
Round Robin	11.82 %	11.67 %	-0.34 %
LRU 1-bit	13.60 %	12.44 %	1.414 %
LRU 2-bit	13.56 %	12.71 %	1.38 %

5.3.2. Testing methods

The new implementations need to be compared to the original design. First, the size of the new implementation with the new replacement policies are presented and compared to the original design. Secondly, all the benchmarks are run on the new platforms, showing the difference compared to the old design. Finally, the results will be discussed and combined to show the total speedup.

5.3.3. Results

The implementation has the size as depicted in Table 5.13. The RAM blocks are a combination of the RAMB18E1 and RAMB36E1 types. The reduction of the execution times in % per benchmark suite relative to the current direct mapped design with the same cacheblock configuration is depicted in Table 5.14. Additionally, the average reduction of the execution times in % per benchmark suite relative to the current direct mapped design with the extra cacheblock tree is depicted in Table 5.15. The graphs of every benchmark comparison are in Section A.2.2. The new implementation of the cache has the same clock frequency as the original design, making it possible to compare the run times directly.

5.3.4. Discussion

Both Round Robin and the LRU implementations are bigger than direct mapped, which is expected. The Round Robin implementation increases the number of used LUTs from 55,482 to 60,957, an increase of about 10%. The LRU with 1 bit policy increases the used number of LUTs to 59,060, which is lower than Round Robin. Even the LRU with 2 bits has fewer LUTs compared to Round Robin. However, LRU uses more RAM memories, namely 236 instead of 228. LRU with 2 bits should require more RAM storage, but on the FPGA those extra needed RAM storage apparently fit in the extra allocated RAM blocks needed for LRU with 1 bit, as both implementations have the same number of RAM blocks. This makes comparing the size of an implementation in silicon difficult, since the sizes of the logic and the RAM blocks are technology dependent.

A summary of the performance and area overhead of the new designs compared to the current

Table 5.15: Average reduction of runtime in % per benchmark suite relative to current direct mapped design with the increased size

execution time	Mibench	SPEC	Polybench
Round Robin	4.06 %	10.22 %	-0.20 %
LRU 1-bit	5.08 %	11.61 %	2.98 %
LRU 2-bit	5.06 %	11.75 %	3.07 %

Table 5.16: Comparison of the newly implemented cache structure and replacement policies relative to the current direct mapped design

compared to current design	cache-testing benchmarks runtime reduction	non-cache-testing benchmarks runtime reduction	hardware increase
Round Robin	11.75 %	-0.34 %	9.87 %
LRU 1-bit	13.02 %	1.41 %	6.44 %
LRU 2-bit	13.04 %	1.38 %	8.46 %

direct mapped design is depicted in Table 5.16. The Round Robin replacement policy reduces the runtime of applications that extensively use the caches, but increases the runtime when the application has a low cache utilization. The LRU replacement policy always reduces the application runtime, and can give runtime reductions of about 13.02% of the runtime for cache heavy applications. When all the cache space is assigned to one instruction group, one cache tree gives big runtime reductions with cache heavy applications, but two cache trees have a higher reduction for applications with low cache usage.

The performance between the 1 bit and 2 bits LRU variants are very similar, while the 2 bits variant needs more area. Therefore the 1 bit variant is more efficient and should be the standard LRU version. The LRU 1 bit policy always reduces the execution time, and can decrease the run time about 13.02%, while the increase in hardware is increased by 6.44%.

5.4. Conclusion

In this chapter the replacement policies and design are implemented and tested. First in Section 5.1, the benchmarks used for testing are presented, namely the Mibench, Polybench and SPEC benchmarks. In Section 5.2, the replacement policies are tested on the simulator. The new replacement policies allow for different configurations of the cache compared to the instruction groups. The Round Robin replacement policy sometimes increases the number of cache misses by 5%, but reduces the misses of the data cache between 10.34% and 22.23% for programs with extensive cache usage. The LRU replacement policies never increases the number of cache misses, and reduces the misses of the data cache between 22.95% and 23.38% for programs with extensive cache usage. There is little performance difference between the number of LRU bits used, so only the 1 and 2 bits versions will be implemented. In Section 5.3 the design is implemented on the FPGA and verified. Additionally, the new design is compared to the original design, both in size and the effects on application run times. All the new designs are about 10% bigger on the FPGA. Round Robin reduces the runtime when the application heavily uses the caches, but increases the runtime when the application has a low cache utilization. LRU always reduces the application runtime, and reduces the run times of cache heavy applications by about 13%. The performance between the 1 bit and 2 bits LRU variants are very similar, while the 2 bits variant needs more area. The LRU 1 bit policy always reduces the execution time, and can decrease the run time about 13.04%, while the increase in hardware is increased by 6.44%.

6

Conclusion

In this chapter the thesis will be concluded. First a summary of this thesis will be presented. Following is a list of the contributions made in this thesis. Finally, suggestions are presented for future work.

6.1. Summary

In Chapter 2, the background needed for the rest of this thesis is presented. In Section 2.1, the general information about caches are presented, with a focus on cache resizing and cache associativity. This information is needed to understand the design choices made in this thesis. In Section 2.2, the power saving methods for memory are listed and explained, which are used to consider the new design. In Section 2.3, all of the common replacement policies are explained, and the advantages and disadvantages of every policy is discussed. This information is useful as the current replacement policy needs to be replaced. In Section 2.4, the general workings of FPGAs are explained, on which the ρ-VEX is based. Finally, in Section 2.5, the general structure and workings of the ρ-VEX are explained with a focus on the cache, as this is where this thesis is based upon.

In Chapter 3, the new structure and replacement policy are determined. First in Section 3.1, the current state of the cache is determined. In Section 3.2, the requirements of the new cache are determined. These requirements require a new cacheblock structure and a different replacement policy. In Section 3.3, three designs are proposed to fulfill the requirements, and the advantages and disadvantages are discussed. Subsequently, multiple designs are combined into a final design. The implementation of this design will be explained in the next chapter. In Section 3.4, all the common replacement policies are listed and discussed for this particular design. The Random policy was not selected because of bad performance, and the LFU and Decay policies are not selected because of a problematic implementation on an FPGA. The selected replacement policies are the FIFO and LRU replacement policies. FIFO will be implemented using the Round Robin method, while LRU will be implemented using a Pseudo LRU method that supports any amount of cache blocks.

In Chapter 4, the implementation of the new structure is presented. First in Section 4.1, the structure of the cache trees is further explained. The problems that come from combining the two trees are presented, and the solutions for these problems are given. In Section 4.1.2, the reason the configuration controller needs to be updated is presented. Additionally, the expanded functionality added to the configuration controller is presented. In Section 4.2, the implementations of the replacement policies are presented and explained. These implementations are the Round Robin and the pseudo LRU methods. After the implementation, the design needs to be tested, which will be done in the following chapter.

In Chapter 5 the replacement policies and design are implemented and tested. First in Section 5.1, the benchmarks used for testing are presented, namely the Mibench, Polybench and SPEC benchmarks. In Section 5.2, the replacement policies are tested on the simulator. The new replacement policies allow for different configurations of the cache compared to the instruction groups. The Round Robin replacement policy sometimes increases the number of cache misses by 5%, but reduces the misses of the data cache between 10.34% and 22.23% for programs with extensive cache usage. The LRU replacement policies never increases the number of cache misses, and reduces the misses of the data cache between 22.95% and 23.38% for programs with extensive cache usage. There is little

performance difference between the number of LRU bits used, so only the 1 and 2 bits versions will be implemented. In Section 5.3 the design is implemented on the FPGA and verified. Additionally, the new design is compared to the original design, both in size and the effects on application run times. All the new designs are about 10% bigger on the FPGA. Round Robin reduces the runtime when the application heavily uses the caches, but increases the runtime when the application has a low cache utilization. LRU always reduces the application runtime, and reduces the run times of cache heavy applications by about 13%. The performance between the 1 bit and 2 bits LRU variants are very similar, while the 2 bits variant needs more area. The LRU 1 bit policy always reduces the execution time, and can decrease the run time about 13.04%, while the increase in hardware is increased by 6.44%.

