UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Low energy cache memory implementation with data compression

Héctor Romero de Blas

Bachelor Thesis

Specialization in Computer Engineering

Director: Ramon Canal Corretger

GEP tutor: Joan Sardà Ferrer

October 2021

Generalitat de Catalunya
Departament de Recerca i Universitats

Unió Europea
Fons Europeu de Desenvolupament Regional

The DRAC project with -file number 001-P-001723- has been 50% co-financed with € 2,000,000.00 by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020, with the support of Generalitat of Catalonia.

Contents

Abstract

Energy consumption on CPUs is reaching a point where the heat is becoming hard to dissipate, and the temperature is hindering the overall performance of the processors. In addition, the device that feeds the data to the processor, the cache memory, it is getting larger, and the larger it is, the more power it uses.

In order to solve the energy problem, this thesis proposes a new design of compressing cache, the 'Ghost Cache'. The proposal for the 'Ghost Cache' tries to extend another existing cache compressing algorithms in order to compress 3 possible values (0, 1 and -1) in 2 bits. Doing this, we reduced the regular 32-bit data banks of caches, into a 8-bit data bank. We analyzed 5 different algorithms, and we tested them on our cache programmed in Verilog to show which algorithms work better on the cache and which work worse.

Resumen

El consumo de energía en las CPUs ha alcanzado un punto en que dificulta la disipación de calor, y la alta temperatura reduce el rendimiento del procesador. Además, el dispositivo que alimenta la CPU con datos, la memoria caché, está creciendo, y cuanto más crece, más energía consume.

Para solventar ese problema, esta tesis propone un nuevo diseño de caché de compresión, la 'Ghost Cache'. Esta propuesta intenta ampliar otros algoritmos de compresión de cachés para comprimir 3 valores distintos (0, 1 y -1) en 2 bits. De esta manera reducimos un banco de datos de 32 bits a uno de 8. También analizamos 5 algoritmos distintos, y los probamos en nuestra caché programada en Verilog para ver en qué casos hay mejoras en nuestra caché y en cuáles no.

1. Context

1.1. Introduction

In the recent decades, computer technologies had expanded and evolved significantly. Every device, whether it is a computer, a car, or a fridge, has some kind of digital circuit

with a processor. This widespread application of computers creates new requirements, from low-power efficient passively cooled processors, to huge multi core fast processors that require water cooling. Nevertheless, even fast inefficient processors try to expel as low heat as possible, allowing the chip to run faster and last longer.

Recent advances in CPU technology have greatly improved the performance of processors. This increase is due to technological enhancements, such as miniaturization; and architectural advances, such as superscalar[1] and pipelined[2] processors. Processor performance improvements have outpaced memory performance enhancements. This difference in performance made CPUs unable to run at full speed while fed by the main memory, which led to a new memory in the memory hierarchy[3] called the Cache Memory.

Modern cache memories are located inside the CPU die, while old CPU caches were installed on the motherboard. The main advantage of having the cache on-die is that the latency on that cache will be the lowest possible. The major drawback is that the die shares the heat spreading device, meaning that cache heat and CPU heat will be shared, heating both devices and presumably reducing performance and energy efficiency.

1.1.1. The project in the context of the FIB

Historically universities were the pioneers of the Computer Science development. From hardware to operating systems and software, research on colleges always has been on the bleeding edge of technology. In this regards, it makes sense to develop studies about hardware improvements on the hardware department of the 'Facultat d'Informática de Barcelona'.

The FIB always has shown a great concern over the environmental impact of comput-

[1]Superscalar refers to the capability of a CPU of executing more than one instruction on a single cycle, usually implementing parallelism.

[2]Another improvement of CPUs that isolates the hardware inside the CPU in 'steps', which greatly increases the maximum frequency of the CPU.

[3]Memory hierarchy refers to a distinction of computer memories based on their response time and capacity. Modern computers have, from slowest to fastest: Persistent Memory, Main, Cache Level 3, Level 2, Level 1 and CPU Registers.

ers of the modern age, making the topic of energy and resource efficient caches interesting to investigate and solve.

I am a student of the FIB and I am very passionate about RISC-V. The idea of making an open-source computer processor is very appealing as a concept. In addition, the DRAC[4] is a project related to the FIB and the DAC[5], making this project a potentially valuable asset for the development of the European processors.

1.2. Concepts

1.2.1. Memory technologies

We explained on chapter 1.1 the memory hierarchy, but not the reasoning behind it. In essence, memory hierarchy is created by using a diversity of physical storage technologies, with their advantages and disadvantages. The more remarkable technologies are the following:

- Magnetic Storage: one of the oldest persistent memory technologies still used to date. It has the worst latency, due to the need for rotating platters and a magnetic head that moves to read each segment of the platter.

- Flash: flash memory is the other widespread persistent memory technology used today. Either pen-drives or solid state drives are made with one of the two MOSFET[6] flash implementations, NAND or NOR[7] flash.

- DRAM: dynamic random-access memory is a volatile semiconductor memory consisting of a MOS transistor and a capacitor. DRAM memory is the foundation of the main memory, usually referred as the RAM memory of a computer. It is also used on high speed Solid State Drives as a cache, to improve read and write speeds.

[4]Designing RISC-V-based Accelerators for next generation Computers, an European project that uses RISC-V architecture to develop general purpose and high performance processors.

[5]Department of Computer Architecture

[6]Metal-oxide-semiconductor field-effect transistor, a type of transistor made by oxidating a semiconductor, i.e. silicon. It is the most produced type of transistor to date.

[7]NAND or NOR flash describes the type of logical gate used for making the memory.

DRAM chips are used on the main memory because it has great speed and the energy consumption is reasonably low. The distinguishing factor of DRAM memory is that uses capacitors, which need to be cyclically re-charged to hold their value.

- SRAM: static random-access memory is the fastest volatile memory used today. It uses latching technology, which consists usually in 6 transistors that constantly feedback and hold a data value inside the memory cell. SRAM cells use a lot of energy[8] compared to other memory technologies. They also use more transistors than other, making them less dense and more expensive to manufacture. All this caveats turn SRAM a non viable option for massive data storage, and they are the reason why we only find SRAM in the cache and registers of a CPU.

1.2.2. Locality of reference

In order to accelerate memory accesses, applications are characterized. The most common characteristics of all software are considered the 'locality of reference'. This term refers to the tendency of the processor to access certain data repeatedly on short periods of times. The two main types of localities are:

- Temporal: when a memory address is referenced, chances are that same address will be referenced again in a short period of time.

- Spatial: when a memory address is referenced it is likely that another close memory value will be accessed. This is due to data being stored contiguously on the memory, both arrayed data and spare variables, making memory accesses to contiguous memory addresses very common.

[8]Energy consumption can greatly change on SRAM chips depending on how many times they are accessed. The idle energy consumption of a SRAM chip can be negligible, but high performance SRAM uses a lot of energy when accessed at high frequencies.

1.2.3. Cache memory

Cache memory is a small fast memory designed to exploit the 'locality of reference' of computer software. This memory reduces greatly the latency of accessing data from the main memory. It stores the recent accessed memory values on SRAM data cells.

The way cache memory keeps track of which memory entries are valid, and which addresses correspond to a certain cache line is with metadata, usually holding a presence bit, a validity bit, and then storing the address. Also, certain types of caches hold the address in different ways, due to the cache working method.

Another distinction made on the cache memory is the level. There are 3 levels: L1, L2 and L3. The lower the level, the smaller the memory gets. This hierarchy is made to further exploit the 'locality of reference'. Increasing the size of a cache also increases the complexity for accessing the data value, so the smaller it is, the faster it can transmit valid data.

Once a data block gets pushed out of the L1, it gets stored on the L2, and the same happens on the L2 cache. This helps to hold the data close to the CPU instead of sending the blocks directly to main memory. It also increases the overall capacity of the cache memory.

1.2.4. Cache associativity

There are 3 common types of cache memory, determined by their associativity. Associativity is the policy of structuring memory addresses in the cache. In order to enforce this strategy, memory addresses are interpreted by sections. On each policy, sections are delimited differently, but the common idea is using the 2-4 latest bits of the address as an offset[9] and the higher bits of the address to know which data is being stored.

The associativity types are the following:

- Direct mapping: the simplest way to store data in a cache. It can be seen as a single line matrix. The way it uses the lower bits of the address to choose the

[9]Every cache line has multiple bytes, the offset determines which byte of the line represents the address. E.g. if a cache line can hold 4 characters there would be 2 offset bits.

cache line creates conflicts when directions are close[10] and the lowest bits coincide. Hence, this type of cache has the lowest hit-rate[11] of all.

- N-Associative: to keep it simple, an N-associative cache is a set of various small direct mapped caches, that circumvent the limitation of close equal-ending addresses. It does this by having N number of "sets" that target the same addresses. Then, each set is assigned to a different memory space. E.g. 0x0000, 0x4000, 0x8000 can fit on a 3-associative.. The only drawback of this solution is that requires more hardware than a direct cache, as well as a replacement policy of which associative sets should be invalidated and reused. However, 8-associative caches are the more common nowadays due their advantages (better hit-rate) weighted against their complexity.

- Fully Associative: a fully associative cache holds the whole "Tag", meaning that the whole address (except the final offset bits) are stored in the line metadata, making this cache capable of holding any data, as close or far apart they might be. The major issue is the replacement mechanism being very complex and time consuming, as well as energy consuming. It also requires more hardware, making it the most expensive of the three, and it does not give a more consistent hit-rate on all scenarios.

1.2.5. RISC and RISC-V

There are two main philosophies into designing the instruction set[12], CISC and RISC.

Complex Instruction Set Computers emphasize the importance of dedicated hardware to create a lot of instructions. CPUs created with CISC designs tend to be very complex, high power, high performance processors. The advantage over RISC is that having more

[10] If the cache holds 'n' bytes, each 'n' addresses will coincide in line

[11] Hit-rate means how many times the demanded value is in cache over how many times it is not

[12] Abstract model that represents each operation executed in a processor as an 'instruction', that describes the effects and the required operands for an operation. It also describes data types, registers and inner workings of the processor, and Input/Output relevant information.

instructions, one for each job, creates less bloated and faster software.

On the other hand, Reduced Instruction Set Computers take the opposite idea. It uses a smaller instruction set, with as little number of instructions as possible, to keep the hardware simple. This creates small, energy efficient processors that usually rely on accelerators[13] for making complex calculations. This processors rely more on software implementations to compensate for the lacking instructions, which create larger codes.

However, as superior as CISC might seem, usually the instructions present on common software are in CISC as well as in RISC, and the lacking instructions are compensated with the accelerators as aforementioned. In addition, RISC CPUs are simpler in design, and are more power efficient. All of those reasons explain the recent rise of RISC architectures like ARM and RISC-V.

RISC-V[11] is a free and open source RISC instruction set. It is becoming a widespread architecture due to it being free, open source, and having implementations intended to run on FPGAs[14].

1.3. Problem to be solved

The problem presented in this Degree Final Project is the excess of heat production and energy consumption on modern caches. As stated on chapter 1.1, nowadays cache memory shares die with the main processor of the computer. This creates inherent heat and power transfer between them, decreasing power efficiency and performance due to thermal throttling[15].

To put in context the type of energy consumption CPUs work with, we show this graph. In it we can observe how CPU improvements increase power density in processors

[13]Small hardware connected directly to the CPU that processes complex operations with a narrow purpose, e.g. a video decoder or a cryptography encoder

[14]Field Programmable Gate Array, a technology that allows to configure chips architecturally, in opposition to regular chips that are only configurable once, when they are built.

[15]Reduction in CPU frequency due to overheating. One of the limitations for CPUs is temperature. Although $130°C$ is the theoretical limit, CPUs do not exceed $100°C$, and they dynamically reduce the frequency to keep the temperature under the threshold.

until it got close to the power density of a nuclear reactor.

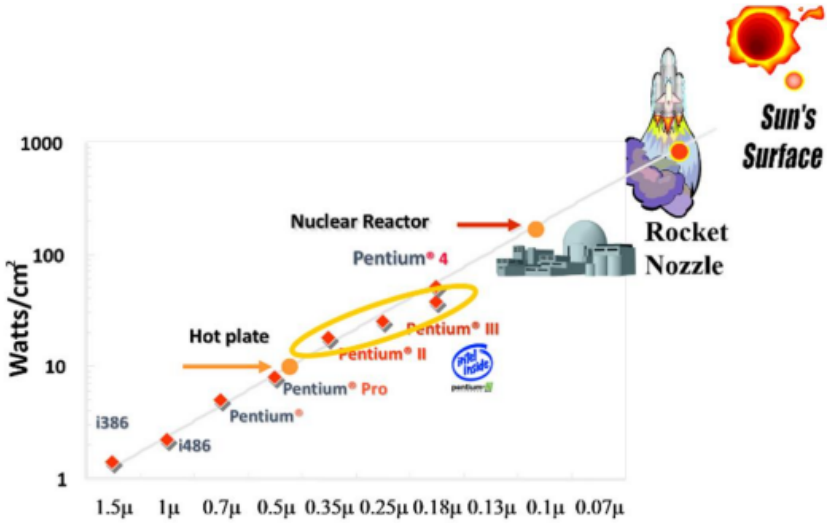

Figure 1: Power density of CPUs

[5]

As shown in figure 1, if the increasing trend continued, CPUs could have reached power densities on excess of nuclear reactors by the $10nm$ lithography[16]. However, this never happened. CPU manofacturers started taking another approach for increasing the performance of CPUs: they increased the number of processing cores and the IPC[17] rather than increasing frequency.

1.3.1. Stakeholders

The stakeholders involved in this project are the director and the researcher. The director of this project is Ramon Canal, associate professor and former Vice Dean of post-graduate studies at the 'Facultat d'Informatica de Barcelona'. The researcher, Héctor Romero de Blas, will research, design, document and implement a cache with the aid and approval of the director.

This project benefits the RISC-V community, as well as the cache designing community. It gives access to the study and design of a working compressing cache that can

[16]Term describing the size of the transistors of a given chip.

[17]Instructions per Clock, meaning how many instructions can perform a CPU on each clock cycle.

improve energy efficiency compared to the current designs.

1.4. Justification

1.4.1. Previous studies

As stated on chapter 1.3, in the recent years there have been grand efforts to increase power efficiency of the computers, due to the impossibility of augmenting their performance with previous methods.

There are a lot of new studies that propose some kind of energy optimization hardware-wise, such as the Dynamic zero compression for cache energy reduction[13], or the Significance Compression[8] papers. We will analyze these papers on chapter 2, with proper evaluations to state why we cannot implement exactly these options, but we can extract the main ideas.

As a summary, Dynamic zero compression requires alterations of the transistor design at a level that our tools would not allow it. Significance Compression also falls out of our scope, due to requiring a general alteration of the inner workings of the CPU, not only the cache.

The most implementable idea is the Zero-Contents Augmented Cache[2], and our proposal will be heavily influenced by this option.

1.4.2. Our target

The main problem with all the theories aforementioned is that they have very limited applications, relying on the presence of zeroes on memory. This cache memories are not appealing to implement due to the small benefits they give compared to the complexity to implement.

