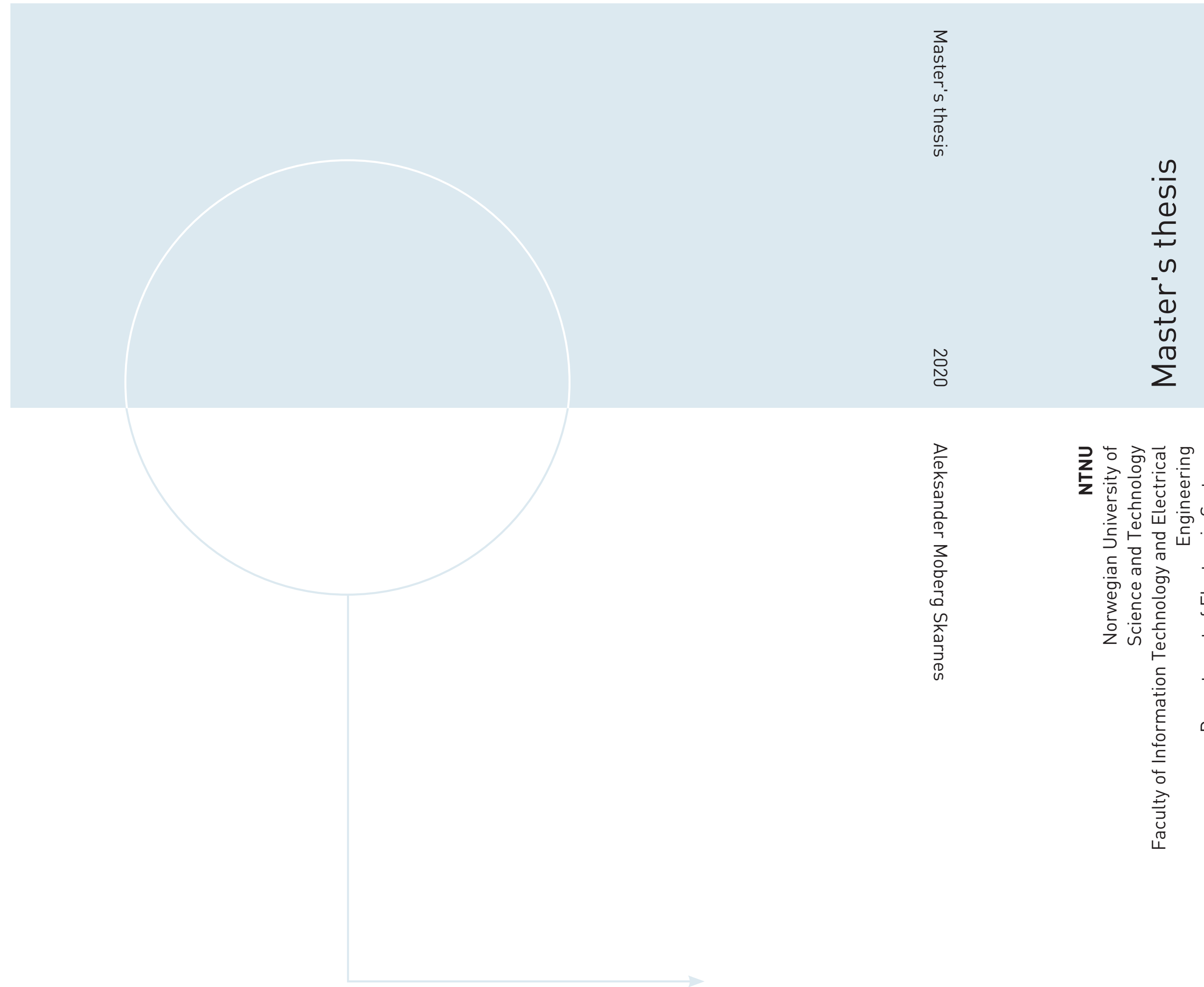


Aleksander Moberg Skarnes

# MPCache: A Novel High-Level Cache Simulation Framework

## for Design Exploration of Embedded Cache Systems

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

# MPCache: A Novel High-Level Cache Simulation Framework

for Design Exploration of Embedded Cache Systems

**Aleksander Moberg Skarnes**

Electronic Systems Design
Submission date: June 2020
Supervisor: Per Gunnar Kjeldsberg, Department of Electronic Systems
Co-supervisor: Eivind Fylkesnes, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronic Systems

# Abstract


Cache memories have been extensively used in computer designs to hide the growing divergence between the speed of the main memory and the CPU. The design space for caches is almost infinitely large, which cuts both ways in a design phase. It presents many candidate cache structures but can make it hard to decide which one to choose for a particular computer system. This decision introduces the need for high-level models that can be used early in the exploration of different designs. Most cache models today are incorporated in sophisticated full-system simulators and are targeted at larger computer systems.

This thesis presents MPCache, a high-level cache simulation framework made for design exploration of cache memories of embedded computer systems. It combines a functional and a physical cache model explicitly made for this purpose of producing results about total execution time and energy usage of applications using a specific cache memory. The functional model has been verified to produce correct results under changes to all design options. The combined results have been compared to RTL simulations of a specific cache structure, which show that the results are reasonable for a selection of benchmarks, but that the features of the physical model should be extended to support more cache configurations and that more RTL simulations of a variety of cache structures are needed to make more judgements about the framework.

The usage of the physical model has been compared to the established physical cache model CACTI. The two models have been compared by analyzing both the plain results and the results used in a combined simulation. The results indicate that MPCache produces poor estimates of caches with a sequential access mode and with increasing cache sizes. However, comparing the CACTI results with the RTL simulations revealed that MPCache gave better estimates for this individual case. This signifies that MPCache can produce reasonable estimates for some structures. Though, because of the issues related to the results of some cache structures, it is not mature for usage in design exploration.

# Sammendrag

Cache-minne har blitt mye brukt i datamaskiner for å skjule det økende avviket mellom hastigheten til hovedminnet og prosessoren. Designmulighetene for cacheminner er nesten uendelige, noe som kan være et tveegget sverd i den designprosess. Det fører til at en gitt datamaskinarkitektur kan ha mange ulike løsninger for sitt minnesystem, men det kan være vanskelig å få frem hvilken som er den beste av dem. Dette introduserer behovet for høynivåmodeller som kan brukes tidlig i utforskningen av ulike design. I dag er de fleste cache-modeller i dag er integrert i sofistikerte simulatorer som modellerer hele datasystemer og som er rettet mot større arkitekturstørrelser.

Denne oppgaven presenterer MPCache, et høynivå cache-simuleringsrammeverk laget for utforskning av cache-minnedesign for innvevde datasystemer. Rammeverket kombinerer en funksjonell og en fysisk cache-modell laget for akkurat dette formålet, og legger til rette for estimering av total kjøretid og energiforbrukbruk av applikasjoner ved bruk av et spesifikt cachedesign. Det er verifisert at den funksjonelle modellen gir riktige resultater for ulike valg av de tillatte designalternativene. De kombinerte resultatene er blitt sammenlignet med RTL-simuleringer, som viser at resultatene er rimelige for et utvalg av ulike applikasjoner for dette cachedesignet.

Den fysiske modellen er sammenlignet med den etablerte fysiske cache-modellen CACTI. De to modellene er blitt sammenlignet ved å analysere både de resultatene hver modell produserer, men også ved bruk i en kombinert simulering. Resultatene indikerer at MPCache gir dårlige estimater av cacher med sekvensielt aksessmodus, og at feilene her øker med størrelsen må cacheminnet. For parallelt aksessmodus gir modellen bedre estimater, men også her øker feilen med størrelsen på cacheminnet. Sammenligning av CACTI-resultatene med RTL-simuleringene viste imidlertid at MPCache ga bedre estimater for dette individuelle tilfellet. Dette betyr at MPCache kan gi rimelige estimater for noen strukturer, men på grunn av problemene knyttet til økende størrelse på cacheminnet er rammeverket ikke modent for bruk i designutforskning.

# Preface

The task of constructing a high-level cache model started in the fall of 2019 with my specialization project and carried over into the spring of 2020 with this master thesis. Eivind Fylkesnes at Nordic Semiconductor proposed the project as a general topic and gave me next to no boundaries for where to take it. It has been a long and bumpy road from that starting point with much experimenting and eventually settling on a topic for this master thesis. I decided to target this project at cache modeling for embedded computer systems to utilize the expertise of Nordic Semiconductor, but also because I saw a gap in physical cache models coverage of this subdomain of computer architectures.

The framework presented in this thesis is named Marco Polo Cache. I abbreviated this name to MPCache to create a double meaning of the letters MP, where the other meaning is ”microprocessor.” This double meaning was made to emphasize the combined focus of the framework - exploration of embedded cache system designs.

I want to thank Eivind Fylkesnes for guiding me through this project and for always being there when I was stuck and needed a helping hand. He has helped me to understand the workflow of Nordic Semiconductor and how to use the tools that were provided by them for me. He has also helped me to understand how a real cache memory can be implemented and has influenced the design of the models through some of his ideas. During the verification of the physical model, he played a significant role in choosing appropriate benchmark programs and compiling these for the RTL system model.

I would also like to thank my supervisor, Per Gunnar Kjeldsberg, who has helped me a great deal in structuring and writing this thesis by providing constructive feedback. He has also helped me through meaningful discussions and has pushed me to work at a steady pace throughout the project period.

I would also like to express my gratitude towards Nordic Semiconductor for allowing me to use their licensed tools for my work, and for making datasheets and one of their RTL cache models available for me to use during the verification of my models. This has made it possible to compare the results of the model to a real-life example of a cache implementation.

Lastly, I would like to pay my respects to my former supervisor at NTNU, Kjetil Arnt Svarstad, who sadly passed away during the work of my specialization project. I will remember you as a brilliant and funny man who had the outstanding ability to make a full auditorium of students

break into laughter during a lecture on hardware verification.

# Table of Contents

**Abstract** **i**

**Sammendrag** **iii**

**Preface** **v**

**Table of Contents** **ix**

**List of Tables** **xi**

**List of Figures** **xiv**

**List of Abbreviations** **xv**

**1 Introduction** **1**
1.1 Outline . . . 2
1.2 Main Contributions . . . 2

**2 Theory** **5**
2.1 Concepts Related to Cache Memory . . . 5
2.2 Concepts Related to Modeling and Simulation . . . 7
2.2.1 Physical Cache Modeling . . . 7
2.2.2 Functional Cache Modeling . . . 8
2.2.3 Model Assessment . . . 8
2.3 Energy and Power . . . 9

**3 Related Work** **11**
3.1 Functional Models . . . 11
3.2 Physical Models . . . 12

**4 MPCache Framework** **15**
4.1 Components of the Framework . . . 15
4.1.1 The Functional Model . . . 15

4.1.2 The Physical Model . . . 17
4.1.3 The Combined Simulation . . . 18
4.2 Overview of Framework . . . 19

**5 Functional Cache Model** **21**
5.1 Implementation . . . 21
5.1.1 Configuration of the Model . . . 22
5.1.2 Memory Access Handling . . . 23
5.1.3 Memory Hierarchy Simulation . . . 26
5.2 Verification . . . 27
5.2.1 Methodology . . . 27
5.2.2 Results and Discussion . . . 30

**6 Physical Cache Model** **33**
6.1 Implementation . . . 33
6.1.1 Configuration of the Model . . . 34
6.1.2 Estimation of Energy Usage . . . 34
6.1.3 Estimation of Leakage Power . . . 40
6.1.4 Estimation of Time Usage . . . 41
6.2 Verification . . . 41
6.2.1 Methodology . . . 41
6.2.2 Results and Discussion . . . 44

**7 Performance Compared to CACTI** **49**
7.1 Comparison of the Models . . . 49
7.2 Comparison of the Performance . . . 50
7.2.1 Methodology . . . 50
7.2.2 Results and Discussion . . . 52
7.3 Reflections on the Comparison . . . 58

**8 Conclusion** **61**

**9 Future Work** **65**

**Bibliography** **67**

**A The MPCache Framework** **71**
A.1 The Functional Models . . . 71
A.2 The Physical Models . . . 71
A.3 Simulation File . . . 72

**B Functional Verification** **73**

B.1 Cache Configurations . . . . . . . . . . 73
B.2 Memory Access Traces . . . . . . . . . . 73

# List of Tables


5.1 Valid values for all the cache configuration parameters and options. . . . . . . . 23

5.2 The default test value for all cache configuration parameters. . . . . . . . . . . 29

5.3 The verification test plan with values for the cache configuration parameters and a corresponding test case number. . . . . . . . . . . . . . . . . . . . . . . . . 30

5.4 The expected results of the different verification tests of the functional model and a remark on whether it matched the actual results or not. . . . . . . . . . . 31

6.1 Configuration parameters used only by the physical model and their valid values. 34

6.2 The custom benchmark suite used in the validation of the physical model. . . . 42

6.3 A list of every tool used in the process of gathering the power estimation results of the RTL model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

7.1 The selection of cache structures used to compare the results of the models. . . 51

7.2 The usage of the CACTI estimates in the MPCache combined simulations. . . . 52

# List of Figures


4.1 Overview of the MPCache framework. . . . . . . . . . . . . . . . . . . . . . . . 20

5.1 Flowchart of the memory access handling of the functional cache model. . . . . 24
5.2 A visual representation of the memory hierarchy described in Listing 5.6. . . . 27

6.1 Organization of SRAM-based cache for direct-mapped or set-associative cache architectures [1]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.2 Access patterns for a conventional n-way set-associative cache (a), a phased n-way set-associative cache (b), and way-prediction n-way set-associative cache (c) [2]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.3 The number of accesses and type of accesses to the RAM block in a parallel access mode cache for all modelled scenarios. . . . . . . . . . . . . . . . . . . 37
6.4 The number of accesses and type of accesses to RAM block in a sequential access mode cache for all modelled scenarios. . . . . . . . . . . . . . . . . . . 38
6.5 The percentage of deviation of the execution time estimated for each benchmark by MPCache compared to the corresponding estimate by the RTL simulation. . 44
6.6 The estimated total execution time of each benchmark by both the RTL simulations and the MPCache framework relative to the benchmark with the largest estimated total execution time. . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.7 The percentage of deviation of the energy usage estimated for each benchmark by MPCache compared to the corresponding estimate by the RTL simulation. . 46
6.8 The estimated total energy usage of each benchmark for both the RTL simulations and MPCache relative to the one with the largest estimated total energy usage. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.9 The estimated total energy usage by MPCache relative to the estimated energy usage of the memory blocks by the RTL simulations. . . . . . . . . . . . . . . 47

7.1 The estimates for dynamic energy usage by the selection of cache structures by using MPCache. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
7.2 The estimates for dynamic energy usage by the selection of cache structures by using CACTI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

7.3 The estimates for leakage power of the selection of cache structures by using MPCache. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
7.4 The estimates for leakage power of the selection of cache structures by using CACTI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
7.5 The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using MPCache. . . . . . . . . . . . . . . . . . 56
7.6 The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using CACTI. . . . . . . . . . . . . . . . . . . . 56
7.7 The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations. 57
7.8 The estimates for leakage power by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations. . . . . 58

# List of Abbreviations

**CPU** Central Processing Unit. 42, 43

**CRC** Cyclic Redundancy Check. 42

**EEMBC** Embedded Microprocessor Benchmark Consortium. 42

**FFT** Fast Fourier Transform. 42

**FIFO** First In, First Out. 23, 30

**GCC** GNU C Compiler. 43

**GPIO** General Purpose Input/Output. 43

**HDL** Hardware Description Language. 42

**iFFT** Inverse Fast Fourier Transform. 42

**ISA** Instruction Set Architecture. 11

**LRU** Least Recently Used. 23, 29, 30, 42

**MCU** Microcontroller. 42

**NoC** Networks on Chip. 12

**RAM** Random Access Memory. xiii, 34, 36–38, 40

**RTL** Register Transfer Level. xi, 2, 41–43

**SoC** System on a Chip. 42

**TLM** Transaction Level Modeling. 41

# Chapter 1

# Introduction

The memory wall describes the growing disparity between processor speed and memory speed. William Wulf and Sally McKee first introduced it in 1994 [3] when they predicted that the system performance would be entirely dependent on the memory speed in the future without a breakthrough in memory technology or if computer scientists did not come up with something to prevent it. Luckily, architectural developments have been able to keep these problems at bay, but they remain highly relevant to this day. Among many solutions to avoid memory speed to dominate the system performance are the increasing use of multiple levels of caches to form elaborate memory hierarchies and the development of sophisticated refill and prefetch schemes used in these caches [4]. To get indications on how different structures of cache memory impacts the system performance before implementation, we need models that can accurately imitate the properties of the cache memory.

Cache models can be separated into two categories: (1) physical models and (2) functional models. A physical model will try to model the physical properties of a cache memory. This means that it will estimate the time and power consumption of accesses made to that particular cache, and the results of the modeling are values to these parameters based on the input of the user. These properties are tightly coupled with the technology node used and the internal structure of the cache. A functional model, on the other hand, is mainly used to run program simulations with a specific memory hierarchy containing the cache structures that are studied. The results will usually contain information about how many accesses are made to the different components of the memory hierarchy and the number of hits or misses for the cache memories. These results are universal for a particular memory hierarchy structure across, for example, technology nodes, but can be combined with specific physical properties to estimate how much energy or time the program execution would cost. Both type of models can be either high or low-level based on their ease of use and how deep they go into detail about the cache memories.

Most high-level cache models are incorporated in full-system simulators, which require extensive configuration to model a particular system and consume a great deal of time in simulations because of the complex models. Besides, these simulators are targeted towards larger computer systems, which might need different timing and power models than smaller systems like embedded computer systems.

The objective of this study is to develop a novel high-level cache modeling framework called *MPCache* (**M**arco **P**olo **Cache**) for simulations of cache memories systems of embedded computers. This will be done by establishing a functional model to represent the functionality of relevant cache structures for this purpose, as well as a physical model to emulate the physical properties of an embedded cache structure. The model will be compared to a lower level RTL model to evaluate the accuracy of the timing and energy consumption estimates. The usage of the framework will also compared to the established physical cache model CACTI using software benchmarks made to represent applications that are likely to run on an embedded computer and a selection of relevant cache structures. To constrain the possibility space of system configurations, it will only consider interfacing between the cache structure and processors commonly used in embedded computer systems. Multicore architectures will also not be considered to simplify the assumptions regarding the models and removing the need for coherency protocols.

## 1.1 Outline

The rest of the report is organized as follows. Chapter 2 presents concepts related to cache memories and their power and energy usage needed to easier understand the majority of the report. Related work in modeling of caches is presented in Chapter 3. An overview of the MPCache modeling framework is given in Chapter 4. The implementation of a high-level functional cache model and the verification of it is described in Chapter 5. Chapter 6 presents the implementation of a high-level physical cache model and the work of attempting to verify it. The work of comparing the performance of the aforementioned models to other cache models is presented in Chapter 7. Chapter 8 presents the conclusion of the work presented in this report, before the potential goals of future work is listed in Chapter 9.

