Tampere University

Kalle Tamminen

COMPARISON OF CACHE REPLACEMENT ALGORITHMS FOR RISC-V

Master's thesis
Faculty of Information Technology and Communication Sciences
Examiners: Timo Hämäläinen
Matti Haavisto
September 2023

ABSTRACT

Kalle Tamminen: Comparison of Cache Replacement Algorithms for RISC-V
Faculty of Information Technology and Communication Sciences
Tampere University
Master’s Programme in Information Technology
September 2023

Cache replacement policies play a significant role in the performance of cache memories. This thesis aims to find new replacement algorithm options for an existing RISC-V multicore processor design, the HPC subsystem of the Ballast System-on-Chip. HPC is a dual core design with two 64-bit CVA6 cores, and it features an L2 cache. The cache was originally designed with a simple but popular LRU replacement algorithm.

This work first looks at the cache replacement algorithms used by other existing RISC-V systems, and then at other research that has been carried out on the topic. Three algorithms are implemented as part of the HPC L2 cache: PLRUm, EBR, and Mockingjay.

The performance of these algorithms is evaluated in RTL simulation with a set of benchmark programs. The results show that more complex EBR and Mockingjay algorithms have a concrete performance improvement over LRU-based solutions in most cases, but the LRU algorithms do have an edge on a few benchmarks. The performance improvement provided by more advanced algorithm comes at a cost, as they require more memory and registers.

Keywords: cache, memory, replacement, policy, algorithms, RISC-V, CVA6

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Kalle Tamminen: Välimuistin korvausalgoritmien vertailu RISC-V -järjestelmässä
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Syyskuu 2023

Välimuistien korvausalgoritmeilla on huomattava vaikutus välimuistin suorituskyvylle. Tämän diplomityön tavoitteena on löytää uusia korvausalgoritmivaihtoehtoja aiemmin toteutetulle RISC-V moniydinprosessorille, Ballast-järjestelmäpiirin HPC-alijärjestelmälle. HPC on 64-bittisiä CVA6-ytimiä käyttävä kaksiydinprosessori, joka sisältää myös L2-välimuistin. Välimuisti suunniteltiin alun perin yksinkertaista, mutta suosittua LRU-algoritmia käyttäen.

Diplomityö tarkastelee ensin muiden RISC-V-järjestelmien käyttämiä korvausalgoritmeja, ja sen jälkeen myös muissa tutkimustöissä esitettyjä algoritmeja. Algoritmeista kolme toteutetaan osaksi HPC:n L2-välimuistia: PLRUm, EBR ja Mockingjay.

Näiden algoritmien suorituskyky arvioidaan RTL-simulaattorilla testiohjelmia käyttäen. Tulokset näyttävät, että monimutkaisemmat EBR- ja Mockingjay-algoritmit parantavat suorituskykyä huomattavasti useimmissa testeissä, mutta joissain tapauksissa LRU-pohjaiset algoritmit suoriutuvat paremmin. Paremman suorituskyvyn hintana on edistyneempien algoritmien suuremmat vaatimukset muisteille ja rekistereille.

Avainsanat: välimuisti, muisti, korvausalgoritmit, RISC-V, CVA6

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

This thesis was made for Nokia as part of the SoC Hub project. I would like to express my gratitude to all the participants of the project for their respective efforts, particularly to Tero Lehtinen for laying much of the groundwork that made this thesis possible.

I would like to thank my examiners Timo Hämäläinen and Matti Haavisto for supervising this work. I am also grateful to my line manager Antti Hirvonen for the opportunity to get my career started at Nokia.

Tampere, 12th September 2023

Kalle Tamminen

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CCX	CPU-Cache Crossbar
CVA6	CORE-V Application Class, 6-Stage Core. Aka. Ariane.
FIFO	First In First Out
FSM	Finite State Machine
HPC SS	High Performance Computing Subsystem
ISA	Instruction Set Architecture
L1	Level 1 Cache
L1.5	Level 1.5 Cache
L2	Level 2 Cache
LFSR	Linear Feedback Shift Register
LFU	Least Frequently Used
LLC	Last Level Cache
LRU	Least Recently Used
LSB	Least Significant Bit
MRU	Most Recently Used
MSB	Most Significant Bit
NoC	Network-on-Chip
RTL	Register Transfer Level
RR	Round Robin
SoC	System-on-Chip

1 INTRODUCTION

Memory use continues to be a bottleneck in processor performance due to long access times. Decreasing the required amount of read and writes to memory can provide a real performance increase for most systems. Caches have been long used to mitigate this issue.

Cache architecture has a lot of variables which can be tuned to modify their behaviour, and this thesis focuses on one of them: the cache replacement algorithm. The purpose of a replacement algorithm, ultimately, is to maximize the accuracy of the cache by attempting to keep the most valuable regions of memory cached.

This thesis investigates how much performance can be gained with different replacement algorithms on a 64-bit RISC-V subsystem of a recently developed system-on-chip. Different algorithms are investigated, then implemented as part of the hardware description, and finally simulated with a set of benchmark programs.

The first chapter introduces the background and motivation of the thesis. In the second chapter, the most important features and design decisions of caches are explained in more detail. The third chapter provides more information on the Ballast SoC and its HPC subsystem, on which this thesis focuses on.

Previous related works are explored in the fourth chapter. This includes a look at other 64-bit RISC-V processors that have been designed, and the kinds of caches they feature. Cache replacement algorithms are also investigated in general, beyond the RISC-V architecture.

Chapter five introduces the methodology of this work, mainly focusing on the benchmarking setup and the criteria used in comparing the replacement algorithms. The actual compared algorithms are introduced in chapter six, and their implementation details are explained in chapter seven.

The results of the algorithm comparisons are shown in the eight chapter. And finally in chapter nine, the conclusions and some of the limitations of the work are discussed.

2 CACHES

Caches’ purpose is to speed up memory access times, as fetching data from memory is very slow relative to the execution speed of processors. Caches can be used to store data as well as instructions. In terms of memory hierarchy, caches are between the processor and the memory of the system. [1], [2]

Figure 1 shows examples of typical memory hierarchies. Memories close to the top of the pyramid are closer to the processor and are faster to access. Further away memories are slower but can be implemented in larger sizes. [2]

Sizes of different memories are limited by their per-bit cost. If not for this cost, the whole memory would be implemented in the fastest available memory technology. However, this is not economical. Instead, a memory hierarchy is used to find a balance between cost and performance. [1], [2]

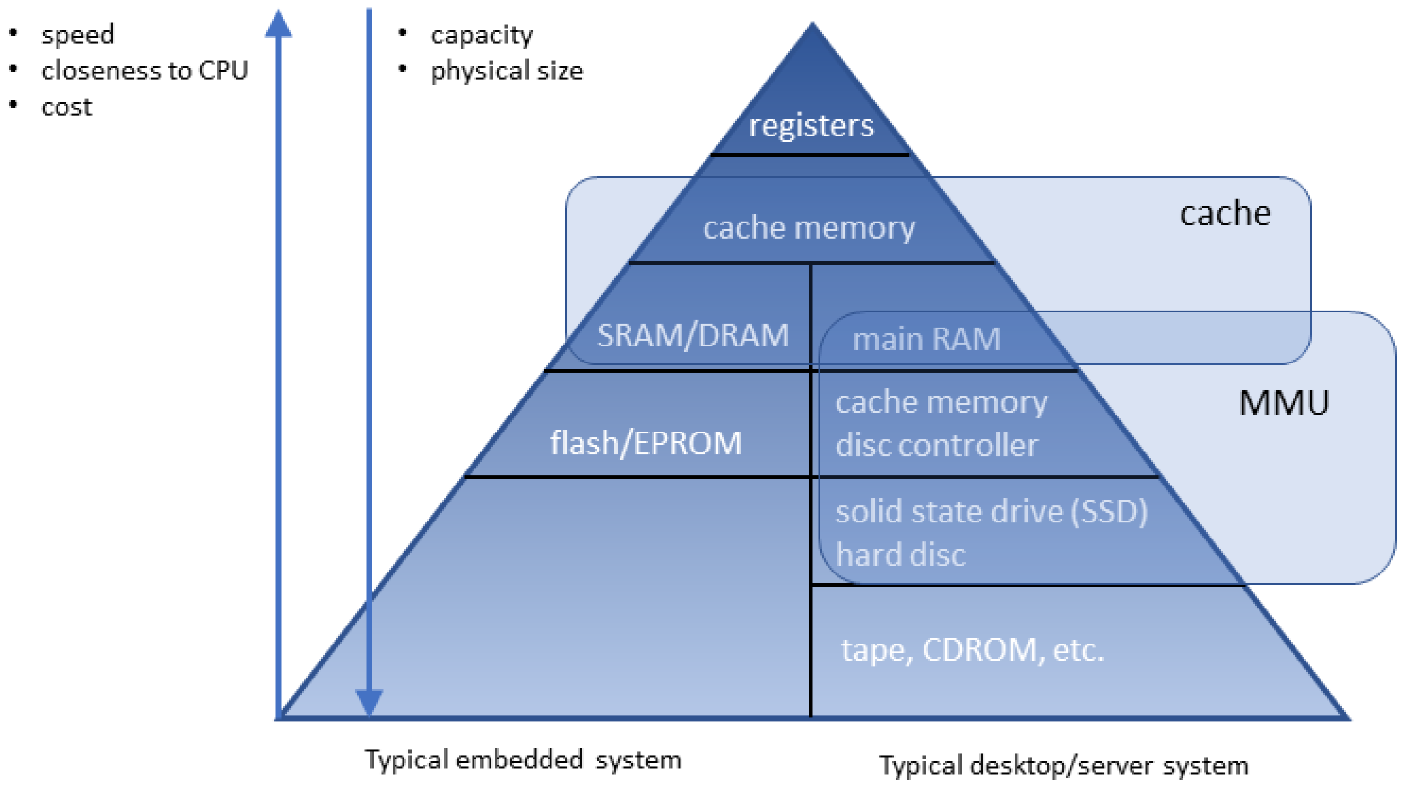

Figure 1: Memory Hierarchy, adapted from [2]

A system can have multiple levels of caches. As with other memories in the hierarchy, high level caches i.e., ones closest to the core, are faster to access but also smaller.

Conversely lower levels of caches are larger but slower to access. When the processor accesses a memory address, the data is always fetched from the closest level of the memory hierarchy that has the desired data at that moment. [1], [2]

The event in which a desired address is found at a particular level of memory hierarchy is called a *hit*. The opposite of hit is a *miss*. The ratio of hits to all accesses is called a *hit ratio*, a number which can be used to measure the performance of a cache. [1], [2]

Caches generally handle data as larger blocks that are called *cache lines* or *cache blocks*. A cache line's size in the magnitude of tens of bytes. [1], [2]

One additional consideration with caches is *coherence*, making sure that all caches match the data held by the memory [1], [2]. This is a challenge both in multiprocessor systems, where there can be both shared and processor-specific caches and memories, but also in systems with input-output devices [1].

2.1 Design

As caches are smaller than the memory, they cannot always hold all the information that is related to the current program context. Instead, attempts are made so that the cached data is from addresses that are likely to be handled soon. To achieve this, caches rely on a concept called the *principle of locality* [1], [2].

Principle of locality is divided into two separate types: s*patial locality* and *temporal locality*. According to spatial locality, when a particular memory address is handled during a program, there is an increased likelihood that the other addresses close to that address will also be soon accessed. According to temporal locality, on the other hand, memory addresses that have been recently accessed, will also be more likely accessed again in the near future. [1], [2]

There are multiple ways in which memory address can be mapped to caches, the most common ones being direct-mapped caches, set-associative caches, and fully associative caches [2], [3]. One example of a less frequently mentioned cache design are subset caches [1].

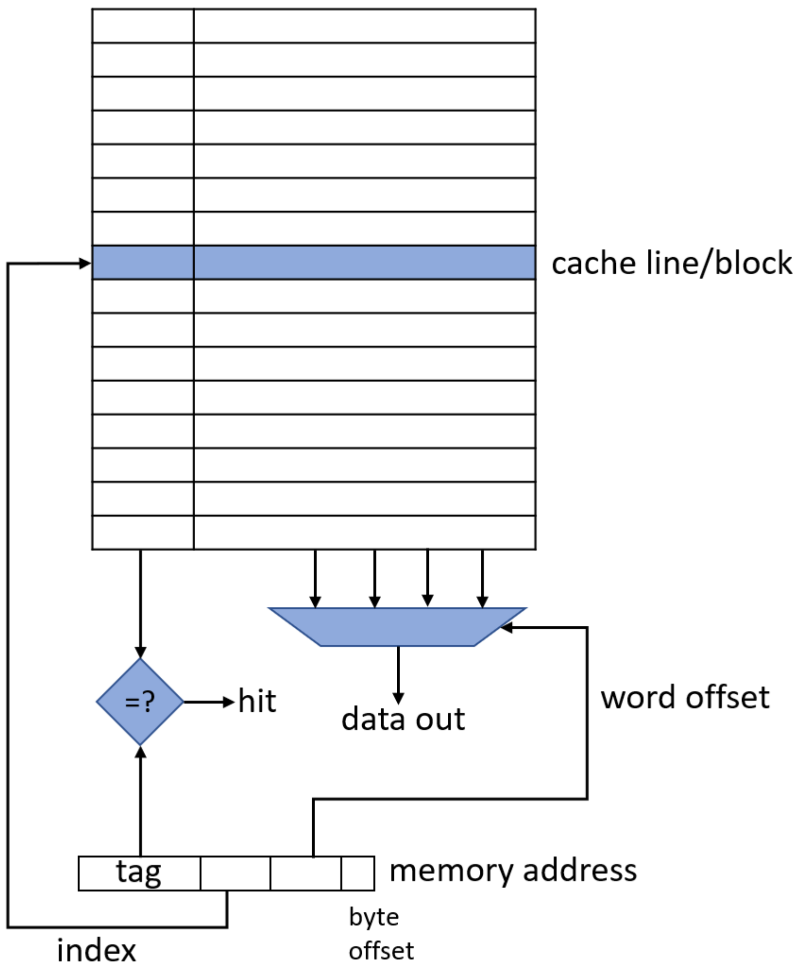

Figure 2: a direct-mapped cache, adapted from [3]

In direct-mapped caches (Figure 2) each memory address maps to only one cache location, i.e. when a particular memory address' contents are cached, the data is always placed in the same location in the cache [3]. The cache line that is to be used for an address is determined by the *index*, the address' lowest significant bits (excluding possible byte and word offset, as can be seen in Figure 2) [1], [3]. Index is also called *line* in some literature [2].