6.2. Contributions

As described in Section 1.2, the problem statement of this thesis is:

How to design a run-time adaptable cache organization for a dynamic multi-processor system?

This is achieved by replacing the replacement policy with a more flexible variant, as well as adding an extra cache tree that allows for the possibility to reallocate cache blocks during runtime. The replacement policy is replaced by round robin and (pseudo) LRU, giving the required flexibility as well as decreasing cache misses.

The requirements for answering the problem statement were given in Section 1.2. How each of these requirements are completed is explained separately per requirement.

1. *The cache of the ρ-VEX must implement a better replacement policy that can support all configurations, as the current replacement policy only supports a power of 2 number of cache blocks.*

   This requirement was accomplished by investigating possible replacement policies, of which Round Robin and (Pseudo) LRU offered the required flexibility. Both of these two replacement policies operate cache blocks independently of each other, meaning all the cache blocks can be freely arranged as opposed to the initial direct mapped replacement policy implementation.

2. *The cache of the ρ-VEX must be able to dynamically reconfigure the cache configuration to allocate more cacheblocks for specified contexts as the current implementation only supports a cache configuration identical to the application configuration*

   This is accomplished by the addition of the second cache tree, allowing for a more flexible cache size assignment as either the small or the larger cache tree can be assigned to a specific instruction group configuration. The assignments of the cache blocks can be reconfigured during runtime.

3. *The cache of the ρ-VEX must return the correct values to the ρ-VEX at all times during and after reconfigurations to maintain the integrity of the applications.*

   The cache is verified to always give the correct values, and give the exact same number of expected cache accesses and misses.

As an additional contribution that was not part of the problem statement:

- The new replacement policies reduce the number of cache misses. The Round Robin replacement policy sometimes increases the number of cache misses by 5%, but reduces the misses of the data cache between 10.34% and 22.23% for programs with extensive cache usage. The LRU replacement policies never increases the number of cache misses, and reduces the misses of the data cache between 22.95% and 23.38% for programs with extensive cache usage. The reduction of cache misses results in better overall performance for the ρ-VEX. Round Robin reduces the runtime when the application heavily uses the caches by 11.7%, but increases the runtime when the application has a low cache utilization. LRU always reduces the application runtime, and reduces the run times of cache heavy applications by about 13%.

6.3. Future work

During the writing of this thesis multiple ideas came up for possible future work. These ideas are:

1. *Changing the cache from the current write-through to write-back.* Write-back caches generally give better performance, but for this platform the dirty data needs to be flushed when swapping contexts, making changing context have a big penalty. The penalty could be reduced by writing dirty cache data when the bus is idle, but this requires considerable more decisions, which was to much effort and therefore outside of the scope for this thesis.

2. *Automatic assigning of cache blocks to instruction groups that requires extra cache space the most.* At this point the cache configuration and allocation needs to be controlled manually. Manually assigning the cache blocks is possible for single applications, but would be impossible for generic usage. The best possible solution would be an addition that automatically measures the performance of the current program, and assigns the cacheblocks based on the needed cache space. What statistics need to be measured and when to assign cacheblocks to an instruction group provide considerable amount of decisions, which was to much effort and therefore outside of the scope for this thesis.

3. *Explore new possibilities made possible by freely exchangeable cacheblocks* The possibility of reallocating cacheblocks is new and needs to be further explored for applications in other systems. For example, the cache block reconfiguration can be used to let the main processor use the cacheblocks of a data processing core when the latter is idle, increasing the possible cache space for the main processor.

A

Experiment graph results

In this Appendix all the results of the experiments are presented in graphs. All the graphs are relative to the original direct mapped design. This will ensure that one test that is significantly longer compared to the other tests does not completely dominate the results. Some of the tests do not use the cache extensively, but these tests will show if the replacement policies still reduce the number of misses for those applications.

A.1. Cache misses in simulator

The first part of the results are comparisons of the replacement policies in the simulator. All the graphs show the number of cache misses relative to the original direct mapped design.

A.1.1. Original cache structure

In these graphs all the tests are run with the exact same cacheblock structure. This ensures only the effect of the replacement policies are tested.

Data cache misses
In this section the number of misses for the data cache are evaluated. The graph for the Mibench benchmark is depicted in Figure A.1, the graph for the SPEC benchmark is depicted in Figure A.2, and the graph for the Polybench benchmark is depicted in Figure A.3.

Instruction cache misses
In this section the number of misses for the instruction cache are evaluated. The graph for the Mibench benchmark is depicted in Figure A.4, the graph for the SPEC benchmark is depicted in Figure A.5, and the graph for the Polybench benchmark is depicted in Figure A.6.

A.1.2. Added cache tree

In these graphs the extra cache tree is added, which doubles the amount of cacheblocks. This doubles the available amount of total available cache space, which lowers the miss rate. The added cacheblocks are of equal size as the original cache tree. The original design will have double the size of its cacheblocks to compensate for the difference in total cache space.

Data cache misses
In this section the number of misses for the data cache are evaluated. The graph for the Mibench benchmark is depicted in Figure A.7, the graph for the SPEC benchmark is depicted in Figure A.8, and the graph for the Polybench benchmark is depicted in Figure A.9.

Instruction cache misses
In this section the number of misses for the instruction cache are evaluated. The graph for the Mibench benchmark is depicted in Figure A.10, the graph for the SPEC benchmark is depicted in Figure A.11, and the graph for the Polybench benchmark is depicted in Figure A.12.

Figure A.1: Results of the Mibench data cache misses relative to the original direct mapped cache with the same size.