Our goal is to improve this design, making compression caches more useful, thus more appealing for the grand market CPU manufacturers.

1.5. Scope

1.5.1. Objectives

The main objective of this project is to propose a compressing cache that reduces energy consumption compared to current caches. This project can be divided as a theoretical part and a practical part.

- The theoretical part will be an analysis of the currently proposed methods to create a compressing cache, and then present our conclusions and the best option to implement. We must provide a thorough description of our proposed option, with diagrams, schematics, etc.

- The practical part will be implementing a working prototype of cache in Verilog[18]. For experimental purposes, we also want to include the cache implementation on a RISC-V CPU, accommodating the data interfaces to the standard for RISC-V 32bit.

1.5.2. Potential problems

- Time constraints. The theoretical part is doable in the provided time, but the practical part can be very time consuming. The complexity of including a new design in a CPU can be hard, because we will need to understand the way the CPU is built by the tool-chain, and then include the new part.

- Inclusion in a CPU. As stated before, the inclusion of the cache on a working CPU can be challenging on itself, not only on time, but on difficulty. We will need to change the way Verilog inserts data in a simulation of a third-party developer.

- Failing to the premises. If our cache does not reduce energy reduction, or it reduces it at the expense of a lot of performance, then our design does not meet

[18]Verilog is a hardware description language. It is similar to a programming language, but it does not compile to an executable file. Instead, other tools known as synthesis tools create a circuit as described by Verilog

the expectations. This kind of conclusion is not uncommon on research projects, meaning that not all researches end up with positive outcomes. The project will document all the point where it failed and why.

1.5.3. Requirements

In this section we will enumerate the functional and non-functional requirements of the project.

- Functional requirements:
  - Compress data.
  - Decompress data.
  - Interact with the processor.
- Non-functional requirements:
  - Compression and decompression without performance compromises.
  - Data integrity of the compressed bytes.
  - Capacity size and capabilities of industry standard caches.

1.6. Methodology and rigor

1.6.1. Methodology

The methodology that I will use is the Scrum framework. The core concept is to incrementally build the project, from basic ideas all the way up to the complex final product. It accepts that the problem cannot be comprehended fully upfront, thus focusing the efforts on the emerging requirements.

1.6.2. Validation

The main target of the practical part is to validate the design of the theoretical part. When we implement the concept proposed on the Project we will be able to gather

valuable information about how successful is the design.

The practical part will be executed and validated on a Verilog Simulator, that interprets the code, correcting for errors, and then simulates the whole design.

2. Related Work

2.1. Dynamic Zero Compression

One of the methods of cache compression is Dynamic Zero Compression [13]. This method aims to reduce energy consumption on the cache when reading or writting zero-bytes. DZC is a very effective method due to the prevalence of zero bytes present in caches. According to the DZC report's abstract[13, p.1], this method is capable of reducing cache energy consumption by 26% on the data cache and 10% on the instruction cache.

Implementing the DZC requieres making a few hardware changes to existing devices on the cache. One of those changes is adding additional bits to the cache to represent the presence of a zero-byte. This bit is called the ZIB (Zero Indicator Bit). 4 ZIBs are needed in total to represent each 32bit number. Other changes require modifications in the CPU store data driver, the Word Line Gating or the addition of a ZIB sense amplifier to the cache.

There are two problems when implementing this method:

2.1.1. Verilog's Switch Level Modeling

The changes aforementioned require modifications at the Switch Level, wich means designing circuitry at the transistor level. For instance, the Word Line Gating changes that prevent bitline discharge when reading zeros require adding PMOS and NMOS transistors directly to the SRAM cells that make each bit of the cache.

Although Verilog is suited for Switch Level Modeling, Verilog's approach makes designing this changes very difficult. Verilog allows NMOS and PMOS transistors to be added as devices in every module, but Verilog doesn't have basic things requiered in SLM like VDD / GND cable systems. This is due to the design philosophy of Verilog. The language is suposed to describe the workings of hardware on a more abstract way (precisely at the Register-Transfer Level), and power, grounding and other things inherent to SLM design are delegated to the Synthesis tools.

The way Verilog expects you to describe the memory is to use the keyword "reg", wich defines a variable. Then the implementation of that variable depends on the interpreter that reads the Verilog file.

All the avobe suggests that Verilog is not the best tool suited when working with SLM, and will hinder the developement of our cache. In fact, it is very uncommon to work in SLM in Verilog in the industry of Hardware Design. SLM seems to be implemented on Verilog as a tool to test certain circuits, but not for final production.

2.1.2. RISC-V implementation of the DZC cache on Verilog

At the current times, RISC-V has gained a lot of popularity due to it's use on FPGAs. RISC-V provides a variety of CPU designs: simplified and minimalistic cores, cores with extensions such as FPUs or MMUs... This versatility combined with the reduction of costs of FPGAs are the reasons why RISC-V is getting this popularity.

Having this in mind, the implementation on this paper should take into account the feasibility of the cache with the current technologies.

FPGA boards usally come with an SRAM module. This is because creating an SRAM in an FPGA is impossible. FPGAs work at a Logic Gate level, whereas in order to create an SRAM cell you need to work at a transistor level. This is a major roadblock on the implementation of DZC.

Regardless of FPGA limitations, having a SLM modified SRAM cell integrated on a non SLM CPU seems challenging at the synthesis level. Conecting the power signals to those generated by the synthesizer is a very complex task, or even impossible.

This two problems render this DZC approach non-viable.

2.2. Significance Compression

Another promising compression method is Significance Compression. The paper[8] studied in this section reports a reduction of activity 30-40% for each pipeline. The main idea conveyed in this paper is to reduce inter-stage data traffic at all stages of the pipeline using "Significance Compression". The study focuses on the reduction of

dynamic energy consumption, due to it being the primary energy consumption of the current CMOS transistor technology.

The compression proposed is basically sending only the significant bits through data lines. The paper explains various encoding methods for data representation[8, p.2]. The report proposes using a granularity of byte for simplicity, as well as adding two extension bits to mark how many extension bytes are. The final proposition is a three-bit extension scheme with more edge cases covered, and with only a 9% overhead. This compression would be applied to data, as well as cache tags.

One important point on this paper is that the methods proposed reduce energy consumption at the expense of performance. It is not forseen for our proposed compressed cache to reduce performance in a significant manner. The introduction of the report[8, p.1] states that 30-40% of activity can be reduced at the pipelines provoking a 79% of CPI(cycles per instruction) on the most basic implementation, but wider pipelines can get only a 2-6% penalty.

Chapters 2.3[8, p.3] and 2.6[8, p.5] are the Instruction Cache and Data Cache proposals.

- The Instruction Cache requires storing the instructions in permuted forms. Unfortunately, the report is made for MIPS ISA, and the methods described for permutations are designed for that architecture. Due to our proposal using RISC-V architectures, other permuting methods are required. Finding those methods would require further research.

- The Data Cache uses the three-bit extension to reduce drastically the activity of the cache. A few benchmarks show a reduction of 31% for the data array and 1% to the tag array.

The only major drawback for this implementation is that our cache optimizations aim to affect only the cache itself. We want a cache that can be incorporated on any design of RISC-V 32-bit processor. In order to make the optimizations described on this paper, the pipelines, data buses and other interfaces should be changed in size, as well as

adding aditional hardware to compress and uncompress at all stages (or at least at the cache input, output and ALU/REGFILE stages). Nevertheless, this paper make great proposals, such as the three-bit scheme, that will be added to our final design after a small redesign to suite our purposes.

2.3. Zero-Contents Augmented Caches

Yet another twist to exploiting the presence of zeroes in the cache. The Zero-Content Augmented Cache, or ZCA, consists of a convenional cache and a specialized cache for memorizing null blocks, the Zero-Content Cache. The paper of this proposal claims that on applications manipulating large amounts of null data blocks, ZC reduce up to 81% the miss rate and memory traffic, specially the write-back traffic[2, p.4]. It also claims to not create any performance loss for applications with low null block rate.

The ZCA is a very simple solution at a Register-Transfer Level, wich makes it very interesting for our purposes. One advantage of this method is that we can create huge Cache memories without using as many transistors as in regular Caches. This would reduce the energy consumed by the cache.

The biggest problem of this cache is that the application of this theory works best only in the presence of huge data structures with a lot of consecutive null values. Another problem is that this solution is heavily intended for Data Cache, meaning that this method is innefective on Instruction Caches. It is not as common to find null values on Instruction Caches.

The last problem, common to all zero-based compression, is the fact that the ZC is a "Read-Only" cache. When a non-zero value is stored in the ZC, the block gets invalidated. This requires additional hardware to synchronize the two caches and to mantain the coherence of both. A potential issue of this hardware is that, for instance, if zero-values are read and put into the ZC, they can be immediately modified, thus invalidating the block, requiring a new block on the regular cache. This new block could also invalidate recent data of the regular cache. If this "worst case scenario" is prevalent, the ZCA could be a very slow cache. This problem will be addressed on our proposal.

2.4. Discarded compression methods

2.4.1. Adaptive Cache Compression for High-Performance Processors

The Adaptive Cache Compression for High-Performance Processors paper[1] proposes an interesting method of storing compressed lines on the L1 with a 2:1 compression ratio, thus doubling the L1 capacity.

The main problem on this cache is the reliance on the L2 cache. The L2 is the cache who tracks whether compression is viable or not in a given application. The final implementation of the cache on this paper does not have a compressed L1, it has a compressed L2 instead. All the aforementioned renders this solution not suitable for our proposal.

2.4.2. Compresso: Pragmatic Main Memory Compression

The Compresso[4] method relies on Memory Compression, rather than Cache compression. The reason why we analyzed this paper was to determinate if the compressed data could be copied from the main memory to the cache still compressed. It seems that the three main compression methods (Frequent Pattern Compression, Lempel-Ziv and Base-Delta-Immediate) would require major cache modifications to accommodate the compressed data, making this solution also non viable. It would need more SRAM cells to hold the dictionaries, or additional hardware for decoding the $B\Delta I$[3], which could severely increase the time of cache hit. $B\Delta I$ compression is also a dynamic compression, which could compress or not compress, depending on the values of the data.

3. Novel proposal: Ghost Cache

From chapter 2 we gathered some ideas that can be implemented in a single cache, further improving the design of the compressed cache. This cache will have to implement compression at a high level, as stated in chapter 2.1. It will also exploit the characterization of software, for instance, the presence of zero-values.

Due to the Register-Transfer level implementation, our plan for reducing energy consumption is to maintain the capacity of the cache while reducing the number of transistors in it, with a method similar to the chapter 2.3.

Our Proposed method is the Ghost Cache, an 8-Way Associative Data Cache with 4 blocks of regular Cache memory, and 4 "Ghost Caches". A "Ghost Cache" is a Cache with a smaller memory module tied to it. The size of this memory will be of 2 bits per byte, adding into 8 total bits to represent a 32 bit integer. Having 2 bits per byte adds granularity enough to support the usage of 8 bit, 16 bit and 32 bit numbers.

This proposal would be similar to the chapter 2.3, but instead of only storing Null values, the GC can also store another value, either 1 or -1.

3.1. Physical layout

Our cache will have three distinct parts: the logical circuitry, the storage device and the interfaces.

The logical circuitry will compute the necessary data to detect hits in cache memory, assessing the next state, calculating the next victim for the replacement policy and other housekeeping calculations for running the cache microcode.

The storage device is divided into two main banks: compressed and non-compressed. There is a tag, a validity bit and a data array on each bank. The main difference is that on the non-compressed bank, the bank holds 32 bits, whereas the compressed one holds 8 bits.

Interfaces on this cache are compatible with the cv32e40p LSU (Load Store Unit).

The following figure shows the diagram of the cache:

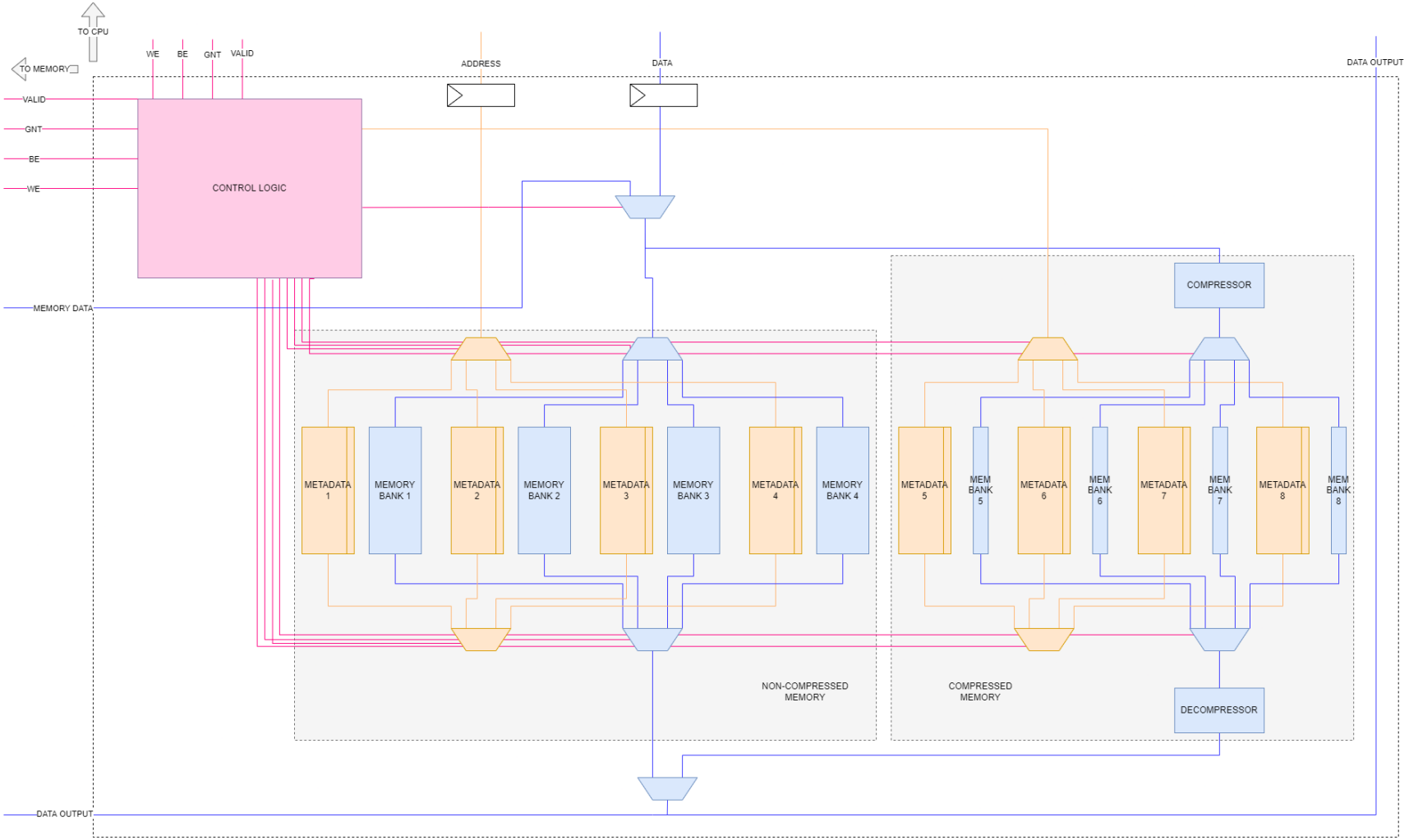

Figure 2: Cache design diagram (Own compilation)

3.2. Design decisions

In order to justify the decisions taken in the making of the design of the cache, we have to take into account the philosophy we applied: *keep it simple*.