The remaining most significant bits of the address, the *tag*, is then used when data in the cache is accessed, to identify whether the memory location stored in the cache line is the correct one, or if it's holding the contents of another address that shares the same index [2]. In addition to the tag and the actual data, the cache line can have status fields, like a valid bit to indicate that the cache has anything in that index [3], and a clean/dirty bit to indicate that the data held in the cache is still matching with the data in the memory [2].

The benefit of direct-mapped caches is that is only one location of the cache needs to be checked for presence of the desired data, but as a downside, multiple memory addresses will compete for the same cache line [2]. This means that if a program is, for example, frequently accessing two addresses that both happen to map to the same cache line, only one of them can be kept in the cache at a time.

Set-associative caches offer some remedy to this issue as they have cache lines in multiple banks, called *ways* [2], [3]. Set-associative caches are named according to the number of ways they have [2]. For example, 4-way set-associative cache (Figure 3) would allow simultaneous caching of up to four memory addresses that have the same index. The set of cache blocks that share the same index can also be called a *row* [1].

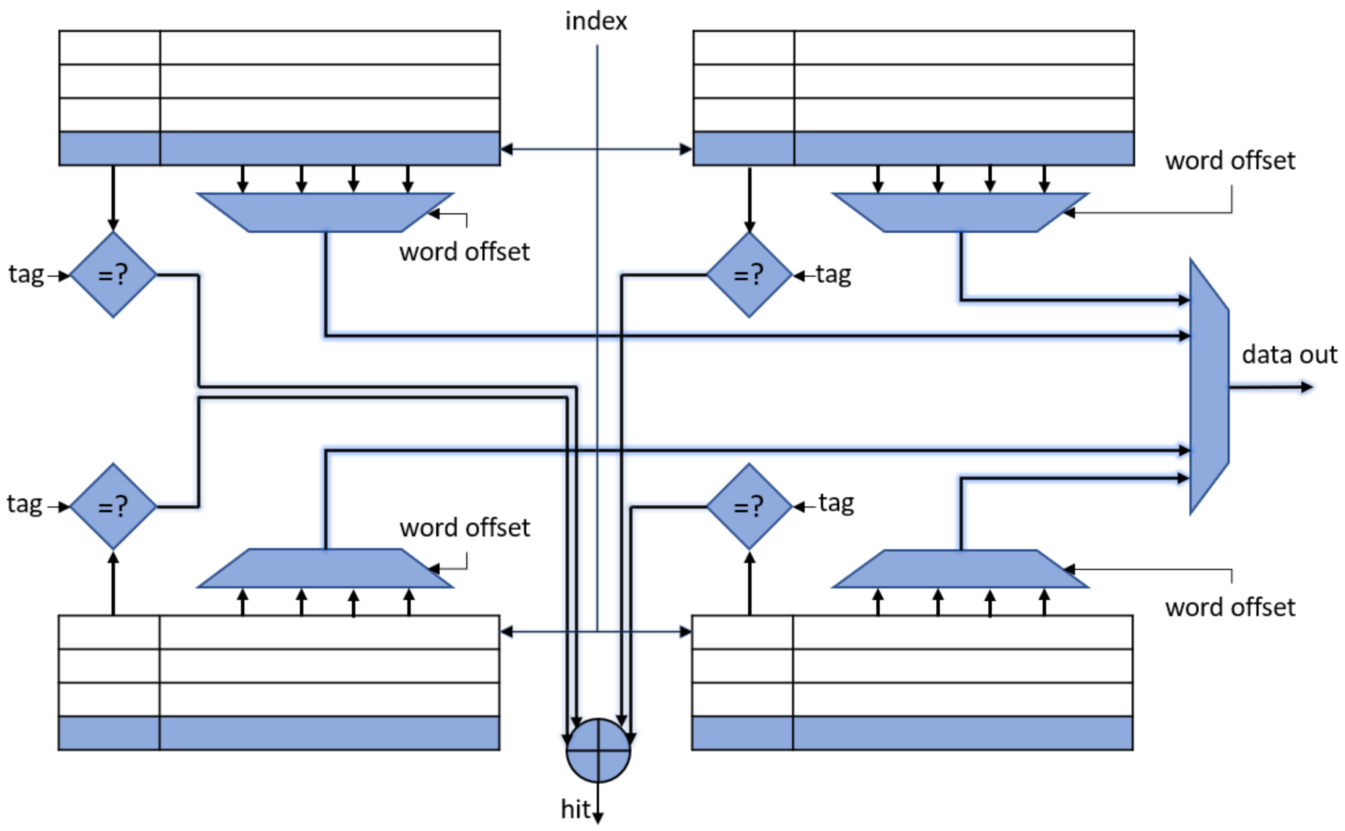

Figure 3: a 4-Way Set-Associative Cache, adapted from [3]

As seen from Figure 2 and Figure 3, the downside of set-associative caches is that every added way also adds more complexity, as they all require comparison logic for the tag [3].

Fully associative caches don't have the placement limitations of direct-mapped and set-associative caches, instead, any memory address can be stored in any location of the cache [2], [3]. This means that there is no indexing [3], but every cache line's tag needs to be checked when accessing memory [1], [2]. Because of this increased need for comparison logic, fully associative caches are rarely used [1].

Caches also have other design choices that affect their functionality. Write misses can be handled in two ways, either *no-write-allocate* or *write-allocate*. When a write miss occurs in a cache that uses no-write-allocate, only the memory of the system is updated, and the address in question remains uncached. Conversely in write-allocate caches, once the write operation has modified the memory, the data is then also stored in the cache. [1], [3]

The way write hits are treated is determined by *write policy*. Write policies can be categorized as either *write-back*, *write-through* or *posted write* aka. *deferred write* [1]–[3].

In write-back caches a write does not go past the cache to the lower-level memory, until the written cache line must be replaced by another location. Instead, when a cache line is modified it's marked as *dirty* by setting a flag bit. When the cache line is replaced, the dirty bit can then be used to determine if the contents need to be written to memory. [1]–[3]

Write-back caches improve performance by reducing the amount memory accesses [1], [3], but they do also have some weaknesses. There is an increased access time when a dirty cache line gets replaced, as it needs to be first written to the memory [3]. Cache coherence needs to be also taken into account [1], [3].

Unlike write-back caches, write-through caches modify both the cache line as well as the memory location on every write. To avoid delay from continuous writes to the memory, a write buffer can be added to the cache. The buffer then takes care of the writes, and next operations can be started while memory is still being written. [1], [3]

In posted write caches writes are done immediately to the cache. The write to the memory is done once the memory bus is free [1], [2].

2.2 Replacement policy

As caches intermittently get filled and have to make space for new data, it is necessary to have mechanism for deciding which older data is evicted. This is the responsibility of a replacement policy. [2], [3]

According to Belady [4], an optimal replacement policy would be one which always chooses for replacement the block of memory that won't be needed for the longest time to the future. This algorithm, called MIN, would require first a pre-run of the program to analyse the memory accesses, and as such is not practical in most cases. However, it can be used for example as a baseline for measuring performance of other replacement algorithms. Conversely to MIN, a worst case for replacement would be to evict a cache line that is used immediately after the eviction.

When it comes to more practical solutions, there are a few of commonly used replacement algorithms which vary in complexity and performance. Such algorithms are Least Recently Used (LRU), Least Frequently Used (LFU), First-In-First-Out (FIFO), and random replacement [2], [3]. Round-robin is also mentioned in some literature [2].

Least Recently Used always replaces the cache line that has not been used in the longest time [1]. In two-way set associative caches an LRU algorithm can be implemented with simple flag bits, which are toggled according to the use [3].

For more complex caches, a stack can be used for LRU bookkeeping. Whenever a cache line is accessed, it’s number is pushed back to the top of the stack, simultaneously shifting other line numbers down the stack. When a line has to be evicted, the number at the bottom of the stack indicates the line to be replaced. [1], [2]

There are also pseudo-LRU algorithms which approximate true LRU, by attempting to mimic the algorithm while requiring less hardware or being easier to implement. One example is using a binary tree for storing the recency information, instead of the stack. [5], [6]

A FIFO replacement algorithm naturally uses a first-in-first-out buffer, and unlike with LRU, line numbers are not pushed back to the top of buffer if they are used again. An older line will get pushed to the bottom of the FIFO even if it has been accessed multiple times after its initial use. As a result, its performance is worse than LRU for cache lines that are accessed continuously. [2]

An LFU replacement algorithm replaces the line that been used least frequently, and as the implementation requires counters and comparison logic, it is more complex in hardware [2], [3]. As a benefit its performance is also very good [2].

A random replacement policy picks the replaced line at random. It is easy to implement as it doesn’t require any bookkeeping. Despite this its performance is relatively good. [2], [3]

Round-robin is a cyclic replacement policy, which replaces each way in turn. It’s said to be easy to implement but to not be very efficient for small caches. [2]

As direct-mapped caches have only one cache line to which a memory location is mapped to, there is no need for a replacement policy [3].

In addition to these more established replacement policies, that are common enough to appear in computer engineering books, there are also other more recently developed algorithms. Some of these are explored in the Related Works chapter, along with an investigation into the specifics of RISC-V cache replacement policies.

3 PROJECT AND PLATFORM

SoC Hub is a consortium project started by Tampere University in collaboration with founding partner companies Nokia, CoreHW, VLSI Solution, Siru Innovations, TTTEch Flexibilis, Procemex, Wapice and Cargotec. SoC Hub officially launched on 27th of February in 2021, and received funding from Business Finland. [7]

The SoC Hub ecosystem aims to collect and spread expertise of SoC development among industry professionals, students, and researchers [8]. The goal is to create three chips, and the first them, Ballast, was taped out near the end of 2021 [7].

3.1 Ballast

Ballast's design and implementation was finished near the end of 2021, when it was taped out. Samples of the chip were received in June of 2022, with a successful wake-up two days later. [7], [8]

At the top level, Ballast (Figure 4) consists of eight subsystems all of which are connected via AXI interconnects. The subsystems are SysCtrl (System Control subsystem), MPC (Medium Performance Computing subsystem), HPC (High Performance Computing sub-

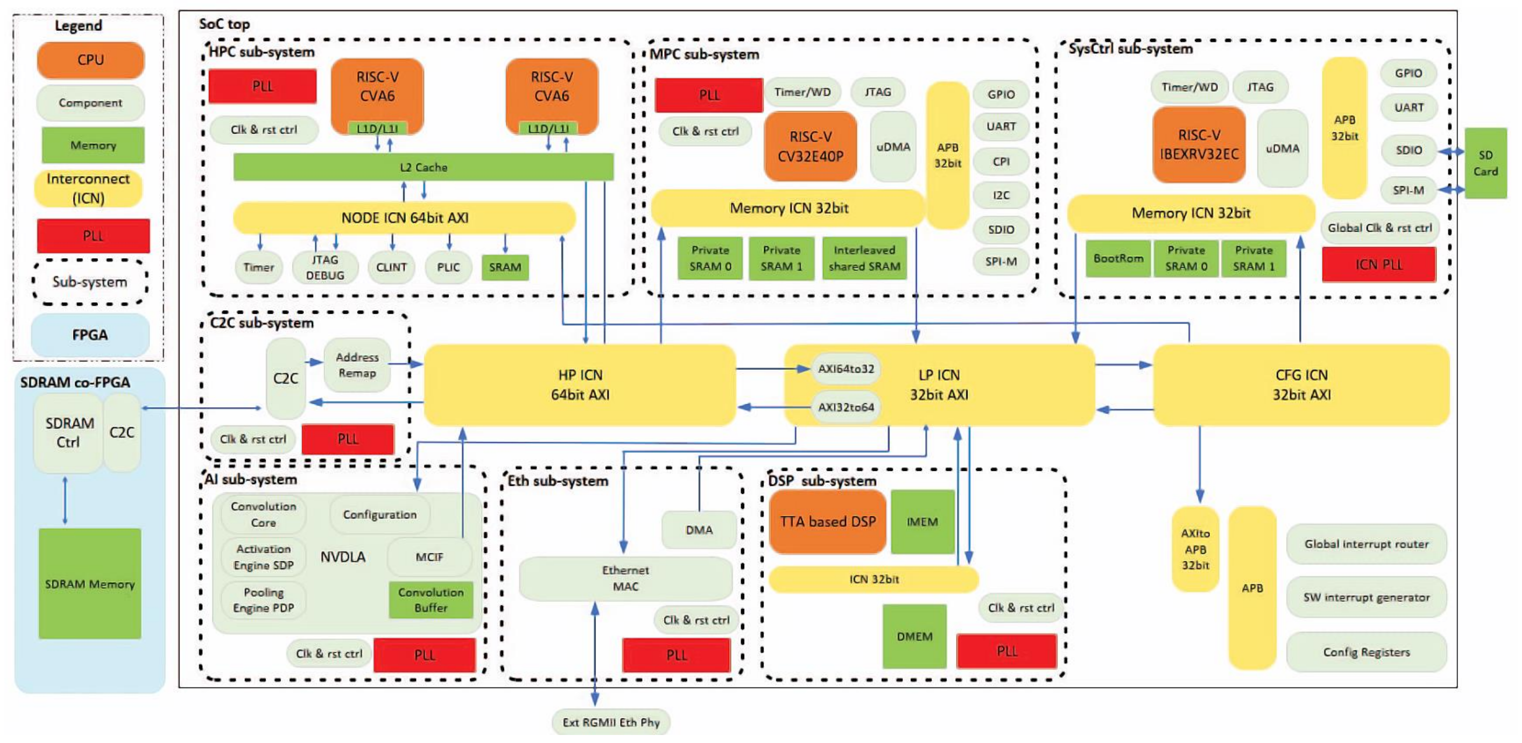

Figure 4: Ballast SoC [9]

system), AI (Artificial Intelligence subsystem), DSP (Digital Signal Processing subsystem), Ethernet subsystem, C2C (Chip-to-chip subsystem), and a top-level peripherals subsystem. [9]

This work focuses on the HPC subsystem. Performance measurements simulate use cases where a single CVA6 core from HPC subsystem uses an SDRAM memory which would be located behind the Chip-to-Chip subsystem. Communications between HPC and C2C happen on the High-Performance Interconnect (HP ICN), which is a 64bit interconnect using an AXI protocol. [9]

3.2 HPC Subsystem

HPC subsystem is built around two CVA6 cores (CORE-V Application Class, 6-Stage Core) [10], [11]. CVA6 is 64-bit RISC-V core that implements an RV64GC instruction set [12]. The core was designed at ETH Zurich, and it was originally called Ariane [12]. In 2020 the core was handed over to OpenHW Group for verification, maintenance, and documentation, and it was renamed to CVA6 [13]. For the sake of clarity this thesis uses the newer name. The CVA6 cores in the subsystem are identical, except for minor configurable details such as their internal ID code.