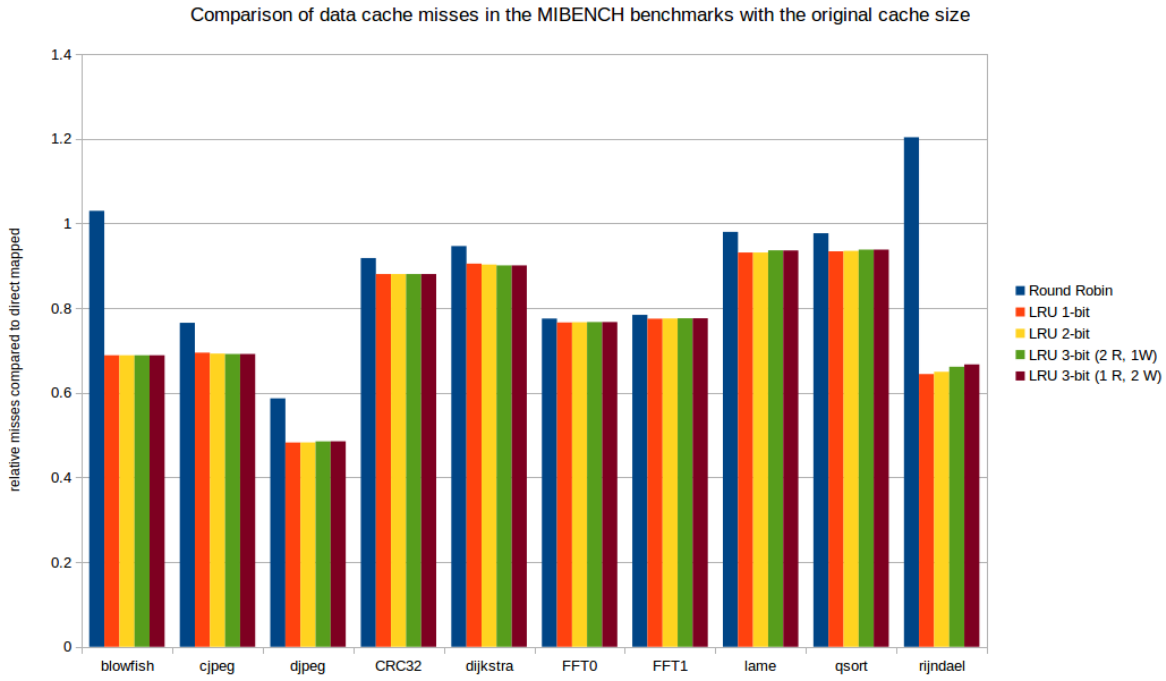
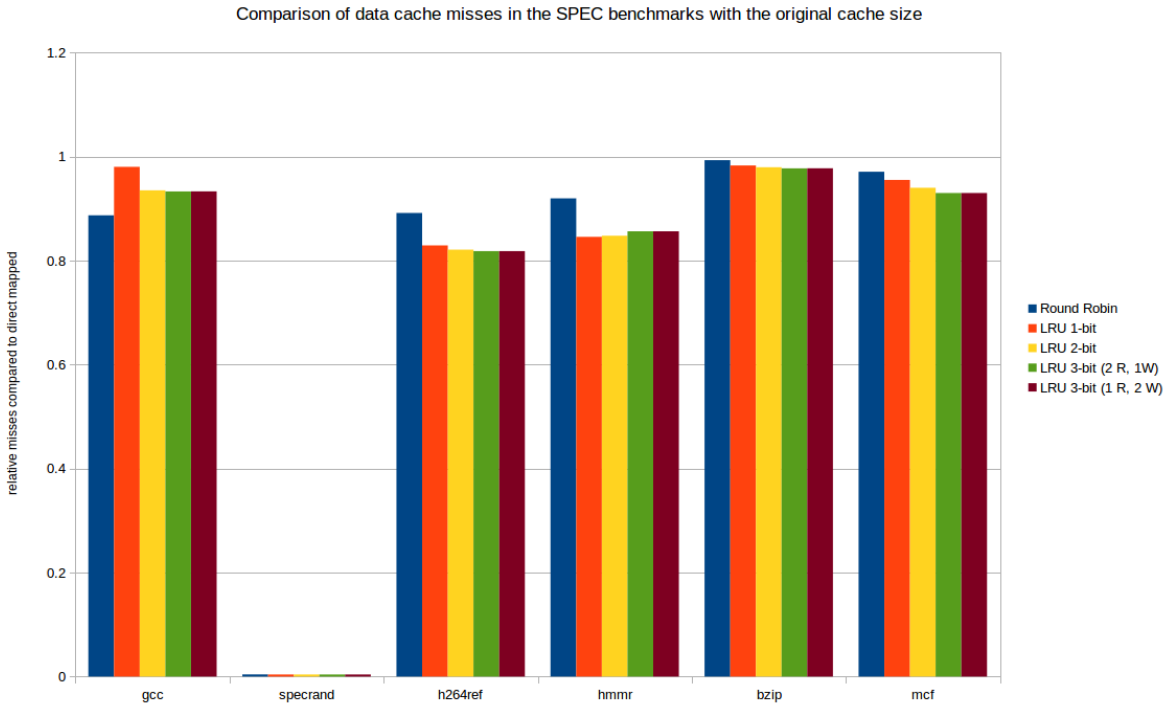


Figure A.2: Results of the SPEC data cache misses relative to the original direct mapped cache with the same size.



A.2. Execution times on hardware

In these graphs the tests are run on the FPGA. All the graphs show the number of used clock cycles needed to complete the program relative to the original direct mapped design.

Figure A.3: Results of the Polybench data cache misses relative to the original direct mapped cache with the same size.

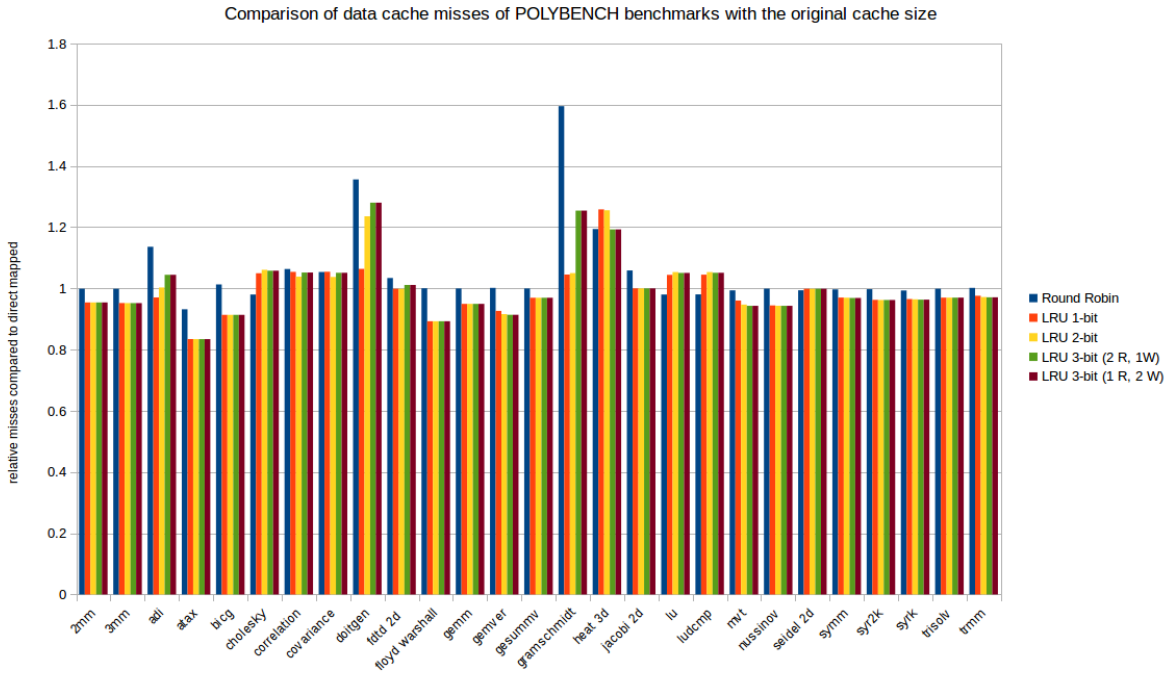
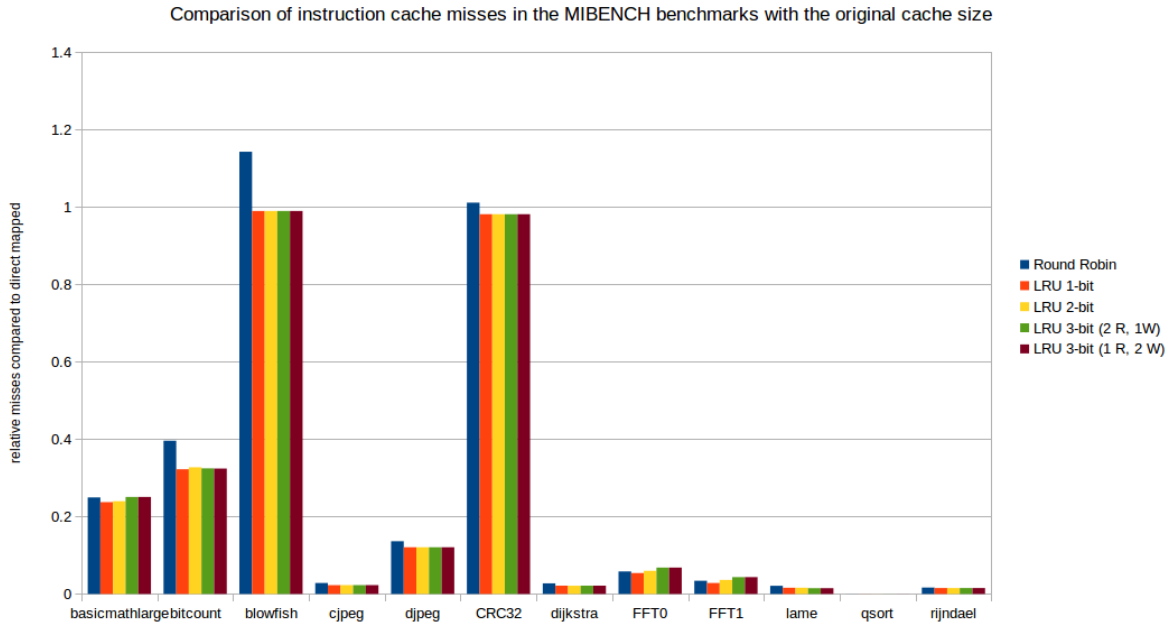