## 1.2 Main Contributions

- Constructed a cache simulation framework that facilitate combined functional and physical cache simulations.
- Implemented a high-level trace driven functional cache memory model.
- Verified the functional model based on a set of traces and configurations highlighting the

different configuration parameters and showed them to be equal to expected results.

- Implemented a high-level combination of an analytical and datasheet physical cache memory model.
- Compared the results of the physical model to power and timing estimations of a RTL model of a 8KB 2-way associative cache and confirmed correspondence in the results even though the results were not accurate.
- Comparison of the framework results with usage of the established physical cache model CACTI and explained the benefits of using MPCache.
- Revealed potential weaknesses in the physical modeling technique used and the impact it has on results.
- Suggestions on the future work to better model cache memories using high-level frameworks.

# Chapter 2

# Theory

This chapter presents the theory and terms related to cache memories necessary to understand the majority of this thesis. It will also introduce some information about power and energy usage related to cache memories.

## 2.1 Concepts Related to Cache Memory

In general, a cache is a component that temporarily stores data from all manner of storage devices so that future requests to that data can be served faster. A cache works on the principle of locality of reference, the tendency of an application to reference a predictably small amount of data in a given window of time [5]. This thesis is focused on hardware cache memories placed between a CPU and the main memory to reduce the time and energy cost of accessing memory by the CPU [5].

A memory hierarchy can consist of multiple *levels* of caches, where the smallest and fastest are placed closest to the CPU and the larger and slower are closer to the main memory [4, 6, 5, 7]. The levels can have different properties based on whether its placement closer or farther away from the CPU. A cache level can also be split into multiple caches, where each cache serves a specific purpose, e.g., one cache for instruction memory references and one for data memory references, respectively referred to a *instruction caches* and *data caches*. A cache level with multiple caches is called a *multilateral* cache level. A cache which can hold any kind of memory reference is referred to as a *unified cache* [4, 6, 5, 7].

A memory request from the CPU can be either a *read* or a *write* [4, 6, 5, 7]. A read request is issued when the CPU is retrieving data from memory, and a write request is issued from the CPU when new data is to be stored at a specified address in memory. A cache *hit* or *miss* is related to whether the address specified in a memory request is found in the cache at the time

of the request or not. In the case of a miss, the request is forwarded to the next memory level in the hierarchy [4, 6, 5, 7].

A cache stores memory references as *cache blocks*, also called *cache lines* [4, 6, 5, 7]. These are collections of neighboring addresses in memory, which are fetched together when one of the addresses contained by it is requested. The size of a cache block is chosen by the designer of the cache and is referred to as *block size* or *line size*. Caches are categorized based on how many possible locations a given cache block can be placed into a cache. This is called the *placement policy* of the cache [4, 6, 5, 7]. If a given block can be placed anywhere in a cache, the cache is *fully-associative*. A cache that has one specific location for every given block is *direct-mapped*. If the block can be placed at a restricted $n$ different locations in a *set*, the cache is *n-way set-associative*, where every possible location in the set is called a *way*. The number of ways a block can be placed in a set is known as the *associativity* of the cache. The total size of the memory references a cache can hold is known as the *cache size* [4, 6, 5, 7].

With the restricted size of the cache, a block might need to be evicted from the cache as another is fetched in response to a miss. For all other cases than direct-mapped caches, a replacement policy is thus needed. There are many replacement policies developed for caches specifically, but some of the simple and commonly used policies are: (1) first-in-first-out (FIFO), (2) least-recently-used (LRU), and (3) random. A FIFO policy will replace the block that has been in the set longest. An LRU policy will replace the block in the set that was used least recently. The random policy will pick a random block to evict [8, 4, 6, 5, 7].

Since there is a fundamental difference between read and write accesses in that writes will modify the data stored at a memory location, and that this memory location might reside at multiple locations in the memory hierarchy at a given time, a cache needs *write policies* to know how to handle certain situations. If there is a write-hit in a cache, there are traditionally two ways of handling it. *Write-through* means that if there is a write-hit in the cache, the write is forwarded throughout the memory hierarchy. *Write-back* on the other hand will wait until the block which is written to is evicted before the new content is forwarded to lower memory levels [9, 4, 6, 5, 7]. In the case of a write-miss, there are also traditionally two ways of handling it. *Write-around* will forward the write request to the next memory level, but the block which is written will not be fetched to the higher levels of the memory hierarchy. The other way is *write-allocate*, which is to fetch the block that was missed with the write. A read does not need such policies, as a read request to an address will always fetch the block in the case of a miss [9, 4, 6, 5, 7].

In a multicore processor, each processor has its own cache in some cases. As we can imagine, this might lead to very complex problems if these processors are sharing some memory locations. This is mainly related to memory that can be written, as there may arise a situation where

the same memory location holds different content at the same time at two different points of the memory hierarchy. This is where *coherency protocols* are introduced to guarantee that all processors see the same content at memory locations at all times. The coherency protocols are a set of rules that applies to all shared memory to make sure that every instance of a memory location is updated if one of the instances has its content updated [4, 6, 5].

## 2.2 Concepts Related to Modeling and Simulation

There are many different definitions of modeling and simulating. Some might say that they are synonymous, but this thesis will separate the terms and use the definitions provided by [10]. They are as follows:

- A model is a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process.
- A simulation is a method for implementing a model over time.

These definitions mean that a model is the representation of a system, while a simulation is the utilization of that model to look at the performance of the system.

### 2.2.1 Physical Cache Modeling

As briefly mentioned in Chapter 1, the goal of a physical cache model is to get energy and timing related values that are connected to a specific cache structure.

[7] presents a workflow for creating predictable memory models for a specific purpose. It is suggested that energy models for memories are developed using one of three general approaches: (1) by the use of datasheet models of the components the memory architecture consists of to get values for energy and time used per access, (2) by making measurement models if the components we intend to use are physically available such that we can measure the energy dissipation and time usage, or (3) by constructing analytical models which are made from observations that are true for all types of a specific memory component.

The datasheet model is the easiest to implement if we have that particular information available. Analytical models are the most versatile, as these model generic memories and thus have almost infinite different design options. A measurement model can only be used for the memory measurements have been done on, but will be the most accurate for this specific memory.

### 2.2.2 Functional Cache Modeling

The goal of a functional cache model is to be able to run simulations of the execution of a program and capturing results regarding hits and misses in the modeled caches. A functional model is dependent on input in the form of memory accesses to make a simulation. The methods for driving the input of a memory simulation can mainly be done in two ways: (1) *execution-driven simulations* and (2) *trace-driven simulations*.

#### Execution-Driven Simulations

Execution-driven simulations are characterized by running an application on a processor model to obtain stimuli of a memory model in the form of real-time memory accesses. This process requires the inclusion of an instruction-set emulator and an interface between the processor model and the memory model. The instruction-set emulator interprets each instruction and directs the memory model's activities [11].

#### Trace-Driven Simulations

A trace-driven simulation employs a file containing a sequence of memory references, called a *address trace*, made to mimic how a real processor might access the memory. A trace-driven memory simulation is sometimes viewed as consisting of three main stages: *trace collection*, *trace reduction*, and *trace processing*. The trace collection is the process of creating the address trace. This file can become very large, so trace reduction techniques are sometimes needed to remove redundant parts of it. Trace processing is the final stage, where the address file is fed to a functional memory model that simulates the behavior of that memory [12].

### 2.2.3 Model Assessment

To make evaluations on how good or bad a model is, we need clearly defined criteria of the assessment of the model. The model's correspondence with reality is known as *fidelity*. Fidelity is characterized by attributes like *accuracy*, *precision*, *timeliness*, potential *error sources* and *uncertainties*, *consistency*, and *repeatability* [10].

This thesis will focus mainly on accuracy, precision, error sources, consistency, and repeatability when addressing fidelity. Accuracy can be measured by comparing the results of a model with the real-world data of what it is trying to model. Precision is the limiting factor of accuracy, as in how precisely the model represents reality. Evaluation of error sources is essential to know what the likelihood of errors altering the results is. Consistency addresses whether the results are biased in any way, and repeatability means that the simulation should produce the same results given the same stimuli [10].

## 2.3 Energy and Power

The terms energy and power are often used synonymously. Although they are related, they are not identical. Electrical power $P$ is defined as the product of Voltage $V$ and Current $I$ [7] and is measured in the unit Watt:

$$P = V \cdot I. \tag{2.1}$$

Energy on the other hand is described by the integral of power $P$ over the time $t$ [7] and is measured in the unit Joule, which is equal to a Watt-second:

$$E = \int P dt = \int V \cdot I dt. \tag{2.2}$$

# Chapter 3

# Related Work

Numerous studies look into the modeling of cache behavior and how to guide design exploration to obtain an optimal cache structure. In order to be sure that the performance of the studied cache structure is correct, we need to have good models, but what defines a good model depends on how it will be used. This section will describe some of the related work done in the development of cache models for different purposes.

## 3.1 Functional Models

*gem5* [13] is a full-system functional simulation framework which constructed as a merger of the best aspects of the *GEMS* [14] and *M5* [15] simulators. The product is a framework has high configuration capability, with multiple ISAs, CPU models, and flexible memory systems. Since it is a full-system simulator and contains a CPU model, it is an execution-driven simulator that can run applications directly, as explained in Section 2.2.2.

Another full-system framework that utilizes execution-driven simulation is *SimpleScalar* [16, 11]. The framework integrates the multi-level cache memory simulator *Sim-Cache*. All the components of the full-system framework are execution-driven. It is done this way to enables greater analysis of dynamic energy usage, as it provides access to all the data produced and consumed during program execution. It also allows for analysis of branch prediction, which is not possible using a trace-driven simulation technique. The cost of this choice is an increased model complexity and the difficulty of reproducing experiments, two general problems for execution-driven simulators.

*mlcache* [17] is an event-driven cache simulator that is made to be integrated into a full-system simulator environment. It was one of the first cache simulators to support various configurations of multi-lateral caches.

*ZOOM* [1] is a simulation framework for cache optimizing and characterizing the performance, energy, and area of low-power caches in the early stages of design. The framework consists of both a timing-sensitive trace-driven functional simulator and an analytical micro-architecture simulator used for physical modeling.

## 3.2 Physical Models

*CACTI* [18, 19, 20, 21, 22, 23, 24], is an analytical physical cache model, which is widely used among computer architecture researchers, according to [7]. It uses a configuration file with design options and design goals to model generic cache memories with relatively high precision. It can also be used to model generic RAM memories and off-chip main memory storage. There have also been developed own versions of the model to account for weaknesses in the supported configurations and models used in the tool. *eCACTI* [25] was developed to expand the leakage models of *CACTI*, and remove limitations of constant gate widths and account for power dissipation in sub-blocks outside of the critical path. These inaccuracies were observed to cause *CACTI* to suggest suboptimal cache structures. *CACTI-D* [26] added support for modeling commodity DRAM technology and support for main memory DRAM chip organization, and thus enabling modeling of complete memory hierarchies with consistent models throughout the hierarchy. *CACTI-P* [27] added support for major leakage reduction approaches. All the enhanced versions of the model are incorporated in the latest release of *CACTI*.

*Wattch* [28] is a framework for power analysis and optimizations on the architectural-level. It uses power models of different parts of computer architectures with a high configuration capability to allow users to accurately model and optimize systems in an early design phase. It was presented in 2000 and was state of the art for such models for a long time. As mentioned in [29], *Wattch* did not scale well into deep-submicron technologies and lacked the ability to model multi-core architectures. *McPAT* [29] addressed these problems and advanced the architectural-level models by integrating models of power, area, and timing, and supporting multi-core architectures. *McPAT* accounts for all power dissipation relevant for the deep-submicron era, and by the use of an *XML* file for results, it can easily be combined with any functional simulator. In [30], sources of error in *McPAT* was quantified, and in spite of accuracy gaps, it was concluded that most of its limitations could often be adequately addressed.

As mentioned, many-core architectures have become increasingly relevant, and thus NoCs and buses are equally as important to model. *ORION* [31] is a tool used for physical modeling of such structures, and can be combined with physical models of other parts of computer architectures to enable better design choices for the architecture as a whole.

All of the physical model mentioned above rely on physical device parameters for different technology nodes to accurately represent real-world components. *MASTAR* [32] is a model

based on the industry-standard *ITRS* [33] and gives prediction of advanced CMOS structure performance knowing only their main technological parameters. It is used by the most state of the art physical modeling tools, e.g., *CACTI*, *McPAT* and *ORION*.

# Chapter 4

# MPCache Framework

This chapter presents an overview of the MPCache framework. Choices made about the components of the framework in the context of the complete framework will be discussed.

## 4.1 Components of the Framework

As mentioned in Chapter 1, MPCache has design exploration of cache structures as its primary purpose. It will be the grounds for the choices taken regarding the framework and its components. This means that the framework should support many of the design options one would consider for real cache memories. It also means that the models contained by the framework must be simple to configure, such that it is uncomplicated to set up several structures that can be compared and that these can easily be altered. The file format used for input and result files will be discussed for each component.

### 4.1.1 The Functional Model

The task of the functional cache model is to mimic the behavior of a specific cache when accessed by a CPU. The results of using the model in a simulation should be the sums of different scenarios that are caused by accesses to a cache. The possible scenarios of an access to a cache seen by the MPCache framework are (1) read-hit, (2) read-miss, (3) write-hit, and (4) write-miss.

#### Modeling Approach

As mentioned in Chapter 2, functional cache models are usually either execution-driven or trace-driven. The choice between the two boils down to the purpose and usage of the model. The model should be focused on the exploration of cache designs and should not be concerned

with the system configurations needed to make it compliant with a CPU model in an execution-driven simulation framework, and possibly other components as well. A trace-driven model must only be able to interpret a file containing a memory access trace made for it. That creates a two-step process of generating traces using a CPU model and then running simulations on the model with the trace file. Still, the trace file can be reused for multiple models as long as the configuration of the underlying main memory is the same. The meaning of this is that the main memory is of the same size, is partitioned equally and that the same addresses are used for each part. When the limited time to develop the model is accounted for, a trace-driven model appears like the most sensible choice. It will set the focus of the development at the features of the model rather than its surroundings. Another advantage of a trace-driven model is that, unlike the execution-driven, the cache is not required to provide data to the CPU. It can, therefore, abstract away the data memory, and only keep track of the tag array, which will simplify the simulation.

#### Input Files

Based on the above discussion, we need to define the format of the trace file that the functional model will utilize. To make it as simple as possible, it will only contain the necessary information, which is the address of the access and the access type. It will be on a similar format to what that used by the trace-driven model *Zoom*, which was described in 3:

```
Address, Access type
```

where the address is a decimal integer, and the access type is either 1 or 0 to represent a write or a read, respectively. *Zoom* used a hexadecimal number to represent the address, but since decimal numbers are easier to handle in Python, they are preferred. With this simple format, the memory access trace can be stored in a plain text file.

A configuration file is needed to hold the information about the cache that is being modeled. The information that is needed by the model is presented in Chapter 5. A number of different configuration file standards exist, e.g. *XML*, *INI*, and *YAML*. These three are all both human-readable and machine-readable, while *XML* files are possibly the most verbose and complicated. The configuration file should support sectioning, such that multiple model configurations can be set up in their section and linked in the same document. A section is used to hold containers containing the properties listed above. *YAML* supports complex data types, but the containers needed for this model only need to support simple strings and numbers. The *INI* format is chosen as it is one of the simplest configuration file standards, while still providing all needed functionality. By keeping it simple, the model will be easy to configure, and the file can be easily adjusted after setting it up.

#### Result Files

Since the model will be used to produce results in simulations, we need to define the format of the file containing the results. For this, two formats will be considered - textitXML and *CSV*. Both are both human-readable and machine-readable. The main difference is that textitXML contains the data in what can be described as a nest of categories, while *CSV* sorts the data in a simple and table-like fashion. The results of functional simulations will be separated based on both which configurations are used and which memory access trace is used in the simulation. One configuration can thus produce many different results, and the nesting functionality of textitXML can prove to be useful for this. Therefore, this is the chosen format for the result files produced by the functional model.

### 4.1.2 The Physical Model

The physical model will be used to assign estimated values of energy usage and timing penalties of accesses to a specific cache structure. It will also estimate the leakage power of the cache. The physical cache model will estimate the dynamic energy usage of the same scenarios that were mentioned in Section 4.1.1: (1) read-hit, (2) read-miss, (3) write-hit, and (4) write-miss. This way, the results of the physical model can quickly be joined with the results of the functional model. As these are the only possible cache access scenarios, we simply need the corresponding results from a functional simulation to quickly calculate the total energy usage of a cache for that simulation.

#### Modeling Approach

Since the model will estimate both energy usage and timing penalty of access, as well as the leakage power of a specific cache, some different approaches are needed for the estimation of the different parameters.

To simplify the modeling of the energy usage and leakage power of a cache, a similar approach to *CACTI*, mentioned in Section 3.2, is taken. There the estimated total dynamic energy of a cache is equal to the dynamic energy usage of the tag array, the data array, and comparators used to compare the tag added together. The total leakage power is the leakage from the two arrays added together. The reason for not using CACTI directly is based on the configuration of CACTI, which is quite complicated if we want to model a specific structure. It is also based on the results of CACTI, which are not made to be used directly in combination with a functional cache model, which will be discussed in Chapter 7. CACTI is made for Unix and Linux platforms, which would restrict the possible platforms that can use the framework.

As stated in Chapter 2, there are three general approaches for creating energy models of memories - (1) datasheets, (2) measurements, and (3) analytical. As the MPCache framework is

made with design exploration as its primary goal, we can not assume that a user can make measurements to be used by the model. The analytical approach models a generic memory array while the datasheet represents a unique memory block. This would mean that a datasheet model should be more, or equally, as precise as an analytical model. Memories do have a highly regular structure and few states to model which make analytical models a good option [7], but it is assumed that developers of embedded cache systems have a predefined set of memory blocks available, which make a datasheet model more accurate for that specific design space and easily reconfigurable as the values from the datasheet can be replaced by other effortlessly. The datasheet values can most likely not be used directly by the model. Thus some analytical calculations must be made to produce the energy estimates, making the model a combination of a datasheet and analytical model.

The timing penalty of an access to a specific cache relies on various properties of the cache and the surrounding system of the cache. These are, e.g., the access time of the memory blocks contained by the cache, its access mode, and bus latency based on the placement of the cache relative to the CPU. Therefore the timing penalty is best modeled by the user providing an accurate prediction about how many clock cycles an access to the cache would take.

**Input Files**

The physical model utilizes the same configuration file as the functional model described in Section 4.1.1. The physical model needs some additional parameters which the functional model does not use, like the datasheet values used in the estimation of the energy usage. However, a user will only need to set up one configuration file for simulation with both models, and all the information about the cache is contained in one place.