The RV64GC architecture of the core includes implementations of the integer, multiplication and division, atomic, single-precision floating point, double-precision floating point, and compressed instruction extensions of the RISC-V ISA specification [14].

The core has out-of-order execution, but in-order commits [13]. It’s Linux-capable, and among other features it has a branch-predictor, a memory management unit, and L1 instruction and data caches. CVA6 pipeline, depicted in Figure 5, has 6 stages: PC generation, instruction fetch, instruction decode, issue, execute, and commit [15].

The L1 data caches in HPC’s two CVA6 cores are configured as 32kB 8-way-associative with 128-bit cache lines. The L1 instruction caches are 16kB 4-way-associative with 256-bit cache lines. Both caches have a random replacement policy, which uses a linear-feedback shift register to generate pseudo-random numbers. In Figure 5, data cache (D$) and instruction cache (I$) are highlighted in blue.

The interface that connects the CVA6 cores to the L2 cache is based on CPU-Cache Crossbar (CCX), which was used by OpenSPARC T1 [16], and more recently by OpenPiton [17]. CCX uses packets for requests and responses. CVA6 does not use all the features of the packets, leaving some of the fields unused.

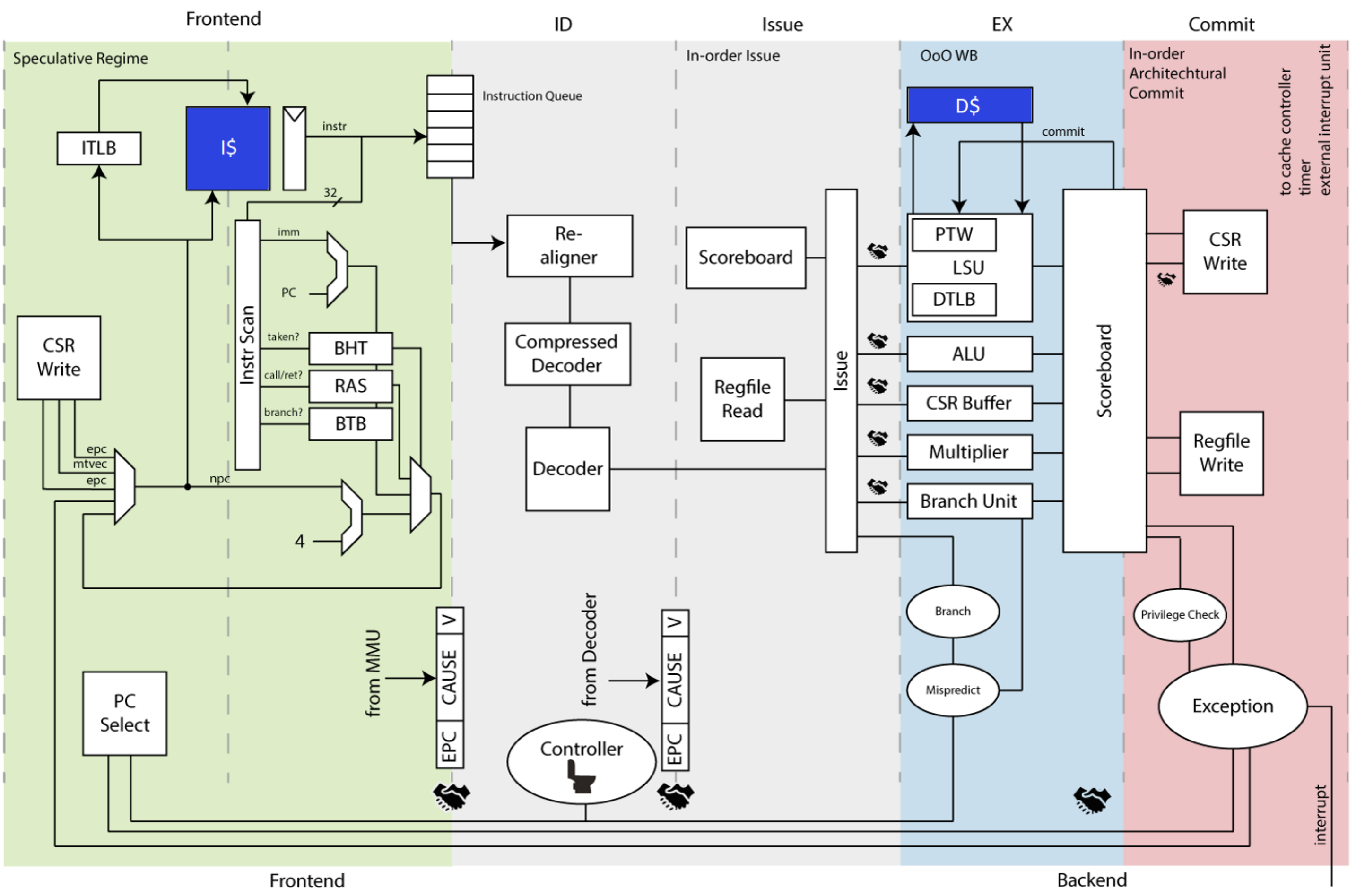

Figure 5: CVA6 processor pipeline [11], modified

HPC's interfaces are routed through the L2 Cache Subsystem, which implements the L2 cache itself, as well as a bypass module which is used if the L2 is disabled. The L2 cache in HPC is an 8-way-associative 256kB cache, and it provides cache coherence. Both the L2 and the cache bypass are controlled by finite state machines, which process the CCX requests coming from the core.

From the L2 cache forward, the subsystem uses AXI protocol for communication. In addition to controlling the state of the L2, the state machines implement the necessary logic for atomic requests, and they are responsible for the CCX-to-AXI and AXI-to-CCX protocol conversions. A separate state machine controls the cache coherence logic. The L2 cache uses a write-back policy, which reduces the need for constant AXI transfers to memory.

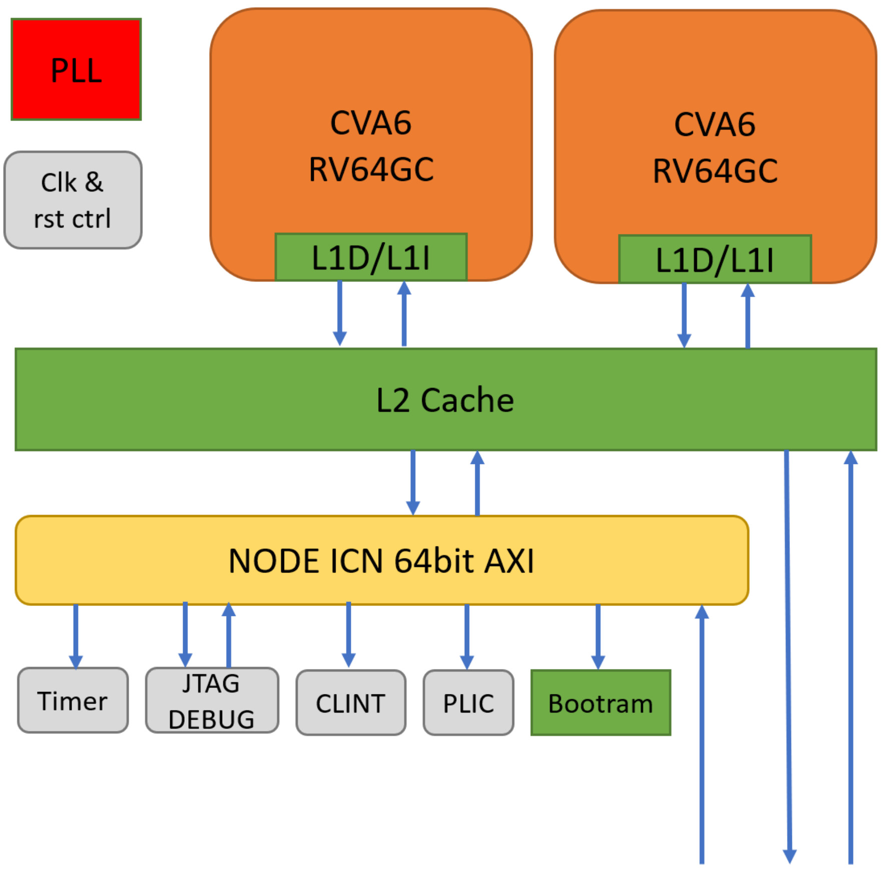

Figure 6: HPC subsystem, adapted from [9]

The replacement policy originally implemented in the L2 cache is a simple LRU implemented with a stack. This LRU is used as the baseline for comparing the performance of other replacement policies implemented in this work.

In addition to the two cores, the HPC subsystem (Figure 6) has a set of peripherals which include CLINT (Core-local Interrupt Controller with an RTC timer), PLIC (Platform-Level Interrupt Controller), a set of APB timers, a debug module with JTAG interface, and an SRAM memory (Bootram). Additionally, there are configuration registers for the subsystem, and separately for the L2 cache.

4 RELATED WORKS

As the goal of this thesis is comparison of cache replacement algorithms, the research to related works had a main goal of identifying potential candidates to be used in the comparison. First stage in this was to study whether any relevant works had been done specifically with CVA6. This required looking for any platforms which had used CVA6, and which also had implemented an L2 cache with a replacement policy algorithm that is of some interest.

Since the replacement policy used by the HPC subsystem L2 was an LRU, comparison with additional standard LRU implementations was not meaningful. Additionally, the focus was on finding algorithms that are more accurate than LRU, meaning that algorithms that perform worse, such as FIFO [2], were not of much interest either.

4.1 CVA6

One project that has utilized CVA6 in combination with an L2 is ESP (*Embedded Scalable Platforms*), which is described as a "open-source research platform for heterogenous system-on-chip (SoC) design and programming". SoCs built with ESP use a tile-based design, and there are four types of tiles available: processor, accelerator, memory, and auxiliary. For the processor tile, two core choices are available: CVA6 and LEON3. Regardless of which core is used, the tile also includes an L2 cache. [18]

The L2 caches on ESP's processor tiles are private write-back caches, and they use a MESI cache-coherence protocol. Additionally, the memory tiles of ESP feature a last level cache (LLC) [18], [19]. The LLC's replacement policy is described as "FIFO-like" [20]. L2 cache's policy is not specified by documentation but appears similar based on reviewing the code.

Another tiled platform that has used CVA6 is OpenPiton [21]. Though originally built on using the OpenSPARC T1 core [22], OpenPiton+Ariane was later developed together with the PULP team from ETH Zürich [21]. Further, OpenPiton has "Bring Your Own Core" (BYOC) architecture, which enables using other processors with different architectures [17]. In addition to OpenSPARC and CVA6, the platform has been used for example with PicoRV32, ao486, and BlackParrot.

OpenPiton is described as a "tiled-manycore architecture" with "extreme scalability". Each of the tiles includes a processor core, an L1.5 cache, an L2 cache, and three Network-on-Chip routers. The NoC routers are used for cache operations, and for connecting each tile to its adjacent tiles to form a 2D mesh. Together the connected tiles form what OpenPiton calls a chip. Within the chip, the L1.5 and L2 caches and networks-on-chip form a cache coherence system called P-Mesh. Each chip connects through a chip bridge to a chipset, which is where DRAM and I/O are found. Chipsets are connected to other chipsets which allows for inter-chip communication. [17]

When it comes to cache replacement algorithms, according to the source code OpenPiton uses an LRU algorithm for L1.5 caches, and a pseudo-LRU implementation for L2 caches [23].

A third platform that supports using CVA6, among many other cores, is the Chipyard framework. Chipyard uses Rocket Chip and other generators to create RTL for SoC designs, and it also bundles together different libraries, simulators, and other tools. [24] [25]

Chipyard's default implementation for L2 caches is to use SiFive's InclusiveCache generator [25]. Due to being a generated design the details of the cache depend on the configuration parameters. However, regardless of the other details, the generated caches always use a random replacement policy. As with CVA6's L1 caches, an LFSR is used for generating pseudo-random numbers. [26], [27]

Wistoff et al. [28] utilized CVA6 in a study of timing channels, a type of security hazard. The work features an L2 cache, however the details of the implementation are covered by a thesis [29] that is not publicly available.

In addition to the works there are multiple other works which have used CVA6, but which do not particularly relate to caches or specify any kind of details of replacement policies. Such cases include researching alternatives to standard floating-point representations [30], run-time variable floating-point precision [31] and logic-locking for increased hardware security [32].

There is also a commercial processor, Mig-V, which uses a CVA6 core [33]. Mig-V is described as a logic-encrypted general-purpose processor. According to the brochure the processor has instruction and data caches, but it is not explicitly stated if these are the same caches CVA6 includes by default [34]. The replacement policies used by these caches are not mentioned.

As such, there are no significant findings when it comes to replacement policies and the CVA6 core specifically. However, other RISC-V designs and their cache implementations were also investigated.

4.2 Other RV64 implementations

As the RISC-V space is very populated by a large number of different designs [35], the focus was limited specifically to 64-bit cores which have close to the same extension set as CVA6. One 64-bit RISC-V design is BlackParrot, which is another tile-based SoC generator [36]. It implements the RV64G ISA, and the C extension is planned [37]. In BlackParrot's architecture, following tiles are available: core, L2 extension, coherent accelerator, streaming accelerator, I/O, and DRAM Controller [36]. The tiles form 2D mesh via BedRock, which is network that each of the tiles is connected to.