The best way to not over-complicate our design is to understand the purpose of our cache. This cache will demonstrate the effectiveness of our compression algorithm on the L1 cache. It will not be the fastest, nor the most memory-cache bus efficient. By having a simple design, we will be able to try some changes in replacement policies or other parts of the algorithm relatively easy.

In addition, the first iteration of the cache will allocate all the compressible data on the compressible data banks, and vice-versa. This is in order to analyze the effectiveness of the compression, how much data can be compressed, and other statistics.

4. Compression

Our compression algorithm will allow 3 different 8 bit values to be compressed into 2 bits:

- 0b00: This corresponds to a 0x00.
- 0b01: This corresponds to a 0x01.
- 0b10: This corresponds to a 0xFF.

With this granularity of compression we allow various things, one of them being compressing an array of booleans. Arrays of booleans are widely used in a large variety of algorithms. Furthermore, optimization techniques, such as *memorization*, usually relies on boolean arrays. In chapter 6 we will describe two algorithms that represent best scenario use-cases for our cache.

In case of being a C like language, where a *boolean* uses 32 bits instead of using 8 bits, it will default to use too much memory. However, our compression also allows *bools* at 8 bits, which means that manually specifying a *char/unsigned char* based structure would be optimal.

5. Replacement policy

5.1. Least Frequently Used

The replacement policy used on our test cache is the Least Frequently Used (LFU). The main reason why we use this replacement algorithm is its simplicity waged against its effectiveness. We want a rather simple algorithm in order to implement the logic for compression on top of the replacement logic, and then testing its efficiency. Our main target of this project is to find out weather compression is useful or not in a cache, so using a complex replacement policy is not necessary.

The implementation of our LFU is made as follows:

- Each cache line has an 8 bit access counter.
- When a cache line is accessed, the corresponding access counter gets increased by one.
- When a cache block is taken from memory, the access counter is reset to 1.
- Every cycle, the LFU logic calculates which is the least frequently used associative block. It does this by using the address sent by the CPU. We have a separate LFU minimum index for the non compressed cache, and another for the compressed.

Although a replacement policy is not strictly necessary on a n-associative cache, we included one to further improve the hit probability and to simplify the process of memory fragmentation when we have an array with mixed compressible and non-compressible data. We want to allow the case of an array being at two different associative blocks at the same time, but this could mean an expulsion at the compressed side or on the non-compressed. Thus, allowing the usage of other associative blocks, using LFU rather than the fixed index logic, would in theory yield better performance.

5.2. Compression replacement policy

Due to our cache having 4 lanes of non-compressed data and 4 lanes of compressed data, we need to establish some special behaviour between them. More specifically, we need to discuss what to do when a compressed memory value changes to a non-compressible, and vice versa.

On the case of a compressed data that becomes non-compressible, we will use the non-compressed LFU victim index to replace the least used value on the non-compressible cache data banks. We will also liberate the compressed block, setting the valid bit to zero. The reason behind this is that we need to keep the most recent value of the data. If the value goes to a non-compressible value, it needs to still be on the cache.

On the reverse case, a non-compressible data that becomes compressible, we will not move it on the compressed cache. The reasoning behind it is that if the value was non-compressible, chances are the data type being stored is coincidentally compressible, and will become non-compressible again. In order to save cycles and possible expulsions, we will not swap from cache banks.

5.3. Improved cache

After the initial tests with the algorithms described at chapter 6, it was clear that cache size would hinder greatly performance metrics. After a brief discussion, we increased the cache line number, from 64 lines / associative block to 1024, making it a 32KB L1 cache. This puts the overall capacity of the cache up to industry standard.

The change was relatively simple due to the use of *parameters* and *defines*[19]. Once all the ranges were recalculated (e.g sectioning the address to create the tag, index and offset).

[19]Both parameters and defines are Verilog mechanisms to set numeric constants, to then apply on the code, e.g. the number of lines or the range of bits that should take from an input

6. Testing Algorithms

6.1. Test methodology

During the whole development for the cache, we used Icarus Verilog as our test platform. The reason why we used Icarus is because it is faster to write small changes, and Icarus uses the verilog language to program the tests. For testing if it works or does not work, it is a great tool. It is also very useful when you are making constant changes.

However, the testing requirements changed when we wanted to test algorithms. We needed a more programming focused language to develop the algorithms and then testing the cache.

6.1.1. Verilator

In order to accommodate the new requirements, we changed our testbench software from Icarus Verilog to Verilator. Verilator creates a representation in C++[20] of your designs written in verilog. This will allow us to program any algorithm, and then use our cache as memory in order to get the statistics.

All the information regarding the algorithms, as well as the pseudocodes[21] come from Wikipedia[6], GeeksforGeeks[10] and Rosetta Code[12]. From these pseudocodes we elaborated codes that interacted with the cache. We made helper functions to manage all the memory interactions with the Verilator generated cache simulation model, as well as implemented a basic stack-based memory management system in order to allocate dynamic memory on some algorithms that required it. These helper functions are the ones that also analyzed and categorized all the values that transited the memory bus, and the Verilator cycle clock gave us the number of clocks passed for all interactions.

[20] A low level general programming language used primarily to make efficient software.

[21] A pseudocode is a description step by step of an algorithm written in plain English instead of a programming language.

6.1.2. Statistics counters

In order to obtain the statistics, we added some hardware counters, and we modified the cache in order to write to those counters according to what happened. We have a counter for:

- Hit counter: counts how many hits are in the non compressed cache side. We consider a hit as every memory access with the validity bit to one. This means that this block was red from the memory on a prior access.
- Miss counter: count how many misses are in the non compressed cache side. We consider a miss as every memory access with the validity bit to zero. This translates to reading a new address as well as writing to a new address.
- Chit counter: counts how many hits are in the compressed cache side.
- Cmiss counter: counts how many misses are in the compressed cache side.
- Swap counter: counts how many times we change a compressed value to the non compressed side of the cache due to the data changing from a compressible value to a non compressible value.
- SwapEx counter: counts how many swaps occur that produce a block expulsion on the non compressed cache.

In addition, we want to further explain how the cache determines whether a value is compressible or not. When we are writing an *int* to the memory, we are able to know if we can compress it in situ. However, in the case of writing a *byte* or a *short*, it might not be as straight forward. If what you are writing is not compressible, then it will not be compressible at all. However, if it is compressible, we need to ask to the main memory for the rest of the data block, to check if it is still compressible or not. This means that if we write a compressible data, but the block is not compressible, we will count this as a *non compressible miss*. On reads, we also need to wait for the memory to know if it is a *cmiss* or a *miss*.

6.2. Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm for calculating the prime numbers between 2 and a given value (n). The name of the algorithm alludes it's creator, Eratosthenes of Cyrene, and it is dated around the 3rd century BC.

This algorithm is often used as a quick comparison benchmark, due to the memory complexity and ease of implementation. Furthermore, it is a very flexible algorithm, being able to benefit from things like lazy evaluation, making it more valuable in benchmarking optimizations.

The time complexity of the classical sieve of Eratosthenes is around $O(n \log \log n)$, although there are CPU optimizations. However, the biggest drawback for this algorithm is memory complexity. It needs to hold the entire range of numbers that are going to be used[22] on a Boolean array. Some memory optimization techniques allow the reduction on memory complexity to $O(\frac{\sqrt{n}}{\log n})$.

This memory requirement makes our cache a very appealing solution. It can double the effective cache storing capacity without doubling the physical cache resources. This will effectively double the cache hit probability.

6.2.1. Statistics with first prototype cache

The following plots show the statistics of the cache while executing the Sieve of Eratosthenes:

[22] Although the lazy evaluated version might help to mitigate this, it will hold until the largest used value, which could lead to the same scenario as the non lazy evaluated.

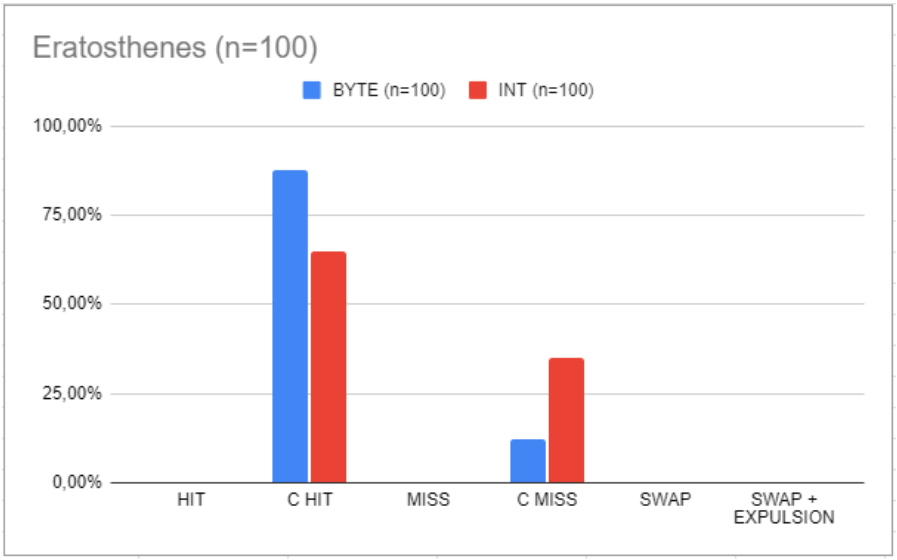

Figure 3: Eratosthenes with N=100 (own compilation)

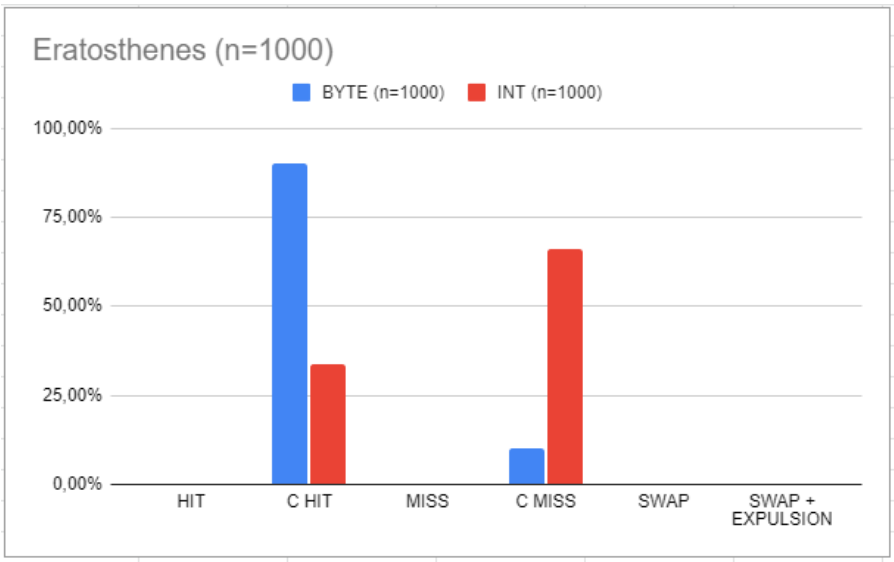

Figure 4: Eratosthenes with N=1000 (own compilation)

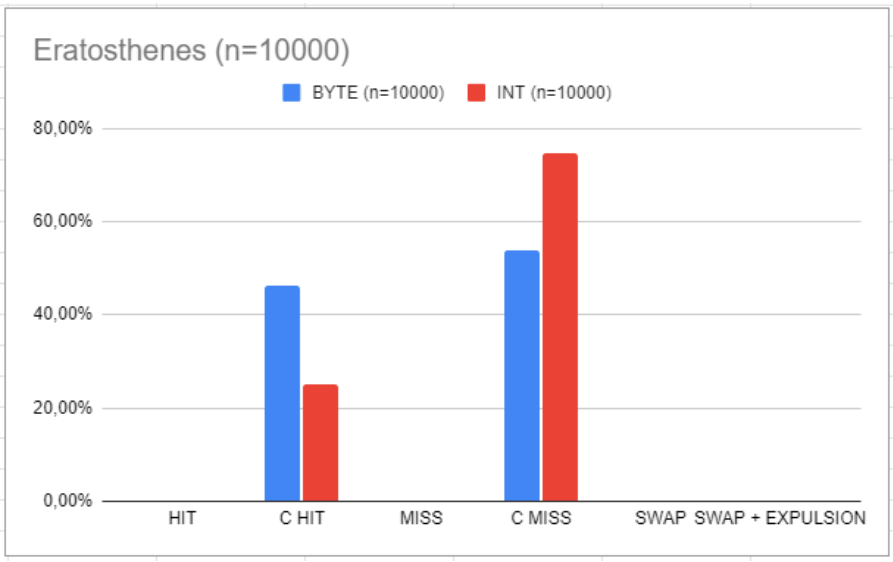

Figure 5: Eratosthenes with N=10000 (own compilation)

On the first plot, figure 3, we can observe a very interesting behaviour. The hit rate stays exactly equal on both *byte* and *int* variant, but misses are a lot higher on the *int* side. This is due to the Erathostenes sieve invoking a *calloc* system call (a system call[23] [24] that allocates dynamic memory[25] with all values initialized at 0). When we allocate memory and set it to zero, we miss every time we bring a block from memory[26]. However, if we use a *byte* structure, we allocate less int blocks ($\frac{n}{4}$ because each *int* contains 4 *bytes*), meaning that we miss less. If on the other hand we reserve int blocks, we will use effectively 4 times more memory (n). This is clearly shown on figure 3 where

[23]System calls are functions that demand resources to the operating system, such as memory or access to a device like the screen or the keyboard.

[24]It must be noted that the *calloc* implementation on our cache writes *ints* in order to be more efficient.

[25]A type of memory reserved during a program's execution specially design to be versatile on size.

[26]In our cache design, we do not bring a block from memory if we write a *int* as an optimization. This is because every line holds an *int* and if we write an *int* we will not lose any data. The only reason for bringing data from memory is when you write a *byte* or a *short*. You do not want to loose the data adjacent to the written data, thus you need to take from memory the rest of the block. Nonetheless this will not affect our statistics because we count this interaction always as a miss. If we do not have a block valid on the cache, it is a miss after all.

we have 25 misses for *byte* and 100 misses for *int*. The rest of accesses hit on the cache because on the process of initializing everything at zero, we brought the data to the cache and it is all present due to the size of the cache fitting all the 100 lines.

Both figure 4 and 5 show a flaw on our prototype cache, the size. When n grows, the miss rate grows very quickly. This also corresponds to the algorithm accessing very disperse memory locations on a wider range of addresses. It was at this point on testing that, after consulting with the director, we agreed to increase the memory cache size to 1024 lines per associative block. The rest of the tests will be performed on the improved version of the cache.

This two figures also show the benefit of using *byte* over *int*. As expected, using a data type 4 times smaller than another yields a better hit rate, and a lower overall memory access due to the aforementioned behaviour of *calloc*.