**Result Files**

Unlike the functional model described in Section 4.1.1, the physical model will yield the same result for all simulations with the same configuration. Therefore, the much simpler result file format *CSV* has been chosen for this model.

### 4.1.3 The Combined Simulation

The MPCache framework will facilitate for combined functional and physical simulations. This facilitation will mainly be done by making the models compatible with each other. However, the framework also contains a simulation file for simulating with both models and combining the results. By combining the results, an estimate for the execution time of an application represented by a trace file can be produced with the total energy usage of each cache involved in the simulation. A methodology for creating memory hierarchy simulations will, therefore, have to be implemented. That is because we need to track the accesses to lower-level memories

when an access to a cache misses to be able to calculate the correct execution time. An example of the usage of the simulation file can be found in Appendix A.3.

**Combining the Results**

Below, the equations used by the simulation file to combine the results of the functional and physical model are shown. The execution time $t_{execution}$ is calculated as

$$t_{execution} = \sum_{i}^{Levels} \left( n_{accesses_i} \cdot t_{penalty_i} \right), \tag{4.1}$$

where $n_{accesses}$ is the number of accesses, and $t_{penalty}$ is the access penalty time. $Levels$ is a representation of every memory level that can be accessed in a memory hierarchy.

The total energy applies the dynamic energy usage and the number of read-hit, read-miss, write-hit, and write-miss accesses, subscripted by $rh$, $rm$, $wh$, and $wm$, respectively. This is combined with the energy consumed by the leakage current $E_{leak}$. These values are the estimates produced by the physical model. The calculation is as follows

$$E_{total} = \sum_{i}^{Levels} \left( E_{rh_i} \cdot n_{rh_i} + E_{rm_i} \cdot n_{rm_i} + E_{wh_i} \cdot n_{wh_i} + E_{wm_i} \cdot n_{wm_i} + E_{leak_i} \right). \tag{4.2}$$

$E_{leak}$ is calculated as

$$E_{leak_i} = P_{leak_i} \cdot t_{execution}, \tag{4.3}$$

where $P_{leak}$ is the leakage power.

**Result Files**

To make the result file handling as easy as possible for the combined cache simulations, the *CSV* format will be used for the file. This is based on the result file discussion in Section 4.1.1.

## 4.2 Overview of Framework

Figure 4.1 presents the overview fo the MPCache framework based on the discussion above, as well as the required input files for the different components and the output files containing

results.

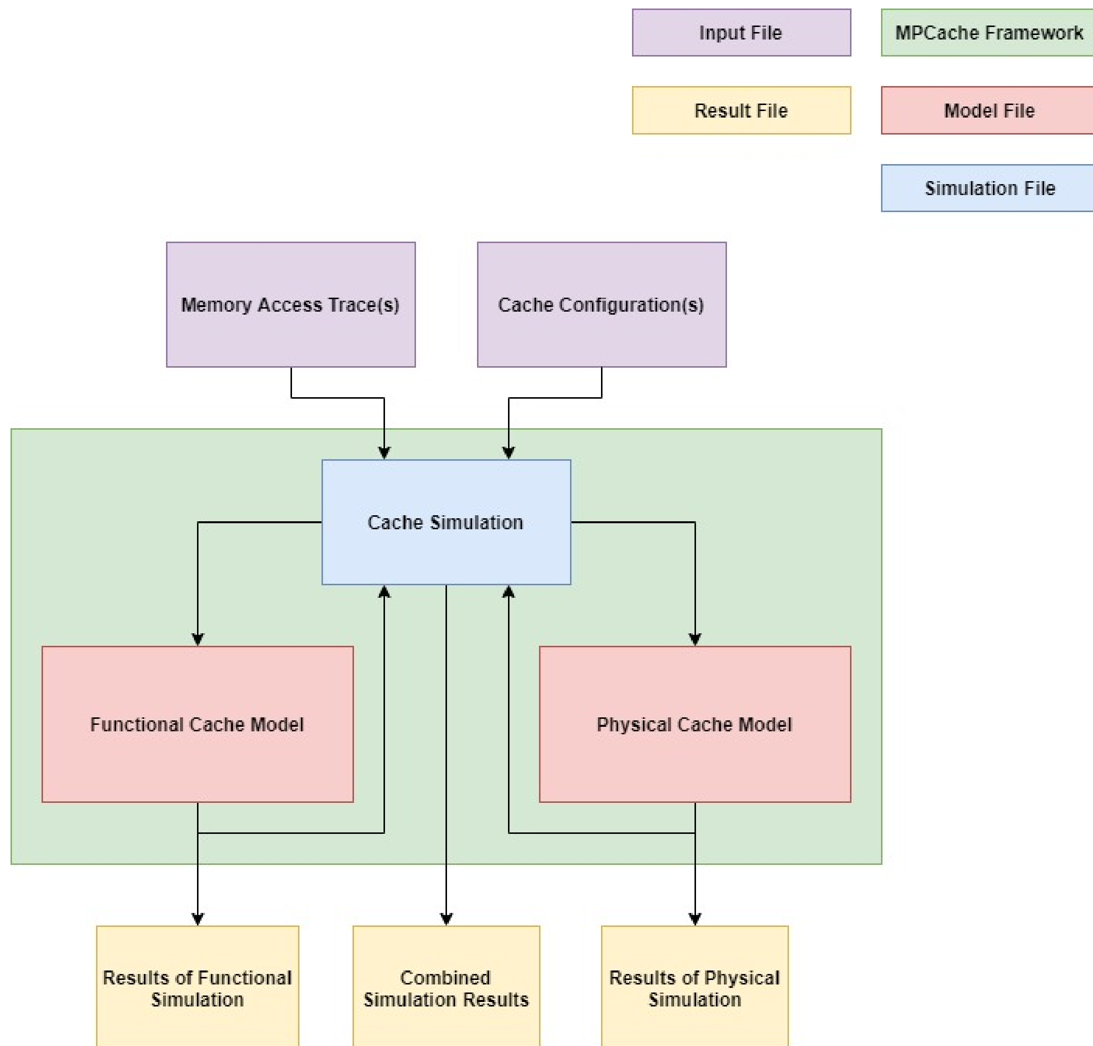


**Figure 4.1:** Overview of the MPCache framework.

As we can see, the framework in itself consists of two models - the functional cache model and the physical cache model - and a simulation file to enable usage of the models in simulations. The models are general cache models, which get configuration files passed to them by the simulation file to represent specific caches. The functional model will also get the memory accesses of the trace files passed to it through the simulation file. Each of the models produces a results file of its own, which the simulation file applies to obtain a joined simulation result. The simulation file can configure multiple models for each simulation.

# Chapter 5

# Functional Cache Model

This chapter presents the implementation of the functional cache model of MPCache and how it is used in a memory hierarchy simulation, as well as the methodology for verifying that the simulation results are as expected and the presentation and discussion of the results from this verification. The purpose of the functional cache model is to accurately calculate the amount of hits and misses of read and write accesses to a given cache structure for a certain sequence of memory accesses. This model can be used in the representation of a complete memory hierarchy with potentially multiple lateral and vertical cache levels, by combining different functional cache models with each other.

## 5.1 Implementation

The functional cache model is a component of the MPCache framework, described in Chapter 4. As mentioned, the functional model must record if an access results in a read-hit, read-miss, write-hit, or a write-miss. To be able to run simulations on a complete memory hierarchy, a functional model must also be made to represent main memory instances. These are meant to be the endpoints of the hierarchy and will always register an access as a hit. Therefore, they are closer to containers for the number of accesses rather than functional models.

The model is implemented in the *Python 3* programming language and can therefore run on any platform that can execute Python code. It is the chosen programming language because it both easy to use and has an understandable syntax. The model could have been implemented in *C/C++*, and this could likely have made the execution faster, but the ease of use of Python is regarded as an higher priority for this model than the absolute fastest and most optimized execution time. The complete source code of the functional cache model and the other functional models needed to run a memory hierarchy simulation is found in Appendix A.1.

### 5.1.1 Configuration of the Model

The ideal functional cache model would support every possible aspect of how a cache can be structured. However, every option the model gives to the user has to be implemented by the model. This means that with limited time to develop the model, a subset of all possible cache structures that still supports a wide range of different possibilities has to be selected. Among all the possible cache configuration options, we find a separation between fundamental properties of a cache, which are found in all cache memories, and optimization features, which supplement or improve the functionality and performance of the cache. Some examples of optimization features are prefetchers, write-buffers, and speculative fetching logic. The functional model will only support what is seen as fundamental properties but will be modular and easily supplemented to support more features. New features can be added by supplementing the part of the source code that targets that particular feature with additional functionality. By having the source code structured according to cache features, the relevant placement in the code should be easy to find.

Deciding precisely what are fundamental properties and what are not can undoubtedly be a subject for discussion but for this model these properties are as follows: (1) associativity, (2) cache size, (3) block size, (4) replacement policy, (5) write-hit policy, and (6) write-miss policy. The nature of the associativity, cache size, and block size makes it possible to allow all possible valid values to be assigned to these parameters, as they only affect the sizing of the tag and data array and one of three arrangements of these arrays based on the associativity. Valid values mean values that create even sets in the cache, e.g., a cache size of 8B and block size of 7B, would not create an even number of sets. For the three types of policies, a set of possible options must be chosen for each, as every option will require a specific implementation. FIFO, LRU, and random are chosen as the replacement policy options because they are common and basic replacement policies in caches, and more advanced replacement policies are usually variations of them. The same approach is taken to the write-hit and the write-miss policies by supporting write-back and write-through, and write-around and write-allocate, respectively, which are the most apparent policies.

The configuration parameters and options for the model are listed in Table 5.1. The valid values of each parameter are listed along with it, as well as the possible alternatives to chose between for the configuration options. As we can see, there are two parameters listed which were not mentioned above: *Address Space* and *Next Memory Level*. These parameters are not directly associated with the configuration of the cache like the others but are needed to construct memory hierarchies. *Address Space* is used to specify which memory accesses a cache should handle and which it should ignore. It is used in multi-lateral caches, e.g., separate instruction and data caches on the same level. *Next Memory Level* is used to hold pointers to one or multiple underlying memory models a cache should send its memory accesses to, e.g., a level 2 cache or

```
[L1]
top_level           = True
memory_type         = cache
address_bits        = 32
associativity       = 1
size_bytes          = 8192
block_size_bytes    = 8
write_hit_policy    = write-back
write_miss_policy   = write-around
replacement_policy  = lru
address_space       = 0,32767
next_memory_levels  = Flash

[Flash]
top_level           = False
memory_type         = main_memory
address_space       = 0,32767
```

**Listing 5.1:** Example of a *INI* configuration file used by the functional model.

flash memory. Using these two parameters, we have the freedom to build almost any hierarchy of memories imaginable.

**Table 5.1:** Valid values for all the cache configuration parameters and options.

| Configuration Parameters | Valid Values |
|---|---|
| Associativity | 0 = Fully-Associative, 1 = Direct Mapped, n = n-Set Associative, $n \in Z_+$ |
| Cache Size | $2^n$ bytes, $n \in Z_+$ |
| Block Size | $2^n$ bytes, $n \in Z_+$ |
| Write-Hit Policy | Write-Back, Write-Through |
| Write-Miss Policy | Write-Around, Write-Allocate |
| Replacement Policy | FIFO, LRU, Random |
| Address Space | Lowest and Highest Address of Valid Address Space |
| Next Memory Level | Pointer to One or More Other Cache or Main Memory Objects |

In Chapter 4, the configuration file format which will be used to hold this information was discussed. An example of the *INI* configuration file format used by the model is shown in Listing 5.1.

There are a few parameters listed in the example which have not been mentioned yet, like `top_level`, `memory_type` and `address_bits`. The two former are used in the structuring of a memory hierarchy, while the latter is used in the calculation of how many tag bits are needed.

### 5.1.2 Memory Access Handling

The memory access handling is the main part of the functional model, as it determines the active part during the simulation. It utilizes the trace file format decided in Chapter 4. The memory access handling of the functional cache model is shown in Figure 5.1. The Figure has been made to show the general flow the model should have. It is made to support handling of accesses based on any of the valid cache configurations listed in Section 5.1.1. We can see that

the handling uses two functions: `Search Tag Array` and `Place Cache Line`. These will of course depend on the configured associativity, write-hit policy and replacement policy.

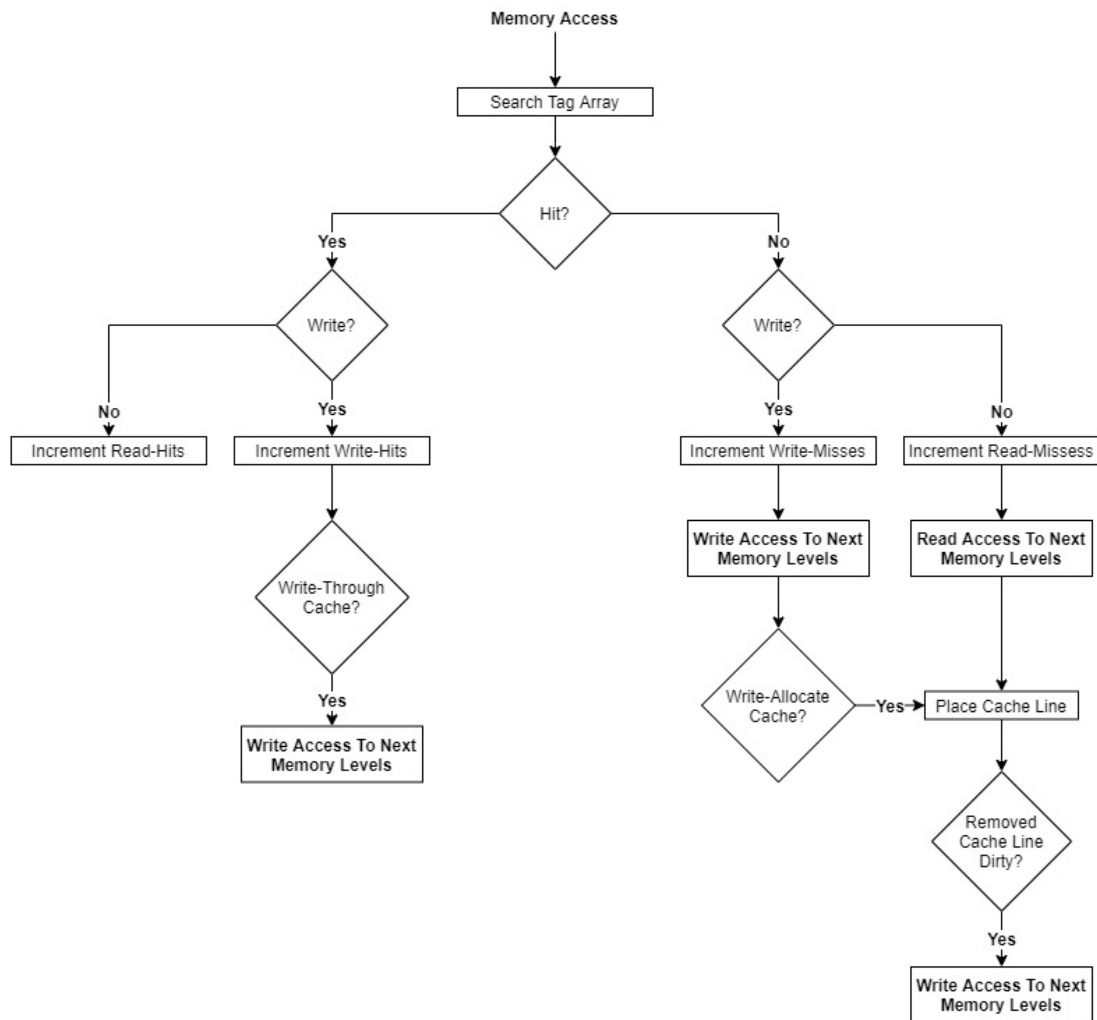


**Figure 5.1:** Flowchart of the memory access handling of the functional cache model.

The Python implementation of the memory access handling described by Figure 5.1 is shown in Listing 5.2. It is is many ways just a practical interpretation of the flow diagram from Figure 5.1.

The functions for searching the tag array and placing a new cache line are implemented for the three different placement policies, and lined to the `_search_tag_array` and `_place_cache_line` functions in the initialization. The implementation of this in Python is shown in Listing 5.3.