The L2 cache in BlackParrot is distributed. Each of the core tiles has an L2 slice, and if there is need for more capacity, the L2 extension tiles can be used. Core tiles have L1 instruction and data caches. The BedRock network implements cache coherency and supports VI, MSI and MESI coherency protocols. [36]

The replacement policy in L1 instruction and data caches of BlackParrot is a tree pseudo-LRU from the BaseJump Standard Template Library. Core tile L2 slices and L2 extension tiles use a cache implementation which is also from BaseJump. These L2s are set associative, write-back, write-allocate caches, and they use the same tree pseudo-LRU replacement policy as L1. [37]–[39]

RiscyOO is an out-of-order RISC-V processor which implements the RV64G (RV64IMAFD) architecture. A multicore system using RiscyOO processors has been implemented on FPGA with an “uncore” which contains an L2 cache, DRAM wrapper for the FPGA's DRAM, MMIO platform, Boot ROM, and a Memory Loader for loading data from a host system. The L2 is a shared cache, which connects to the cores' L1 data and instruction caches with FIFO interfaces. [40]

The L2 implementation in RiscyOO's uncore is an LL cache [41], and it uses a random replacement policy [40]. The L1 data and L1 instruction caches use an LRU replacement policy [41].

Another 64-bit RISC-V core is the NOEL-V. It implements an RV64GC instruction set, and it is part of the GRLIB IP Library. GRLIB is an open-source library which includes a variety of different IP, such as the LEON SPARC processors, timer units, and SRAM, USB and I2C controllers. The library also includes L2 cache controller implementations, and the L2C-Lite variation of the cache is compatible with NOEL-V processors. The L2C-Lite line and way sizes are modifiable, and the cache is configurable as either direct-mapped or set-associative. Replacement policy can be selected either as pseudo-random or pseudo-LRU. [42], [43] There is also Lagarto I, a core with a RV64IMA instruction set, which utilizes the OpenPiton architecture. However, the core was not implemented with new cache designs, rather it reuses the L1 from CVA6, and L1.5. and L2 from OpenPiton. [44]

Shakti is a family of multiple different processor designs. The processors are organized into three base classes, three multicore classes, and additionally two experimental classes. The bases designs include a low-power embedded E-Class, a mid-range Linux-capable C-Class, and a high performance out-of-order I-Class. The multicore designs are M-Class, S-Class, and H-Class, and they use different combinations of processors from the C- and I- base classes. The experimental processors are security and fault tolerant T- and F-Class. Most information is available on the E-Class and C-Class processors, which have also been taped out with different configurations. [45]

The C-Class processor has configurable instruction and data caches. The configurability extends to replacement policy, which can be set as either random, pseudo-LRU, or round-robin. [46]

Overall, there appears to be little diversity in replacement algorithms, at least in the field of 64-bit RISC-V processors. Most systems discovered in this research implement either a random policy, FIFO, or some variation of LRU. This notion is somewhat corroborated by Ghasemzadeh and Fatemi [47], who state that there have been numerous studies on various replacement algorithms, but only a few of them get regularly implemented on hardware.

4.3 Cache replacement algorithms

It was considered that there may be good reasons for the perceived lack of variation in replacement algorithms in existing implementations. Possibly the mentioned algorithms perform well enough, and there is no appreciable increase in performance when using other algorithms. Another possibility was that there may be a notable performance increase, but that the implementation of better performing algorithms is not worth the required effort or cost.

To study this, further research into alternative replacement algorithms was carried out. As strong candidates for the algorithm comparison were not identified from other RISC-V systems, they would need to be discovered from other research.

As already presented in this chapter, the LRU algorithm is very popular. In further investigation it was found that there have been multiple attempts to increase LRU accuracy by augmenting the basic least-recently-used philosophy with additional features.

One such work is presented by Wong & Baer [48], who modify LRU by introducing more temporal locality into the replacement decision. Cache lines on the LRU stack are divided into two categories, those which display “sustained temporal locality”, and those which do not. Cache lines with sustained temporal locality are preferred to be kept in the cache, even in cases where such lines may be least recently used. Two different algorithms are presented for profiling which category a cache line belongs to.

Another LRU-based algorithm is the Enhanced LRU (ELRU). ELRU has counters on the accesses to each cache block. During evictions, the average counter value of the blocks is calculated, and the actual LRU is only applied to cache blocks that have below average counter value. According to the provided results this algorithm does perform better than basic LRU, however the provided results are based only on randomly generated memory accesses. [49]

One simple pseudo-LRU implementation is the MRU-based pseudo-LRU (PLRUm). In PLRUm, every cache line is given an MRU-bit (*most recently used*). This bit functions as a flag that is set to 1 when the line is used. If all other flags are already set to 1 when the last way is toggled, all MRU-bits of the set are reset back to 0, and only the most recent one is toggled. Whenever a line has to be evicted, the algorithm simply picks the first line which MRU-bit is 0. PLRUm should match or even outperform pure LRU. [50], [51]

Modified Pseudo-LRU (MPLRU) is an LRU approximation based on a binary tree implementation. MPLRU takes a Basic Pseudo-LRU (BPLRU) and introduces additional Multiple Block Access Indicators (MBAI) to its binary tree structure. These MBAI bits are used to indicate the history of accesses, by storing both the current and previous status of the binary tree, instead of only the current status. MPLRU performs better than the BPLRU, though does not match the accuracy of LRU. [5]

Another pseudo-LRU is the pseudo-FIFO LRU. Like many pseudo-LRU implementations, this algorithm focuses on improving the memory footprint by removing the requirement for a stack. In addition, it aims to improve the timing requirements to increase the maximum clock rate. According to the provided results the pseudo-FIFO has slightly higher miss rate than a full LRU, i.e., it’s less accurate. [47]

There are also designs that attempt to combine features of multiple different algorithms. One such case is Effectiveness Based Replacement, which aims to combine features of both LFU and LRU into a single algorithm. Each cache line has two counters, one for frequency and one for recency. The frequency counter increments whenever the cache line is accessed, and the recency counter whenever the cache line is not referenced for multiple misses. Based a on these counters, and additional weight variables, an effectiveness value is calculated for each line. The evicted cache line is chosen based on lowest effectiveness. [52]

Another algorithm that similarly aims to combine LRU and LFU is the Least Recently/Frequently Used (LRFU). Similar to EBR, LRFU calculates a value called Combined Recency and Frequency (CRF) for each cache line. The CRF value is based on amount of references to the cache block, and the times they occurred. Since storing the entire history of accesses to each block would take a lot of memory, only the calculated CRF is stored. When a new reference occurs, the CRF is again recalculated, based on the age and value of the previous CRF. [53]

Basic, Dynamic, and Adaptive Cost-Sensitive LRU (BCL, DCL, ACL) are a set of replacement algorithms which have been designed to consider the cost of replacing a cache block, in addition to recency. Even if a cache block is the least recently used one, it can be kept in the cache if evicting it is more costly than evicting some other block. Such blocks are referred to *reserved*. Costs of different blocks can be estimated based on the miss latencies, bandwidth consumption, or other effects, and can depreciated over time. [54]

State-based random replacement is an algorithm that is specifically designed for multiprocessor environments with shared MESI coherent memories. It aims to improve the performance of random replacement by considering cache lines' coherence state, as well as the state of most recently used cache line. Instead of picking a random cache line from the whole cache, the randomization only applies to a subset where all cache lines have the same state. For example, highest priority is Invalid cache lines, followed by Shared, Exclusive, and lastly Modified. In addition to the predefined priority order, a limitation is set so that the MRU state is considered for eviction last. [55]

Hawkeye is a predictive replacement policy which won the 2nd Cache Replacement Championship held in 2017 [56], [57]. Following this, a more advanced version called Mockingjay was later developed [58], [59]. Mockingjay samples the memory accesses done during execution, and then uses those samples to predict future memory operations. Based on these, addresses which are predicted to be used sooner are prioritized to be kept in the cache, while addresses with longer estimated reuse can be evicted.

This concludes the Related Works chapter. The algorithms which were selected for comparison are introduced in further detail in the Algorithms chapter.

5 ALGORITHMS

After the research described in the Related Works chapter was carried out, a set of algorithms had to be selected for implementation and comparison. To select the replacement algorithms which would be included in this work, the different options were collected and processed based on a few simple qualitative criteria.

5.1 Selection

The initial selection could not go very deep into the fine details each algorithm, as that was estimated to be too time-consuming. Therefore, the criteria used was only roughly defined: good performance, accurate enough specification or description, variety, and recency.

The number of candidates was not initially restricted to a specific number, but it was estimated that an amount of three to five was realistic given the size of the work. Adding more algorithms would increase the time required for both the RTL implementation and the performance measurements.

Performance was the highest criteria. As the goal of this work was to increase the performance of HPC, the requirement was that any implemented algorithms would have to surpass, or at the very least match the performance of the LRU algorithm. Due to the popularity of LRU, most papers published on cache replacement algorithms do use it for comparison, making it a convenient benchmark.

The second consideration was the level of description that was available for the candidate algorithm. An algorithm can be specified in many ways. The most accurate ways are a full source-code or a more abstract pseudocode. Different diagrams or other graphical descriptions can also be used, and they are often utilized for visualization along other types of specification. A purely verbal documentation can also be used.

There is no one answer for what is the correct type of specification. However, the more complex the algorithm, the more detailed the documentation should be. The more structured descriptions can also be more accurate and thus less open to interpretation or mistakes in implementation. In practice, algorithms which only had a high-level conceptual description were not considered for implementation. Some otherwise interesting algorithms were excluded due to their specification being too high-level.

As previously mentioned, the amount of selected algorithms had to be restricted. Therefore, the focus was put on having a handful of algorithms, but with each of them being different in nature and in complexity.

Additionally, the recency of the algorithms was considered. This requirement was added mainly to catch any latest developments in cache replacement algorithms.

All considered algorithms are listed in Table 1. Algorithms were searched for in the two manners described in Related Works. Firstly, other high-performance RISC-V platforms were investigated to find out what cache replacement algorithms they use, and secondly, sources that specifically focused on cache replacement algorithms were investigated. Algorithms in the first group are ones that have their respective platform or processor included in the table. There is some overlap in this group, as many different RISC-V platforms use the same algorithms. Some platforms have multiple algorithms if they are configurable or use different algorithms for different cache levels. Year refers to date of related publication, which is not necessarily the same date the algorithm was originally designed.

Final decision was to choose the algorithms PLRUm, EBR, and Mockingjay for the actual implementation and comparison. Each of these algorithms is described in greater detail in following subchapters.

Table 1: Cache Replacement Algorithms

Algorithm	Ref.	Platform	Specification or language	Year	Selected
Least Recently Used (LRU)		Ballast (HPC)	SystemVerilog		Baseline
Mockingjay	[58], [59]		C++ model, diagrams	2022	Yes
Effectiveness Based Replacement (EBR)	[52]		Pseudocode, diagrams	2014	Yes
MRU-based Pseudo-LRU (PLRUm)	[50], [51]		Diagrams	2004	Yes
Pseudo-FIFO LRU	[47]		Pseudocode, diagrams	2005	
Modified Pseudo LRU (MPLRU)	[5]		Diagrams	2006	
Basic Cost-Sensitive LRU (BCL)	[54]		Pseudocode, diagrams	2003	
Dynamic Cost-sensitive LRU (DCL)	[54]		Diagrams	2003	
Adaptive Cost-sensitive LRU (ACL)	[54]		Diagrams	2003	
State-based priority random replacement	[55]		Diagram	1998	
Least Recently/Frequently Used (LRFU)	[53]		Pseudocode, equations	1999	
Reference Locality Replacement (RLR)	[48]				
Enhanced LRU (ELRU)	[49]		Pseudocode	2017	
Hawkeye	[56]		C++ model, diagrams	2016	
FIFO-like	[18]–[20]	ESP (CVA6)		2018	
Pseudo-LRU (L2) and LRU (L1.5)	[17], [21]–[23], [44]	OpenPiton (CVA6/OpenSparc/Lagarto)	Verilog RTL	2016	
Random	[24]–[27]	Chipyard (CVA6/Rocket/Others)	Chisel RTL	2020	
Pseudo LRU (L2 and L1)	[36], [37]	BlackParrot	SystemVerilog RTL	2020	
Unknown	[33], [34]	Mig-V (CVA6)	Unknown		
PLRU, random	[42], [43]	NOEL-V	VHDL	2020	
Random, LRU	[40], [41], [60]	RiscyOO	Bluespec SV	2019	
Random, RR, PLRU	[45], [61], [62]	Shakti (C-Class)	Bluespec SV	2016	

5.2 PLRUm

PLRUm was chosen to represent the vast amount of LRU and pseudo-LRU algorithms. Its specification was simple, but unlike many other pseudo-LRUs, the accuracy was claimed to be higher than pure LRU making it a very compelling algorithm. Al-Zoubi et al. [51] specifically state that "For second level unified cache L2U, both PLRUm and PLRUt outperform LRU for even more cache organizations than in first level caches.", which makes PLRUm very appealing for the case of HPC L2. The mentioned PLRUt is not included in this thesis as Al-Zoubi et al. also note that PLRUm outperforms it. Additionally, PLRUm had been chosen as the best option in a another Master's thesis [50], which also made it appear a proven choice.

PLRUm's functionality is built around an array of MRU-bits. In this array, every cache line of the cache has its own flag bit which indicates if the line has been recently accessed.

The update and eviction sequences go as follows. At initialization, all bits are set to 0. Whenever a line of the cache is accessed by read or write, the associated flag bit is toggled to 1. If after this toggle every bit of the same set is 1, all the bits in the same set are reset to 0, except for that one that was just toggled (stage "Write e" in Figure 7).

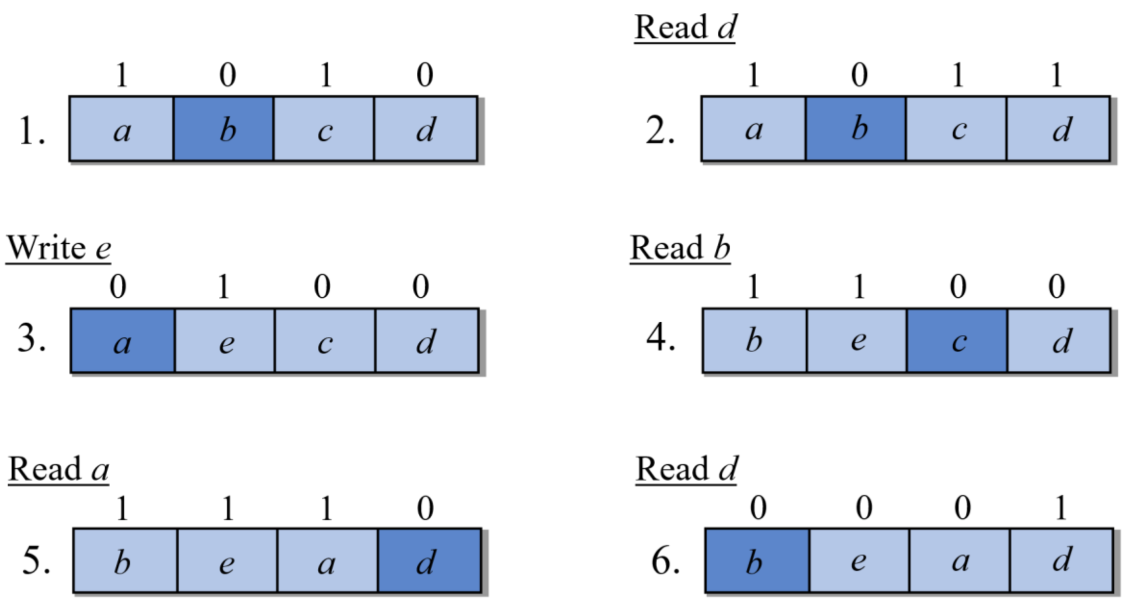

Figure 7: PLRUm example sequence, adapted from [50]

Whenever a cache line needs to be evicted, the first line with MRU bit 0 is chosen as the victim. In Figure 7 the victim way is highlighted.