Regardless of the size, we can see that the compressed cache can hold all the data used on the Eratosthenes sieve, making it 100% efficient on our cache. In order to increase the hit rate, we should either make the cache larger, or consider holding compressible data on the non-compressible side of the cache.

The following plot will show the data values that were sent between the cache and the CPU while executing the algorithm:

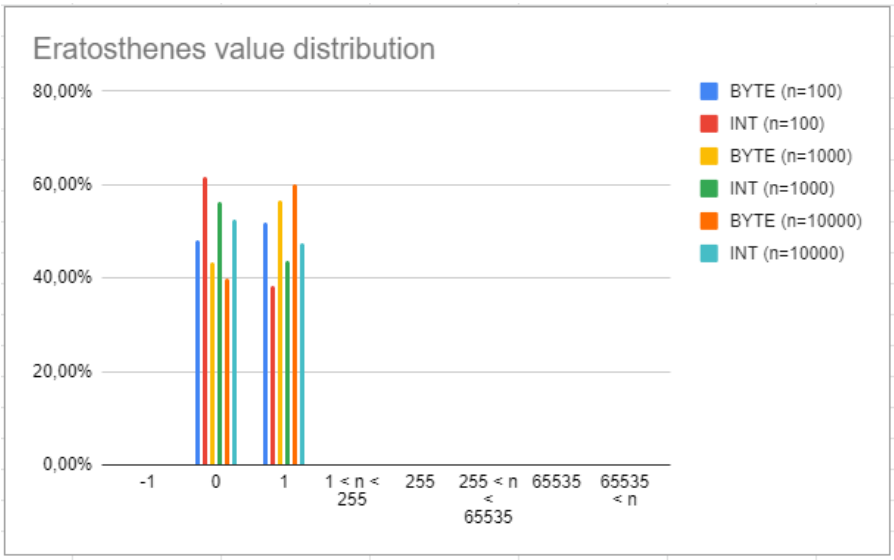

Figure 6: Eratosthenes data values (Own compilation)

As expected, the algorithm only uses a 0 and a 1 as values in a Boolean array.

6.2.2. Statistics with improved cache

The following plots show the statistics of the improved cache with 1024 lines per associative block while executing the Sieve of Eratosthenes:

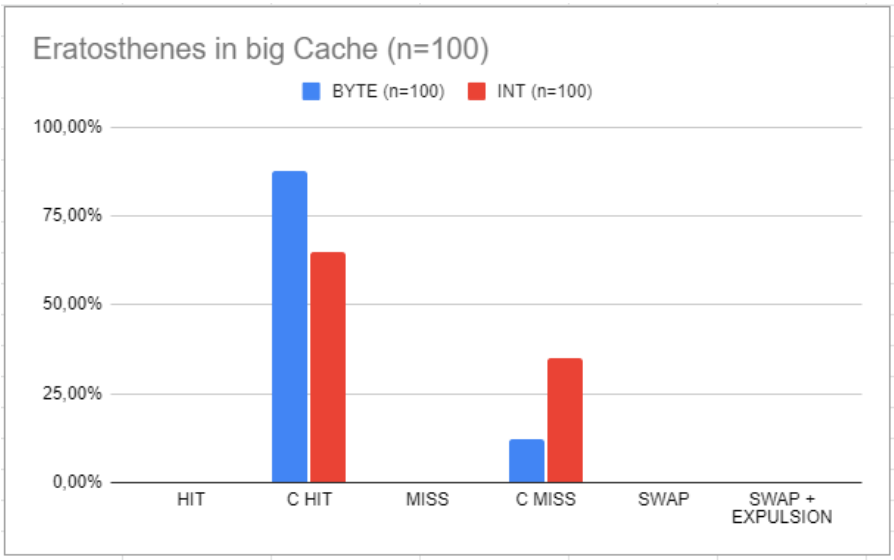

Figure 7: Eratosthenes with N=100 (own compilation)

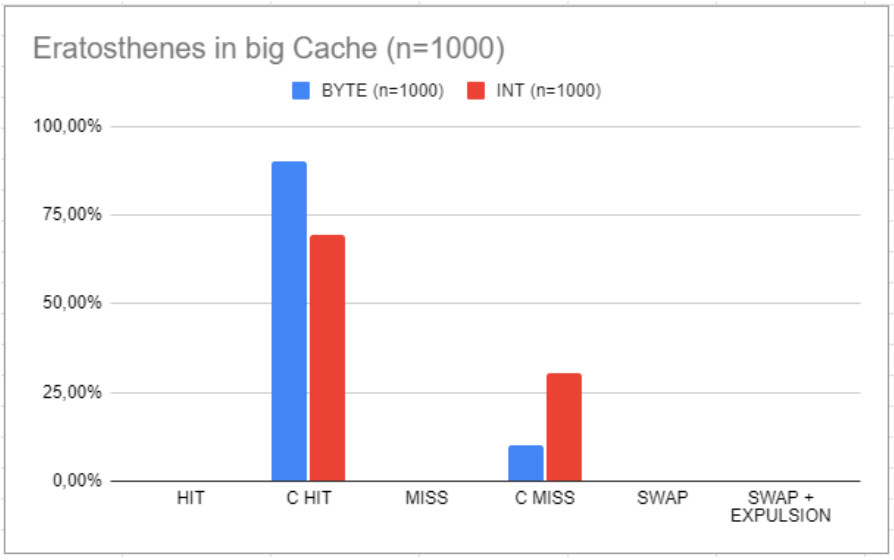

Figure 8: Eratosthenes with N=1000 (own compilation)

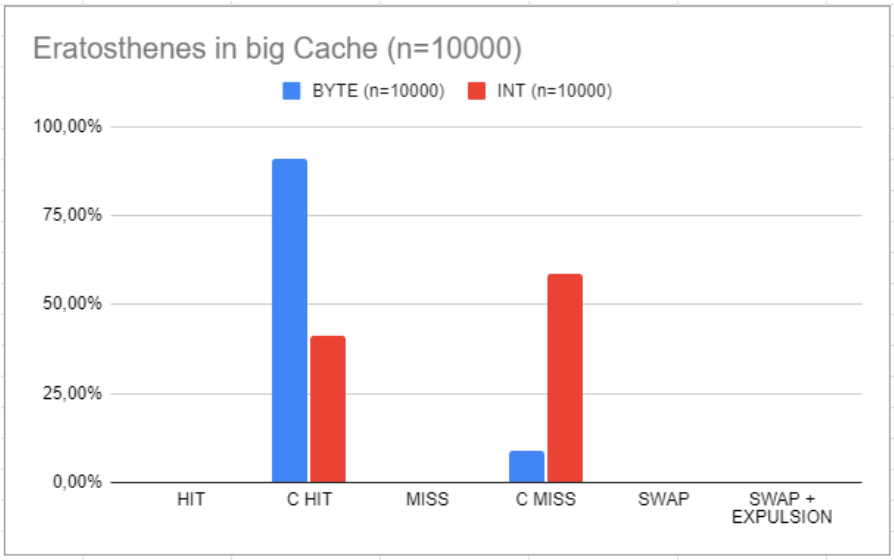

Figure 9: Eratosthenes with N=10000 (own compilation)

With the new cache design, we expected to see a better hit rate compared to the old version of cache. While examining the results, we can see that on $n = 100$ have the same exact statistics (figure 3 and figure 7). This is an expected outcome, because for small n values, both caches should behave exactly the same.

On figures 4 and 8 however, we see a new difference. The hit rate is much better on

the new cache, while in the old cache is approximately 48.7% worse. Consequently, we can deduce that our cache resizing was successful, and that our new cache will be able to handle bigger workloads.

Lastly, figure 9 shows a big improvement on the *byte* test and not a not so big improvement on the *int* over the results in figure 5. This is expected due to the *int* test exceeding the overall cache size. With these tests we established that our new cache is better performing than the old one, and that the interaction between main memory stills intact, meaning that data does not get corrupted on memory transactions.

6.2.3. Statistics without *calloc*

The Eratosthenes algorithm requires a memory region that guarantees all values to be 0. On a regular computer, this region is obtained using the aforementioned *calloc* system call. However, on our testing methodology we control the initial state of the cache, and the current setup is to have all the memory starting at 0. This means that we could theoretically execute directly Eratosthenes without the system call.

Nonetheless, we do not value the following test as a valid, representative performance metric of our cache on Eratosthenes, but rather we value it as a test to weather our theory about the *calloc* creating the difference in *C MISS* on the different tests (theory described at the big paragraph on figure 6.2.1). We think that it is very important the complete understanding of the results in order to reach correct conclusions. Thus we need to check if our guesses and the results coincide.

The following two figures show the tests without a *calloc*:

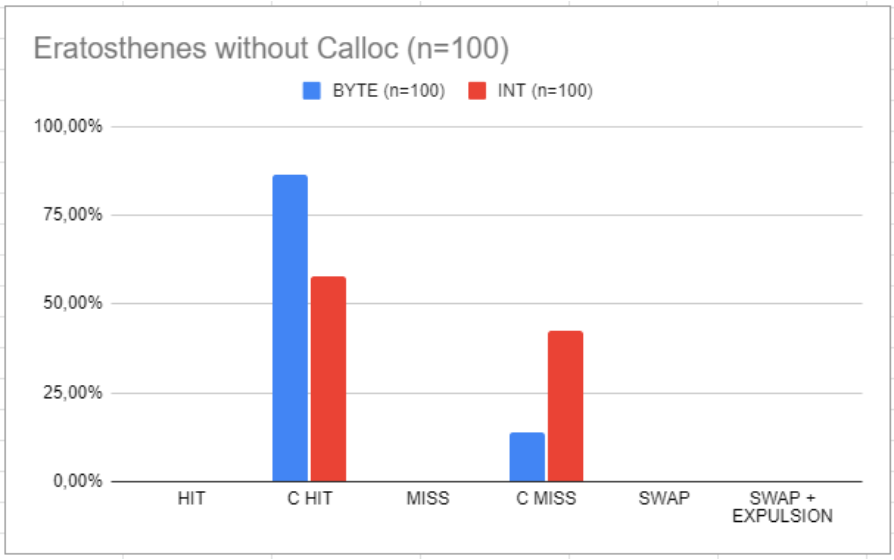

Figure 10: Eratosthenes with N=100 (own compilation)

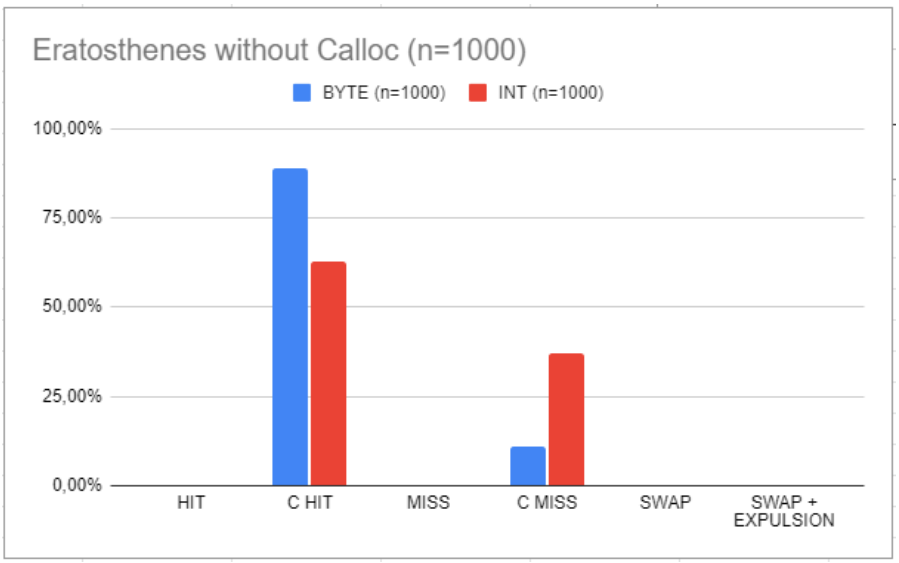

Figure 11: Eratosthenes with N=1000 (own compilation)

Both figure 10 and 11 show the exact same proportion of hit and miss for each data type. Thus, we find the *calloc* to be the culprit in the discrepancy of misses on the prior tests.

6.3. Trémaux's algorithm

The Trémaux's algorithm is a maze solving algorithm that relies on marks. It is an efficient maze solving method that guarantees to give a path for all well defined mazes, although it might not be the shortest path possible.

The reason why we mention this algorithm is that it uses 2 possible marks, meaning that the array won't be necessarily of *bools*, rather some data type that allows 3 different numbers, a *short* for instance. This method proves the improved versatility of our cache compression policy. You could implement this algorithm with two arrays of *bools* instead of one of *shorts*, but this would hinder performance. Our cache is pseudo[27] 8-way associative, meaning that the CPU can access 8 different *bool* arrays at the same time, but having two arrays would effectively double the amount of memory accesses, which would also double the amount of cache misses, thus increasing the overall energy consumption of the algorithm.

Another reason is that the Trémaux algorithm could be interpreted as a more complex version of a broad-first-search or a deepth-first-search algorithm[28]. This is because the Trémaux algorithm shares the core principle of marking visited corridors, but it uses a three state matrix instead of a boolean matrix, increasing efficiency of finding an exit at the expense of using more memory than the BFS or DFS. Thus, this algorithm is representing also a worst case scenario of the performance expected from those BFS and DFS, due to more memory complexity.

This algorithm works as follows:

- You never go through a passage with two marks.
- You mark once every path you follow. This mark is at every end of the passage, meaning that you will have a mark on the entrance and another on the exit of the passage.

[27] It's only 8 way associative if the data used is compressible and non-compressible at the same time by the algorithm, otherwise it is effectively 4 way associative, and stores half of the capacity.

[28] BFS and DFS are both pathfind algorithms that usually use a boolean array to check whether a given position has been visited or not.

- If you find a junction without marks, choose a passage and mark it.
- Else if the path that lead you to an intersection with all marked entries, you turn around, marking again your used path, making it a two mark entrance.
- Else if that intersection has free entries instead, traverse one arbitrarily.

When the algorithm reaches the exit, the *only marked once paths* will give the valid path from the entrance to the exit.