This implementation is one of the advantages of using Python, because we have the ability of assigning functions to others, making the code more flexible for different configurations. As an example of what the search and placement functions look like, their implementations for a set-associative structure is shown in Listing 5.4 and 5.5.

```
1     def memory_access(self, address, write):
2         if ((address >= self.address_space[0]) and (address <= self.address_space[1])):
3             # Valid address space
4             address_bits = "{}".format("{0:b}".format(address).zfill(self.address_bits))
5             hit = self._search_tag_array(address_bits, write)
6             if (hit):
7                 if (write == 1):
8                     self.write_hits += 1
9                     if (self.write_hit_policy == 'write-through'):
10                        for next_memory_level in self.next_memory_levels:
11                            next_memory_level.memory_access(address=address, write=1)
12                else:
13                    self.read_hits += 1
14                return
15            else:
16                if (write == 1):
17                    self.write_misses += 1
18                else:
19                    self.read_misses += 1
20                for next_memory_level in self.next_memory_levels:
21                    next_memory_level.memory_access(address=address, write=write)
22                if ((write == 0) or (self.write_miss_policy == 'write-allocate')):
23                    removed_cache_line = self._place_cache_line(address_bits)
24                    if (removed_cache_line.dirty):
25                        for next_memory_level in self.next_memory_levels:
26                            next_memory_level.memory_access(address=int(removed_cache_line.
      address, 2), write=1)
27                return
28        else:
29            # Not valid address space
30            return
```

**Listing 5.2:** The Python implementation of the memory access handling of the functional model shown in Figure 5.1.

```
1     if (self.associativity == 0):
2         self._search_tag_array = self._fully_assoc_search
3         self._place_cache_line = self._fully_assoc_placement
4     elif (self.associativity == 1):
5         self._search_tag_array = self._direct_mapped_search
6         self._place_cache_line = self._direct_mapped_placement
7     else:
8         self._search_tag_array = self._set_assoc_search
9         self._place_cache_line = self._set_assoc_placement
```

**Listing 5.3:** The assignment of the appropriate search tag array and place cache line functions.

```
1     def _set_assoc_search(self, address, write):
2         index = int(address[self.tag_bits:self.tag_bits+self.index_bits], 2)
3         for i, cache_line in enumerate(self.tag_array[index]):
4             if (cache_line.address[0:self.tag_bits] == address[0:self.tag_bits]):
5                 if ((write == 1) and (self.write_hit_policy == 'write-back')):
6                         cache_line.dirty = True
7                 if (self.replacement_policy == 'lru'):
8                     self.tag_array[index].insert(0, self.tag_array[index].pop(i))
9                 return True
10        return False
```

**Listing 5.4:** The implementation of the search tag array function for a set-associative structure.

```
1      def _set_assoc_placement(self, address):
2          removed_cache_line = None
3          index = int(address[self.tag_bits:self.tag_bits+self.index_bits], 2)
4          if (self.num_occupied_ways[index] == self.associativity):
5              if (self.replacement_policy == "random"):
6                  i = random.randint(self.associativity)
7                  removed_cache_line = self.tag_array[index].pop(i)
8              else: # FIFO, LRU
9                  removed_cache_line = self.tag_array[index].pop()
10         else:
11             removed_cache_line = self.tag_array[index].pop()
12             self.num_occupied_ways[index] += 1
13         self.tag_array[index].insert(0, CacheLine(address, (self.write_hit_policy == 'write-
       back')))
14         return removed_cache_line
```

**Listing 5.5:** The implementation of the place cache line function for a set-associative structure.

### 5.1.3 Memory Hierarchy Simulation

The functional cache and main memory models can be used to create simulations of memory hierarchies. This is done by using a functional model of a memory hierarchy, which is just a supporting model which consists of functional cache and main memory models. The memory hierarchy is constructed by configuring one or multiple functional cache models and specifying how they are connected to each other and main memory models in the *INI* configuration file. The memory accesses can be read one by one from the file and be supplied to the top memory levels of the memory hierarchy using the memory access methods. An example of a memory hierarchy configuration using the *INI* format is shown in Listing 5.6. This example only show the parameters used in the structuring of the hierarchy in a simulation. A visual representation of what the hierarchy would look like is shown in Figure 5.2. The arrows shown in the Figure represent memory accesses.

```
1  [I-Cache]
2  top_level            = True
3  memory_type          = cache
4  next_memory_levels   = L2
5
6  [D-Cache]
7  top_level            = True
8  memory_type          = cache
9  next_memory_levels   = L2
10
11 [L2]
12 top_level            = False
13 memory_type          = cache
14 next_memory_levels   = Flash
15
16 [Flash]
17 top_level            = False
18 memory_type          = main_memory
19
20 [RAM]
21 top_level            = True
22 memory_type          = main_memory
```

**Listing 5.6:** Example of a *INI* configuration file containing a specified memory hierarchy.

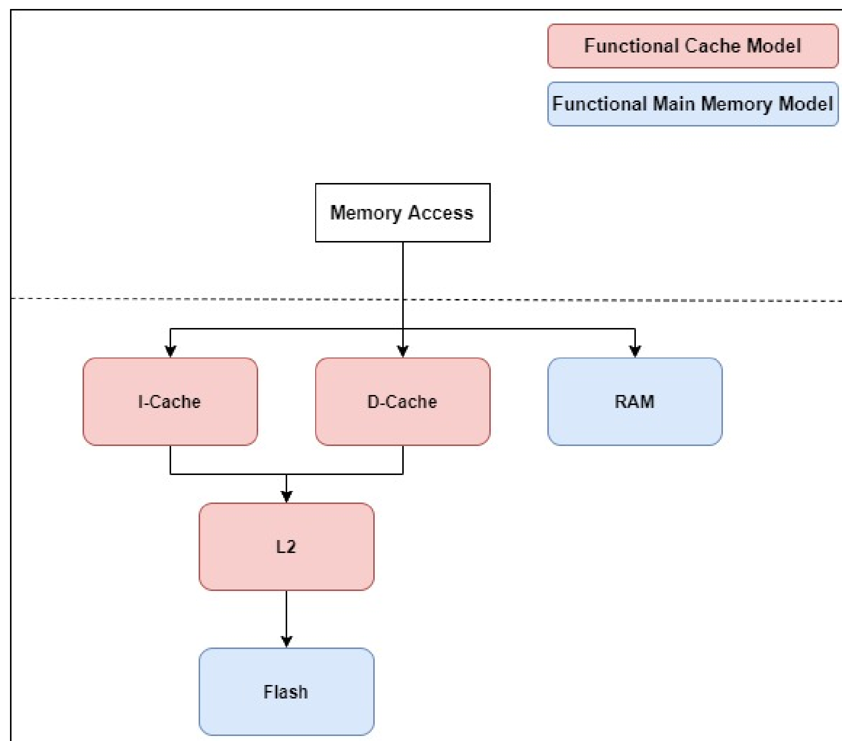


**Figure 5.2:** A visual representation of the memory hierarchy described in Listing 5.6.

# 5.2 Verification

## 5.2.1 Methodology

To be confident that the functional model produces the correct results for all cache structures and memory hierarchies it supports to model, we need some verification that indicates whether

the model does what it is supposed to or not. Some different approaches could be taken to do this. It could be done by the use of another, preferably established, functional cache model. By configuring the same cache structures for both models and driving them with random stimuli, we could compare the results to verify the correctness. The problem with this approach is that in the event of an error, the results would provide limited indications of what is wrong with the model. However, that could be solved by instead creating specific test scenarios, which would target different parts of the functionality. In this case, we would not need the other model, but could rather analyze the structure we are trying to model analytically.

For this functional model, it should be done in a way where we get to verify that changes to all adjustable parameters do not affect the correctness of the model. The verification should indicate whether the model is scalable, such that it can accurately model cache memories of any size.

The verification will, therefore, be done by setting up a test plan which involves which cache structures and memory hierarchy variations will be tested and which memory access sequence will be applied to test each. The cache structures included in the test plan will consist of one default structure and two variations of the default structure for each adjustable parameter, creating three data points for each of the parameters. For the cache model options of the write-miss policy, write-hit policy and replacement policy, all possibilities will be tested. For the modeling options of lateral and vertical multi-level caches, the test plan will include one example of each to compare it to the standalone default cache structure. Every possible combination of the parameters that will be chosen are not tested, simply because it would create too many different test cases.

From this test plan, the expected results for each cache structure applied with the given memory access sequence will be derived analytically. The expected result will be used as a comparison with the results from the corresponding simulation with the functional model. A match with the expected results will indicate that the model is behaving correctly, and a mismatch means that either the expected result is wrong or there is an error in the model. In the case of a mismatch in the results, the first step would be to reanalyze the memory access sequence to review the expected result. If the same expected result is derived or there is still a mismatch, the model must be investigated for errors. If an error is found and corrected, all the tests must be done over again.

To clearly see the variations in performance between the different cache structures, the memory access sequence must be constructed to highlight the differences in the cache structures. A general memory access trace has been made to accomplish this task. It is tailored to highlight the differences of cache structures of the size tested here, i.e., the results should not be the same for all the configurations. It can be found as **general_trace.txt** in Appendix B.2. This will be

**Table 5.2:** The default test value for all cache configuration parameters.

| **Default Test Configurations** | |
|---|---|
| Associativity | 2 |
| Cache Size | 16B |
| Block Size | 2B |
| Lateral Cache Levels, Address Space | 1, Unified |
| Vertical Cache Levels | 1 |
| Write-Hit Policy | Write-Back |
| Write-Miss Policy | Write-Around |
| Replacement Policy, Way (Set-Associative) | LRU |
| Replacement Policy, Set (Fully-Associative) | LRU |

used for all cache structures other than the ones testing write-hit policies, write-miss policies, and replacement policies. These will be tested with memory access sequences targeting these specific events, which can be found as **write_hit_trace.txt**, **write_miss_trace.txt** and **replacement_trace.txt** in Appendix B.2. Since the model handles read and write accesses identically, except for where the write-hit and write-miss policies are considered, the access traces only use read accesses wherever these policies are not tested.

The benefits of this verification methodology are that we get to test all functionalities we intended the model to have based on our own expectations of what results it should produce, by analyzing the memory access traces by hand for the test configurations and comparing this to the simulation results of the model. The potential drawback of this is that a misinterpretation of general cache behavior can lead to an error in both the expected results and the model, which will then go undetected. Another benefit of the methodology is that a continuous correct result as the value of a single parameter is adjusted signifies that the model is scalable for that parameter. The correctness of the results produced by the model should, therefore, be maintained for all legal values to the verified parameters.

The default values for the verification test cache configuration are listed in Table 5.2.

The verification test plan with the chosen values for the different cache configuration parameters are listed in Table 5.3. For each entry, there is a corresponding test case number to make it easier to identify the associated result of that particular test. It should be mentioned that some of the test cases overlap across the test parameters creating equivalent cache configurations, e.g., test case #3 and #5. These are kept to show a complete picture for each test parameter.

**Table 5.3:** The verification test plan with values for the cache configuration parameters and a corresponding test case number.

| **Test Parameter** | **Parameter Value** | **Test Case** |
|---|---|---|
| Associativity | 0 | #1 |
| | 1 | #2 |
| | 2 | #3 |
| Cache Size | 8B | #4 |
| | 16B | #5 |
| | 32B | #6 |
| Block Size | 1B | #7 |
| | 2B | #8 |
| | 4B | #9 |
| Lateral Cache Levels, Address Space | 1, Unified | #10 |
| | 2, Split 50/50 | #11 |
| Vertical Cache Levels (Cache Sizes) | 1 (16B) | #12 |
| | 2 (16B/32B) | #13 |
| Write-Hit Policy | Write-Back | #14 |
| | Write-Through | #15 |
| Write-Miss Policy | Write-Allocate | #16 |
| | Write-Around | #17 |
| Replacement of Way (Set-Associative) | FIFO | #18 |
| | LRU | #19 |
| | Random | #20 |
| Replacement of Set (Fully-Associative) | FIFO | #21 |
| | LRU | #22 |
| | Random | #23 |

### 5.2.2 Results and Discussion

The expected results of the verification test cases listed in Table 5.3 are shown in Table 5.4. The last column indicates whether the results of simulating the test case with the functional model matches the expected results or not.

We can see that the results match for all entries. Test case #20 and #23 was simulated multiple times to show that it could produce all of the possible results for the random replacement policy, and also to check if it ever was outside the valid results space.

The matching column of Table 5.4 is the last product of an iterative process. The first simulation run produced multiple mismatches compared to the expected results. Most of the mismatches was a product of a wrong manual analysis of the memory access traces leading to incorrect expected results, which was corrected. However, there was a mismatch for test cases #20 and #23 that originated in the functional model. This was due to a faulty replacement method, which made the random replacement option malfunction. After this error was corrected, all test cases was redone with the updated functional model.

**Table 5.4:** The expected results of the different verification tests of the functional model and a remark on whether it matched the actual results or not.

| **Test Case** | **Expected Results** | | | | | | **Match** |
|---|---|---|---|---|---|---|---|
| | **Cache Reads** | | **Cache Writes** | | **Main Memory** | | |
| | Hits | Misses | Hits | Misses | Reads | Writes | |
| #1 | 204 | 100 | 0 | 0 | 100 | 0 | Yes |
| #2 | 172 | 132 | 0 | 0 | 132 | 0 | Yes |
| #3 | 180 | 124 | 0 | 0 | 124 | 0 | Yes |
| #4 | 148 | 156 | 0 | 0 | 156 | 0 | Yes |
| #5 | 180 | 124 | 0 | 0 | 124 | 0 | Yes |
| #6 | 240 | 64 | 0 | 0 | 64 | 0 | Yes |
| #7 | 88 | 216 | 0 | 0 | 216 | 0 | Yes |
| #8 | 180 | 124 | 0 | 0 | 124 | 0 | Yes |
| #9 | 226 | 78 | 0 | 0 | 78 | 0 | Yes |
| #10 | 180 | 124 | 0 | 0 | 124 | 0 | Yes |
| #11 | 117 / 64 | 43 / 80 | 0 | 0 | 123 | 0 | Yes |
| #12 | 180 | 124 | 0 | 0 | 124 | 0 | Yes |
| #13 | 180 / 60 | 124 / 64 | 0 | 0 | 64 | 0 | Yes |
| #14 | 0 | 12 | 4 | 0 | 12 | 2 | Yes |
| #15 | 0 | 12 | 4 | 0 | 12 | 4 | Yes |
| #16 | 0 | 16 | 0 | 4 | 16 | 4 | Yes |
| #17 | 8 | 8 | 0 | 4 | 8 | 4 | Yes |
| #18 | 4 | 16 | 0 | 0 | 16 | 0 | Yes |
| #19 | 8 | 12 | 0 | 0 | 12 | 0 | Yes |
| #20 | Min 4, Max 8 | Min 12, Max 16 | 0 | 0 | Min 12, Max 16 | 0 | Yes |
| #21 | 4 | 16 | 0 | 0 | 16 | 0 | Yes |
| #22 | 8 | 12 | 0 | 0 | 12 | 0 | Yes |
| #23 | Min 4, Max 8 | Min 12, Max 16 | 0 | 0 | Min 12, Max 16 | 0 | Yes |

# Chapter 6

# Physical Cache Model

This chapter presents the general assumptions made and implementation details for the physical cache model of the MPCache framework, as well as the verification methodology and the results of the verification. The physical model aims to estimate the energy and time usage of cache memories. The results can be used in combination with the results of a simulation using a functional cache model with the same cache structure to determine the total energy and time usage of the simulated program execution.

## 6.1 Implementation

Like the functional model described in Chapter 5, the physical model is implemented in the *Python 3* programming language. The reason for this is that it will make the two models coherent and easier to use in combination with each other. The complete source code of the physical cache model can be found in Appendix A.2. The Appendix also contains an example for a physical main memory model, which can be used in combined memory hierarchy simulations.

As discussed in Chapter 4, the physical model is implemented as a combination of a datasheet model and an analytical model. The model will not include estimates for the energy usage of any control logic or registers contained by the cache, or wires used for interconnecting the memory cells to the inputs and outputs of the cache. The model will also only make energy estimates for the cache, and not the bus used to transfer the data to and from the cache.

A datasheet can provide precise values for how much energy is consumed by an access to a memory block. However, a cache will have multiple memory blocks which can be structured in a number of different ways, depending on design choice. The model will hence have to analytically calculate how many accesses and which type of access is made to each of the memory blocks contained by the cache for all the scenarios a cache access can cause.

Thus the modeling is making several assumptions about the energy usage of a cache. These assumptions will be verified in Section 6.2.

### 6.1.1 Configuration of the Model

The model uses the same configuration file as the functional model described in Chapter 4. This way, a user only needs to set up one configuration file for simulation with both models, and all the information about the cache is contained in one place. This configuration file includes the parameters listed in Table 5.1, but the physical model needs some additional parameters which the functional model does not use. These are listed in Table 6.1. The physical model has one clear difference compared to the functional model described in Chapter 5, which is that it can not model fully associative caches.

**Table 6.1:** Configuration parameters used only by the physical model and their valid values.

| Configuration Parameters | Valid Values |
|---|---|
| Number of Tag RAM Blocks | $n_{t_blocks} \in Z_+$ |
| Datasheet Read from Tag Energy | In nano joule (nJ), $E_{t_read} \in R^*_+$ |
| Datasheet Write to Tag Energy | In nano joule (nJ), $E_{t_write} \in R^*_+$ |
| Datasheet Tag Leakage Power | In micro watt (µW), $P_{t_leak} \in R^*_+$ |
| Number of Data RAM Blocks | $n_{d_blocks} \in Z_+$ |
| Number of Data Banks | $n_{d_banks} \in Z_+$, $\frac{n_{d_blocks}}{n_{dbanks}} \in Z_+$ |
| Datasheet Read from Data Energy | In nano joule (nJ), $E_{d_read} \in R^*_+$ |
| Datasheet Write to Data Energy | In nano joule (nJ), $E_{d_write} \in R^*_+$ |
| Datasheet Data Leakage Power | In micro watt (µW), $P_{d_leak} \in R^*_+$ |
| Access Penalty to Cache | In Number of Clock Cycles, $n_{penalty} \in Z_+$ |
| Clock Period | In nano seconds, $T_C \in R^*_+$ |

### 6.1.2 Estimation of Energy Usage

As mentioned in Chapter 4, the model calculates estimates for the dynamic energy consumption for four different scenarios of accesses to a cache: (1) read-hit, (2) read-miss, (3) write-hit, and (4) write-miss. It also calculates the total leakage power of all the memory blocks contained by the cache. If the simulation is done with a memory hierarchy with multiple caches, each cache needs a model configured like it.

The model assumes that the cache uses a SRAM-based organization of the cache which is similar to the architectural assumptions made by *Zoom* [1] and *CACTI* [24]. In Figure 6.1, which is taken from [1], we can se the basics organization of a SRAM-based cache.

These types of caches are used for direct-mapped and set-associative caches, meaning that we can not adequately model fully-associative caches using this approach, but need CAM based caches to do it [1]. These will not be considered here to simplify the task. The tag and data memories are separated and are accessed by a decoder which finds the appropriate memory

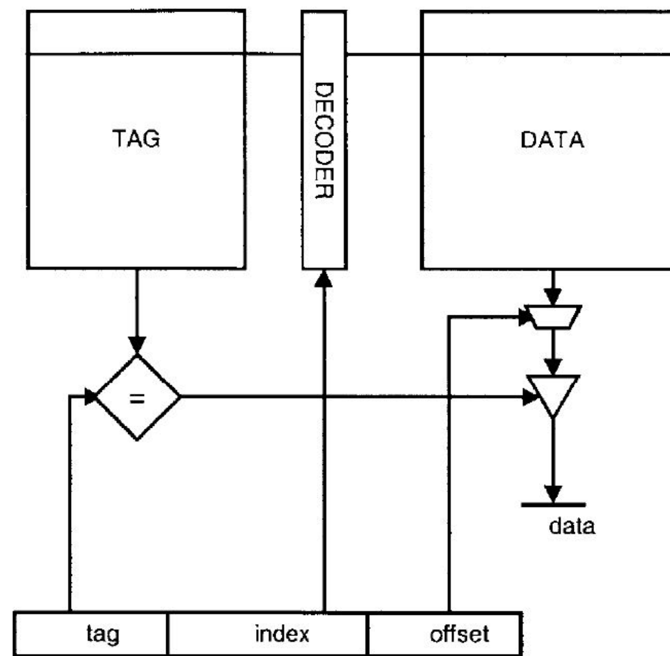


**Figure 6.1:** Organization of SRAM-based cache for direct-mapped or set-associative cache architectures [1].

location based on the index of the access. The tag and data memory blocks can also be divided up based on the ways of a set-associative cache, and the data memory can be divided into banks in which a cache line can be distributed over. We can also introduce different access modes to decide whether the memory blocks should be accessed in parallel or sequentially, as described by [22].

By analyzing these structures analytically, we can deduce how many accesses must be made to every memory block in the cache. Figure 6.2, taken from [2], shows an example on how this can be done for read accesses to: (a) a conventional, parallel cache, (b) a phased, sequential cache, and (c) a cache which implements way-prediction.

We can see that the parallel accessed cache will need read accesses to all tag and data memory blocks, while the sequentially accessed cache needs a read access to all tag blocks, but only to one data block. The last cache implements a conditional way-prediction, and the amount of read accesses made to memory blocks of the cache is dependent on whether the way-prediction is successful or not.