5.3 EBR

EBR was chosen as it is a hybrid of the features of LRU and LFU algorithms, but with a practical set of requirements. The LRFU algorithm [53] was also considered in its stead, due to the similar nature of the two, but the implementation did not seem as straightforward as EBR.

The cost based BCL, DCL or ACL [54] would have also been similar in that they assign a value for each cache block, to aid in replacement decisions. However, this set of algorithms seemed more geared towards multiprocessor systems with multiple different memories. While any of these algorithms would likely be very interesting to use on the full Ballast SoC, the planned single-core single-memory simulation setup used for the benchmarking in this thesis may have not shown its full capabilities, and practical implementation of the cost function would have likely been challenging. EBR's counter based design therefore seemed to better align with the parameters of this thesis.

To explain EBR in closer detail, the algorithm assigns two counter registers for each line of the cache: *R-counter* and *E-counter*. R-counter is used for tracking references to each block, while E-counter is used for tracking how much time has elapsed from last reference. Additionally, two weight values are used to emphasize the different counters: frequency weight *f*, and recency weight *r*. These counters and two weight values are used to calculate the effectiveness value *E* of each way using equation (1).

$$\boldsymbol{E(i) = \frac{r * R_{counter}(i)}{f * E_{counter}(i)},} \tag{1}$$

When a line needs to be evicted, the effectiveness value is calculated for all ways of the same set, and the way with the lowest effectiveness is chosen as victim. In the case that multiple cache lines have the same lowest effectiveness value, the choice between them is randomized.

The R-counter is incremented every time the related cache line is accessed, but E-counter's handling has more steps. Each sets' E-counters have one shared counter assisting them. This counter's purpose is to count misses which occur in its set. The E-counters are then only incremented when this miss counter overflows. E.g., if the miss counter is a 2-bit register, it will only overflow on every 4th miss. Whenever the 4th miss happens, all E-counters in the same set are incremented by 1.

Unlike the miss counters, the E- and R -counters are saturating i.e., once they reach their maximum value, they do not overflow. E-counters are reset when the associated cache line is hit, or if the cache line evicted to bring in new data. R-counters are only reset on evictions.

EBR also has a more advanced Dynamic (EBR-D) variant. However, in this thesis only the simpler variant is implemented. The main difference between the regular and dynamic variants is that EBR-D adjusts the recency and frequency weights *r* and *f* during execution, while in the regular version both weights are static. Tian & Liebelt's [52] guideline for the static weights was that recency weight *r* should be much greater than frequency weight *f*.

5.4 Mockingjay

Mockingjay [59] is a predictive replacement algorithm, which aims to replicate Belady's optimal algorithm by estimating how soon different cache lines will be reused. It was selected as the third algorithm due to its performance. As discussed in chapter 4.3, the older Hawkeye version was shown to be a very high accuracy algorithm, and Mockingjay itself should outperform it.

The algorithm (Figure 8) consists of three main components: Sampled Cache, Reuse Distance Predictor, and the ETR (Estimated Time Remaining) counters. Additionally, there are a group of timestamp registers, one for each set, and similarly a group of ETR clocks, also one for each set.

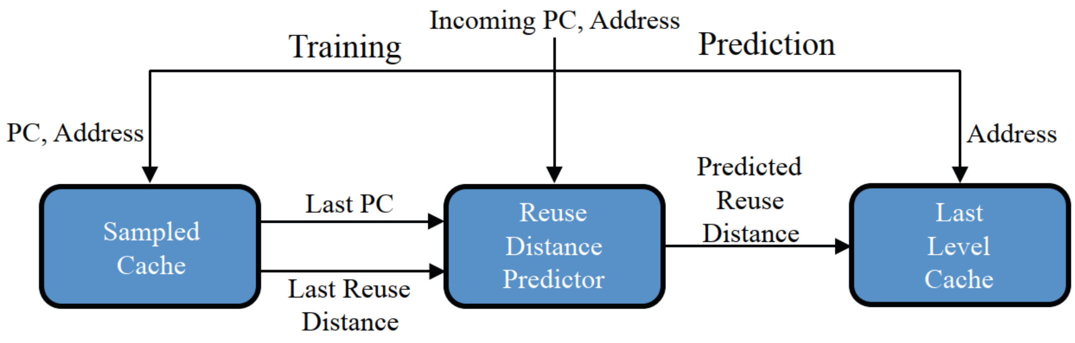

Figure 8: Mockingjay algorithm

Sampled cache is a separate 5-way set-associative cache, included only for the use of the Mockingjay replacement algorithm. Its purpose is to collect data on the past accesses to the actual L2 cache. It doesn't log every reference, but rather only those that target sampled sets of the L2. Every 32nd set is sampled, meaning that 32 of HPC's 1024 sets are logged. With a history length of 8 in an 8-way cache, this means that each sampled sets' last 64 accesses would be logged, making the full-size sampled cache require 2048 cache lines. In practice, however, only references to unique addresses are stored, and the sampled cache's size with its 5 ways is limited to only 1280 cache lines.

Each cache line in the sampled cache consists of a valid bit, a 10-bit tag from the requested address, an 11-bit program counter signature, and an 8-bit timestamp. The PC signature is a hashed value that is calculated from part of the actual program counter associated with the request, a hit/miss bit, and the core ID of the request. The timestamp is assigned according to the set's current timestamp value at the moment the request is logged.

Like other caches, the sampled cache also needs a replacement policy. As the sampled cache lines include a timestamp by design, an LRU policy can be implemented without requiring a separate stack or other additional data.

Reuse distance predictor (RDP) is the second major component. As the name suggests, its goal is to predict how long a time will take until a given PC signature is encountered again. The RDP functions like a direct-mapped cache, with PC signatures serving as the index. Each line in the RDP only stores a valid bit and a 6-bit reuse distance. RDP entries are updated using temporal difference [63], which considers both the previous predicted distance and the newly witnessed one.

The last main component are the ETR counters. Each cache line in the L2 has its dedicated counter, which maintains an estimation of how soon the line should be reused. These counters are based on RDP's predictions. The ETR value can be either a positive or a negative value. Positive ETRs are those which should take place in the future, while negative ETRs are those which, according to the estimation, should have already occurred.

ETR counters provide the data for the actual eviction decisions, with the longest ETR of the set always being the victim. Since the ETR values are signed, the comparisons are done using the absolute values.

The algorithm also maintains registers of timestamp counters, one for each set, which are used when a new line is entered into sampled cache, as well as when calculating the amount of time that has passed since the last access to the set.

An additional group of ETR clocks, also one for each set, are used to for counting down the main ETR counters. Each ETR clock is incremented on every access to its set. Every time the ETR clock overflows, the main ETR counters are decremented. Consequently, the purpose of the ETR clocks is comparable to the miss counters in EBR algorithm (chapter 5.3).

Mockingjay also requires definitions for multiple different parameters that direct its behaviour. Of them, a few are particularly important. Infinite reuse distance (INF RD) is the highest possible value for reuse distance, i.e., 63 in the case of HPC's 6-bit RDP registers. While the literal number value is not infinite, it represents PC signatures which are predicted to be not reused in the current lifetime in the cache. Shah et al. [59] refer to such non-recurring accesses as scans. Cache lines that are a part of a scan are considered low-priority and therefore are preferred for eviction. Another related parameter is maximum reuse distance (MAX RD). Any predicted reuse distances above the maximum are considered infinite. For the HPC implementation, maximum reuse distance is defined as 53. To match INF_RD, the ETR counters have their own infinite value, called INF_ETR. For HPC this value is defined as 7, for the signed 4-bit registers. While other ETR counters count down to estimate the time of next access, any that are set to INF_ETR retain that maximum value.

Mockingjay's predictions are updated as follows. When a new request comes into the L2, the algorithm first checks if the line is part of a sampled set. If it is, the update process is longer. First the sampled cache tags are checked to see if the same address already exists in it. If the line is found in the sampled cache, the algorithm calculates how much time has passed since the last access to the address by using the timestamp from the last sample and the current timestamp from the timestamp counters.

The result of this calculation can then be used to update the relevant PC signature in the RDP. If the PC signature already has a valid prediction in the RDP, temporal difference is used to update it. Otherwise, the RDP is initialized to the amount of time that had passed according to the timestamps. After this the Mockingjay state machine goes to its sampled cache LRU process. If the sampled cache line was not found earlier, the previous steps are skipped and the LRU is entered directly.

During sampled cache LRU, a victim sample line is chosen. Invalid lines are identified, but if none are present, the line with the oldest timestamp can be used. Additionally, lines that have passed MAX_RD are also identified. This allows updating those lines PC signature in the RDP, by setting them to INF_RD. Sampled cache lines that relate to these expired RDP lines are also invalidated. After any invalidations are made to the sampled cache, the current request is written in.

Last step in the update process is setting a new ETR value. Any accesses that are to non-sampled cache lines, only have to run this part of the update process.

During the ETR update, the cache line's ETR counter and the related ETR clock are updated. This is done based on the RDP line that has a matching PC signature. If the matching PC signature has no valid prediction, the ETR counter is set to the INF_ETR value. The set’s ETR clock is incremented, and if it overflows, other ETR counters of the set are decremented.

Addresses that have no valid RDP or which RDP is a maximum value, have their ETR set to the maximum value. For the rest of lines, ETR is updated according to RDP value divided by 8. The division is required as ETR uses less precise values than RDP to save memory, and the 3-bit ETR clocks compensate for it.

Finally, when a line has to be evicted from the cache during a miss, the choice is based on the ETR of the cache lines. Each of the set’s ways’ ETRs are checked, and the one with the longest absolute value is chosen as victim.

6 METHODOLOGY

For the comparison, a set of attributes were selected as the criteria. Amounts of hits and misses were used to calculate hit-ratio, which was used to measure the accuracy of the algorithms. Hit-ratio was collected from all hits and misses from the start of the benchmark programs.

No training period was used at the start of the program, which means that the effect of a completely empty cache is also included in the hit-ratio. Alternative method would be to run each program enough to fill the cache to some degree, and only then start collecting hits and misses.

Simulation clock cycles of the benchmarks were used to measure overall performance differences. Additionally amounts of different types of evictions and state transitions of the algorithms were counted as a form of sanity check.

The comparisons were carried out in RTL simulation. Largely the existing Ballast RTL code was used, but some minor modifications were added to enable collection of performance data. These were done using simple SystemVerilog variables to not interfere with the actual functional RTL logic. Simulations were performed on a single core, as realistic multi-core use cases would ideally require an operating system.

Benchmarks used for the performance measurements were primarily from RISC-V International [64]. Additionally, two custom LFSR (*linear feedback shift register*) tests were used, mainly to troubleshoot the testbench prior to adding the rest of the benchmarks. All used benchmarks are listed in Table 2.

Table 2: Used benchmark programs.

Benchmark	Description	Data set size
median	Finds median value in data set	50000
mt-matmul	2D Matrix multiplication	10000
mt-vvadd	Floating point vector addition	10000
multiply	Multiply two arrays with a software multiplication operation	45000
qsort	Sorts array with quicksort	25000
rsort	Sorts array with radix sort	16384
vvadd	Integer vector addition	25000
lfsr64bit	Calculates pseudo random data with an LFSR operation, stores it in an array.	50000
lfsr64bit_rand	Same as above, but store indices are also randomized	50000

Different data set sizes were tested to find out points where different benchmarks would cause a significant number of evictions with each algorithm. This testing was carried out by simply trial-and-error. If a benchmark caused multiple thousands or more evictions, without taking tens of hours, it was considered usable for the actual benchmarking phase.

RISC-V International does have multiple other benchmarks available as well, but any that didn't cause evictions in the L2 were discarded from the final results. If a benchmark shows no evictions, the cache replacement algorithm has not been used for any eviction decisions, and therefore the benchmark provided no useful data for this thesis.

It's possible that even larger than tested data sets would have at some point caused more evictions in these discarded benchmarks, but it's challenging to estimate how long the execution times would have been and how much testing would have been necessary to reach that point.

The execution time of the final run of the benchmark set was approximately 225 hours. This number does not include the multiple simulation runs that were ran while verifying the algorithms or testing the different benchmarks.

These benchmarks were mainly selected to their known compatibility with RISC-V, and the fact that they run as baremetal, i.e., they do not require an operating system or other supervising or scheduling processes to manage them. While the CVA6 cores are Linux-compatible as mentioned earlier, running a full operating system in RTL simulation in addition to the actual benchmark process would add a significant overhead. While Linux or some other operating system would allow for more options in benchmarking, it was not realistic for this work.

Worth emphasizing is that results presented in this thesis are for an L2 cache, with a write-through L1 caches above it in the memory hierarchy inside the processor core. This means that any L1 read hits are not seen by the L2, its replacement algorithm, or the benchmark data collectors. However, every L1 write hit is seen due to the write-through policy. This is likely to have some amount influence on a few aspects. First, the replacement algorithms may perform quite differently on another system with a different cache architecture. Second, the hit-ratio presented in Results chapter is only for the L2 and does not account for any of the read hits that are encountered in the L1 caches.

Introducing multiple cores would also provide different results, as coherency control would force shared data to be occasionally evicted from L1. Naturally, the amount of memory accesses to the L2 cache would also be increased.

7 IMPLEMENTATION

As large part of this thesis was practical RTL implementation of different algorithms, a separate chapter is dedicated to different decisions and observations made on the topic.