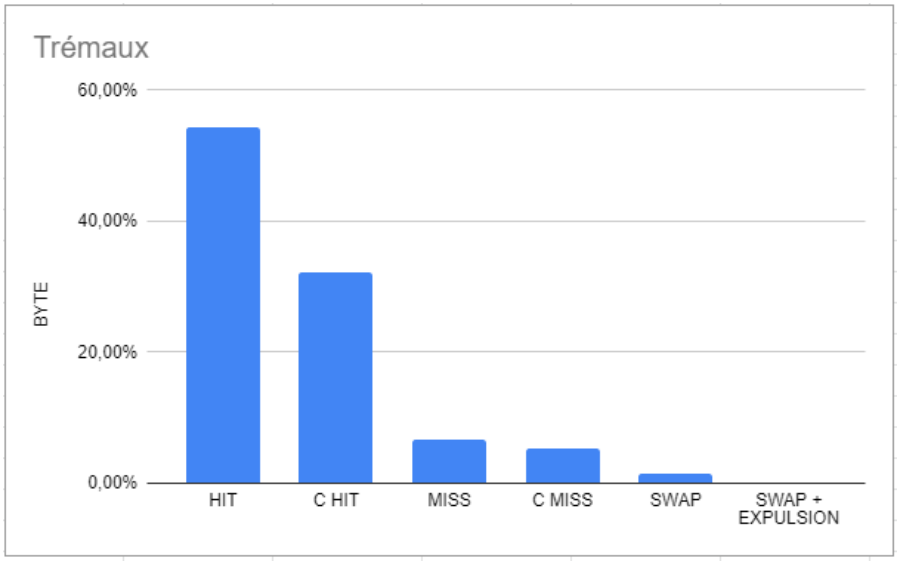

Figure 12: Trémaux solving a 32 by 16 maze (own compilation)

Figure 12 we see a very high hit rate of both the compressed and uncompressed cache. This means that our cache is working for the purpose that was especifically designed: reading an uncompressed array (the map) and a compressed array (the byte array) at the same time. Figure 13 shows exactly that data value distribution. The 1s and 255s are the compressed values, and the values between 2 and 254 are the map. Zeroes are both present on the map and the compressed array.

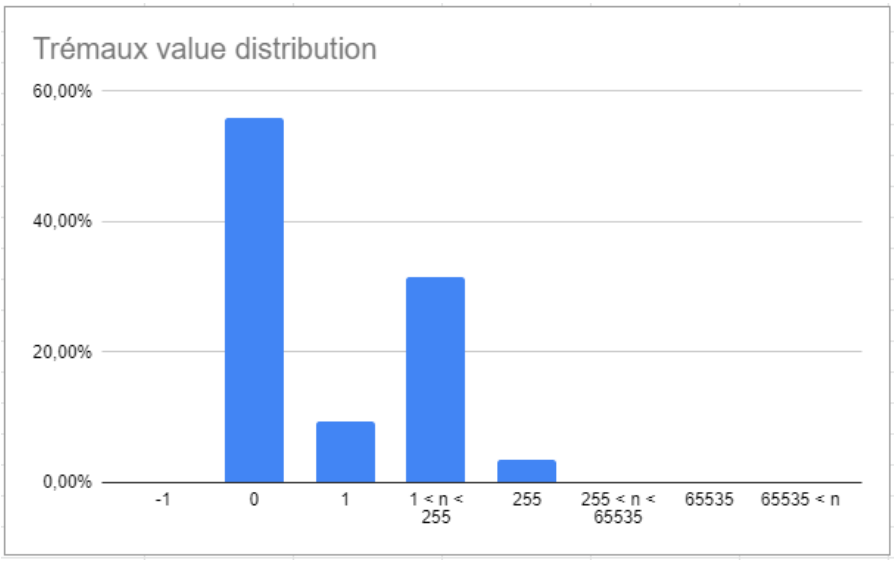

Figure 13: Trémaux data values (own compilation)

6.4. Pigeonhole sort

The pigeonhole sort algorithm is an algorithm that sorts arrays on a very time efficient manner ($O(N+n)$, where N is the range of key value and n the number of items to sort). This algorithm uses the pigeonhole principle[29] to reduce time complexity into one of the most time efficient sorting algorithms in existance. However, this fast speed cannot be achieved without drawbacks. Those drawbacks manifest in a very high memory complexity compared to the other sorting algorithms ($O(N+n)$, when the most commonly used sorting algorithms use $O(n)$, $O(\log n)$ or $O(1)$. This characteristic makes the pigeonhole sort an interesting testbench for our cache.

6.4.1. Statistics of the pigeonhole sort

The following figure shows the statistics for the pigeonhole sort on our cache:

[29]If n items go into m containers, and there are more items than containers, at least one container must contain two or more items.

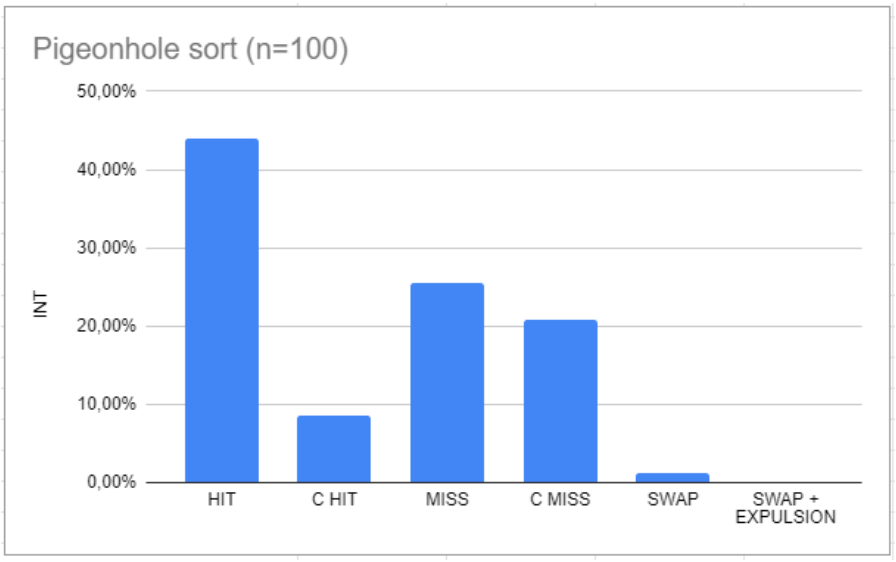

Figure 14: Pigeonhole sort with N=100 (own compilation)

This data shows a mixed usage between the non compressed and the compressed cache. However, there are more accesses to the non compressible zone, which make sense because our algorithm should not contain a lot of compressible data. This behaviour is probably due to the versatility we gave to the cache when we decided to allow fragmentation of the data and to have an array on two or more associations, rather than keeping a whole array into a compressible or non compressible associative block.

Although the more basic version of this algorithm, the counting sort, would be more compressible; the pigeonhole algorithm is more common, and more representative of a real world application[30]. However, the counting sort would be more suitable to use on a system that uses our cache, rather than the pigeonhole sort.

[30]'Real world application' in this context describe software that aims to be on a final product or application, rather than being academic or experimental software.

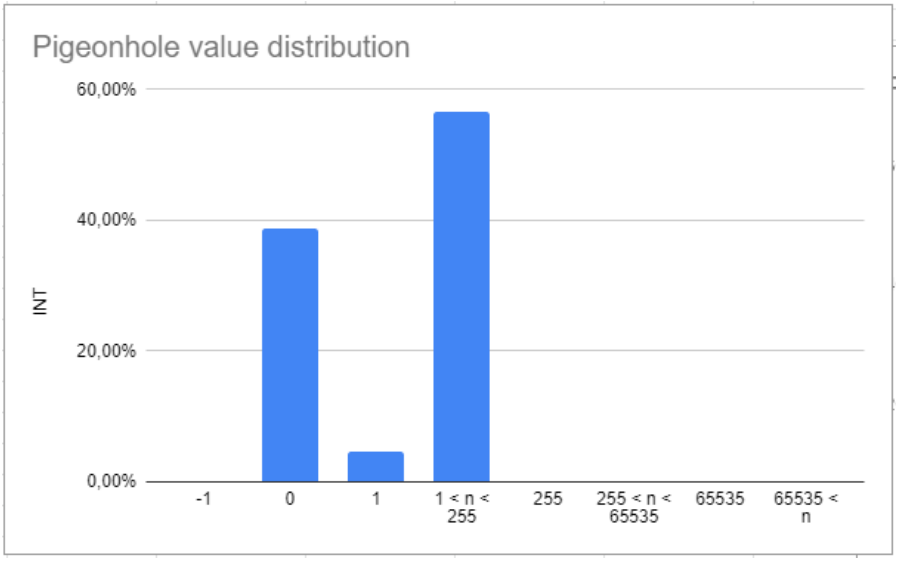

Figure 15: Pigeonhole sort data value distribution (own compilation)

On figure 15 we can see a prevalence in zeroes, which corresponds to the hole array, a big auxiliary memory array that the pigeonhole sort uses in order to sort. This array contains zeroes where there is no data, and in this case, there are a lot of holes empty.

6.5. Mergesort and quicksort

On the following subsections we are going to analyze two more commonly found sorting algorithms: mergesort and quicksort. These algorithms are probably the two benchmarks that represent better a 'real world application' on the tests that we did.

6.5.1. Quicksort

The quicksort algorithm is a sorting algorithm that does not use a lot of memory, because it is usually implemented as an in-place[31] algorithm. In terms of speed, quicksort shows an average case of $O(n \log n)$ and a worst case of $O(n^2)$.

Quicksort is a relevant algorithm. It is one of the three pilars of introsort, a hybrid algorithm that is included in the standard library as 'the sorting algorithm'. Introsort

[31] In-place means that the algorithm does not use extra memory. It uses the input data memory to compute and store the output.

uses quicksort, heapsort and insertion sort to optimize as much as possible the sorting speed. The quicksort is also commonly used as a standalone sorting algorithm, due to the ease of implementation weighted to its speed and low memory complexity.

Our test case and code evaluate the worst case possible: the input data is inversely sorted, and the algorithm takes the leftest item as a pivot. The following charts show the statistics of quicksort:

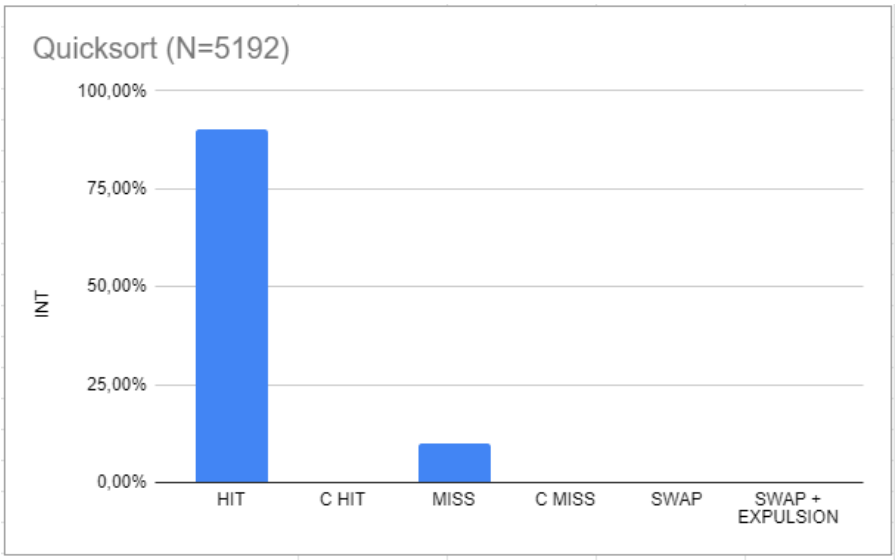

Figure 16: Quicksort with N=5192 (own compilation)

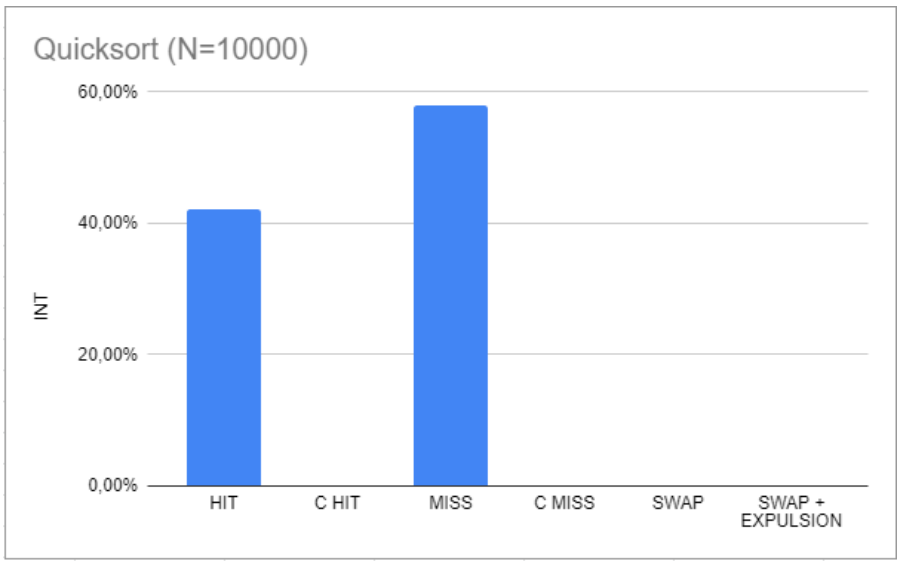

Figure 17: Quicksort with N=10000 (own compilation)

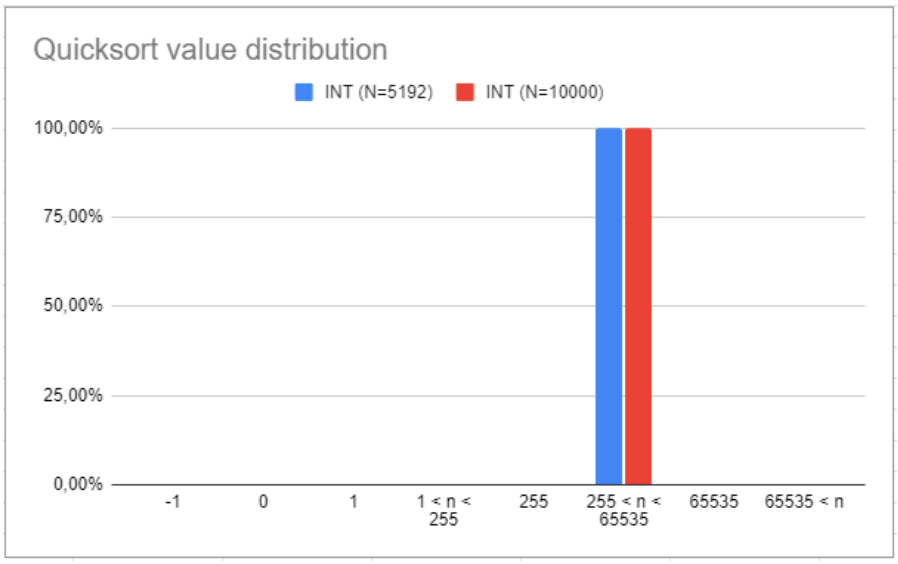

Figure 18: Quicksort data value distribution (own compilation)

As expected, the quicksort uses non compressible values. The algorithm itself does not operate with data structures, it only sorts the input data. This means that if the input data is non compressible, it will not use that side of the cache. This lack of compressible data is shown on figure 18. Therefore, figures 16 and 17 show no use of the compressed cache side.

6.5.2. Mergesort

Mergesort is a sorting algorithm that uses the 'divide et impera'[32] principle in order to sort an array. In opposition to the quicksort, the mergesort uses more memory, because it allocates new memory regions each time a recursive 'division' happens.

The use of mergesort is less common than the quicksort, but it is still widely used. The differences between the mergesort algorithm and the quicksort algorithm are that the mergesort uses more memory in exchange of a constant complexity of $O(n \log n)$, whereas the quicksort can be slower, having the worst case complexity of $O(n^2)$. This means that the mergesort will be used on applications that require a guaranteed time complexity of $O(n \log n)$. It is also used as the default algorithm to explain how the 'divide et impera' logic is applied to algorithms.