This physical model builds on the idea presented in Figure 6.2. The energy consumption for each of the four scenarios described will depend on the structure of the cache and the write-policies it support. In Figure 6.3 and 6.4, the types of memory access and to which memory blocks the accesses are made to are shown for a parallel and sequential access mode, respectively. The access mode signifies the order the tag and data memory of a cache is accessed. The

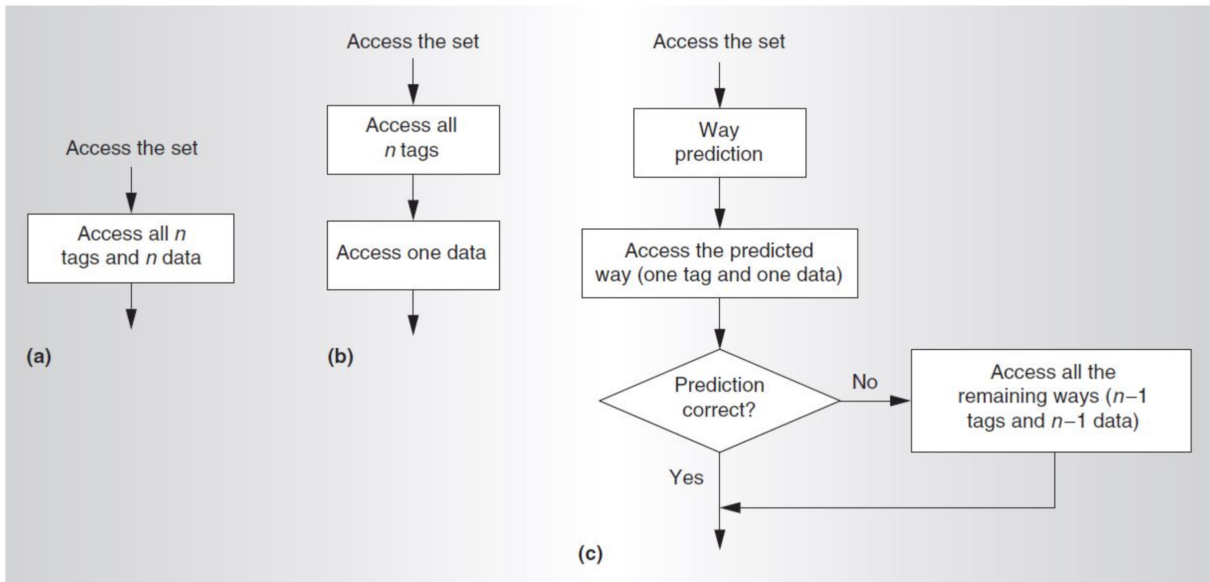


**Figure 6.2:** Access patterns for a conventional n-way set-associative cache (a), a phased n-way set-associative cache (b), and way-prediction n-way set-associative cache (c) [2].

scenarios for the two modes are shown for both write-hit and write-miss. Most accesses are self-explanatory, but the write accesses at the end of read-miss and write-miss (only for write-allocate) are there to represent the replacement of the missed cache line. These scenarios are common for both direct-mapped and set-associative caches. A fully associative cache would need its own analytical models, but they are not accounted for in this work.

From the figures we can see that we end up with two possibilities for the total number of RAM block accesses for three of the four scenarios, yielding only one possibility for the write-hit policy, because the access count for write-hit and write-miss are the same for both access modes. The number of RAM block accesses for each scenario is used in combination with the energy cost of the different accesses supplied by the user in the configuration file to estimate the total energy usage of every scenario. The calculations are done by the model, with the datasheet values described in Section 6.1.1, to estimate the energy usage of every scenario for the different configurations. The detailed estimates are presented below, with a comment on the thinking behind each one.

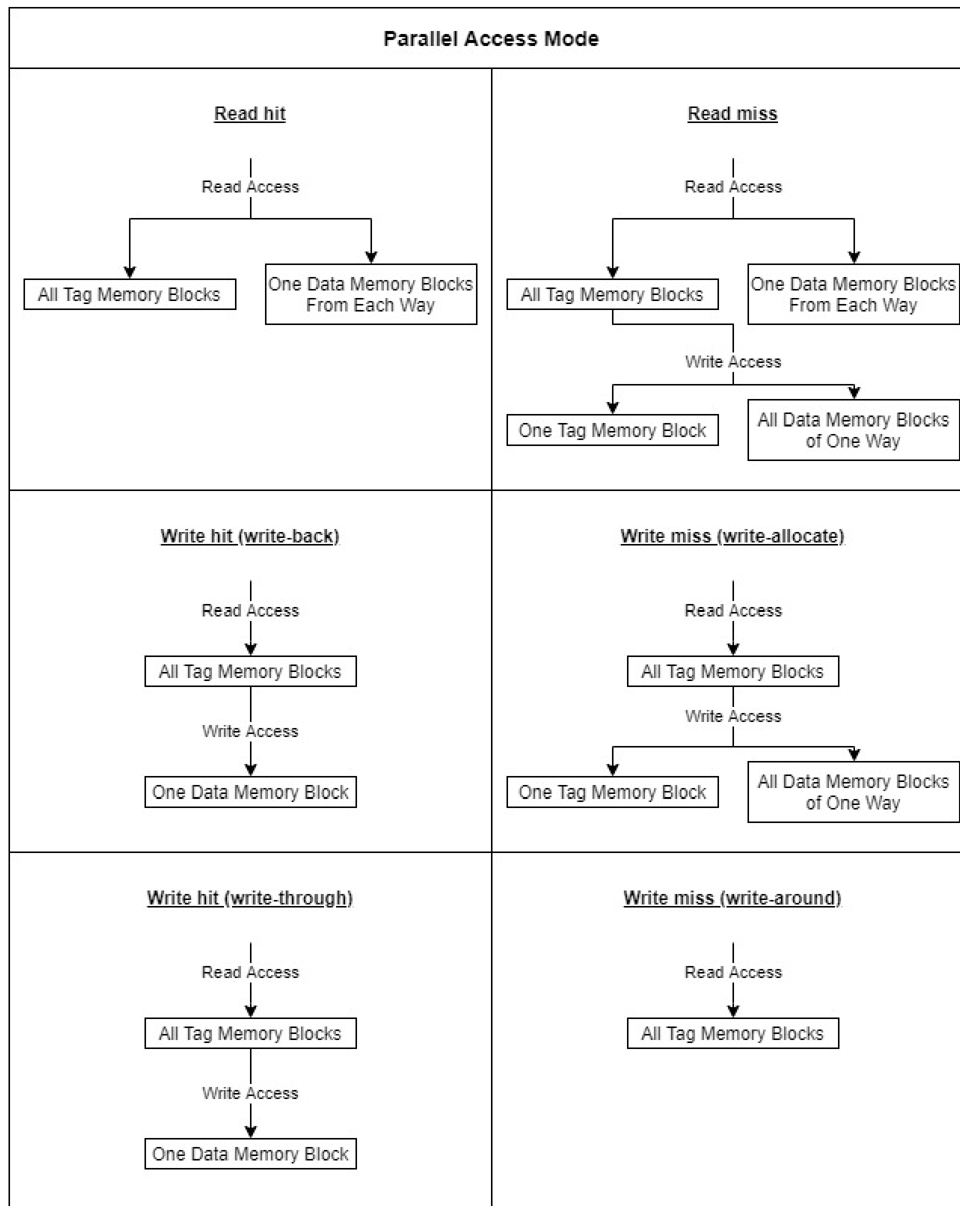


**Figure 6.3:** The number of accesses and type of accesses to the RAM block in a parallel access mode cache for all modelled scenarios.

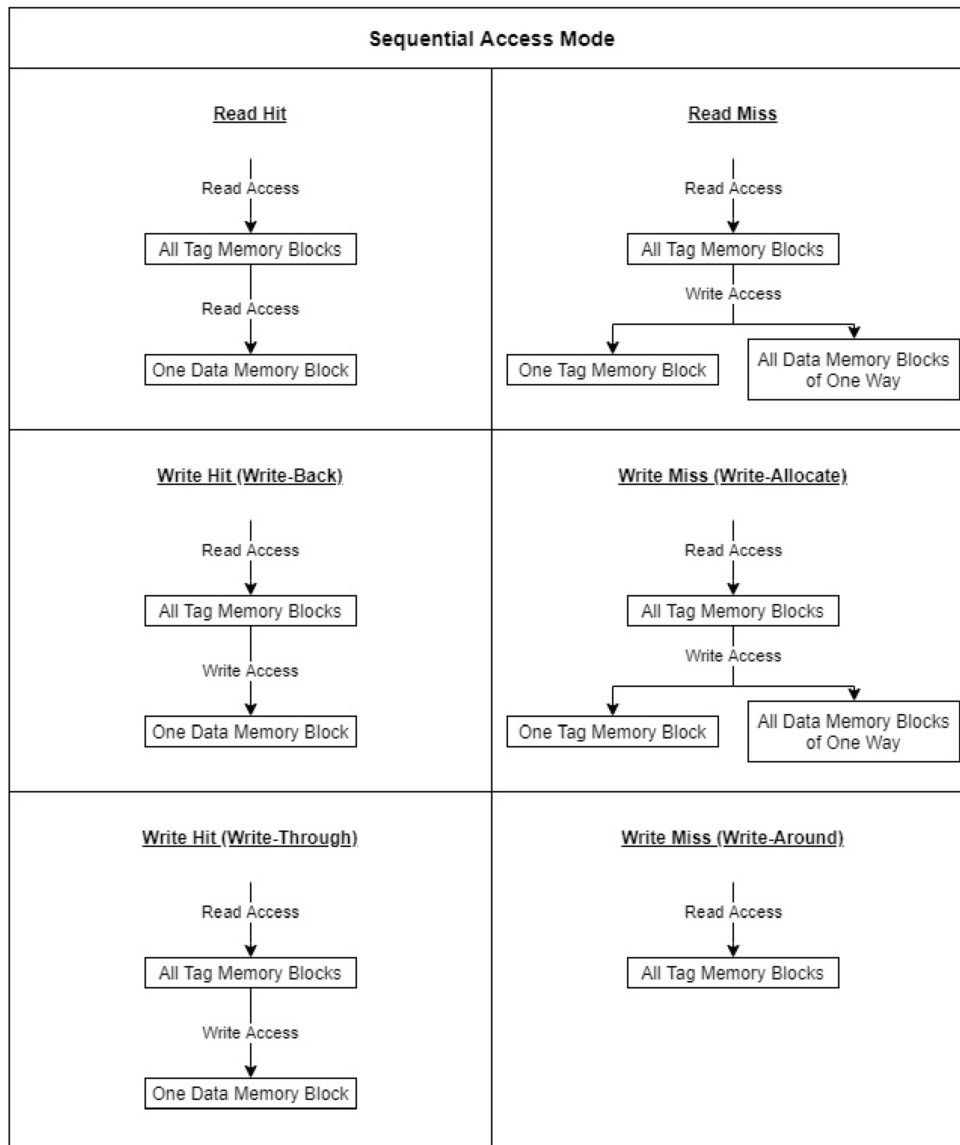


**Figure 6.4:** The number of accesses and type of accesses to RAM block in a sequential access mode cache for all modelled scenarios.

**Read-Hit**

The two different energy usage estimations for the read-hit scenario is related to whether the cache implements a sequential or parallel access mode. The estimated energy usage for the parallel access mode $E_{rh_parallel}$ is calculated as

$$E_{rh_parallel} = (E_{t_read} \cdot n_{t_blocks}) + (E_{d_read} \cdot A), \tag{6.1}$$

where $A$ is the associativity of the cache, and thus also the number of ways in a set. The read energy of accessing all the tag blocks is summed together with the read energy of accessing one data block from all the ways in the set because it is not known in which way the wanted data is due to the readouts happening in parallel. For the calculation of the estimated energy usage for a sequential access mode $E_{rh_sequential}$ we get

$$E_{rh_sequential} = (E_{t_read} \cdot n_{t_blocks}) + E_{d_read}, \tag{6.2}$$

which is the same as for the parallel access mode $E_{rh_parallel}$, except that we only need to add the read energy of accessing one data block. This is because the tag and data readouts happen after each other in time, so the placement of the wanted data is known.

**Read-Miss**

Just as with the read-hit scenario, it is the access mode that decides the calculation of the estimated energy usage of a read-miss scenario. The calculations for the parallel access mode calculation is

$$E_{rm_parallel} = (E_{t_read} \cdot n_{t_blocks}) + (E_{d_read} \cdot A) + E_{t_write} + (E_{d_write} \cdot n_{d_banks}). \tag{6.3}$$

The first two terms are equal to the parallel read-hit estimation $E_{rh_parallel}$, but in this case, the data is not present in the cache. The data will, therefore, be fetched from the next level of the memory hierarchy, and the tag will be written into one of the tag blocks, and the data will be written to as many data blocks as one cache line is spread across, defined by the number of banks $n_{d_banks}$. The calculation of the estimated read-miss energy usage for the sequential access mode $E_{rm_sequential}$ is

$$E_{rm_sequential} = (E_{t_read} \cdot n_{t_blocks}) + E_{t_write} + (E_{d_write} \cdot n_{d_banks}). \tag{6.4}$$

As we can see, it is identical to the parallel access mode calculation $E_{rm_parallel}$, except that it omits the data readout term due to the readout of the tag leading to a miss.

#### Write-Hit

Contrary to the read scenarios, the write scenarios only depend on the write policies and not the access mode. This is simply because all write operations in caches are sequential, as it is needed to know if the data block that is written to is present in the cache before writing the data. The calculation of the energy usage estimate of the write-hit scenario is the same for both the write-through and the write-back write-hit policies. This calculation is

$$E_{wh} = (E_{t_read} \cdot n_{t_blocks}) + E_{d_write}, \tag{6.5}$$

which is just the read energy of reading all the tag blocks and the write energy of writing to a single data block. The difference between them is that the memory access will be passed on to the next memory level in a combined simulation using write-through, leading to higher energy usage in that memory level.

#### Write-Miss

For the write-miss scenario, it is the write-miss policy that decides which calculation should be made for the estimated energy usage. In the case of a write-around policy, the calculation of the estimated energy usage $E_{wm_around}$ is

$$E_{wm_around} = E_{t_read} \cdot n_{t_blocks}. \tag{6.6}$$

This represents the readout energy of all the tag blocks that are needed to confirm that the data is not contained by the cache, and no more RAM blocks contained by the cache are accessed.

$$E_{wm_allocate} = (E_{t_read} \cdot n_{t_blocks}) + E_{t_write} + (E_{d_write} \cdot n_{d_banks}), \tag{6.7}$$

### 6.1.3 Estimation of Leakage Power

The estimation of leakage power done by the model is based on the datasheet values of the leakage of the memory blocks used in the configuration presented in Section 6.1.1. The values for leakage for data and tag memory $P_{d_leak}$ and $P_{t_leak}$ are used in combination with the number of data and tag memory blocks $n_{d_blocks}$ and $n_{t_blocks}$ to calculate the total leakage power of the cache $P_{total}$:

$$P_{total} = n_{d_blocks} \cdot P_{d_leak} + n_{t_blocks} \cdot P_{t_leak}. \quad (6.8)$$

### 6.1.4 Estimation of Time Usage

The time usage estimation done by the model relies heavily on the user of the model to provide an accurate prediction about how many clock cycles an access to the cache would take, $n_{penalty}$. Combining this with the clock period $T_C$, we can get a simple estimate on what the time usage penalty $t_{penalty}$ is for access:

$$t_{penalty} = n_{penalty} \cdot T_C. \quad (6.9)$$

The reason for the simplicity of the time usage estimation is because of the lack of information about the surrounding system. By omitting details about the surrounding system, the usage of the model is kept plain and straightforward. The complexity of estimating the time usage based on anything other than an estimate provided by the user could easily have undermined the point of keeping the model simple and easily configurable.

## 6.2 Verification

### 6.2.1 Methodology

The goal of the physical model is to accurately estimate the real energy consumption associated with every access to a specific cache structure, but these values are tough, if not impossible, to measure precisely. Therefore, we can focus on another goal of the model, which is to be a high-level model in a design exploration process. This means that it is to be used early in a design process and that the estimates produced should be similar to what one would get from a lower-level model coming later in the design process. The model will, therefore, be validated by using advanced power measurement tools on a RTL model of a memory hierarchy and comparing that to the results of trying to model that exact hierarchy using the physical model.

Instead of making measurements on single accesses, a benchmark suite will be used to generate a bundle of memory access sequences that can be used to get total energy usage for the RTL model. These results can be compared to the estimated total energy usage using the combination of the physical and functional models. The RTL model and the power estimation tools used to analyze it will function as a golden reference model, a concept widely used in TLM based functional verification of the RTL design process [34]. The result of the power estimation on the RTL model are the results that we are seeking to generate with the much simpler Python models, and thus it can be considered a golden reference model in this case.

**Table 6.2:** The custom benchmark suite used in the validation of the physical model.

| Benchmark Name | Description |
|---|---|
| TeleBench/autcor00 | Auto-correlation computation of three different datasets – pulse, sine, and speech. |
| TeleBench/conven00 | Performs an algorithm often used to improve the performance of digital radio, mobile phones, satellite links, and Bluetooth implementations on three different datasets. |
| TeleBench/fbital00 | Spreading of three different streams of data over a series of buffers, modulation and ”transmitting” of these on a telephone line. |
| TeleBench/fft00 | Performs FFT and iFFT on three different datasets. |
| TeleBench/viterb00 | Recovery of output data packets from an encoded input data packets of three different datasets in an embedded IS-136 channel coding application. |
| CoreMark | Performs list processing, matrix manipulation, determines if an input stream contains valid numbers, and CRC. |

### RTL Model and Benchmark Suite

The RTL model that will be used as a golden reference model is a similar cache to the instruction/data cache used in Nordic Semiconductor's nRF5340 SoC [35]. It is an 8KB 2-way associative cache with a block size of 16 bytes and the LRU replacement policy implemented, and here it is used as an instruction cache. The model is written in the Hardware Description Language SystemVerilog and is connected to an ARM Cortex-M33 processor [36], which will run a custom benchmark suite consisting of a collection of different applications, to get a variety of different memory access patterns. The custom benchmark suite is listed in Table 6.2 with descriptions of what each application does. The applications are taken from EEMBC's TeleBench benchmark suite [37] and CoreMark benchmark [38].

The CoreMark benchmark is chosen because it is designed to test the performance of MCUs and embedded CPUs, and therefore the memory access pattern of the application should be similar to applications regularly used in embedded systems. The TeleBench benchmarks suite contains applications that appear in many telecommunication systems, which is what many embedded systems are being used for today. The combinations of these will cover different access patterns to the cache memory, which are very relevant to test on a cache memory used in an embedded system.