Figure A.4: Results of the Mibench instruction cache misses relative to the original direct mapped cache with the same size.



A.2.1. Equal size

In these graphs all the tests are run with the exact same cacheblock structure. This ensures only the effect of the replacement policies are tested.

In this section the execution times for completing the tests are evaluated. The graph for the Mibench benchmark is depicted in Figure A.13, the graph for the SPEC benchmark is depicted in Figure A.14,

Figure A.5: Results of the SPEC instruction cache misses relative to the original direct mapped cache with the same size.

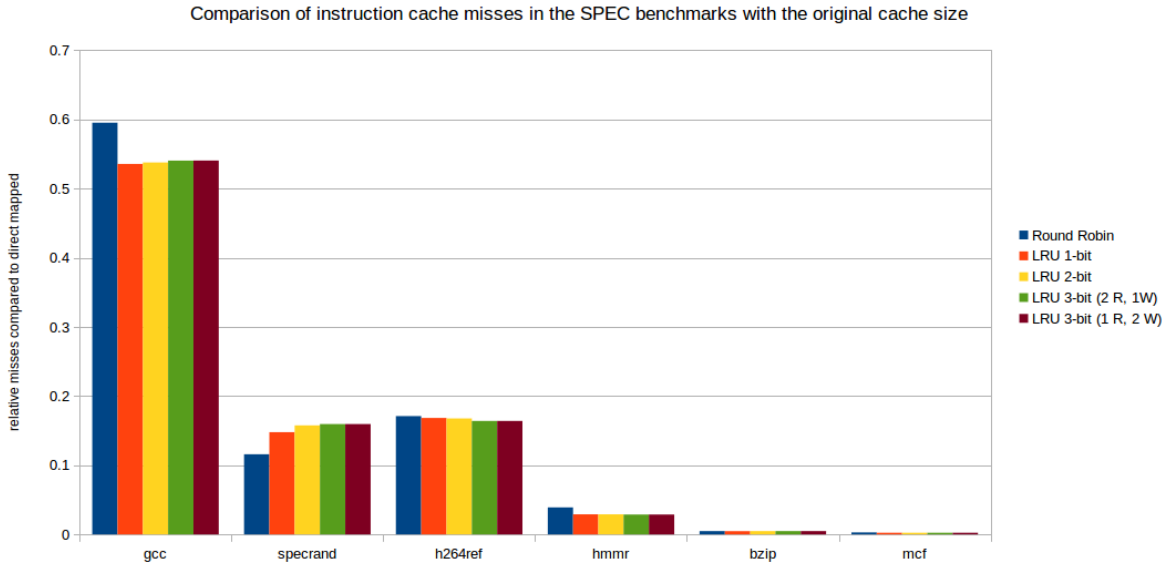
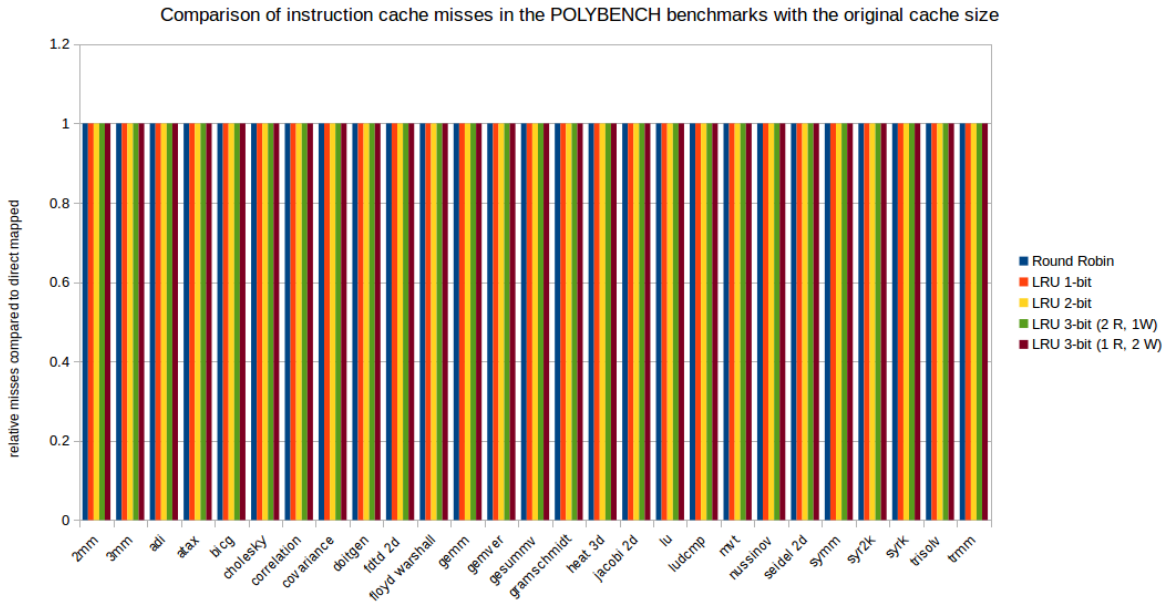


Figure A.6: Results of the Polybench instruction cache misses relative to the original direct mapped cache with the same size.



and the graph for the Polybench benchmark is depicted in Figure A.15.

A.2.2. Added cache tree

In these graphs the extra cache tree is added, which doubles the amount of cacheblocks. This doubles the available amount of total available cache space, which lowers the miss rate. The added cacheblocks are of equal size as the original cache tree. The original design will have double the size of its cacheblocks to compensate for the difference in total cache space.

In this section the execution times for completing the tests are evaluated. The graph for the Mibench benchmark is depicted in Figure A.16, the graph for the SPEC benchmark is depicted in Figure A.17,

Figure A.7: Results of the Mibench data cache misses relative to the original direct mapped cache with the increased size.