The following charts show the test results of the mergesort algorithm:

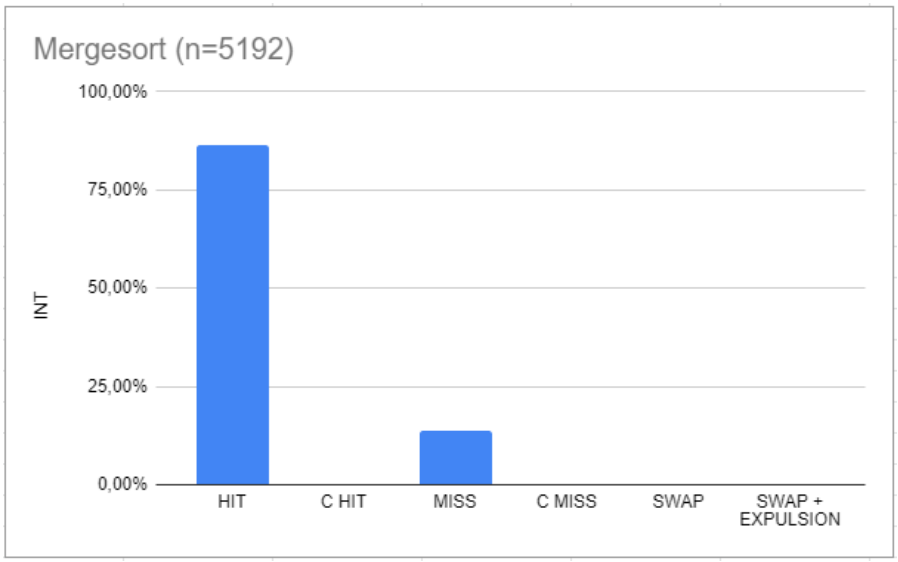

Figure 19: Mergesort with N=5192 (own compilation)

[32] 'Divide et impera', 'divide et vinces' or divide and conquer in english, when talking about computer science, refers to a strategy of recursively dividing a problem into smaller problems until the problems are simple enough to solve them directly.

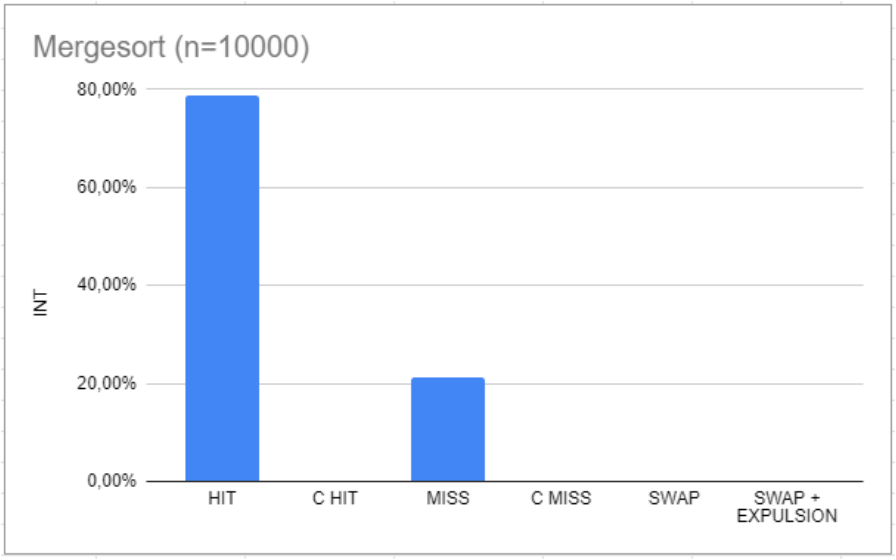

Figure 20: Mergesort with N=10000 (own compilation)

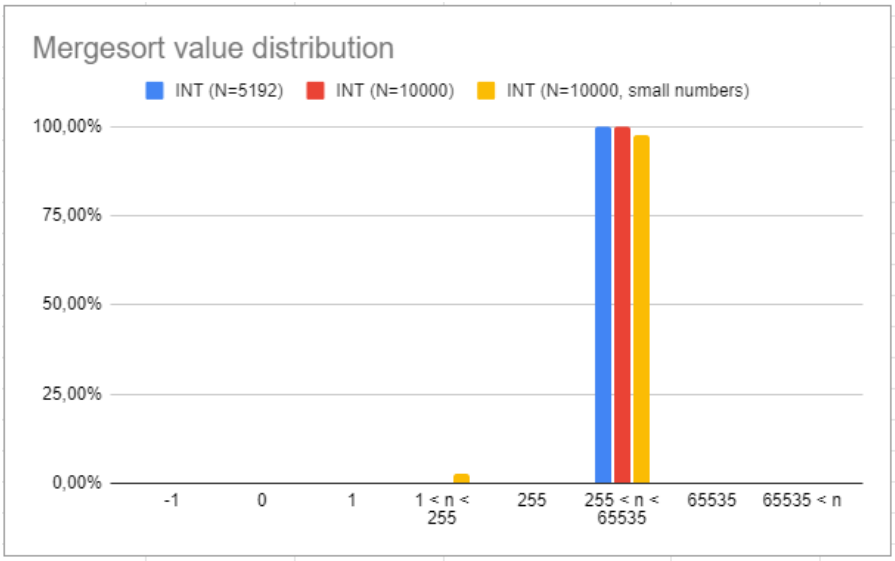

Figure 21: Mergesort data value distribution (own compilation)

However, we tested the algorithm with smaller numbers and we got different results. In order to see that data, we need to use a table, because it is not really visible on a chart. The following table shows the data of the test for $n = 10000$ versus the test for $n = 10000$ with smaller numbers:

	INT (N=10000)	**INT (N=10000) and small numbers**
HIT	530111	530084
C HIT	0	65
MISS	143568	143491
C MISS	0	39
SWAP	0	35
SWAP + EXPULSION	0	35

Table 1: Mergesort with N=10000 vs N=10000 and small numbers (own compilation)

Similar to the quicksort, the mergesort does not actively use compressible data on its operation. It just sorts data, without using auxiliary data structures. This means that the compressibility of this algorithm is tied to the input data. The more compressible the input data, the more compressed memory it will use. However, having only 3 numbers that are compressible really hinder the performance on this cache, and the data on table 1 confirms that.

6.6. Performance metrics analysis

The following table shows the relevant performance metrics of the cache:

Algorithm	Clock cycles	Memory accesses	CPMA	Hit Rate	Miss Rate
Eratosthenes					
BYTE (n=100)	937	214	4,378504673	87,85%	12,15%
INT (n=100)	1462	289	5,058823529	65,05%	34,95%
BYTE (n=1000)	10467	2524	4,146988906	90,06%	9,94%
INT (n=1000)	15717	3274	4,800549786	69,43%	30,57%
BYTE (n=10000)	113068	28352	3,988007901	91,18%	8,82%
INT (n=10000)	226699	35852	6,32318978	41,29%	58,71%
Trémaux					
BYTE (n=32*16)	6325	1468	4,308583106	86,44%	13,56%
Pigeonhole sort					
INT (n=100)	2977	600	4,961666667	52,56%	47,44%
Quicksort					
INT (N=5192)	33682559	13509581	2,493234912	90,18%	9,82%
INT (N=10000)	245207759	50059997	4,898277541	42,06%	57,94%
Mergesort					
INT (N=5192)	1446287	325328	4,445627182	86,24%	13,76%
INT (N=10000)	3147457	673680	4,672035685	78,69%	21,31%
INT (N=10000, small numbers)	3147211	673680	4,671670526	78,69%	21,31%

Table 2: Clocks, memory cycles, hit and miss ratio (own compilation)

Table 2 shows the clock cycles of the cache memory, the number of memory accesses, the division of the previous two (clock / total accesses), and the hit rate and miss rate on the cache.

We observe a very stable cycle to memory access cost of arround 4 cycles. Taking into account the fact that the main memory responds in 1 cycle, that shows a really fast cache, and its relatively independent of the hit ratio.

On the last 3 Eratosthenes runs, there is an inverse correlation between the hit ratio and the average cycles per access. The best hit ratio has the lowest CPM (Clock per memory access), where as the lowest hit ratio has the highest CPM. This is the expected behaviour of a cache, because on a miss the cache usually has to wait for memory. As we alluded before, our memory responds so quickly that this effect is hardly present on the testing.

This inverse relationship is again noticeable on the Quicksort tests. The highest hit rate on our testing is the one with the lowest CPM of all.

In addition, we also want to add a table with the comparison between the hit rate of our cache, and the hit rate of a regular cache. In order to do that, we created a regular 8-way associative cache and a 4-way associative cache, with the same replacement algorithm. However, the compressed placement logic is disabled. The following table shows the comparison between the three:

Algorithm	**Hit rate (GC)**	**Miss rate (GC)**	**Hit rate (8-way)**	**Miss rate (8-way)**	**Hit rate (4-way)**	**Miss rate (4-way)**
Eratosthenes						
BYTE (n=100)	87,85%	12,15%	87,85%	12,15%	87,85%	12,15%
INT (n=100)	65,05%	34,95%	65,05%	34,95%	65,05%	34,95%
BYTE (n=1000)	90,06%	9,94%	90,06%	9,94%	90,06%	9,94%
INT (n=1000)	69,43%	30,57%	69,43%	30,57%	69,43%	30,57%
BYTE (n=10000)	91,18%	8,82%	91,18%	8,82%	91,18%	8,82%
INT (n=10000)	41,29%	58,71%	61,33%	38,67%	41,29%	58,71%
Trémaux						
BYTE (n=32*16)	86,44%	13,56%	87,87%	12,13%	87,87%	12,13%
PigeonHole						
INT (n=100)	52,56%	47,44%	51,33%	48,67%	51,33%	48,67%
Quicksort						
INT (N=5192)	90,18%	9,82%	99,96%	0,04%	90,18%	9,82%
INT (N=10000)	42,06%	57,94%	89,04%	10,96%	42,06%	57,94%
Mergesort						
INT (N=5192)	86,24%	13,76%	95,82%	4,18%	86,24%	13,76%
INT (N=10000)	78,69%	21,31%	87,43%	12,57%	78,69%	21,31%
INT (N=10000, small numbers)	78,69%	21,31%	87,43%	12,57%	78,69%	21,31%

Table 3: Comparison between our Ghost Cache, a regular 8-way associative cache and a 4-way associative cache (own compilation)

As we can see in table 3, the 8-way cache has an edge on the capacity, making hit rates greater on cases where the smaller caches would run out of memory. We can see the smaller Eratosthenes tests having the exact same hit rates, whereas the big ones fall behind on the Ghost Cache and the 4-way. Both Trémaux and pigeonhole sort fall into margin of error, meaning that the difference would not be noticeable. This happens because the input sizes of both algorithms is not big enough to saturate the caches. Lastly, quicksort and mergesort show a big improvement on the 8-way over both other caches.

A potential problem that we find with this comparison is that our cache and the 4-

way show an almost exact behaviour. However, with the extensive testing that we have done previously, we know why this happens: none of the test actually used optimally the Ghost cache to the point that would differ from a 4-way cache. This is because it either used 100% compression, or it did not at all. Trémaux and pigeonhole sort were the ones that used best the cache, but by having a memory usage that fits in a 4-way, the 4-way shows comparable results.

Unfortunately, we could not make the Trémaux algorithm work with larger input data. However, we could get pigeonhole sort with a size of 5000, in order to saturate the 4-way cache. The following chart shows the comparison of this pigeonhole sort with the three caches:

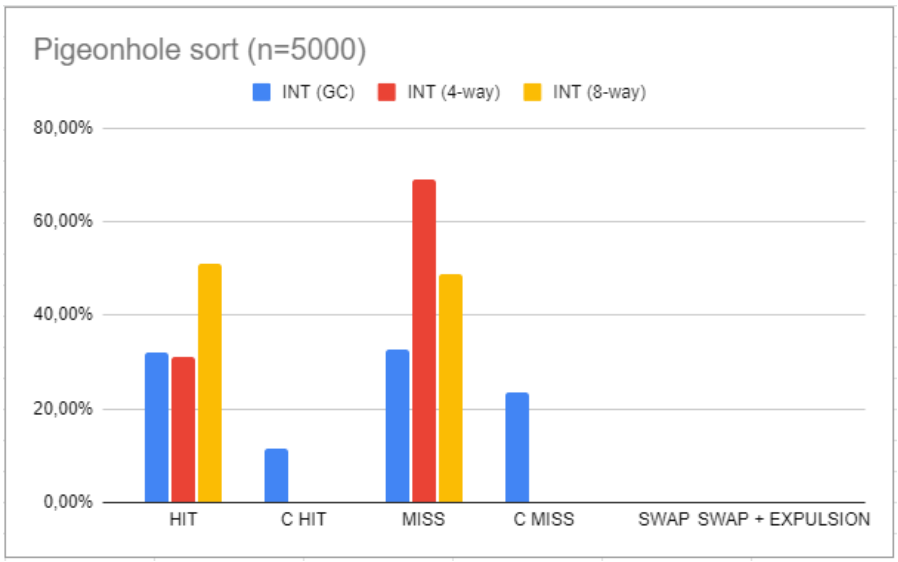

Figure 22: Pigeonhole sort comparison (own compilation)

Figure 22 shows how the Ghost Cache has a better hit rate than the 4-way, due to the compressibility of this algorithm. The following chart compares the hit rates and miss rates:

	Ghost Cache	**4-way**	**8-way**
HIT	43,58%	31,12%	51,16%
MISS	56,42%	68,88%	48,84%

Table 4: Pigeonhole sort n = 5000 comparison (own compilation)

Table 4 comparison further proves our theory about the small input sizes equalizing the hit rates of our cache and the 4-way.

Table 3 also shows the reason why CPU manufacturers choose to keep doing big caches, rather than compressing caches. Their hit rates are comparable or better (specially in the more memory complex algorithms), and they are simpler in design and operation.

Finally, we wanted to also compare the speed in cycles on our cache, against the 8-way and 4-way associative cache. This cycle comparison is not very effective, because the design of the 8-way and 4-way derive from the Ghost Cache, but it is also an interesting comparison. The following table shows the speedup of the other caches over the Ghost Cache:

Algorithm	**8 way / cache**	**4 way / cache**
Eratostenes		
BYTE (n=100)	0,00%	0,00%
INT (n=100)	0,00%	0,00%
BYTE (n=1000)	0,00%	0,00%
INT (n=1000)	0,00%	0,00%
BYTE (n=10000)	0,00%	0,00%
INT (n=10000)	17,65%	0,00%
Trémaux		
BYTE (n=32*16)	0,00%	0,00%
PigeonHole		
INT (n=100)	0,00%	0,00%
Quicksort		
INT (N=5192)	19,63%	0,00%
INT (N=10000)	47,96%	0,00%
Mergesort		
INT (N=5192)	7,36%	0,00%
INT (N=10000)	5,52%	0,00%
INT (N=10000, small numbers)	5,51%	0,00%

Table 5: Cycle speedup compared to the Ghost Cache (own compilation)

Table 5 shows how the 8-way is faster than the Ghost cache, but the 4-way is effectively as fast as our cache.