One consideration in implementation was the extent of required changes. An algorithm which does not need changes for example to the processor pipeline or to the cache's structure is significantly easier to implement than ones which do need such changes. Structurally, each of the new algorithms added in this work followed the same principle as the existing LRU algorithm, where each algorithm has two separate sets of control logic: update logic and eviction logic.

Update logic is used whenever the algorithm's internal state needs to be updated. This means whenever a hit or a miss occurs in the L2 cache's own primary state machine. In practice the replacement algorithm snoops the signals of the L2 controller and activates when the correct combination of signals is present. The update logic then handles all necessary operations, and then goes back to waiting. The complexity of update logic is highly dependent on the specific algorithm.

Eviction logic, on the other hand, is directly built in as part of the primary L2 controller's state machine. The goal of this logic is to simply indicate to the controller which of the cache lines in the 8 ways of the cache can be evicted. As with the update logic, the amount of complexity required for this depends on the replacement algorithm.

Each cache replacement algorithm requires some registers or memories for bookkeeping. The full amounts have been calculated and are shown in Table 3.

Table 3: Required registers or memories.

Algorithm	Structure name	Size (bits)	Size (KiB)
LRU	**l2_mem_tag_lru_array**	3072	**3**
PLRUm	**plrum_mru_bits**	8192	**1**
EBR	ebr_r_counters	40 960	5
	ebr_e_counter	24 576	3
	ebr_mitt_counter	2048	0,25
	EBR Sum		**8,25**
Mockingjay	sampled_cache	38 400	4,6875
	etr	32 768	4
	etr_clock	3072	0,375
	current_timestamp	8192	1
	rdp	14 336	1,75
	Mockingjay Sum		**11,8125**

As can be seen from the table, different replacement polices have varying amount of needs to storage data. These requirements increase the area that would be needed to implement said algorithms in hardware. Furthermore, area would be also increased by the control logic of each algorithm. However, to obtain detailed information on that, the subsystem would need to be synthesized with each different algorithm, which was beyond the scope of this work.

The memory requirements also scale up if cache size or associativity is increased. Other implementation details are discussed in following subchapters.

7.1 PLRUm

On the simplest of the three algorithms, PLRUm, the only observation made was the ease of implementation. Only a handful of code lines and one set of registers were necessary. The algorithm was simple enough that even though no pseudocode was found, it could be easily understood and implemented based on a short explanation and a diagram (Figure 7).

As with most pseudo-LRU, the hardware requirements are modest. PLRUm requires only one bit per cache line, less than the traditional LRU.

7.2 EBR

EBR was also straightforward to implement. The logic consists of three sets of counters (R-, E- and miss counters) which are incremented during hits and misses, and of the effectiveness calculation which is done during eviction. In an effort to keep the design simple, the dynamic EBR-D variant was not implemented. However, this did mean that the weights used in effectiveness calculation (Equation 1) had to be selected.

The frequency weight *f* was set to be a constant 1, as the guideline was that that recency weight *r* should be significantly larger *f*. Initially, a recency weight of *r*=8 was used for running the benchmarks. Following this, an experimentation was carried out to see if other weights might perform better.

Running the full test set with many different weights would have been extremely time consuming, and therefore a simple experimentation was carried out with just one test case. Used benchmark was the lfsr64bit test, as it was the one to have largest differences in hit-ratio between the other algorithms.

Considered recency weights were mostly powers of 2, as in hardware they could be implemented with a simple left shift operation. Tested recency weights were 1, 2, 4, 8 and 16. The results of this experimentation are in Figure 9.

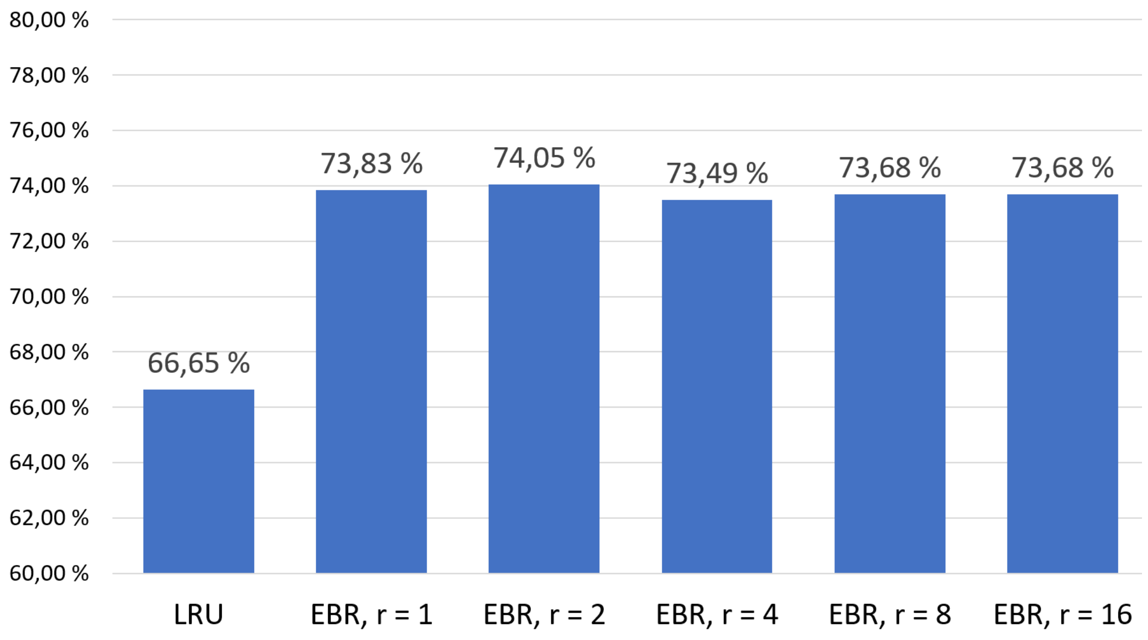

Figure 9: Hit-ratio for benchmark lfsr64bit with varying recency weights

As can be seen, *r* weight 2 had the best hit rate with lfsr64bit, though the variance between different weights is small. The full benchmark set was then also tested with this value. Also noteworthy is that with this one test, equal *f*, and *r* weights ("EBR, r=1") resulted in the second highest hit-ratio. Full test results with both EBR *r* weights and the other algorithms are in chapter 8.

In the EBR implementation some complexity is still added as the algorithm does require division logic to calculate the final effectiveness values. Additionally in the case that multiple ways have an equally low effectiveness value, the victim will have to be chosen randomly from the subset. This makes the actual victim selection of EBR more complicated than the other algorithms, as a pseudo-random number may have to be generated.

Number of bits used for each counter can be modified, but with this implementation, a 5-bit R-counter, 3-bit E-counter, and 2-bit miss counter were used. As seen from Table 3, this brings the overall required memory to 8,25 KiB.

Further testing could be done to optimize the counter widths, as each bit removed from the R- and E- counters would reduce the needed memory by 1 KiB.

7.3 Mockingjay

Mockingjay is a significantly more complicated algorithm. Despite this, its SystemVerilog implementation was also greatly aided by the C++ model [58] which had been used to simulate the algorithm in previous works. The model was close enough in detail to RTL level that many constructs were directly translated to SystemVerilog.

However, there were also some challenges present, due to the fundamental differences of hardware and software. The C++ model was mainly realized as one large function, which was to be called whenever a hit or a miss occurred. Additional smaller helper functions had been added for commonly used operations.

For the SystemVerilog version, this larger main function of the algorithm was broken down into an 8-state finite state machine. Like the update-function of the C++ model, this state machine activates whenever a hit or a miss occurs in the cache, otherwise waiting in an idle state.

One challenge brought on by the complexity of the algorithm state machine is that the state update process can take longer than it takes for the next memory request to arrive to the cache. This can lead to a situation where the actual L2 controller FSM has to wait in idle state so that the replacement algorithm can finish its update. With tested benchmarks the number of cycles “wasted” in this manner was minimal, less than 0,3% of runtime with all benchmarks. However, in a multi-core system with more frequent memory request this number would likely rise, and the effect could be more significant. Further analysis on the algorithm logic would be necessary identify which parts could be more effectively parallelised.

Certain details of the C++ model were not documented in detail, such as some of the cache parameters that were only defined as constant numbers. This led to additional work in identifying what values the parameters should have in the case of HPC L2. Different variables and structures had to be also carefully examined, to identify correct data widths, and whether they should be signed or not. The Mockingjay paper detailed many of the hardware requirements, which did aid in the implementation.

Mockingjay was the only one of the algorithms implemented in this thesis that required modifications to not only the cache logic, but also to the processor core itself. Namely it was necessary to add program counter value to each memory request. While the modification to the L1.5 interface itself was trivial, finding the correct way to route the PC value through the various pipeline stages and other modules of the processor was more challenging.

8 RESULTS

After running the benchmarks presented in chapter 6 the statistics were collected to analyse the results. Figure 10, Figure 12, and Figure 11 show hit-ratios for each testcase and algorithm. EBR8 refers to EBR with *r* weight of 8, and EBR2 to EBR with *r* weight of 2. Mockingjay is referred to as MJAY.

The results for hit-ratio show LRU generally performing the worst, expect for the median benchmark and the two sorting algorithms. PLRUm matches, and in most cases outperforms the standard LRU, except for the median and qsort cases where it's behind. Even in those cases the difference in hit-ratio by less than 0,05%

Both EBR versions generally perform better than the two LRU variants, except for the mentioned median, qsort, and rsort. Mockingjay has the highest hit-ratio in most tests except the two sorting algorithms, and in lfsr64bit.

Overall, little difference is seen between the two EBR implementations, staying within 1% of each other. Neither version seems to have a clear edge over the other.