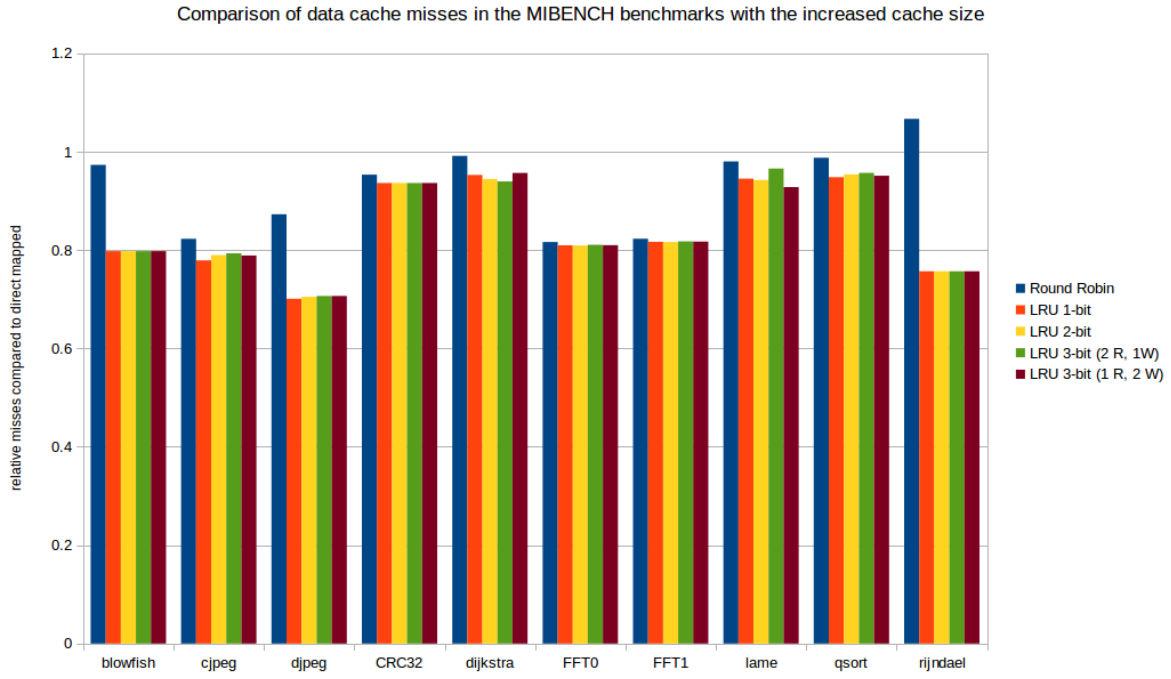
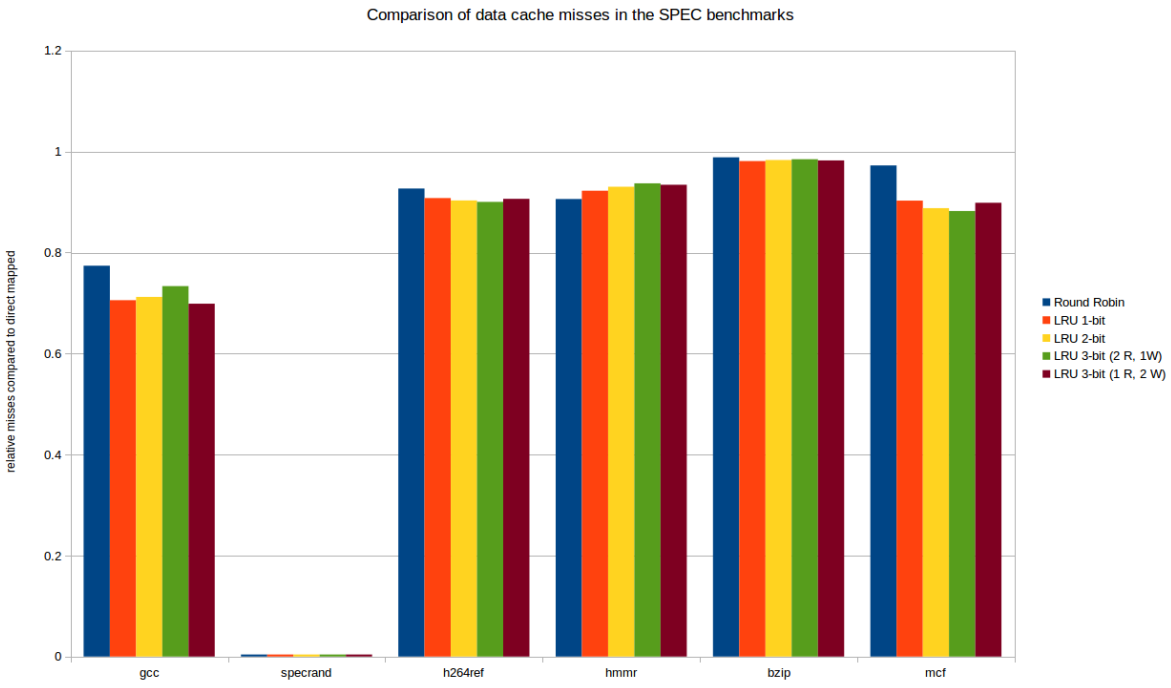


Figure A.8: Results of the SPEC data cache misses relative to the original direct mapped cache with the increased size.



and the graph for the Polybench benchmark is depicted in Figure A.18.

Figure A.9: Results of the Polybench data cache misses relative to the original direct mapped cache with the increased size.

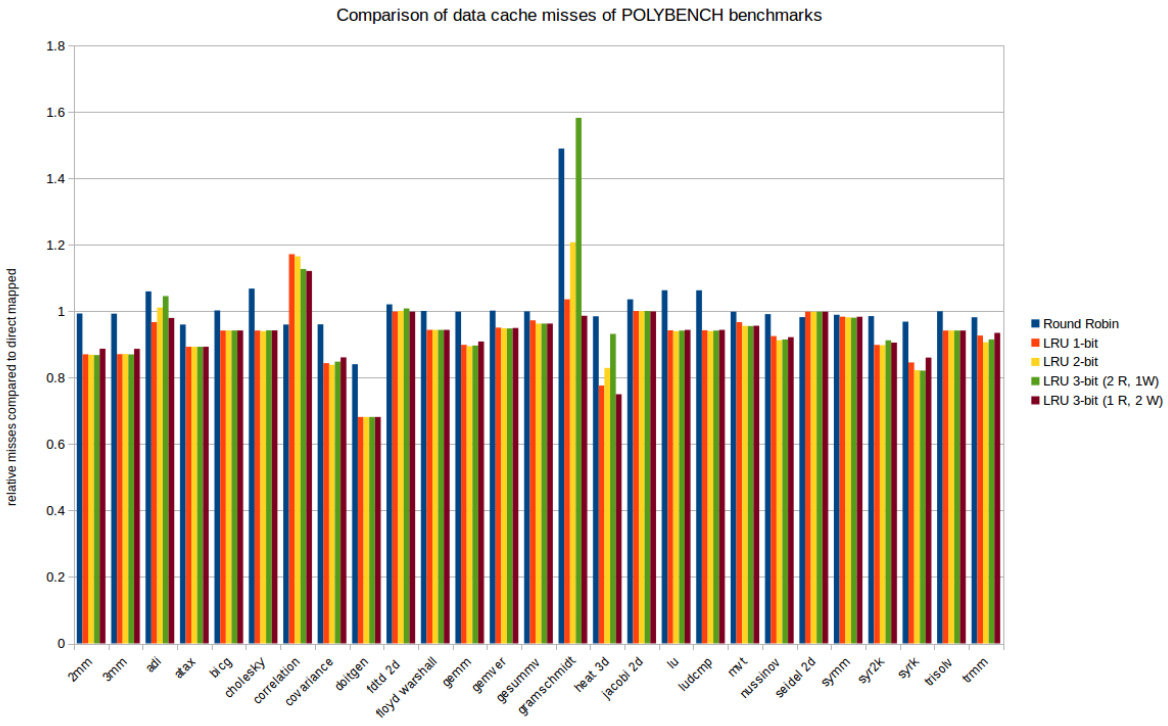


Figure A.10: Results of the Mibench instruction cache misses relative to the original direct mapped cache with the increased size.