6.7. Discarded benchmarks

Although we tested 5 different algorithms on our cache, we investigated a lot more. We analyzed weather or not was worth it to code and adequate the benchmarks to our cache memory system, but in the end, the following algorithms were discarded: binary-tree test, mandelbrot set, julia set, pi digit calculation, n-body and regex-redux.

Some algorithms, such as the pi digit, or the regex-redux did not use an extensive amount of memory, and it did not make lots of sense evaluating CPU intensive algorithms.

Other algorithms, like the binary-tree or the n-body required some high level memory calls, that required lots of time to develop for our benchmark.

Lastly, the mandelbrot and julia sets were very complex, and they are more CPU heavy than memory heavy. Thus, waging the amount of hours needed to adequate one of these algorithms against the value of the data that would give us, we decided against testing them.

Nonetheless, the algorithms that we chose to evaluate our cache were representative enough of a computer workload to achieve reasonable conclusions. Both the Eratosthenes sieve and the Trémaux algorithm were a best case scenario for our cache, whereas the mergesort, quicksort were a worst case scenario. Lastly we chose the Pigeonhole sort as a middle ground, representing a different algorithm principle that could exploit cache compression, but the algorithm was not fine tuned to use our cache in a way that counting sort would, for instance.

6.8. Initial energy analysis

The methods to obtain certainty on the usage of energy from our new cache are very time consuming for the short time of this thesis. However, we can make an educated guess about how much energy it will use.

6.9. Static energy

From the data bank perspective, we are reducing a 32bit cell into a 8 bit cell. This is a 4 time reduction in size. Since we have 8 banks, 4 compressed and 4 non compressed, this means that we will have the equivalent of 5 non compressed grids, over 8 that regular caches have.

Giving that the data banks are made of data and metadata, we only reduce partially the size of the bank. The metadata is made of 29 bits, without taking into account

the replacement policy counters, and 37 bits with it. To simplify the math, we will assume that the metadata is also 32 bits. With a simplistic approach to the maths, if we had 5 grids on our cache over 8 on the original one, now we add 8 to both, getting 16 grids on the original and 13 on the compressed. Solving the next formula, $1 - \frac{13}{16} = (1 - 0.8125) = 0.1875$, we get that the energy reduction is about 18.75%.

To this number we need to add the increase of energy consumption due to the compressor and decompressor, and also take into account the rest of the transistors dedicated to the logic control of the cache. We know from existing cache memories that the size of the logic compared to the data banks is ridicule, but just for being sure, we can assume than the best case scenario is a 18.75% reduction in energy (the one obtained above) and a worst case scenario of a 10% reduction.

A 10% reduction of energy would a small improvement, albeit not sufficient to reduce significantly the heat of a processor.

6.10. Dynamic energy

The concept of dynamic energy is directly tied to the activity of the hardware and data lanes. On our cache, a memory access for a *byte*, will not use the same energy that a *int* access. On our testing, we can assume that a *byte* access will use $\frac{1}{4}$ the energy that an *int* access would. With this information and our testing data, we can obtain the reduction of dynamic energy.

Aplying the energy reduction formula, we can get the energy reduction of any test. The formula would be as follows: $EnergyReduction = 1 - \frac{E_d}{E_n} = 1 - \frac{N_{int} + \frac{N_{byte}}{4}}{N_{int} + N_{byte}} = 1 - \frac{N_{int} + \frac{N_{byte}}{4}}{1} = 1 - (N_{int} + \frac{N_{byte}}{4})$.

Both Quicksort and Mergesort show a 0% improvement, due to the input data being all non-compressible integers. On the other hand, Eratosthenes would only use compressed byte memory accesses, meaning that it would reduce a 75% of the dynamic energy. Trémaux would yield a 28% improvement (with a 37.47% compressed byte accesses), and lastly Pigeonhole sort would yield a 22% dynamic energy reduction (with a 29% compressed byte access).

In conclusion, our testing shows a very strong correlation between the compression potential of an algorithm and the dynamic energy reduction. Ranging between the 75% best case scenario energy reduction, to the 0% of a non-optimized algorithm, it is hard to take conclusions. Noting that the average case, being something similar to Trémaux or Pigeonhole, where we have a 25% energy reduction, and also guessing that a general purpose CPU would use a lot of non-optimized algorithms, we deduce that the dynamic energy reduction would be around 20% to 10%.

6.11. Design conclusions

As a summary, these are our observations from the results of the testing phase of our project:

- In terms of replacement policies, and the mixed usage of compressible and non-compressible data, we can conclude that our cache does in fact predict correctly where to place the data with a high degree of confidence. All the tests show an almost non-existent amount of 'swap' and 'swap with block expulsion', which means that once it places the data on a block, it does not need to be moved from the compressed to the non compressed blocks.

  We can also conclude that our replacement policy works as expected, because our miss rate is usually below 33%. The cases where the miss rate increases, like in figure 17 or figure 14 are due to various expected reasons. First, the cache only loading 32 bits each time it access the memory, meaning that if we use 32 bit-size variables, we will miss every time, where as if we use an 8 bit-size variable, we will bring 32 bits, dividing the miss rate by 4 effectively. This effect can be seen at figure 5, but this algorithm introduces the next problem: some algorithms access memory on a random way, and read that data again in a long span of time, enough for the data to not be present in the cache. This last problem cannot be solved easily, hence why there are caches that hold to recently removed blocks, such as the L2 or L3 caches.

- In terms of compressibility, we can conclude that compressibility is directly related to the application (or algorithms) executed. In all our tests, only the benchmarks that we expected to compress (Erathostenes, Trémaux, and pigeonhole as the middle case) were able to compress the data. The rest of benchmarks showed no usage of compressed memory at all, even with our aggressive compression policies. This direct relation to the software executed means that a cache with this design would not improve a computer just by including it on the design. It would require either this computer to execute compressible algorithms only, or to use the hardware knowledge and awareness and design software in a way that exploits our compression design (e.g. using boolean or a three state variable array).

- In terms of speed, the tests were not as thorough as possible, but we obtained enough data to establish that the Ghost Cache is slower than an 8-way equivalent cache. It is approximately 20% slower. However, speed was never our focus, but rather the energy efficiency.

7. Integration with RISC-V

One interesting thought about this project is the integration of our cache on a RISC-V core.

Unfortunately, due to time constraints, the final design of the cache has not been tested thoroughly on the RISC-V core. However, the interfaces of our cache, as well as the protocol used for the transactions on both the CPU side as well as the Memory side are all CV32E40P compliant.

The documentation for including the cache memory on the existing core benchmark is almost non existing, due to the nature of that test bench. The idea is for you to make your own test bench for the CPU core. Moreover, in addition to the complexity of changing that test bench, in order to test the cache the right way, would be to change it on the official repository for testing, the *core-v-verif* on *GitHub*. This would involve changing the *Makefile* and learning a complex test environment with a documentation that does not explain a lot about changing the core design[33].

[33]This is not their fault. The documentation that we could find talked about how to make a test bench (without changing the core, just for testing core related functionalities) and things about the standard namings and conventions to follow if we want to contribute to the project.

8. Scheduling

This project started on the 13th of September 2021, and it will last about 250h in a best-case scenario. The delivery deadline is the 20 of December. The time of the oral defense is yet to be disclosed.

It is foreseen to work from 5 to 8 hours a day in this project, and the scheduling and Gantt diagram are made with that assumption.

8.1. Task description

8.1.1. Control tasks

This tasks are recurrent over the development of the project. They are the communications between the researcher and the director.

The list of tasks is the following:

- [$C1$] Initial deliberations: The researcher and the director need to narrow down the target of the project and focus it on the expected features of a Final Degree Project from the 'Facultat d'Informatica de Barcelona'. This takes about 1 hour.
- [$C2$] Regular communication: The researcher needs to inform regularly the director about the state of the project, as well as asking for help whenever needed. This task takes about 0.5 hours each 2 weeks.

8.1.2. Theoretical part

The main goal of the theoretical part of this project is to propose an implementable RISC-V 32 bit compatible compressing cache. In order to achieve this target, we need to segment the tasks using the Scrum methodology.

The tasks are the following:

- [$T1$] Brainstorming: The first task on every theoretical part is to use the common knowledge to write down every possible way to develop the project. The main purpose of brainstorming is to start narrowing down the scope on what to do, and

finding the next task, which is investigating sources. This task will require 8 hours at best.

- [$T2$] Finding sources: One of the fundamental tasks on a project is to investigate the 'State of the art'[34]. This task requires 8 hours.

- [$T3$] Analysis of the sources: Once we have every source, we need to analyze every paper of the project's topic, to gather the ideas for making our design. This task takes about 24 hours.

- [$T4$] Making our design: This task ties to the conclusion of the analysis of the sources. While making our design, we need to describe the architecture of the cache, make diagrams for helping the architecture explanation and write the state diagram of data in the cache. This task takes 24 hours.

- [$T5$] Making speculative calculations for the design: The last task on the theoretical part of the job is to make some calculations to obtain the expected performance and behaviour of our design. This task will take about 6 hours.

8.1.3. Practical part

The practical part of this project has two purposes: checking the functionality of our cache design, and also testing the integration with RISC-V. The tasks will be the following:

- [$P1$] Learning the tools: On this project we work with RISC-V. RISC-V, being open source, uses a modern tool-chain to synthesize and simulate processor designs. The main programming language for RISC-V developement is Verilog. Verilog is a relatively modern HDL[35]. We need to learn to use this language, and the tools that comes with (Verilator, Icarus...). This will take about 12 hours.

[34] State of the art means what has been done or investigated on the field in which you are going to work on.

[35] Hardware Describing Language

- [$P2$] Finding code: Once we know how to code in Verilog, we need to analyze the public RISC-V cores to further improve our Verilog designing and programming skills, as well as understanding the way different modules interact between them. We also need to decide which core will be used in the final design for testing. The most difficult thing about this task is understanding the Makefile[36] of the RISC-V cores. This will take about 14 hours.

- [$P3$] Starting with the cache prototype: To code the cache we will start on an isolated environment. We will first try to create a regular simple direct cache module, with tags and the memory cells. We also need to test this design and check that works as it should. In this stage, there are no external interfaces, just the internal cache module. This task will be about 20 hours.

- [$P4$] Making the associative cache: The next step to have our compressing cache is making it associative. At this point, we only have the regular cache module, and we will use this module 4 times to make a 4 associative cache. This task will take about 20 hours.

- [$P5$] Making the compressing cache module: On this task, we will take the design of the theoretical part and implement it on Verilog. This task will take around 6 hours.

- [$P6$] Changing the associative logic and including the compressing cache modules: This task is one of the hardest on this project. We need to change the associative placing algorithm to bias the compressing caches when they can be effective. We also need to make a complex logic that will change a block from the compressing cache to the regular cache when needed. This task will take about 22 hours.

- [$P7$] Creating the external interface: This is the last step in making our cache. We

[36]Make is a software that handles dependencies defined by a Makefile. Make is usually used for software compilation, or on this case, to execute the tool-chain and synthesize and test the Verilog code.

will connect the inner circuits to an interface compatible with the RISC-V core. This task will take about 10 hours.

- [$P8$] Including the cache in the RISC-V core: This is the hardest task on the project. This task requires a perfect comprehension of the T1 and T2 tasks. What we need to do is change the Makefile that defines how the CPU is created, and add our new cache. We also will need to change some Verilog files from the core to add the cache. This task could take up to 35 hours.

- [$P9$] Cache testing: once we have the CPU assembled, we can simulate it and check for the correct working of the cache. This task takes about 12 hours.

8.2. Summarized task table

Code	Description	Time	Dependencies	Resources
C1	Inital deliberations	1h	*NONE*	PC
C2	Regular communication	0.25h/week	*NONE*	PC
C	***Control Subtotal***	***4h***		
T1	Brainstorming	8h	*NONE*	PC
T2	Finding sources	8h	*NONE*	PC, internet
T3	Source analysis	24h	T2	PC, papers
T4	Designing	24h	T3	PC, analysis results
T5	Speculative analysis	6h	T4	PC, theoretical design
T	***Theoretical Subtotal***	***70h***		
P1	Learning the tools	12h	*NONE*	PC, Verilog/Verilator manuals
P2	Finding code	14h	*NONE*	PC, Github / RISC-V webpage
P3	Starting cache prototiping	20h	P1	PC, Verilator / Icarus
P4	Creating associativity hardware	20h	P3	PC, Verilator / Icarus
P5	Compressing cache module	6h	P4	PC, Verilator / Icarus
P6	Modifying cache structure	22h	P5	PC, Verilator / Icarus
P7	Adding external interfaces	10h	P6	PC, Verilator / Icarus
P8	Cache installation in CPU	35h	P7	PC, core documentation / Makefile
P9	Cache testing	12h	P8	PC, Verilator / Icarus
P	***Practical Subtotal***	***151h***		
	TOTAL	**225h**		

Table 6: Summarized task table (own compilation)

8.3. Gantt diagram

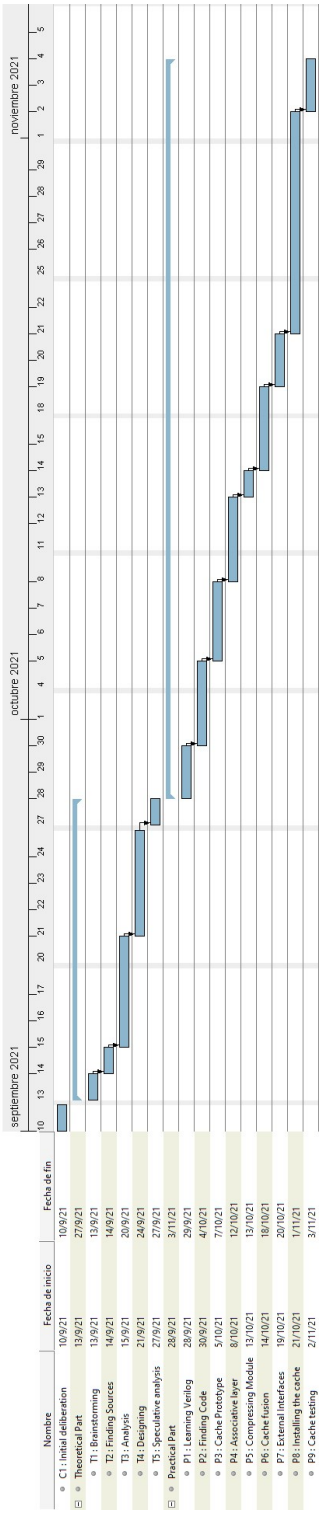

Figure 23: Gantt diagram (Own compilation)

8.4. Risk management

In this section we will discuss the alternative planning for every problem that can occur during the development of the project.

The three problems stated on the chapter 8.4 are the feasibility of incorporating the cache in the RISC-V core before the deadline, general time constraints and failing to meet the expectations of performance.

The two first problems combined would mean that the testing on the cache could not be done on an actual RISC-V processor, and would require some other method to check the correct working.