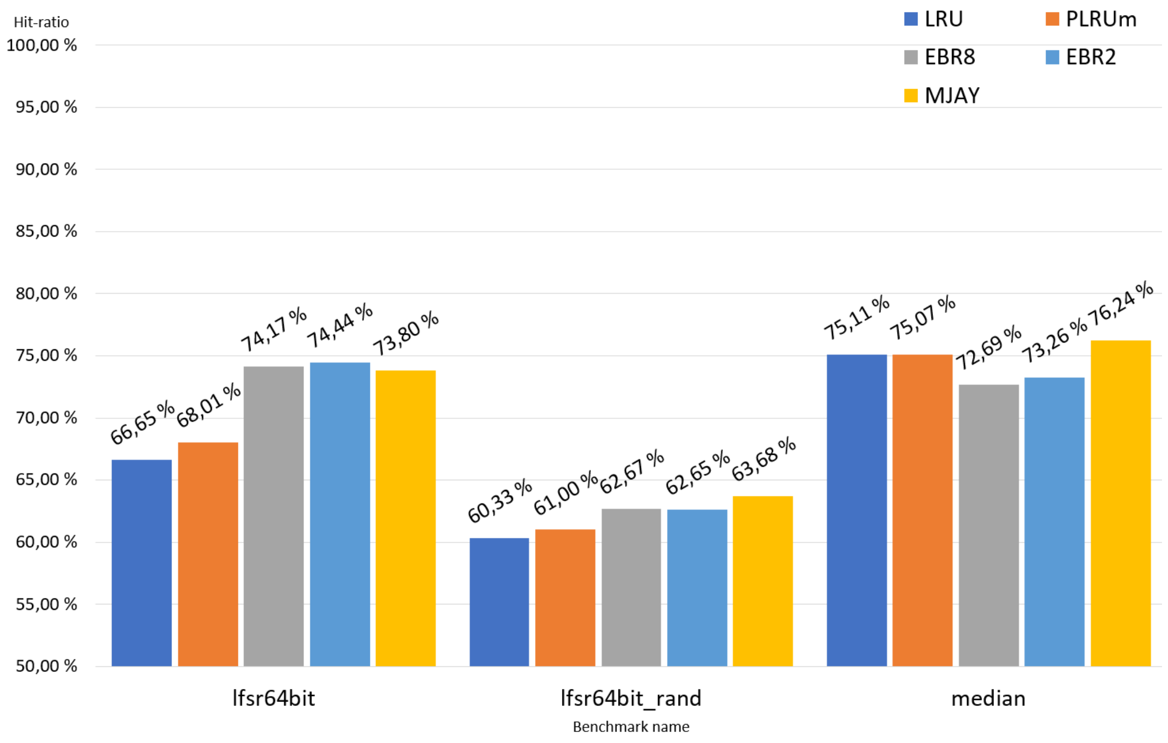

Figure 10: Hit-ratios with lfsr64bit, lfsr64bit_rand, and median

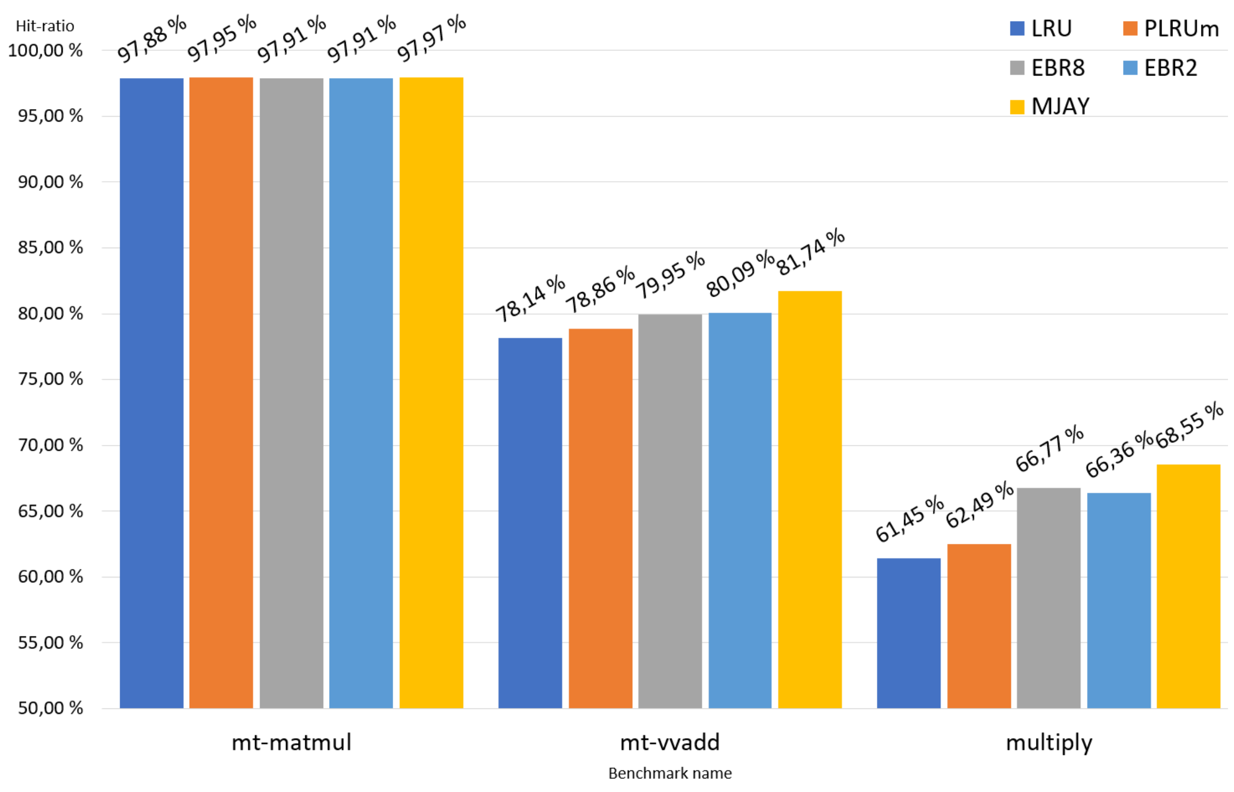

Figure 12: Hit-ratios with mt-matmul, mt-vvadd, and multiply

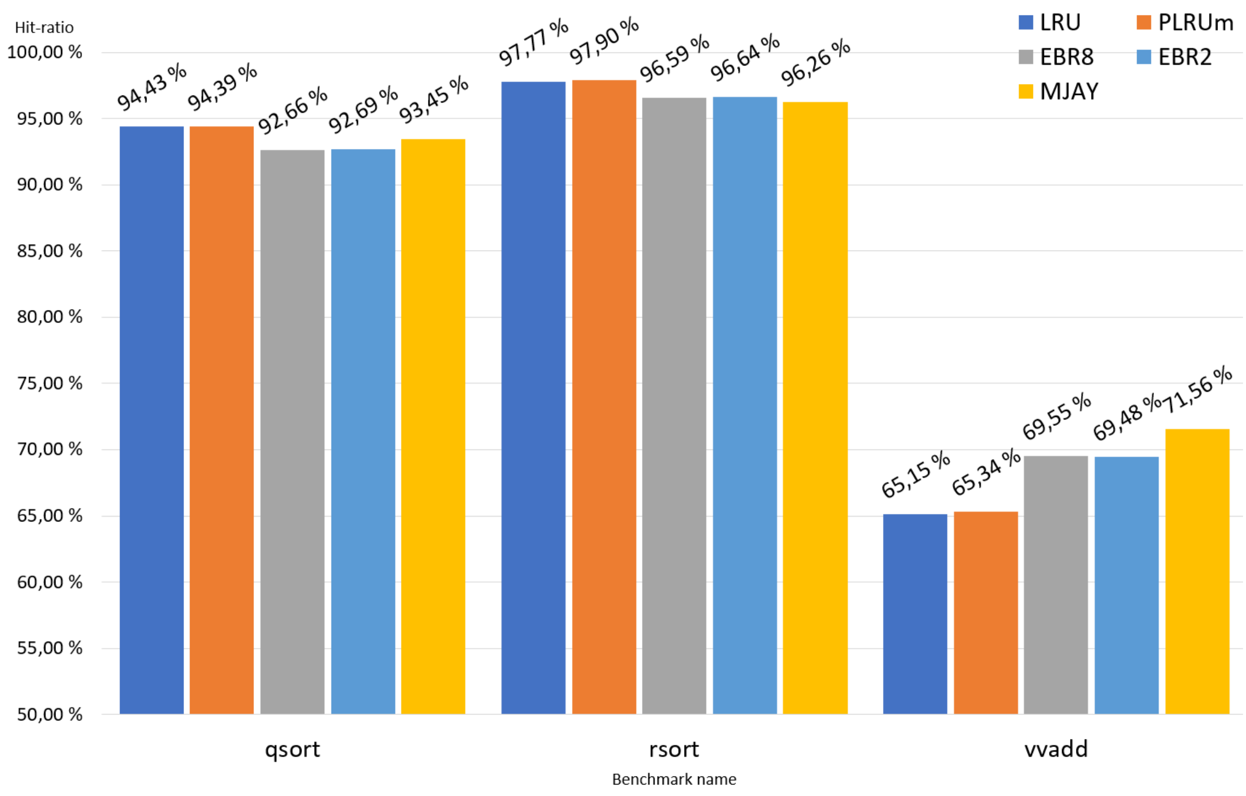

Figure 11: Hit-ratios with qsort, rsort, and vvadd.

Figure 13, Figure 15, and Figure 14 show the run time for each benchmark, with each algorithms' results normalized to LRU's performance. A result of 100% would be equal to LRU's performance for the given benchmark, less than 100% is a faster execution time, and more than 100% is slower. These comparisons are calculated from the raw amount of simulated clock cycles taken by each algorithm on the given benchmark.

What should be noted is that these results may not fully match the exact performance of a real system based on this RTL, as actual performance would be affected for example by the exact type of memory used, as well the exact clock rates used by the different subsystems involved.

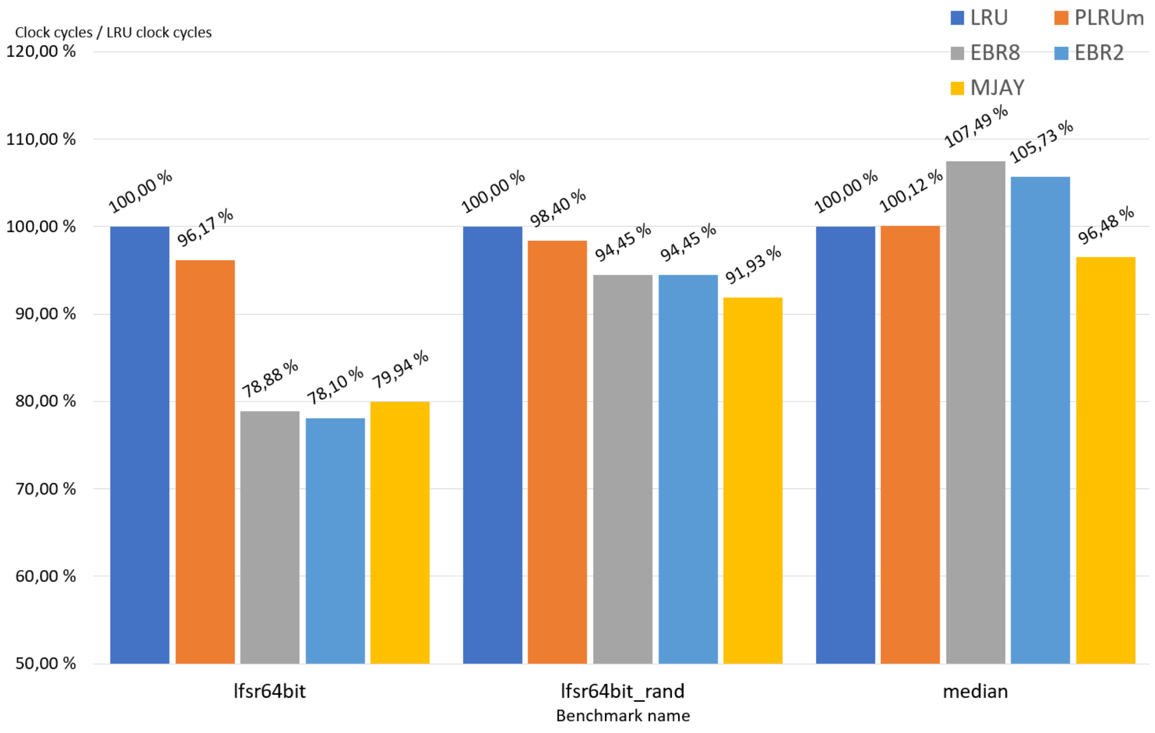

Figure 13: Clock cycles/LRU clock cycles, with lfsr64bit, lfsr64bit_rand, and median

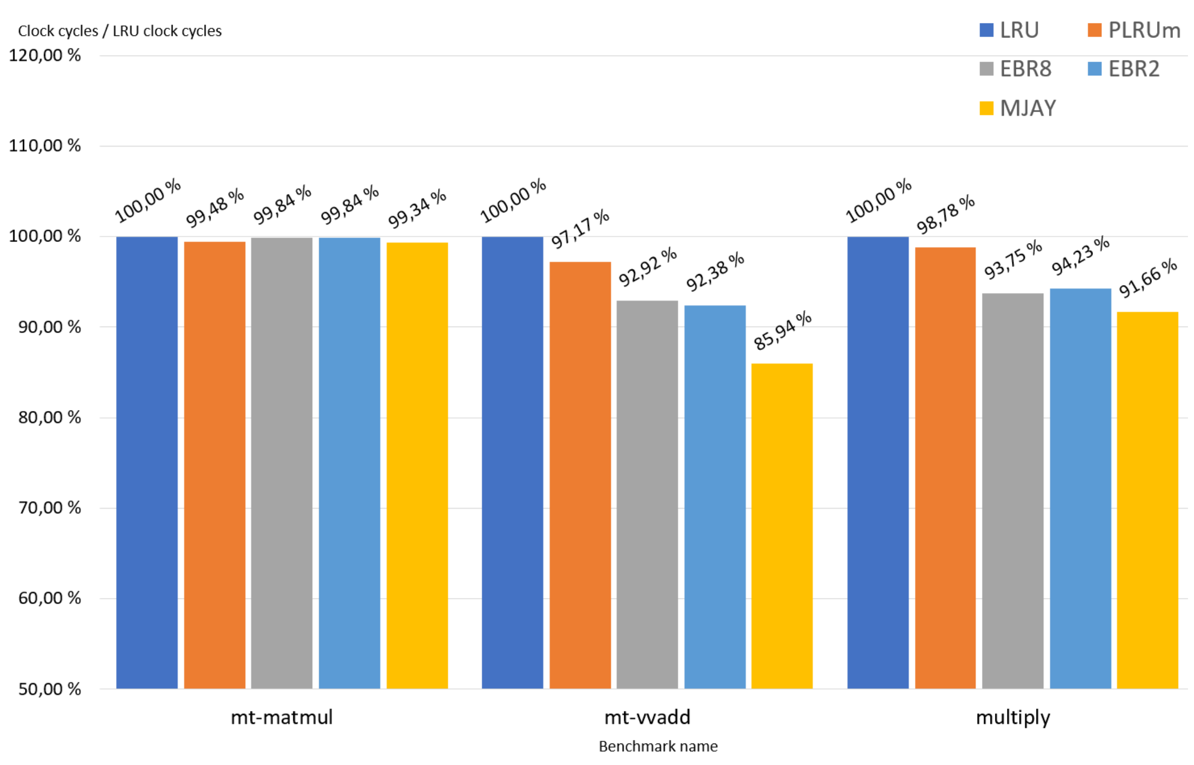

Figure 15: Clock cycles/LRU clock cycles, with mt-matmul, mt-vvadd, and multiply

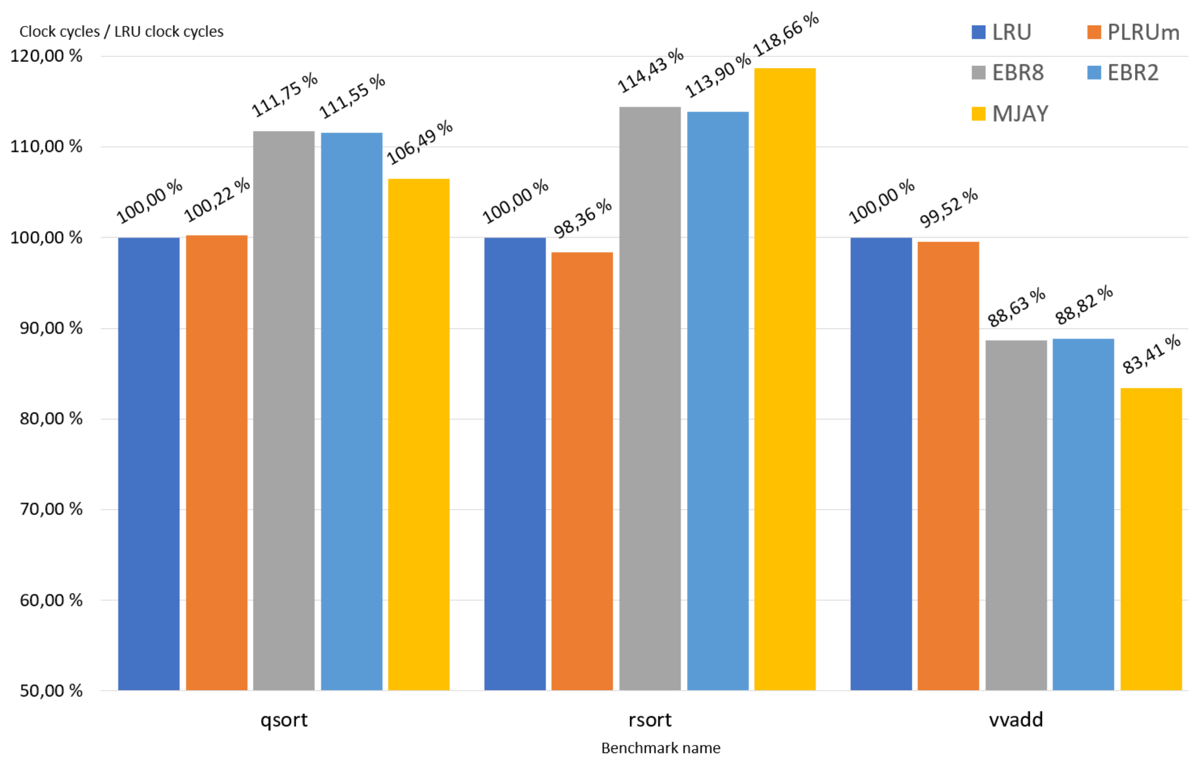

Figure 14: Clock cycles/LRU clock cycles, with qsort, rsort, and vvadd

Nevertheless, these results do demonstrate in a practical manner the benefit of higher hit-ratio, though the performance is not directly proportional. Clearest example of this are the two sorting algorithm benchmarks, where EBR and Mockingjay have only slightly worse hit-ratio than the LRU-based replacement algorithms, but the effect on execution length is much more significant. Rsort in particular is a long testcase with a lot of memory accesses, which emphasizes the runtime difference caused by worse hit ratio.