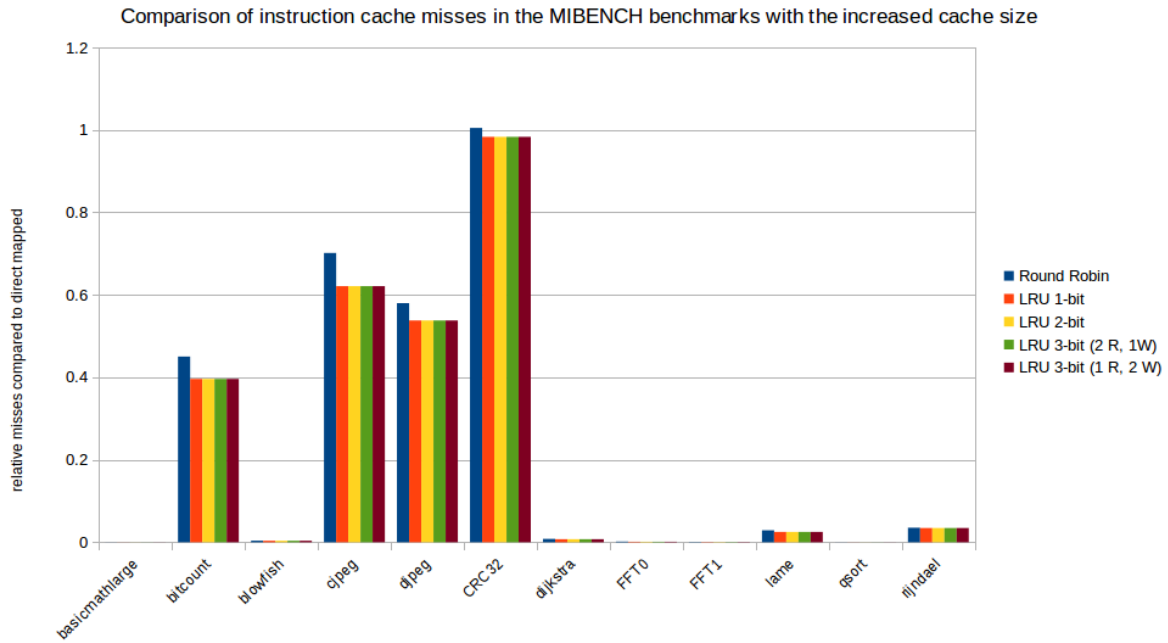


Figure A.11: Results of the SPEC instruction cache misses relative to the original direct mapped cache with the increased size.

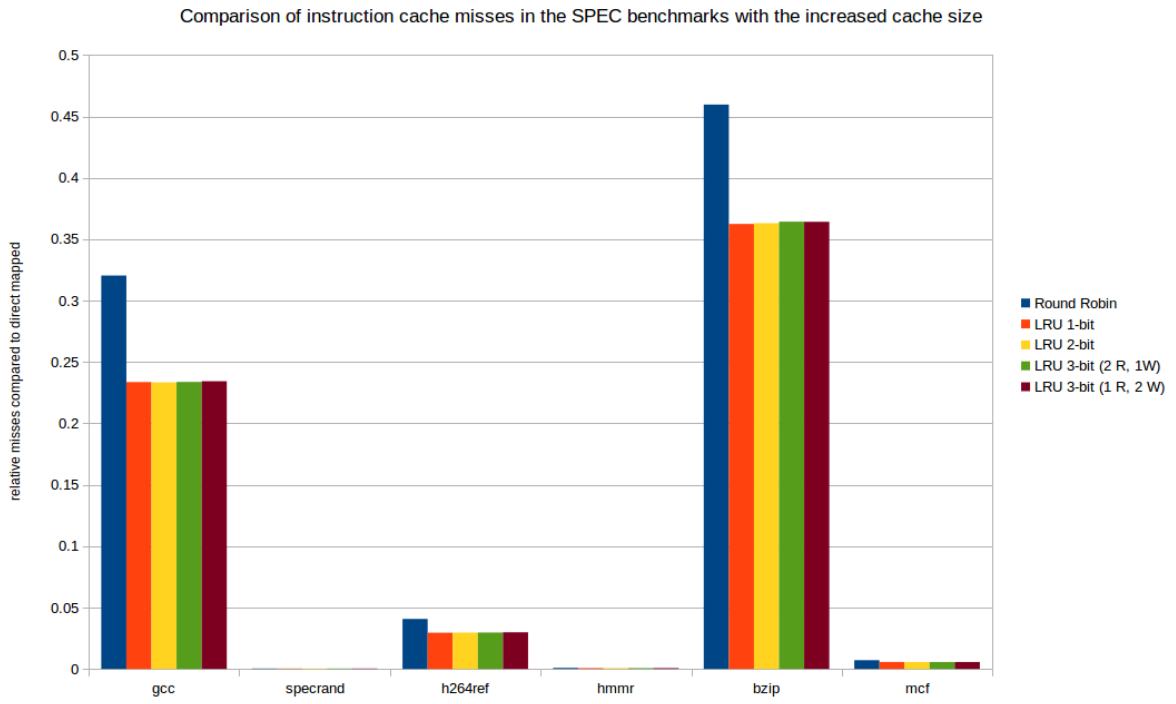


Figure A.12: Results of the Polybench instruction cache misses relative to the original direct mapped cache with the increased size.

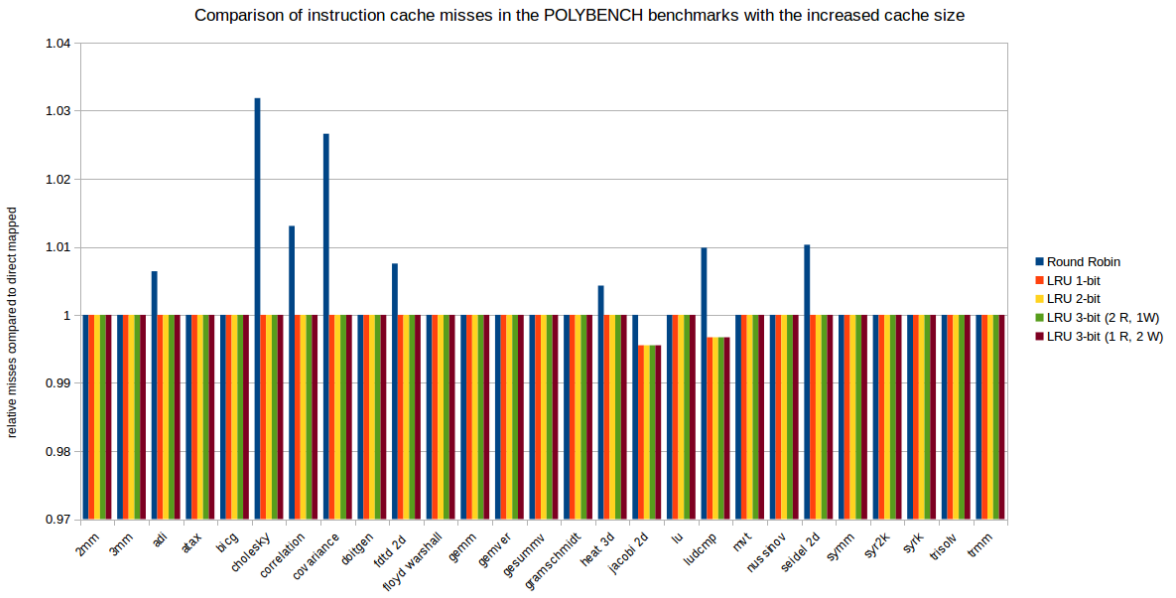


Figure A.13: Results of the Mibench execution times on the hardware with the same size.