The simplest way to solve this is ditching the idea of using a pre-built CPU and making our own RISC-V simulator. We would need to implement a Verilog program, or a Verilator C++ script. This program should interface with the cache as a regular RISC-V CPU, with the instruction and memory interactions as a regular program would do on a RISC CPU. This new task would need at least 25 hours, and worst case 50 hours of extra work.

The final possible risk is failing at the premise of making an efficient compressing cache. This means that we could not achieve successful or expected results on this project. In the event that this happened, we should report as much information as possible about how we failed, what failed, and how could this failure be overcame on future projects. This task wold be around 20 hours.

Another potential risk that is not only related to this project, but with the whole research scenario, is data loss, which will be prevented with regular backups. In addition, the practical part of the project has programming on it, which will be preserved and version controlled via git[37]. This also will help into making backups.

[37]The most common program for version control, used ubiquitously on every project computer science related.

9. Budget

In this section we will discuss the economical cost of the project. Although this project will be developed by a single person with the help of the director, we will assume that there are two different agents in the development of this project, a documentation curator and a hardware designer.

Both agents will need office equipment, as well as access to the internet and other expenses. Everything will be detailed below.

9.1. Costs

In order to develop this project we do not need esoteric technologies, unlike others projects that require expensive FPGA / experimental boards. We need to cover mobility expenses and office equipment. The following table show the prices for every expense:

Name	**Type of cost**	**Cost**
Document curator(DC)	OpEx	27€/hour
Hardware designer(HW)	OpEx	35.1€/hour
Internet connection	OpEx	50€/month
T-usual metro ticket	OpEx	40€/ticket
Electricity	OpEx	0.29762€kWh

Table 7: Project CapEx and OpEx (Own compilation)

Note that salaries in the table 7 already take into account the x1.35 factor of the Social security. Also note that we do not have any CapEx, because all our software is free and open source, and we do not need any extra equipment besides the computers.

9.2. Contingency budget and incident management

On every project, there should be a contingency budget in case of any unexpected expense. The contingency budget will be a 15% of the total cost of the project.

In terms of incident management, we should be aware that we disclosed risks on the section 8.4 that could occur during the development of the project. In order to take them into account, we will also add the cost of OpEx during the estimated times of the alternative tasks.

9.3. Total costs

Task / Concept	Quantity	Cost	Subtotal	**Notes**
Internet Connection	3 months	50€	150€	
T-usual metro ticket	6 tickets	40€	240€	3 tickets per person, 2 people
Electricity	13.5kWh	0.2976€/kWh	4.0176€	
OpEx			**394,0176€**	
C1	1h	35.1€/h	35.1€	Hardware Designer wage
C2	3h	62,1€/h	186.3€	Hardware Designer + Document Curator wages
T1	8h	35.1€/h	280.8€	Hardware Designer wage
T2	8h	35.1€/h	280.8€	Hardware Designer wage
T3	24h	35.1€/h	842.4€	Hardware Designer wage
T4	24h	62,1€/h	1490.4€	Hardware Designer + Document Curator wages
T5	6h	35.1€/h	210.6€	Hardware Designer wage
P1	12h	35.1€/h	421.2€	Hardware Designer wage
P2	14h	35.1€/h	491.4€	Hardware Designer wage
P3	20h	35.1€/h	702€	Hardware Designer wage
P4	20h	35.1€/h	702€	Hardware Designer wage
P5	6h	35.1€/h	210.6€	Hardware Designer wage
P6	22h	35.1€/h	772.2€	Hardware Designer wage
P7	10h	62,1€/h	621€	Hardware Designer + Document Curator wages
P8	35h	62,1€/h	2173.5€	Hardware Designer + Document Curator wages
P9	12h	62,1€/h	745.2€	Hardware Designer + Document Curator wages
Tasks			**10165.5€**	
Project Expected Subtotal			***10559.5176€***	
Contingency			1583.92764€	15% of the Project Expected Subtotal
Delay 1	50h	35.1€/h	1755€	Making our own RISC-V simulator (HW wage)
Delay 2	20h	62,1€/h	1242€	Documenting the failures (HW + DC wages)
TOTAL			**15140,45€**	

Table 8: Total costs of the project

9.3.1. Amortization

When an enterprise buys materials, an amortization study should be done. Because the computers will be used only three months, we know that the amortization will not come with positive results. This is because the legal maximum amortization on our country applied to computer equipment, the EPI[38], only allows a maximum of 25% amortization each year. Because we will analyze the amortization by month, we will use a (25/12)% monthly amortization rate.

The following table will show the amortization:

Month	**Cummulative amortization**	**Net tax value**
0	0€	3900€
1	81,25€	3818,75€
2	162,5€	3737,5€
3	243,75€	3656,25€
TOTAL	**243,75€**	**3656,25€**

Table 9: Lineal amortization table (Own compilation)

As we can see on the table, the amortization of the materials show a net value of 3656,25€, due to the use of only 3 months, rather than the expected of 4 years by the law. In order to have an even amortization, we should have a 0 in net tax value.

[38]"Equipos para procesos de información", the maximum yearly amortization value applied to a computer device on Spain.

10. Sustainability

The sustainability study will determine the ecological footprint of the project, as well as the long term viability of the hardware proposed. The way we are going to analyze the sustainability is by comparing the current hardware with the hardware proposed, see the differences and then arguing the benefits of one design over the other one.

10.1. Ecological footprint

10.1.1. CPU production

The process to produce a CPU is a very energy and resource consuming process. It uses very rare materials, such as pure silicon extracted from special sands. This pure silicon is then subject to a set of procedures that are expensive and energy inefficient, such as doping the silicon crystals or creating the electrical circuits and gates on the silicon wafer[39].

Those procedures are very susceptible to failure: even a single spot of dust can destroy multiple CPUs on the wafer, meaning that the machines and the environments where the processors are produced are very sterile and controlled to improve yields. Also, improving the machinery required to make CPUs is also very expensive.

All the aforementioned means that having a smaller design (in size) could improve yields due to fitting more processors in a wafer, thus reducing the losses when malformations happen in some CPUs on the wafer. The next formula shows the relationship between processor area size and the amount that processor that can fit on a wafer:

[39]The way CPUs are produced is on silicon discs called wafers. Each wafer measures about 30-45cm in diameter. In each wafer you produce multiple CPUs.

$$DPW = \left\lfloor \frac{\pi d^2}{S} \right\rfloor \quad (1)$$

where

$$d = \text{diameter (mm)}$$

$$S = \text{size of the processor } (mm^2)$$

In the formula 1 we can see an inverse relationship between S and the DPW, meaning the smaller the size, the larger the amount of processors per wafer.

Due to our cache design being smaller than the current ones (about 20-40% smaller), the number of dies[40] per wafer would increase, increasing the yield.

10.1.2. Energy consumption of CPUs

As we eluded in the introduction, cache modules in the CPU consume a lot of energy. In addition, some of the papers studied in this project[8][13] state that energy consumption of caches is becoming a problem.

This project aims to reduce the overall energy consumption of CPUs, making it a very positive design for the environment. Due to the widespread use of CPUs, a reduction of energy of a processor can affect millions of different devices, from vending machines to office printers, computers, etc. Less energy consumption directly affects energy production, requiring less energy to maintain the electronics, thus reducing pollution due to less energy generation.

In addition, heat is one of the factors that intervene in the longevity of semiconductors. The colder they operate, the longer they live. This means that a chip with a compressing cache could last longer, requiring less replacements and reducing obsolescence of the devices, thus making it more Eco-friendly than competitors.

[40] Technical word to describe the silicon processor chip.

10.1.3. FAQ on Environmental Dimension

Have you estimated the environmental impact of the project? Yes, it is thoroughly explained on sections 10.1.1 and 10.1.2.

Did you plan to minimize its impact, for example, by reusing resources? Reusing resources in CPU production is a mixed bag. While some materials, such as gold, are being reused, the silicon and other primary materials of the processors cannot be recycled.

How is currently solved the problem that you want to address (state of the art)?, and how will your solution improve the environment with respect other existing solutions? As we stated on section 10.3, no manufacturer is using any kind of energy saving measure in existing caches. Our solution aims to bring the compressing cache technology to another level, increasing the effectiveness of the technology, making it more appealing for hardware manufacturers.

10.2. Economical impact

In terms of economical effects of this project, the aforementioned improvements on yields makes wafers more profitable. This yield improvement would increase the production of the factories, reducing the price of processors and the cost in raw resources.

Besides the industrial efforts, changing the design of the cache as we propose is not very inexpensive in a current generation of processors, but each time a new design generation is implemented (e.g. from generation 1 to generation 2), the incorporation of the new cache is less expensive in terms of the physical template for the processor. Nevertheless, an incorporation of a newly designed module in a CPU requires thorough testing, even more than we will ever do in the project, and that testing will be expensive.

Nonetheless including the design in a processor should be interpreted as a CPU improvement for a new generation, thus justifying the cost of implementing it. The cache we are developing is a no compromise, more energy efficiency version of the current caches.

10.2.1. FAQ on Economic Dimension

Reflection on the cost you have estimated for the completion of the project The cost of the project seems on par with a project of this caliber. 15k€ is a relatively small sum, especially compared to the amount of money that could save on production costs of CPUs.

How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)?, and how will your solution improve economic issues (costs ...) with respect other existing solutions? The problem I want to address does not have a lot of competition. The market of compressing caches is basically non-existing. Nevertheless, our approach to the project is to pack as much features as possible in our cache, collapsing 3 or 4 designs of different caches into one, with as much bennefits as possible. This should increase the value of our solution over the other ones.

10.3. Social impact

10.3.1. FAQ on Social Dimension

What do you think you will achieve -in terms of personal growth- from doing this project? This project not only will give me a better understanding on how caches work and can be improved, but also give more ideas and possible solutions, which eventually will reach production states. It will also help me put together all the knowledge I have been gathering since entering the degree of computer science.

How is currently solved the problem that you want to address (state of the art)?, and how will your solution improve the quality of life (social dimension) with respect other existing solutions? No big hardware manufacturer is using any type of hardware compression cache in its designs. A compressing cache is not the only option to aid in energy consumption of the memory, but the problem is that they still do not use another method, meaning that caches stayed almost untouched for decades.

Is there a real need for the project? This project aims to bring into attention

the issue with cache heat and energy consumption issue that other papers tried to arise. This is a ever growing problem in the hardware design department of CPUs for at least 20 years. There is an urgency to solve this issue, otherwise improvements in the CPU performance will be minimal in the coming years.

10.4. Conclusion

As a summary, there is not a great disadvantage in implementing a compressing cache. The only reason why no one has included one on a commercially available processor is because they do not give a great performance increase, meaning that they cost a lot of money and do not give back perceivable returns. This also means that if our design also does not deliver in performance metrics, it will not be used as well.

11. Conclusion

Our goal was to establish weather a compressing level 1 cache would be an improvement over the current caches used in the CPU industry.

In those regards, we can establish that a general purpose CPU with a compressing cache would not be feasible as we designed it. This is due to the fact that our compression only benefits certain algorithms and optimized software, whereas non-optimized software (the majority of the software used today) would use only half of the cache capacity. Nowadays, the trend of CPUs is to make larger and larger caches, so having a cache that halves its capacity on the majority of software would not be competitive (we saw the effect of halving the capacity on table 3, where it behaved almost as a 4-way instead of the pseudo 8-way cache that it actually is). If we then add the relatively small energy usage reduction (around 10% - 20%, as seen in chapter 6.8) we can see that it is really not feasible at all.

However, our cache design has shown that on certain tasks the Ghost Cache has an edge, and that we could get rid of the non-compressing section, and use only a compressed cache as a main cache. This means that, on certain use cases where the algorithms could be optimized to use compression, such as a robot that explores mazes or caves, or a microcontroller for a Hard Drive, it could be highly beneficial to the cache physical size and energy requirements. By changing the non-compressed memory banks for compressing memory banks, we would double the data bank size reduction, and we would also double the energy reduction, achieving a more convincing 40% (doubling the previous 20%) maximum energy efficiency improvement.

On the topic of custom requirements, the RISC-V platform is now used exactly for those: custom designed processors. A hard drive company called Western Digital moved from micro-controllers for their drives to RISC-V[14] processors optimized for hard drive operations. And they are not the only ones switching to RISC-V[7]. This means that our cache might be a usable option for a RISC-V specific processor that only operated on compressible data, or a mixed used of compressible and non-compressible.

References

[1] A.R. Alameldeen and D.A. Wood. "Adaptive cache compression for high-performance processors". In: *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* ISSN: 1063-6897. June 2004, pp. 212–223. DOI: 10.1109/ISCA.2004.1310776.

[2] Julien Dusser, Thomas Piquet, and André Seznec. "Zero-content augmented caches". In: *Proceedings of the 23rd international conference on Supercomputing.* ICS '09. Yorktown Heights, NY, USA: Association for Computing Machinery, June 2009, pp. 46–55. ISBN: 9781605584980. DOI: 10.1145/1542275.1542288. URL: https://doi.org/10.1145/1542275.1542288 (visited on 09/20/2021).

[3] Gennady Pekhimenko et al. "Base-delta-immediate compression: Practical data compression for on-chip caches". In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT).* Sept. 2012, pp. 377–388.

[4] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. "Compresso: Pragmatic Main Memory Compression". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* Oct. 2018, pp. 546–558. DOI: 10.1109/MICRO.2018.00051.

[5] Daniel Etiemble. "45-year CPU evolution: one law and two equations". In: *arXiv:1803.00254 [cs]* (Mar. 2018). arXiv: 1803.00254. URL: http://arxiv.org/abs/1803.00254 (visited on 09/26/2021).

[6] Wikimedia community. *Wikipedia Main Page.* en. Page Version ID: 1004593520. Feb. 2021. URL: https://en.wikipedia.org/w/index.php?title=Main_Page&oldid=1004593520 (visited on 01/12/2022).

[7] Jeremy Hsu. *RISC-V Star Rises Among Chip Developers Worldwide.* en. Apr. 2021. URL: https://spectrum.ieee.org/riscv-rises-among-chip-developers-worldwide (visited on 01/12/2022).

[8] R. Canal, A. Gonzalez, and J.E. Smith. "Very low power pipelines using significance compression". In: *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. ISSN: 1072-4451, pp. 181–190. DOI: 10.1109/MICRO.2000.898069.

[9] Crystal Chen, Greg Novick, and Kirk Shimano. *RISC vs. CISC*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ (visited on 01/12/2022).

[10] GeeksforGeeks.org. *GeeksforGeeks — A computer science portal for geeks*. en-us. URL: https://www.geeksforgeeks.org/ (visited on 01/12/2022).

[11] RISC-V Internationa. *About RISC-V*. en-US. URL: https://riscv.org/about/ (visited on 01/12/2022).

[12] rosettacode.org. *Rosetta Code*. URL: https://www.rosettacode.org/wiki/Rosetta_Code (visited on 01/12/2022).

[13] Luis Villa, Michael Zhang, and Krste Asanović. "Dynamic zero compression for cache energy reduction". In: MICRO 33 (), pp. 214–220. DOI: 10.1145/360128.360150. (Visited on 07/31/2021).

[14] westerndigital.com. *RISC-V — Western Digital*. en. URL: https://www.westerndigital.com/solutions/business/risc-v (visited on 01/12/2022).