The applications have been altered only to contain the initialization and the benchmark computation, which is timed, omitting all result checking. This is because it is only the timed portion that contains the memory access pattern we are looking to test the models with. A start signal before and a stop signal after the timed portion has also been added to easier know when the

benchmark is running in the simulation of the RTL model. These signals make the CPU set a specific GPIO pin to high and low, respectively.

Regarding the TeleBench benchmark suite, it originally contains a test harness to control each application individually. To make the process of compiling easier, each application has been extracted from this test harness, and every dependency to it has been removed. This way, each application can more easily be compiled and run individually. Each application also contains different datasets that are all supposed to be used by the benchmark portion, but which are incorporated into the executable file. This means that the application has to be compiled once for each of the datasets.

### Tools

The process of gathering the results explained above requires a number of different tools. The compilation of the benchmark programs are done with the ARM specific variation of the GCC compiler [39]. This generates the binary files, which are used in the SystemVerilog test bench containing the RTL cache model and the CPU. The simulations with this test bench are done using QuestaSim from Mentor [40]. Verdi [41] from Synopsys is used to dump FSDB files from the simulations. The last step is to gather the power estimation results from the simulation using SpyGlass Power from Synopsys [42]. To do this, we also have to link it to the specific technology libraries we want to use. The tools and the version of each that was used in the process of gathering the power estimation results of the RTL model are listed in Table 6.3.

**Table 6.3:** A list of every tool used in the process of gathering the power estimation results of the RTL model.

| Tool | Version |
|---|---:|
| ARM GCC Compiler | et_5-2016-q3 |
| QuestaSim | 10.c_7 |
| Verdi | 2018.09 |
| SpyGlass Power | 2019.06-sp2 |

### Memory Access Trace Generation

The functional model needs a memory access trace to run to be able to simulate the benchmarks. To make memory access traces that can represent the benchmarks being run on the RTL model, the SystemVerilog test bench is modified such that it will write every memory access made from the CPU during the timed portion of a benchmark to a file. The memory accesses written to the file are on the form to functional model uses, described in Chapter 5. This way, the stimuli to the RTL model of the cache and the MPCache model will principally be the same.

### 6.2.2 Results and Discussion

This section presents the results of the verification methodology described above. The results are represented in a relative fashion to compare the performance of the MPCache model with the results of the RTL simulation.

#### Configuration of the Model

The MPCache model has been configured to resemble the structure of the RTL model cache to the best of its ability. The general structure is, to a large extent, the same, but the RTL model still has some features implemented to improve its performance, which is not supported by the MPCache models. This is a potential error source that can lead to discrepancies between the results of the two models.

#### Estimates of Execution Time

Figure 6.5 shows the estimated execution time by MPCache for each benchmark as a percentage of deviation compared to the corresponding estimate by the RTL simulation.

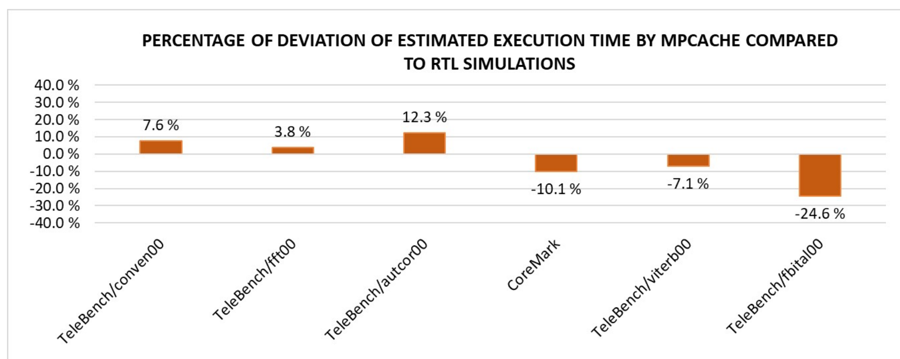


**Figure 6.5:** The percentage of deviation of the execution time estimated for each benchmark by MPCache compared to the corresponding estimate by the RTL simulation.

We can see that the accuracy of the estimates made by MPCache varies for the different benchmarks, by -24.6% to 12.3% at the extremities, but are reasonably close for most of them. The estimation of the execution time done by MPCache is done based only on the memory accesses and disregards the time used by the CPU between accesses. This means that the estimates will only be accurate if there are constantly active memory accesses during the execution time of the benchmark. The estimated execution time of *TeleBench/fbital00* is the one with the largest deviation, which could indicate that the CPU uses a substantial amount of time with no pending memory accesses. This problem could be fixed by altering the trace file format to include a form of timestamping or time between accesses. It is not directly tied to the physical cache model, but rather the way MPCache combines the results of the functional and physical models.

We can also see that the MPCache estimates are both larger and smaller compared to those

made by the RTL simulation. Based on the method of estimating the execution time by MP-Cache discussed above, we would assume that it would never exceed the estimate made by the RTL simulations. This overestimation could be caused by the way the physical model has one constant access penalty associated with the cache. In contrast, a real cache might have different access penalties based on different features implemented to improve performance. An example of this was mentioned in Section 6.1.2, where a cache that implements way-prediction will have a conditional access pattern based on whether the prediction is correct or not.

Figure 6.6 presents the execution time estimated by MPCache and the RTL simulation for each of the benchmarks. The results are represented as the relative value of the estimate compared to the largest estimate. In this case, the largest estimate is the execution time of *TeleBench/fbital00* made by the RTL simulation.

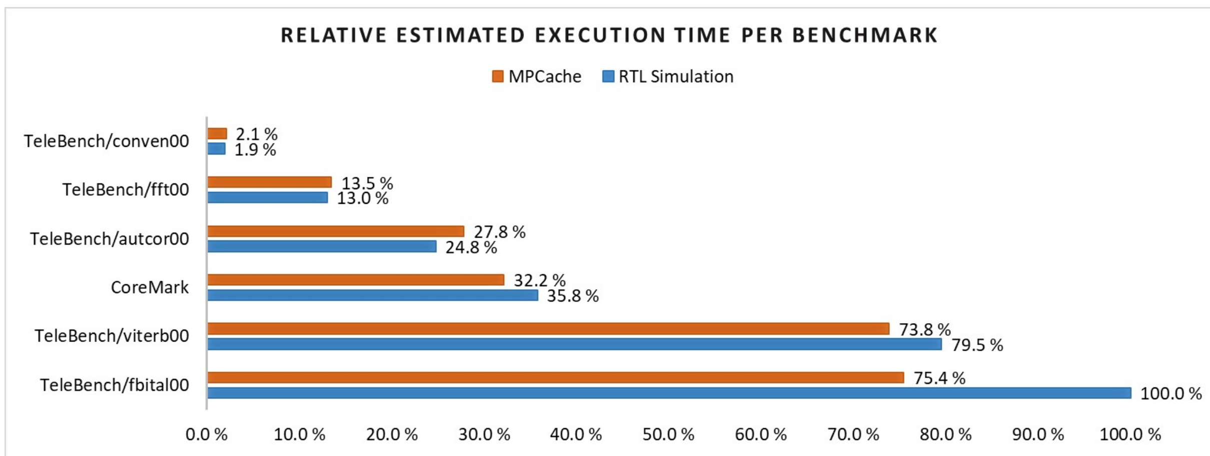


**Figure 6.6:** The estimated total execution time of each benchmark by both the RTL simulations and the MPCache framework relative to the benchmark with the largest estimated total execution time.

This representation shows that even though the estimates produced by MPCache are not accurate, the ranking of the benchmark execution times by MPCache is equal to that of the RTL simulation. This statement is slightly misleading since the difference between the individual benchmarks can be very wrong compared to the RTL simulations, as exemplified by *TeleBench/vierb00* and *TeleBench/fbital00*. Therefore the ranking could easily have been wrong, making it clear that changes, like the ones mentioned earlier, to improve the estimates of execution time is essential to produce results with high fidelity for experimental designs.

**Estimates of Energy Usage**

Figure 6.7 shows the estimated total energy usage by MPCache for each benchmark as a percentage of deviation from the corresponding estimate by the RTL simulation.

It is evident that the estimates produced by MPCache are very inaccurate compared to the RTL simulations, deviating by close to -55% for all except for *Telebench/autcor00* benchmark which

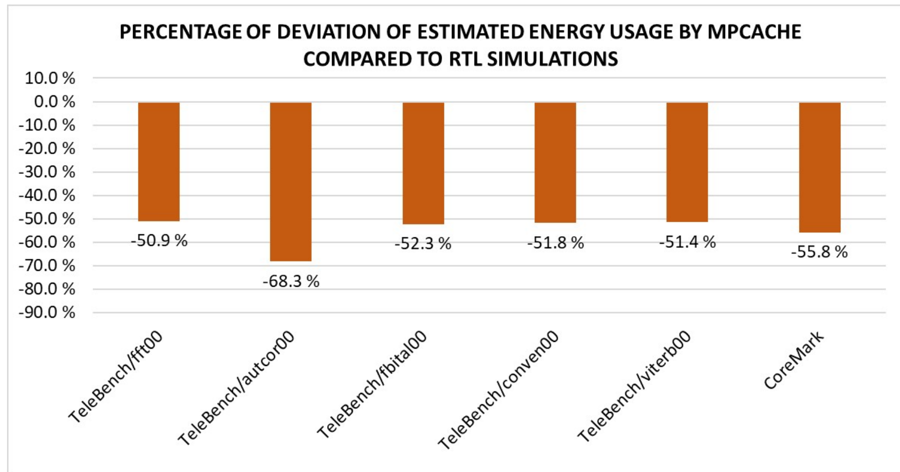


**Figure 6.7:** The percentage of deviation of the energy usage estimated for each benchmark by MPCache compared to the corresponding estimate by the RTL simulation.

deviate by 68.3%. One thing to notice is that even though the results are not accurate, they seem to have some consistency between the different benchmarks. As mentioned in Section 6.1, the physical cache model only considers the energy usage of the tag and data memory of the cache, which obviously limits the precision of the estimates. However, the results can still be used to classify cache designs as long as the inaccuracy is consistent.

Figure 6.8 presents the estimated energy usage for each of the benchmarks by MPCache and the RTL simulations. The results are represented as the relative value compared to the largest estimated energy usage. In this case, the largest estimate is the RTL simulations estimate of the energy usage of *CoreMark*.

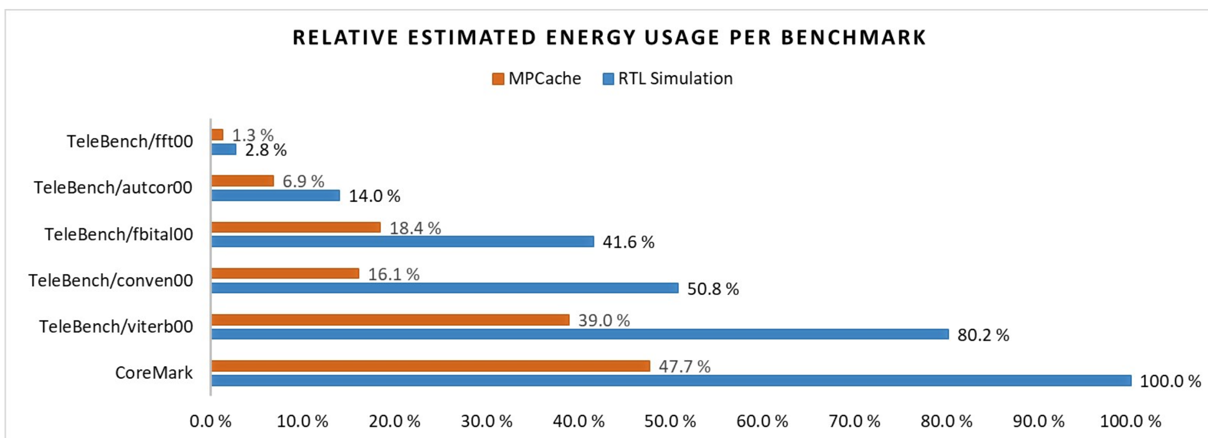


**Figure 6.8:** The estimated total energy usage of each benchmark for both the RTL simulations and MPCache relative to the one with the largest estimated total energy usage.

Here, we can see that the estimates of energy usage made by MPCache almost ranked in the same order as for the RTL simulations. The only difference is that *Telebench/conven00* and *Telebench/fbital00* have switched places in the ranking, but are still placed correctly compared

to the rest of the benchmarks. This flaw is due to the discrepancy in energy consumption by parts of the cache not represented by the physical model of MPCache between the two benchmarks. It could be an indication that the physical model needs to be elaborated to support the modeling of more parts of the cache than the memory arrays to improve the precision of the results. Yet, it could also be caused by an overestimation of the energy usage of these parts by the RTL simulation. The estimation is based on technology libraries, which are different for the memory arrays and the remainder of the cache. A disparity between these libraries could cause an unbalance in the energy consumption estimates of the different parts of the cache.

### Energy Usage of the Memory Blocks

The energy usage estimates produced by the RTL simulation provide a distribution of how much each section of the cache contributes to the total energy usage. By comparing the contribution of energy usage by the memory blocks contained by the cache to the estimates of total energy usage made by MPCache, we can get information about the assumptions made by the model. Both the tag and data memory blocks are included. As mentioned in Section 6.1, MPCache only makes energy estimates based on the memory blocks contained by the cache. Figure 6.9 presents the energy usage estimates made by MPCache as relative values compared to the energy estimates by the RTL simulation on the memory blocks of the cache.

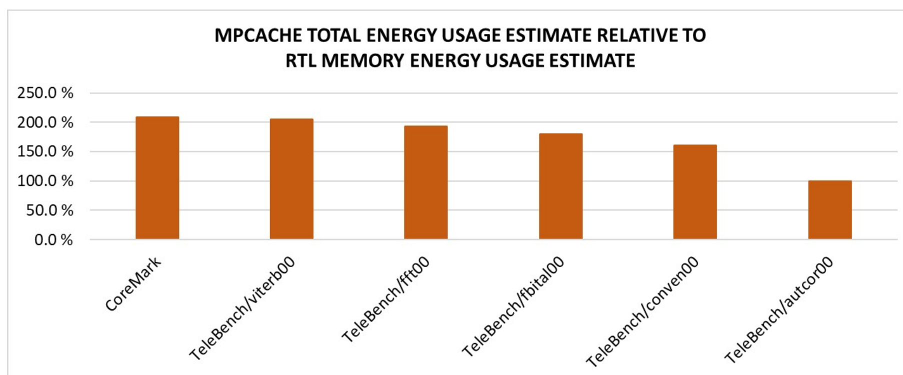


**Figure 6.9:** The estimated total energy usage by MPCache relative to the estimated energy usage of the memory blocks by the RTL simulations.

As we can see, MPCache overestimates the energy usage of the memory by between 70% and 100% for most of the benchmarks, except *TeleBench/autcor00*, which is almost correct. This overestimation implies that the accuracy of the estimated total energy consumption would have been even worse if the model was closer in its estimates of the energy consumption of the memory arrays. This increases the significance of the discussion above about expanding the models of energy usage.

The miscalculation of the energy usage of the memory could likely be a result of the aforementioned difference between the models. If the RTL model contains performance-enhancing features not possible to model using MPCache, it would undoubtedly lead to such a miscalcu-

lation and is a sign of missing design options for the physical model.

### Reflections on the Comparison

The results by MPCache proved to be rather accurate for the estimation of execution time compared to the RTL simulation. The estimates of total energy usage were less accurate when compared to the RTL simulations, but had close to the same ranking of different benchmarks. The results are therefore acceptable for this cache structure, but to correctly categorize this structure compared to other structures, the same tendencies must be demonstrated for them as well. Hence, there is a need to compare the framework to RTL simulations of a variety of cache structures. More features also need to be considered to develop the physical model further, such that it can better compare to an RTL implementation of a cache.

# Chapter 7

# Performance Compared to CACTI

This Chapter presents a comparison of the results produced by the physical model of MPCache, described in Chapter 6, and the well established physical cache model, CACTI, described in Section 3.2. The differences and similarities between the models are discussed before the methodology of comparing the results of the two models is presented. Lastly, the results of the comparison will be shown with a discussion about the findings.

## 7.1 Comparison of the Models

As described in Section 3.2, CACTI is a physical cache model that uses the analytical modeling technique to estimate energy, area, and timing usage of specific cache structures. Unlike the physical model of MPCache described in Chapter 6, CACTI models the cache structures by utilizing assumptions about a generic memory array and combining them with physical parameters produced by MASTAR, described in Section 3.2. The physical model of MPCache applies datasheets of memory arrays to model a specific cache, rather than modeling the cache using generic memory arrays, like CACTI. Another difference is that CACTI also considers the wires interconnecting the memory arrays and comparators used to check the tags in its estimates.

Due to the analytical approach of CACTI, the configuration of the model is very detailed to make sure that a cache with the correct memory array is modeled correctly. CACTI also allows for setting design goals to optimize the cache structure for, e.g., dynamic energy usage, leakage power, or access time. This is where MPCache uses datasheet values of specific memory arrays to more naturally know that the correct memory array is being modeled.

The results produced by the two models have a significant difference in that the physical results of MPCache are ready to be combined those of the functional model. The CACTI results, on the other hand, are general access parameters of the memory section of a cache. The physical model

of MPCache describes the dynamic energy cost of the *read-hit*, *read-miss*, *write-hit*, and *write-miss* scenarios. At the same time, CACTI does this only for general *read* and *write* accesses, closer to what is typical for a regular memory array. For the timing parameters, CACTI produces *access time* and *cycle time* estimates, which are the time it takes the cache to provide the data from an access and the time it takes for the cache to be ready for a new access, respectively. For SRAM, the cycle time is equal to the access time [43]. The corresponding result produced by the physical model of MPCache is the *access penalty* of the cache. While the access penalty is the time consumption of the cache related to clock cycles, the access time and cycle time are precise numbers that must be linked to a clock cycle to know how much time an access to the cache consumes.

Both models make estimations of the leakage power of the cache, which are closely related, as described in Section **??**. As mentioned above, CACTI estimates the area usage of the cache. The physical model of MPCache does not support this feature.

## 7.2 Comparison of the Performance

This Section will present the methodology of comparing MPCache and CACTI and discuss the results of this comparison.

### 7.2.1 Methodology

The MPCache framework and the CACTI model has differences and similarities, as can be apparent from Section 7.1. Therefore, a procedure for comparing the two must be chosen that is fair to both. This comparison will focus on the physical model of MPCache since this is the part of the framework that is comparable to CACTI. The usage of the models, in combination with a functional model, is also an interesting point of discussion. Because of the differences in results produced by the models, this comparison will need to be tailored for this purpose. The functional model of MPCache will be used for the combined simulations to avoid introducing additional components. As mentioned in Section 7.1, there are fundamental differences between the timing results of the two models. Therefore, the access penalty of MPCache will be used to determine the execution time estimate. The comparisons will thus only assess dynamic energy usage, leakage power, and total energy usage.

A selection of cache structures will be used to display the differences in the models. This selection is listed in Table 7.1.