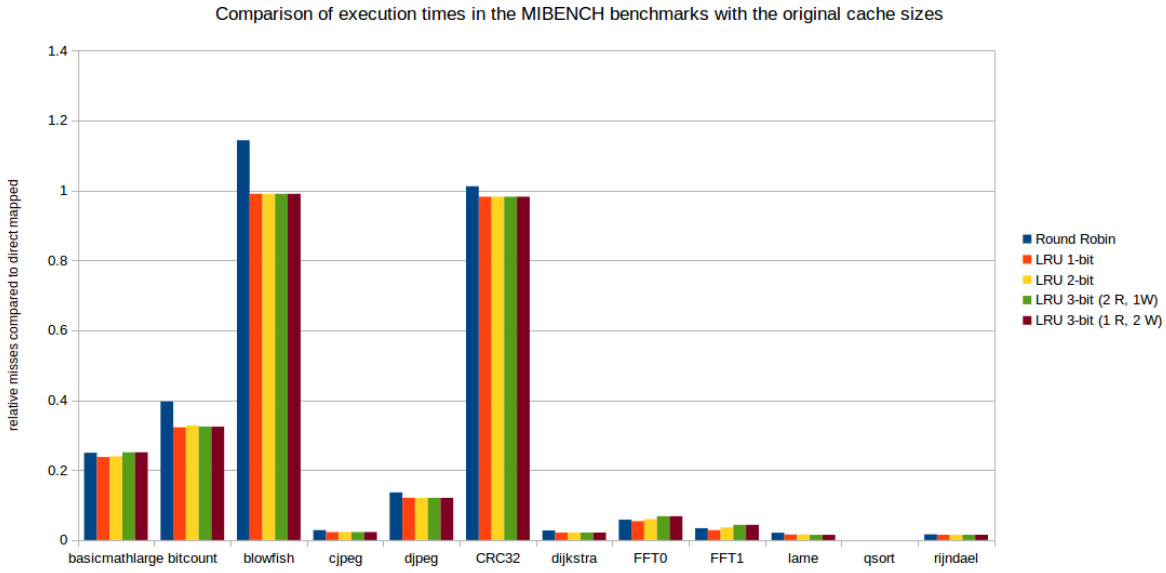


Figure A.14: Results of the SPEC execution times on the hardware with the same size.

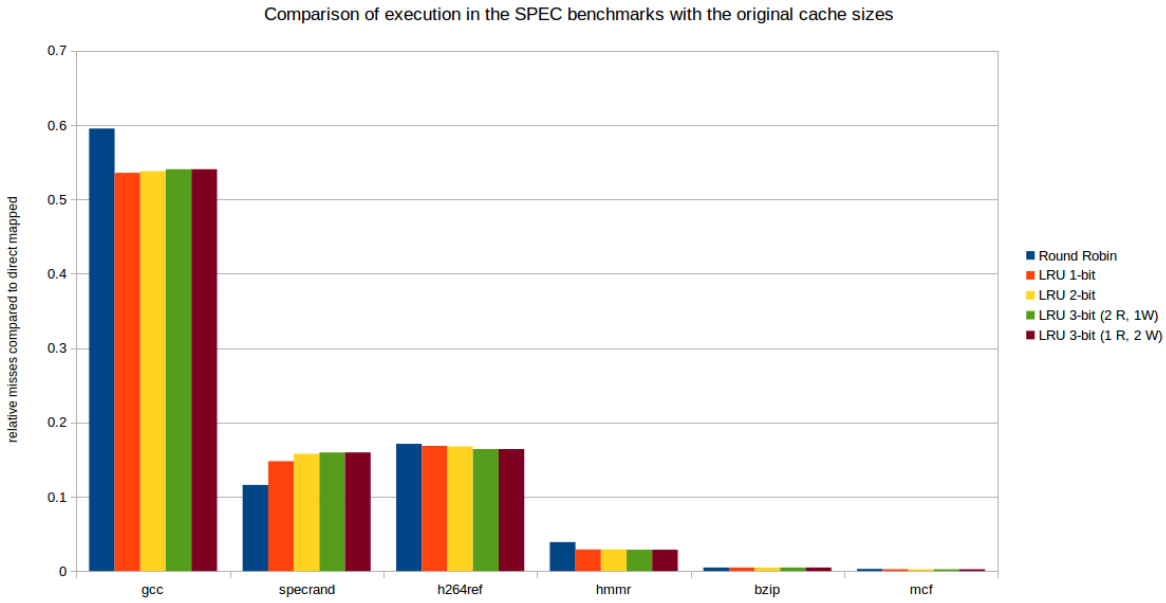


Figure A.15: Results of the Polybench execution times on the hardware with the same size.

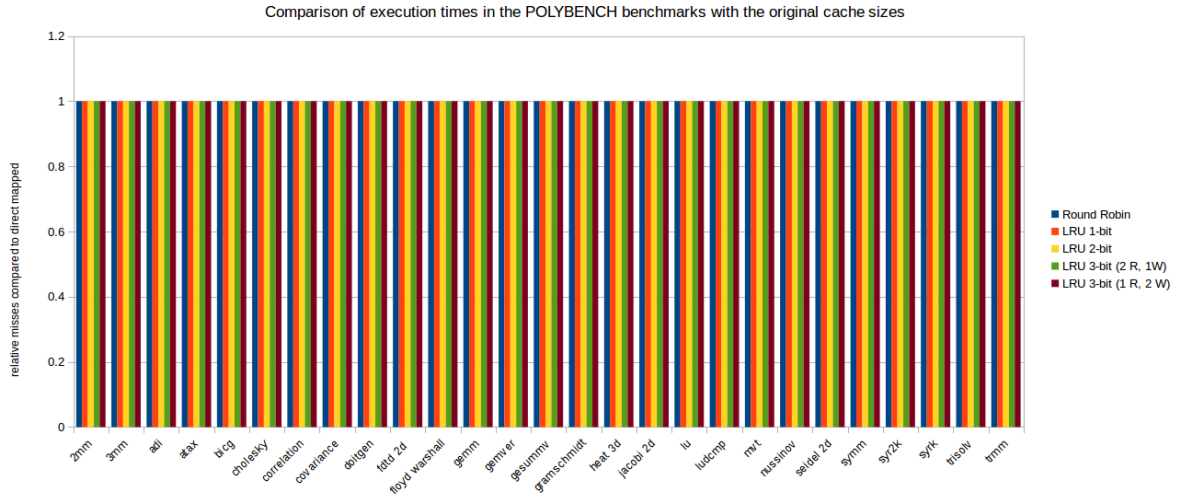


Figure A.16: Results of the Mibench execution times on the hardware with the increased size.

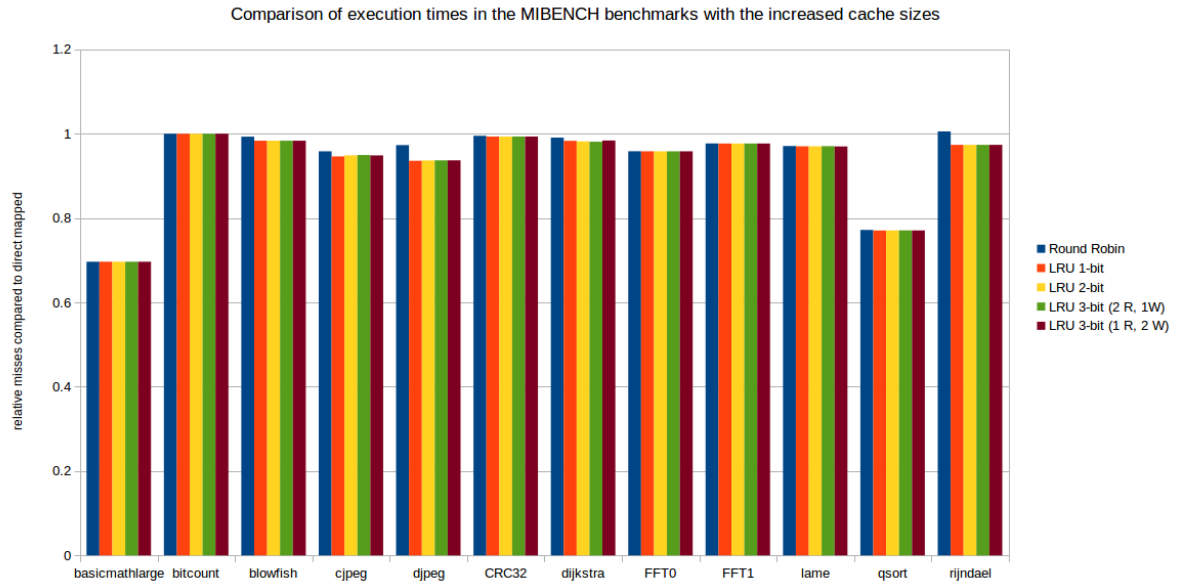


Figure A.17: Results of the SPEC execution times on the hardware with the increased size.