9 CONCLUSIONS

As the Results chapter shows, there is no single clear winner in this comparison. While the more advanced EBR and Mockingjay algorithms beat the LRU-implementations in most cases, there are some exceptions too. To explicitly state that which algorithm is better than the other would require narrowing down the subsystem use cases to a specific scenario. Based on the results presented here, it can only be said that EBR and Mockingjay generally perform better than LRU-based algorithms.

Similarly, it can be said that Mockingjay generally performs better than EBR, but the performance comes at a cost. The amount of work required for the implementation is more significant, and based on just the memory requirements, without considering the control logic, Mockingjay also requires more area on chip. As such, EBR may be a more practical choice.

PLRUm's performance results are also very valuable. This pseudo implementation of LRU not only matches traditional LRU accuracy but overtakes it in most cases, while still requiring less memories. This makes it a very appealing LRU-approximation.

Some limitations of this thesis work were discussed in relevant chapters. Overall, there are a few areas that would provide significant value if researched further.

Perhaps the most significant ones are physical requirements of different algorithms. Some information was obtained from RTL implementation, such as approximate memories required, which shed some light on the area required by different algorithm implementations. However, this does not consider the amount of required control logic for each algorithm, which may or may not be very area intensive.

Another consideration are timing requirements of different algorithm implementations. Pure RTL simulation does not necessarily match actual performance on physical hardware. While one algorithm might need less cycles to perform an operation, it's possible that another algorithm would outperform it due to having shorter critical paths and therefore being compatible with higher clock rates. It's also possible that the only bottleneck would be the existing cache control logic, in which case timing of the replacement policies could prove irrelevant.

Another interesting topic would be more advanced benchmarking on FPGA or ASIC. RTL simulation is inherently slow and limited by the simulation platform's performance as well as the available memory. For example, benchmarking more complex programs running on a Linux or another operating system would provide valuable information on very practical use cases. This could also be extended to multi-core cache testing.

It is worth noting that this work is not, nor does is it aim to be, an exhaustive look at all possible cache replacement algorithms ever devised. Rather it’s a brief look at some of the available designs, practicalities that can be considered in their implementation, and the performance they offer on one simulated platform.

There are bound to be other very well performing algorithms that are not mentioned in thesis. Despite a certain kind of stagnation that can be seen in actual hardware implemented cache replacement algorithms, the topic is worth exploring in detail.

REFERENCES

[1] G. Blanchet and B. Dupouy, *Computer Architecture*. in Computer engineering series. London: ISTE, 2013. doi: 10.1002/9781118577431.

[2] I. V. McLoughlin, *Computer systems: An Embedded Approach*. New York, N.Y: McGraw-Hill Education, 2018.

[3] J. Nurmi, *Processor design: System-on-chip computing for ASICs and FPGAs*. Dordrecht: Springer Netherlands, 2007. doi: 10.1007/978-1-4020-5530-0.

[4] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966, doi: 10.1147/sj.52.0078.

[5] H. Ghasemzadeh, S. Mazrouee, and M. R. Kakoee, “Modified pseudo LRU replacement algorithm,” in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS’06)*, IEEE, 2006, p. 6 pp. – 376. doi: 10.1109/ECBS.2006.52.

[6] A. Shimizu, D. Townley, M. Joshi, and D. Ponomarev, “EA-PLRU: Enclave-Aware Cache Replacement,” in *ACM International Conference Proceeding Series*, in HASP ’19. NEW YORK: ACM, 2019, pp. 1–8. doi: 10.1145/3337167.3337172.

[7] “SoC Hub.” www.sochub.fi (accessed Aug. 02, 2022).

[8] A. Aatinen and S. Lammi, “The new Finnish SoC Hub strengthens global system-on-chip design expertise,” Oct. 06, 2020. https://www.tuni.fi/en/news/new-finnish-soc-hub-strengthens-global-system-chip-design-expertise (accessed Aug. 02, 2022).

[9] A. Rautakoura, T. Hämäläinen, A. Kulmala, T. Lehtinen, M. Duman, and M. Ibrahim, “Ballast: Implementation of a Large MP-SoC on 22nm ASIC Technology,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*, Aug. 2022, pp. 276–283. doi: 10.1109/DSD57027.2022.00045.

[10] “CORE-V Cores.” https://github.com/openhwgroup/core-v-cores (accessed Aug. 02, 2022).

[11] “CVA6.” https://github.com/openhwgroup/cva6 (accessed Aug. 02, 2022).

[12] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019, doi: 10.1109/TVLSI.2019.2926114.

[13] F. Zaruba, “Energy-efficient high-performance computing,” ETH Zurich, 2021.

[14] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213.” RISC-V International, 2019. [Online]. Available: https://riscv.org/technical/specifications/

[15] “OpenHW Group CVA6 User Manual.” https://docs.openhwgroup.org/projects/cva6-user-manual/ (accessed Sep. 02, 2022).

[16] “OpenSPARC T1 Microarchitecture Specification.” 2008. [Online]. Available: https://www.oracle.com/technetwork/systems/opensparc/t1-01-opensparct1-micro-arch-1538959.html

[17] J. M. Balkind, “Open Source Platforms for Enabling Full-Stack Hardware-Software Research,” dissertation, ProQuest Dissertations Publishing, 2022.

[18] D. Giri, P. Mantovani, and L. P. Carloni, “NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators,” *2018 12th IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2018*, 2018, doi: 10.1109/NOCS.2018.8512153.

[19] P. Mantovani *et al.*, “Agile SoC development with open ESP,” in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2020. doi: 10.1145/3400302.3415753.

[20] “esp-caches.” https://github.com/sld-columbia/esp-caches (accessed Aug. 26, 2022).

[21] J. Balkind *et al.*, “OpenPiton at 5: A Nexus for Open and Agile Hardware Design,” *IEEE MICRO*, vol. 40, no. 4, pp. 22–31, 2020, doi: 10.1109/MM.2020.2997706.

[22] J. Balkind *et al.*, “OpenPiton: An Open Source Manycore Research Framework,” *Operating systems review*, vol. 50, no. 2, pp. 217–232, 2016, doi: 10.1145/2954680.2872414.

[23] “OpenPiton.” https://github.com/PrincetonUniversity/openpiton (accessed Aug. 05, 2022).

[24] A. Amid *et al.*, “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” *IEEE MICRO*, vol. 40, no. 4, pp. 10–21, 2020, doi: 10.1109/MM.2020.2996616.

[25] “Chipyard 1.7.1 documentation.” https://chipyard.readthedocs.io/ (accessed Sep. 01, 2022).

[26] “sequencer/block-inclusivecache-sifive at ic_doc.” https://github.com/sequencer/block-inclusivecache-sifive/tree/ic_doc (accessed Sep. 01, 2022).

[27] “sifive/block-inclusivecache-sifive.” https://github.com/sifive/block-inclusivecache-sifive (accessed Aug. 31, 2022).

[28] N. Wistoff, M. Schneider, F. K. Gurkaynak, L. Benini, and G. Heiser, “Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core,” *Proceedings -Design, Automation and Test in Europe, DATE*, vol. 2021-Febru, pp. 627–632, 2021, doi: 10.23919/DATE51398.2021.9474214.

[29] W. Rönninger, “Memory subsystem for the first fully open-source RISC-V heterogeneous SoC,” Master’s thesis, ETH Zurich, 2019.

[30] D. Mallasén, R. Murillo, A. A. Del Barrio, G. Botella, L. Piñuel, and M. Prieto, “PERCIVAL: Open-Source Posit RISC-V Core with Quire Capability,” pp. 1–11, 2021.

[31] N. Ait Said, M. Benabdenbi, and K. Morin-Allory, “Self-adaptive run-time variable floating-point precision for iterative algorithms: A joint HW/SW approach,” *Electronics (Switzerland)*, vol. 10, no. 18, p. 2209, 2021, doi: 10.3390/electronics10182209.

[32] D. Šišejković, F. Merchant, L. M. Reimann, R. Leupers, and S. Kegreiß, “Scaling Logic Locking Schemes to Multi-module Hardware Designs,” in *Architecture of Computing Systems – ARCS 2020*, in Lecture Notes in Computer Science, vol. 12155. Cham: Springer International Publishing, 2020, pp. 138–152. doi: 10.1007/978-3-030-52794-5_11.

[33] “MIG-V – HENSOLDT Cyber.” https://hensoldt-cyber.com/mig-v/ (accessed Oct. 12, 2022).

[34] “MiG-V 1.0 Datasheet.” 2022.

[35] “RISC-V Exchange: Cores and SoCs.” https://riscv.org/exchanges/cores-socs/ (accessed Apr. 13, 2022).

[36] D. Petrisko *et al.*, “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs,” *IEEE MICRO*, vol. 40, no. 4, pp. 93–102, 2020, doi: 10.1109/MM.2020.2996145.

[37] “BlackParrot: A Linux-Capable Accelerator Host RISC-V Multicore.” https://github.com/black-parrot/black-parrot (accessed Sep. 12, 2022).

[38] “BaseJump STL Cache Library.” https://docs.google.com/document/d/1AIjhuwT-bOYwyZHdu-Uc4dr9Fwxi6ZKscKSGTiUeQEYo/edit (accessed Sep. 07, 2022).

[39] “BaseJump Standard Template Library (STL) Repository.” https://github.com/bespoke-silicon-group/basejump_stl (accessed Sep. 12, 2022).

[40] “RiscyOO: RISC-V Out-of-Order Processor.” https://github.com/csail-csg/riscy-OOO (accessed Sep. 08, 2022).

[41] “coherence.” https://github.com/csail-csg/coherence (accessed Sep. 08, 2022).

[42] “GRLIB IP Library.” https://www.gaisler.com/index.php/products/ipcores/soclibrary/soclibrary-overview (accessed Oct. 12, 2022).

[43] “GRLIB IP Core User’s Manual.” Aug. 2022. Accessed: Oct. 12, 2022. [Online]. Available: https://www.gaisler.com/index.php/downloads/leongrlib

[44] M. Torres and J. Klapp, “Lagarto I RISC-V Multi-core: Research Challenges to Build and Integrate a Network-on-Chip,” in *Supercomputing*, in Communications in Computer and Information Science, vol. 1151. Switzerland: Springer International Publishing AG, 2019, pp. 237–248. doi: 10.1007/978-3-030-38043-4_20.

[45] “Shakti Processors.” http://shakti.org.in/processors.html (accessed Jul. 10, 2023).

[46] “C-Class Core Generator — C-Class Core Generator beta-2.0.0 documentation.” https://c-class.readthedocs.io/en/latest/index.html (accessed Jul. 10, 2023).

[47] H. Ghasemzadeh and S. O. Fatemi, “Pseudo-FIFO Architecture of LRU Replacement Algorithm,” in *2005 Pakistan Section Multitopic Conference*, IEEE, 2005, pp. 1–7. doi: 10.1109/INMIC.2005.334496.

[48] W. A. Wong and J.-L. Baer, “Modified LRU policies for improving second-level cache behavior,” in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, IEEE, 2000, pp. 49–60. doi: 10.1109/HPCA.2000.824338.

[49] Manish Kumar Verma and P K Singh, “Enhanced Approach for Cache Behaviour and Performance Evaluation,” *International journal of advanced research in computer science*, vol. 8, no. 3, 2017.

[50] D. Gille, “Study of Different Cache Line Replacement Algorithms in Embedded Systems,” Master’s thesis, KTH Royal Institute of Technology, 2007.

[51] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, “Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite,” in *ACM Southeast Regional Conference: Proceedings of the 42nd annual Southeast regional conference; 02-03 Apr. 2004*, in ACM-SE 42. ACM, 2004, pp. 267–272. doi: 10.1145/986537.986601.

[52] G. Tian and M. Liebelt, “An effectiveness-based adaptive cache replacement policy,” *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 98–111, 2014, doi: 10.1016/j.micpro.2013.11.011.

[53] D. Lee *et al.*, “On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies,” in *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Atlanta Georgia USA: ACM, May 1999, pp. 134–143. doi: 10.1145/301453.301487.

[54] J. Jeong and M. Dubois, “Cost-sensitive cache replacement algorithms,” in *International Symposium on High-Performance Computer Architecture*, LOS ALAMITOS: IEEE, 2003, pp. 327–337. doi: 10.1109/HPCA.2003.1183550.

[55] F. Mounes-Toussi and D. J. Lilja, “The effect of using state-based priority information in a shared-memory multiprocessor cache replacement policy,” in *Proceedings. 1998 International Conference on Parallel Processing (Cat. No.98EX205)*, Aug. 1998, pp. 217–224. doi: 10.1109/ICPP.1998.708489.

[56] A. Jain and C. Lin, “Back to the Future: Leveraging Belady’s Algorithm for Improved Cache Replacement,” in *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, 2016, pp. 78–89. doi: 10.1109/ISCA.2016.17.

[57] “The 2nd Cache Replacement Championship.” https://crc2.ece.tamu.edu/ (accessed Aug. 06, 2022).

[58] I. Shah, A. Jain, and C. Lin, “Mockingjay.” Apr. 06, 2023. Accessed: Jun. 21, 2023. [Online]. Available: https://github.com/ishanashah/Mockingjay

[59] I. Shah, A. Jain, and C. Lin, “Effective Mimicry of Belady’s MIN Policy,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Piscataway: IEEE, 2022, pp. 558–572. doi: 10.1109/HPCA53966.2022.00048.

[60] Sizhuo Zhang, A. Wright, and T. Bourgeat, “Composable Building Blocks to Open Up Processor Design,” *IEEE MICRO*, vol. 39, no. 3, pp. 47–55, 2019, doi: 10.1109/MM.2019.2910012.

[61] “shakti,” *GitLab*, Apr. 19, 2022. https://gitlab.com/shaktiproject (accessed Jul. 10, 2023).

[62] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, “SHAKTI Processors: An Open-Source Hardware Initiative,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, Jan. 2016, pp. 7–8. doi: 10.1109/VLSID.2016.130.

[63] R. S. Sutton, “Learning to Predict by the Methods of Temporal Differences,” *Machine learning*, vol. 3, no. 1, pp. 9-, 1988, doi: 10.1023/A:1022633531479.

[64] “riscv-tests.” RISC-V Software, Jun. 18, 2023. Accessed: Mar. 24, 2023. [Online]. Available: https://github.com/riscv-software-src/riscv-tests