These cache structures vary the four design options: (1) cache size, (2) associativity, (3) access mode, and (4) block size. An S or a P denotes the access mode, signifying if it is accessed sequentially or parallelly, respectively. The structures are grouped by their access mode and

**Table 7.1:** The selection of cache structures used to compare the results of the models.

| Cache Name | Cache Size | Associativity | Access Mode | Block Size |
|---|---|---|---|---|
| 8KB_1A_S_16B | 8KB | 1 | S | 16B |
| 8KB_2A_S_16B | 8KB | 2 | S | 16B |
| 16KB_2A_S_8B | 16KB | 2 | S | 8B |
| 16KB_2A_S_16B | 16KB | 2 | S | 16B |
| 16KB_2A_S_32B | 16KB | 2 | S | 32B |
| 32KB_2A_S_16B | 32KB | 2 | S | 16B |
| 8KB_1A_P__16B | 8KB | 1 | P | 16B |
| 8KB_2A_P_16B | 8KB | 2 | P | 16B |
| 16KB_2A_P_16B | 16KB | 2 | P | 16B |
| 32KB_2A_P_16B | 32KB | 2 | P | 16B |

are sorted within the group by the cache size, associativity, then block size. This approach is made to enhance the differences between the structures in the results, and the same goes for the selection of structures. The naming of the caches is done to represent the structure of the cache. The different structures will be modeled using the methodology of the MPCache framework. Then CACTI will be used to create the same configurations. As mentioned earlier in this Chapter, CACTI uses a very detailed configuration and design goals to make estimates for different cache structures. Therefore, it can be difficult for someone who is not an expert to know positively that the modeling has been done correctly. This uncertainty is a possible error source for the results.

The remaining design option that could affect the physical model of MPCache is the write-miss policy, and it will be fixed at write-around. This choice has been made to restrict the number of cache structures being simulated.

The two models will be compared using three general methods. The methods are to compare the models by studying the:

1. plain results of the models.
2. results used in combined simulations with a functional model.
3. simulation results in relation to the RTL simulation from Chapter 6.

The two first methods will utilize all the cache structures listed in Table 7.1. The comparison with the RTL simulation will only use the cache structure corresponding to the RTL model. The two last methods will use the benchmark suite described in Section 6.2.1 for the simulations. One difference between them is that for the comparison with the RTL simulations, the cache will continue to be used as an instruction cache. In contrast, for the other simulations, it will be a unified cache. This is to increase the number of accesses going through the cache.

For the second method listed, the CACTI results will need to be adapted to the functional

model of MPCache. More precisely, the transformation from read-hits, read-misses, write-hits, and write-misses to reads and writes must be facilitated. Table 7.2 presents how the CACTI estimates will be used in the combined simulations of MPCache.

**Table 7.2:** The usage of the CACTI estimates in the MPCache combined simulations.

| | **Reads** | **Writes** |
|---|---|---|
| **Read-Hit** | 1 | 0 |
| **Read-Miss** | 1 | Block Size / Bus Width |
| **Write-Hit** | 0 | 1 |
| **Write-Miss** | 1 | 0 |

A read-hit and a write-hit correspond to one read and one write, respectively. A read-miss result in one read and as many writes as is necessary to fill in the new cache line with the given bus width, e.g., with a block size of 16 bytes and bus width of 64 bits this would mean two writes. For a write-hit, only one read is necessary. This is because of the write-around write-miss policy. If write-allocate had been used, it would be equal to the read-miss shown.

### 7.2.2 Results and Discussion

This section presents the results of the comparison of MPCache and CACTI using the methodology described above. The results are grouped into the different methods used to produce them, as described in Section 7.2.1.

#### Plain Results

The plain results are the results produced by the physical model of MPCache and CACTI on their own for the different cache structures. The results in this segment are represented relatively compared to the largest value in each respective set. This means that the estimates produced by MPCache are relative to the largest value produced by MPCache. It is done this way to see the trends within one model, but for reference, the largest dynamic energy estimate by CACTI is 5.7% larger than the largest of dynamic energy usage made by MPCache. The largest estimate for leakage power made by MPCache is 873% larger than the largest made by CACTI for leakage power. These relations will be further discussed using the second and third comparison method.

Figure 7.1 presents the estimates for dynamic energy usage of the different access scenarios modeled in MPCache for the selection of caches.

We can see that the energy usage estimated for write-hit and write-miss are almost the same for all the different cache structures. This stationary trend makes sense when taking into consideration the modeling principle for these scenarios. A write-hit will only write to one memory block, and if the cache is built up of mostly the same blocks, the energy usage of this scenario will be the same for all structures. The slight variation will be because of the increasing tag

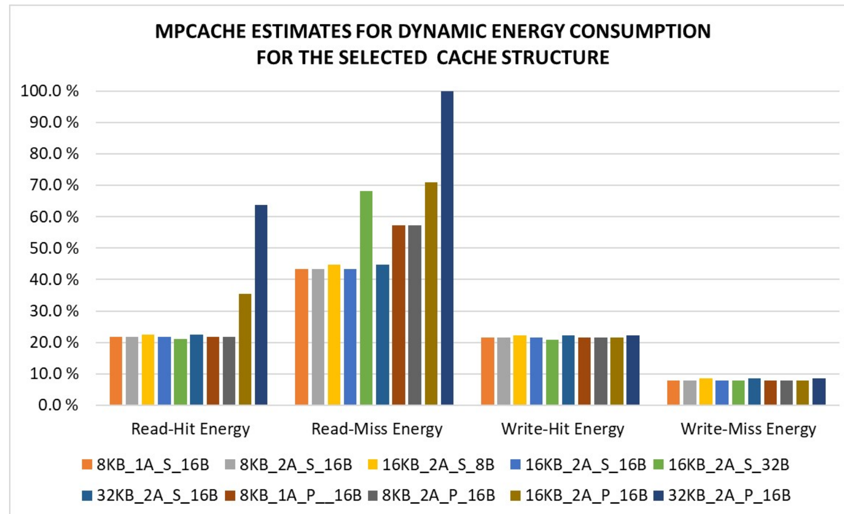


**Figure 7.1:** The estimates for dynamic energy usage by the selection of cache structures by using MP-Cache.

memory. As a result of choosing the write-around policy, the write-miss energy usage is equal for most structures, because this only requires one read of the tag memory. If the write-allocate policy been chosen, we would look for differences between the various block sizes.

These trends also seem to be relevant for the energy usage of the read-hit and read-miss scenarios. Here, the energy usage is equal for many of the structures, for mainly the same reasons. For read-hit, the parallelly accessed caches are the exception. These are modeled to account for multiple read accesses to memory blocks, which increase energy usage. However, the *8KB_2A_P_16B* structure has the same energy usage of a read-hit as most of the others. This fact is peculiar, as it should read twice the number of data blocks as those with comparable energy usage, and could indicate an error in the implementation. For the energy usage of read-misses, we see the same pattern again, without the possible error for the *8KB_2A_P_16B* structure. The increased block size also makes a definite impact on the energy, but the reduced block size does not. The energy usage is closely coupled with how the memory blocks of the cache are structured, i.e., this difference could be a result of bad structural decisions in the modeling.

Figure 7.2 presents the estimates for dynamic energy usage of the different access scenarios modeled in CACTI for the selection of caches.

Here, we can see more of the trends that are natural to see. The energy usage of a read access is increasing with increasing cache size. The energy usage also increases more rapidly for the parallel access mode, which is natural to assume, as more of the data memory accessed. We can also see that the different block sizes have similar energy usage for both read and write accesses when the other parameters are constant. Besides the general trends, we see a clear difference between the energy usage of write accesses compared to the energy usage of write-

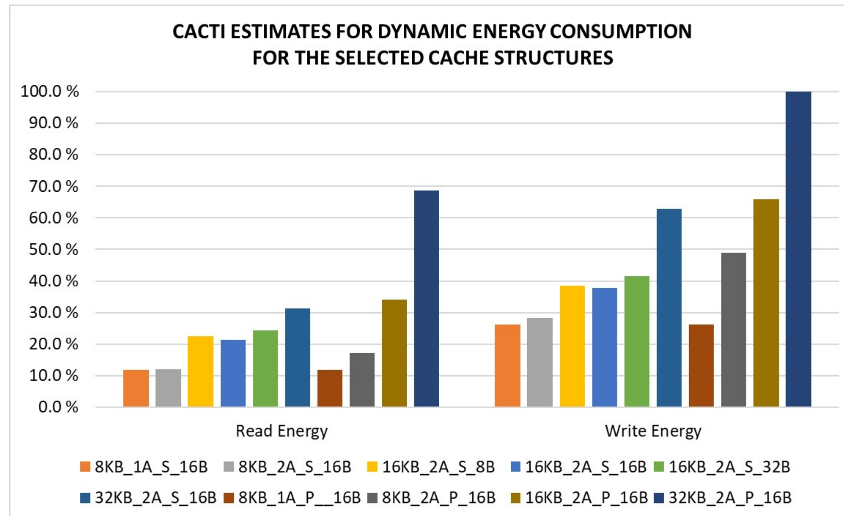


**Figure 7.2:** The estimates for dynamic energy usage by the selection of cache structures by using CACTI.

hits of MPCache. These two inequalities between the models clarify that the physical model of MPCache is missing a vital part to differentiate between cache structures and see real impacts of write accesses. This shortcoming is most likely originating in the missing modeling of wires interconnecting the memory blocks contained by the cache. Neglecting the effects of these on the energy usage of caches seem like a definite weakness, especially for sequentially accessed caches and for the energy usage of write-hits.

Figure 7.1 and 7.4 presents the estimates for leakage power for the selection of caches when modeled using MPCache and CACTI, respectfully.

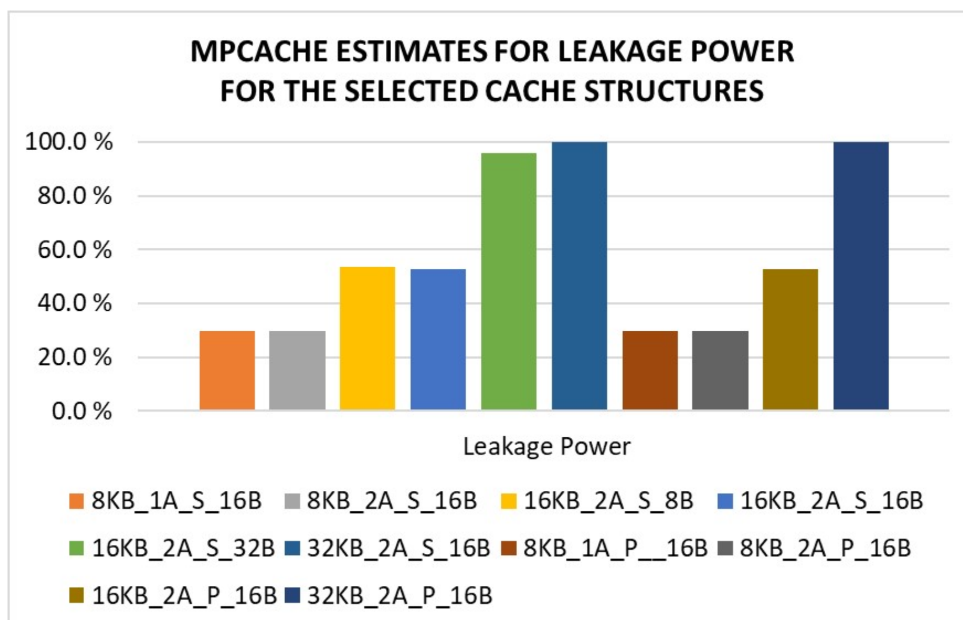


**Figure 7.3:** The estimates for leakage power of the selection of cache structures by using MPCache.

We can see that the two models have a very similar trend for the estimated leakage of the dif-

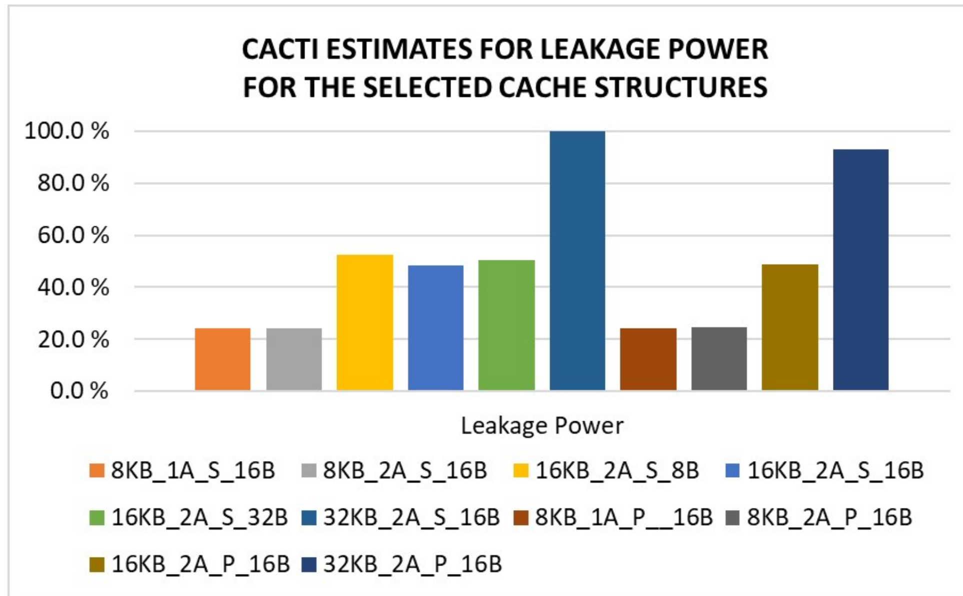


**Figure 7.4:** The estimates for leakage power of the selection of cache structures by using CACTI.

ferent cache structures. One exception is the *16KB_2A_S_32B* structure, which has almost twice the leakage compared to the comparable cache structure. This could come from the modeling uncertainties mentioned earlier, or it could potentially be an error in the model implementation.

However, the estimates for leakage power for the two models are vastly different relative to each other. As mentioned, the largest estimate of MPCache is 873% bigger than the largest estimate by CACTI. This disparity indicates that either both models are very wrong, or that one of the models make better estimates on this point. This will be discussed in the comparison with the RTL simulation.

### Combined Simulation Results

The combined simulation results show the estimated total energy usage by combining the results of CACTI and the physical model of MPCache with the results of the functional model of MPCache. These simulations are done for the selected cache structures described in Section 7.2.1 using the benchmarks described in Section 6.2.1. The results in this segment are represented relative to the largest value in both sets. This means that the estimates produced by MPCache are relative to the largest value produced either using MPCache or CACTI. The reason for this is to emphasize both the trends within one model and how the models fare compared to each other.

Figure 7.5 presents the estimated total energy usage done using the physical estimates of the MPCache framework.

From these combined simulation results, we can see indications of the aforementioned problems

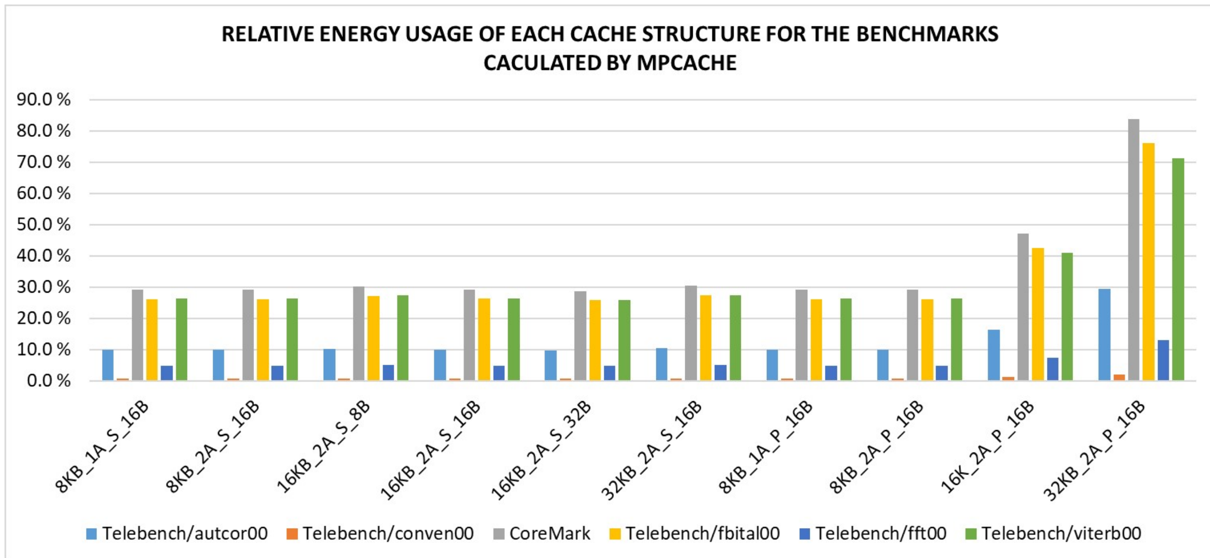


**Figure 7.5:** The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using MPCache.

with the energy usage modeled by the physical model of MPCache demonstrated. It is hard to see any real variation for most of the structures in any of the simulations. The exceptions are the same as for the plain physical results; the two larger parallelly accessed structures - *16KB_2A_P_16B* and *32KB_2A_P_16B*.

Figure 7.6 presents the estimated total energy usage done using the physical estimates of CACTI.

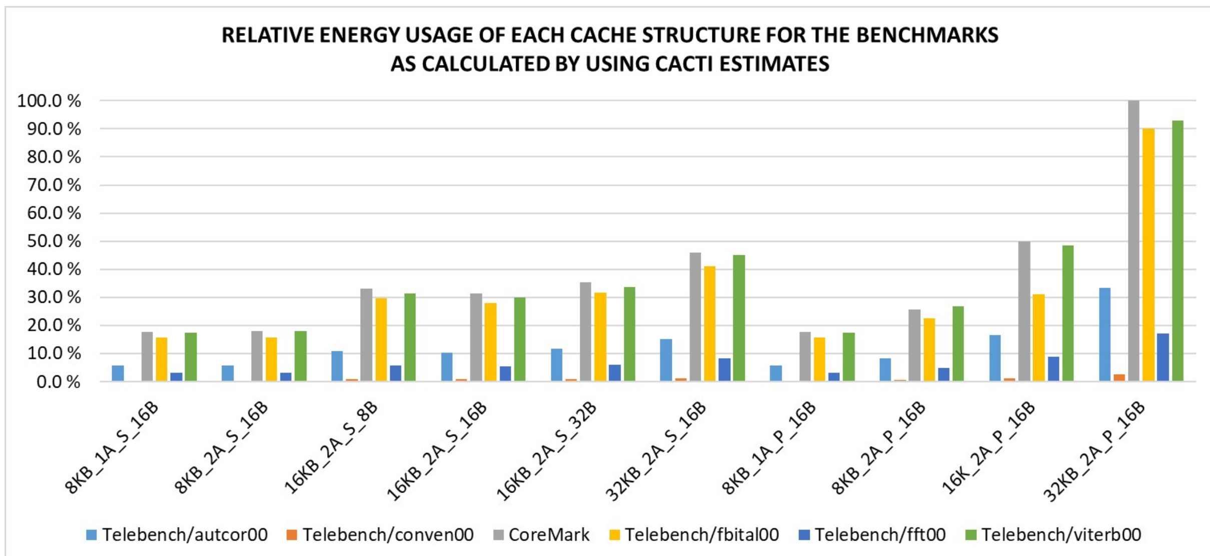


**Figure 7.6:** The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using CACTI.

The results of the combined simulations with the physical results of CACTI are in some ways similar to the results shown for MPCache in Figure 7.5. However, comparable to the plain physical results of CACTI, we can see distinct differences between the cache structures, which

are natural to see for the variation of cache structures simulated here. This contrast emphasizes the need for changes to the physical model of MPCache, more specifically with additions of wire modeling. This change is most crucial for the sequentially accessed caches, but we can also see hints that the disparity of the estimate for energy usage of parallelly accessed caches between the models increases for larger caches.