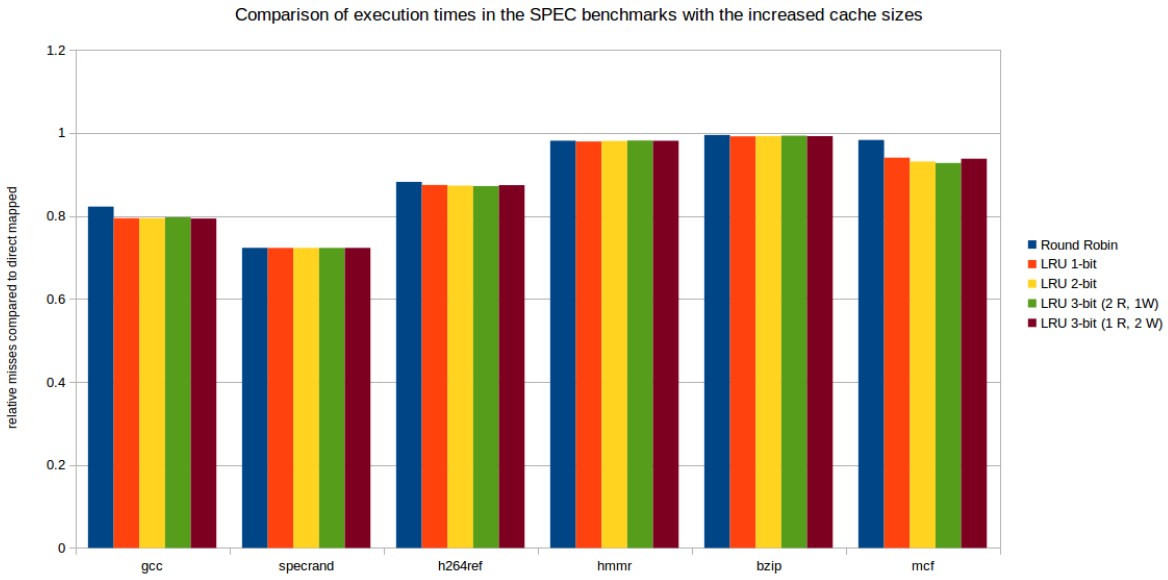
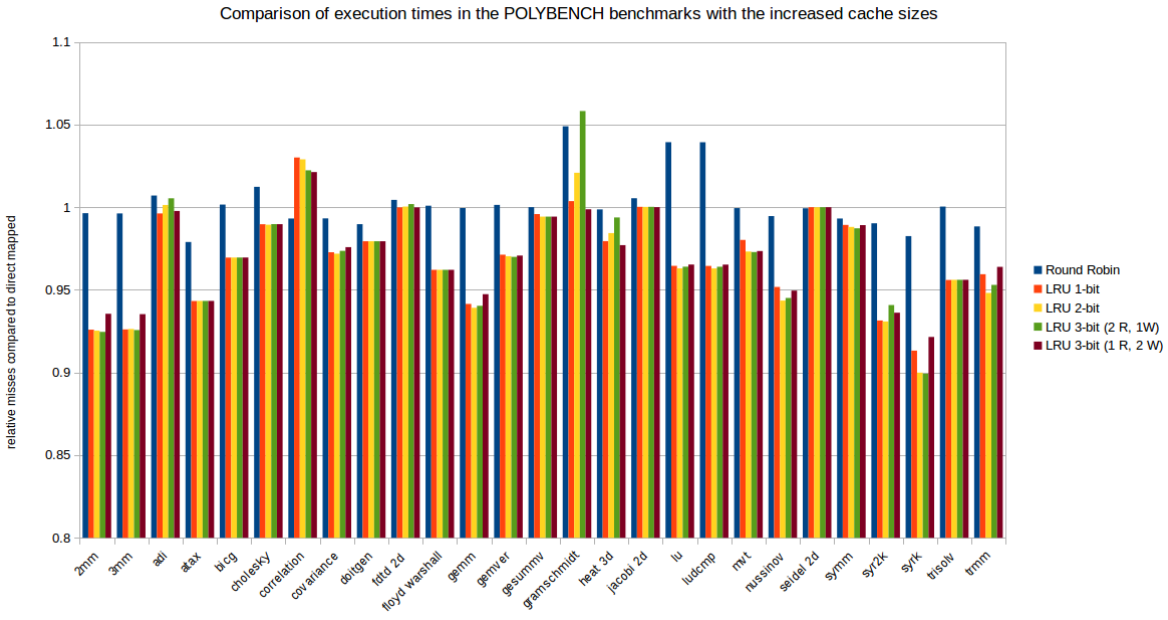


Figure A.18: Results of the Polybench execution times on the hardware with the increased size.



Bibliography

[1] Memory hierarchy. https://classconnection.s3.amazonaws.com/149/flashcards/3088149/png/memory_hierarchy1367201501848.png.

[2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, 1999. doi: 10.1109/MICRO.1999.809463.

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001. doi: 10.1109/WWC.2001.990739.

[4] H. Hanson, M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static energy reduction techniques for microprocessor caches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):303–313, June 2003. ISSN 1063-8210. doi: 10.1109/TVLSI.2003.812370.

[5] John L. Hennesey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 500 Sansome street, Suite 400, San Francisco, CA 94111, 4 edition, 2007. ISBN 9780123704900.

[6] John L. Hennesey and David A. Patterson. *Computer Organization and Design*. Morgan Kaufmann, 500 Sansome street, Suite 400, San Francisco, CA 94111, 3 edition, 2007. ISBN 9780123706065.

[7] J. L. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, 33 (7):28–35, 2000.

[8] S. Kaxiras, Zhigang Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 240–251, 2001. doi: 10.1109/ISCA.2001.937453.

[9] G. Keramidas, C. Datsios, and S. Kaxiras. A framework for efficient cache resizing. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 76–85, July 2012. doi: 10.1109/SAMOS.2012.6404160.

[10] K. T. Sundararajan, T. M. Jones, and N. Topham. Smart cache: A self adaptive cache architecture for energy efficiency. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 41–50, July 2011. doi: 10.1109/SAMOS.2011.6045443.

[11] the Ohio State University. Polybench/c: the polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[12] David A. Wood, Mark D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '91, pages 79–89, New York, NY, USA, 1991. ACM. ISBN 0-89791-392-2. doi: 10.1145/107971.107981. URL http://doi.acm.org/10.1145/107971.107981.

[13] S. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 147–157, 2001. doi: 10.1109/HPCA.2001.903259.

[14] Se-Hyun Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 151–161, Feb 2002. doi: 10.1109/HPCA.2002.995706.

[15] Huiyang Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive mode control: a static-power-efficient cache design. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 61–70, 2001. doi: 10.1109/PACT.2001.953288.