### Results Compared to RTL Simulation

Here, the usage of CACTI to estimate the total energy usage is introduced to the comparison with the RTL simulation described in Section 6.2. The leakage power produced by CACTI for the relevant cache structure is also compared to the results of MPCache and the RTL simulations. The results in this segment are represented relative to the largest value produced by any of the models.

Figure 7.7 presents the estimates for total energy consumption by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations.

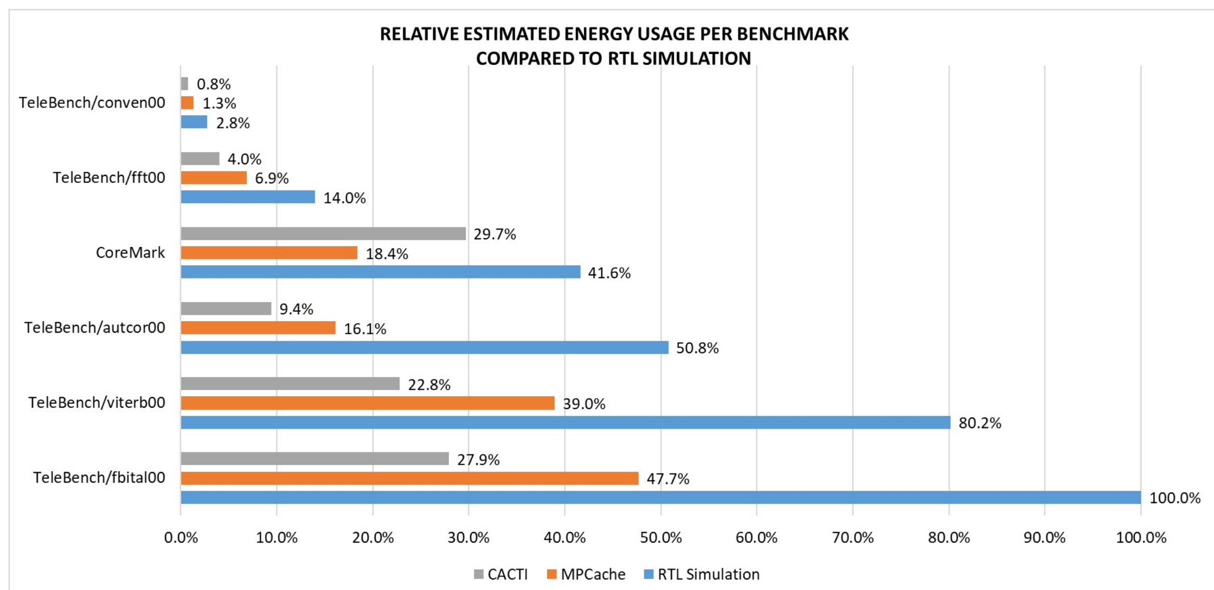


**Figure 7.7:** The estimates for total energy consumption by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations.

We can see that the estimate of the total energy usage by MPCache is closer to the estimate made by the RTL simulations than the one using CACTI for all the benchmarks, except for *CoreMark*. However, the *CoreMark* estimate is the largest estimate made by the combined simulation with CACTI, which is not the case compared to the RTL simulation. This error makes the ranking of the energy usage of the different benchmarks disrupted compared to the ranking done by the RTL simulation, and overall the estimates made by MPCache seem moderately more useful in this comparison. It is not clear what leads to this error, but the most apparent difference between the physical model of MPCache and CACTI was the energy usage of writes. It could mean that

CACTI overestimates the energy usage of writes accesses.

Overall we see some evidence that MPCache can be as good or better at simulating caches of this size than with the usage of CACTI. When the cache is of this size (8KB), fewer memory blocks are used, which seem to mitigate the problem of the energy estimate with an increasing amount of memory blocks mentioned earlier.

Figure 7.8 the estimates for leakage power by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations.

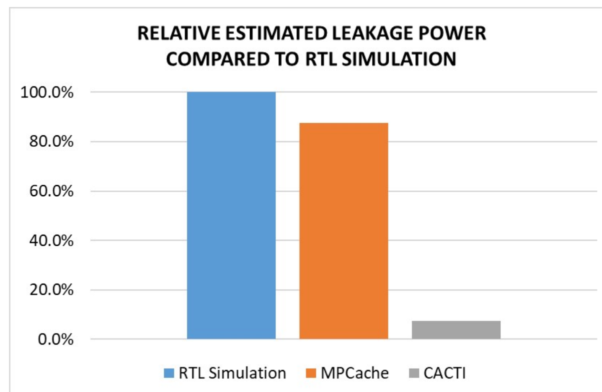


**Figure 7.8:** The estimates for leakage power by the selection of cache structures for the different benchmarks by using MPCache, CACTI, and RTL simulations.

These results show that the datasheet values utilizes by MPCache are much closer to the results of leakage power one would get for an RTL simulation than the generic modeling of CACTI. It should be mentioned that CACTI provides additional transistor models than the ones used in these simulations, but that the usage of these would dwarf the leakage power estimates by MPCache and the RTL simulation, and thus becoming a very significant factor in the estimate for total energy usage.

## 7.3 Reflections on the Comparison

The proposed usage of the two physical models is slightly different, where CACTI focuses on the direct usage of its results to make design decisions. In contrast, MPCache is made to be combined with a functional simulation in addition to providing the plain results. MPCache is therefore considered more suitable for a combined simulation since the framework lays the groundwork for this. The configuration of the MPCache is also more straightforward, as long as datasheets of possible memory blocks are available.

However, it was apparent that the results of MPCache did not show the natural trends of energy usage of larger caches. The estimation of energy usage solely based on datasheet values, therefore, seems unreliable to use for the categorization of different caches. CACTI estimates are

based on the memory blocks of the cache, the wires used to interconnect them, and the decoding logic and comparators used for the signals. The physical model of MPCache should use more of the same techniques to better model caches of larger sizes, especially considering adding the wire modeling, since this will include a definite contribution to the energy usage of a larger cache, because of the increased amount of memory blocks.

We saw from the comparisons that with the RTL simulation that MPCache performed slightly better for the estimate of total energy consumption than the combined simulation using CACTI. The leakage estimates were significantly better than the ones produced by CACTI. These accomplishments show the potential of MPCache, but it is evident that the deficiencies in the physical modeling mentioned above must be addressed for the framework to be reliable in a design exploration phase.

# Chapter 8

# Conclusion

This thesis aimed to develop a novel high-level cache modeling framework for simulations targeted at cache memories of embedded computers. The focus of the framework was to make fast and accurate estimates for the performance of cache structures to guide to be able to guide a design exploration phase. Based on an analysis of the results of simulating caches using this framework, MPCache, it can be concluded that the framework has an easy modeling strategy, but that the results of the combined simulations have a varying degree of accuracy. The results indicate that cache structures consisting of fewer memory blocks, and with a parallel access mode are modeled more correctly than those with more memory blocks and sequential access mode.

In Chapter 5, the implementation of a functional cache model in the *Python 3* programming language was described. This model is highly flexible and can be configured to represent any viable cache size, block size, and associativity, as well as supporting a selection of replacement, write-hit, and write-miss policies. It is also developed to be used in simulations of memory hierarchies by allowing the connection of multiple models to each other by specifying the next memory levels of a model, and by defining the accesses a model should handle by assigning an address space to it. A main memory model was constructed to be used as the endpoint of such hierarchies.

Verification of the functional cache model was conducted to confirm that it worked properly. This verification was done by configuring a selection of cache structures with a representative variety in the parameters and constructing four trace files that would highlight the strengths and weaknesses of the different configurations. The expected results of a simulation with the selected trace for each configuration was derived analytically, and compared to a real simulation. It was concluded that it is reasonable to assume that the functionality of the module works correctly and that the model is scalable for the adjustable parameters.

In Chapter 6, the development of a physical cache model was described. This model was implemented in the *Python 3* programming language to be coherent with the functional cache model. The model can represent direct-mapped and set-associative caches of any viable size and with any block size. It estimates the dynamic energy usage of the read-hit, read-miss, write-hit, and write-miss scenarios of accesses to a cache. This estimation is done based on datasheet values of energy usage of the memory blocks contained by the cache. These values are used to calculate the estimates analytically by assuming how many accesses are done to each memory block by taking in to account the access mode, the write-hit policy, and the write-miss policy. The model also makes simple estimates of the leakage power and access time penalty of a cache. The leakage estimate is made by summing the leakage of all the memory blocks contained by the cache. The access time penalty is mainly estimated by the user, by providing how many clock cycles an access to the cache is assumed to take and the clock period. This penalty is used to estimate the execution time during combined simulations.

The physical cache model was verified by utilizing the combined simulations of the MPCache framework in comparisons with an RTL simulation. The MPCache framework was configured to mimic the RTL model and used a trace file of the execution of each of the benchmarks applications used in the RTL simulation. The estimates of execution time made by MPCache were reasonably close but deviated by -24.6% to 12.3% at the extremities. However, the execution time estimates for the benchmarks were ranked equally by both simulations. The total energy usage estimates by MPCache deviated by close to -55% for the different benchmarks compared to the RTL simulations. The ranking of the total energy usage estimates was close to equal for both simulations except for two benchmarks, which changed places in the ranking for MPCache. Analysis of the contribution to the total energy usage in the RTL simulation showed that MPCache overestimated the energy usage of the memory blocks significantly for most of the benchmarks, but that this helped to make the estimate closer to the energy usage of the entire cache. It was concluded that the MPCache estimates for this particular cache structure were acceptable as a high-level simulation. However, that, especially the physical model, needs more features to model real caches accurately. It is also necessary to compare the performance to more RTL models to get a clearer picture.

Chapter 7 describes comparisons of MPCache with the established physical cache model CACTI. When comparing the plain results of the two models for a selection of cache structures, we could see that the MPCache estimates do not scale naturally with the cache size for the sequential access mode. There was also a large discrepancy between the energy usage of write accesses of the two models. The models were also compared using combined simulations with the functional model of MPCache. These results demonstrated the same trends, and it was concluded that MPCache needs to model the energy consumption of additional parts of the cache to scale well for larger caches using more memory blocks, like wires. When CACTI was used in comparison with the RTL simulation, we saw that the estimates for total energy usage were less accurate and

the ranking of the off by more than those produced by MPCache. The leakage power estimate of CACTI compared to the RTL simulation was significantly wrong, while that of MPCache was reasonably close. It was concluded that even though MPCache performed better in comparison with the RTL model, the deficiencies regarding the modeling must be addressed for the framework to be reliable in a design exploration phase.

# Chapter 9

# Future Work

In future work, the functional model of MPCache can be improved by adding support for more features, which would be interesting to include in the exploration of different designs. Including support for prefetchers is an essential object, as they are used extensively in modern designs. It would most likely require that the functional model is elaborated with a better perception of time because the performance of prefetchers is very dependent that the fetch happens at the right time. As of now, its only knowledge of time is the chronology of memory accesses made to it. This enhancement could be done by adding some timestamping to the trace-file format. With this improved perception of time by the model, it would be worth looking into adding some instructions for the cache, such that it would be possible to model power-states and locking of the cache. The improvement could also improve the estimation of execution time, since the time when the processor working while no memory accesses are pending could be accounted for.

The physical model of MPCache can be improved by adding modeling of wires to the estimation of energy usage. This would generally make the estimates more accurate but is vital for modeling of sequentially accessed caches to see the impact on the energy usage for larger cache sizes. The model would also benefit from more options of access modes, e.g., adding support for speculative fetching. Speculative fetching is a conditional access mode since the energy usage of the access would depend on the fetch being correct or not. This fact means that this information would need to be determined during the functional simulation, leading to alterations to the functional model as well.

In Chapter 6, the dynamic energy usage of write-hit scenarios was calculated equally using the write-through or the write-back policy. This calculation is not correct since, in the case of write-back, the data memory will need to be read when a dirty cache line is replaced. Future work can improve this calculation, but will also need modifications to the functional model since information about how many dirty cache lines are replaced is needed.

It would be interesting to compare the results produced by MPCache with more RTL simulations, to get more data points on different cache structures. This information would give us better indications of how the framework manages to perform for different parts of the design space, and provide feedback on what parts it models good, and where it needs to improve. In this context, it would also be useful to test the memory hierarchy capabilities of the MPCache framework by comparing it with an RTL simulation using multiple cache levels.

The functional model of MPCache has not been compared with any established functional cache models. It would be interesting to measure its performance compared to other models to chart out its strengths and weaknesses.

# Bibliography


[1] R. Sam, "Zoom: A performance-energy cache simulator," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[2] Z. Zhu and X. Zhang, "Access-mode predictions for low-power cache design," *IEEE micro*, vol. 22, no. 2, pp. 58–71, 2002.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[5] B. Jacob, D. Wang, and S. Ng, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[6] K. Hwang and N. Jotwani, *Advanced computer architecture, 3e*. McGraw-Hill Education, 2016.

[7] L. Wehmeyer and P. Marwedel, *Fast, Efficient and Predictable Memory Accesses*. Springer, 2006.

[8] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42nd annual Southeast regional conference*, 2004, pp. 267–272.

[9] N. P. Jouppi, "Cache write policies and performance," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 191–201, 1993.

[10] M. L. Loper, *Modeling and simulation in the systems engineering life cycle: core concepts and accompanying lectures*. Springer, 2015.

[11] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[12] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys (CSUR)*, vol. 29, no. 2, pp. 128–170, 1997.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[15] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *Ieee micro*, vol. 26, no. 4, pp. 52–60, 2006.

[16] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH computer architecture news*, vol. 25, no. 3, pp. 13–25, 1997.

[17] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson, "mlcache: A flexible multi-lateral cache simulator," in *Proceedings. Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. 98TB100247)*. IEEE, 1998, pp. 19–26.

[18] S. J. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of solid-state circuits*, vol. 31, no. 5, pp. 677–688, 1996.

[19] G. Reinman and N. P. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Western Research Lab Research Report*, vol. 7, 2000.

[20] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," 2001.

[21] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0," Technical Report HPL-2006-86, HP Laboratories Palo Alto, Tech. Rep., 2006.

[22] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[24] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti

7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[25] M. Mamidipaka and N. Dutt, "ecacti: An enhanced power estimation model for on-chip caches," *Center for Embedded Computer Systems, Technical Report TR*, pp. 04–28, 2004.

[26] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 51–62, 2008.

[27] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 694–701.

[28] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 83–94, 2000.

[29] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[30] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 577–589.

[31] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 423–428.

[32] J. Lacord, G. Ghibaudo, and F. Boeuf, "Mastar va: A predictive and flexible compact model for digital performances evaluation of cmos technology with conventional cad tools," *Solid-state electronics*, vol. 91, pp. 137–146, 2014.

[33] K. Jeong and A. B. Kahng, "A power-constrained mpu roadmap for the international technology roadmap for semiconductors (itrs)," in *2009 International SoC Design Conference (ISOCC)*. IEEE, 2009, pp. 49–52.

[34] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005, vol. 2.

[35] Nordic Semiconductor. CACHE - Instruction/data cache. Nordic Semiconductor ASA, Trondheim, Norway. Accessed on: May 5, 2020. [Online]. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fps_nrf5340%2Fcache.html&cp=3_0_0_4_3_0

[36] ARM Developer. Cortex-M33. ARM Holdings, Cambridge, UK. Accessed on: May 8, 2020. [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33

[37] EEMBC. Introduction to the EEMBC TeleBench™ Performance Benchmark Suite. Embedded Microprocessor Benchmark Consortium, Hillsboro, OR, USA. Accessed on: May 8, 2020. [Online]. Available: https://www.eembc.org/telebench/

[38] EEMBC. CoreMark®, An EEMBC Benchmark. Embedded Microprocessor Benchmark Consortium, Hillsboro, OR, USA. Accessed on: May 8, 2020. [Online]. Available: https://www.eembc.org/coremark/

[39] ARM Developer. GNU Arm Embedded Toolchain. ARM Holdings, Cambridge, UK. Accessed on: June 9, 2020. [Online]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

[40] Mentor Graphics. Questa® Advanced Simulator. Mentor Graphics, Wilsonville, OR, USA. Accessed on: June 9, 2020. [Online]. Available: https://www.mentor.com/products/fv/questa/

[41] Synopsys. Verdi. Synopsys Inc., Mountain View, CA, USA. Accessed on: June 9, 2020. [Online]. Available: https://www.synopsys.com/verification/debug/verdi.html

[42] Synopsys. SpyGlass Power. Synopsys Inc., Mountain View, CA, USA. Accessed on: June 9, 2020. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-power.html

[43] A. J. Anderson, *Foundations of computer technology*. CRC Press, 1994.

# Appendix A

# The MPCache Framework

This Appendix contains the source code for the MPCache framework. The source code is separated into the functional models, the physical models, and the simulation files, and can all be found in the ZIP-file attached to this thesis (**Appendices.zip**).

## A.1 The Functional Models

The functional models of MPCache are located in the **__init__.py** file in the folder named **functional_cache_model**. The file contains the Python classes listed below.

- CacheFunctionalModel - The main functional model of MPCache. A class made to be configured to imitate the functionality of specific cache memories.
- CacheLine - A class used like a struct to represent cache lines in the CacheFunctionalModel class.
- MainMemoryFunctionalModel - A class used to represent endpoints in a memory hierarchy. Mostly a container to hold number of accesses.
- MemoryHierarchyModel - A class used to construct memory hierarchies consisting of CacheFunctionalModel and MainMemoryFunctionalModel objects.

## A.2 The Physical Models

The physical models of MPCache are located in the **__init__.py** file in the folder named **physical_cache_model**. The file contains the Python classes listed below.

- CachePhysicalModel - The main physical model of MPCache. A class made to be con-

figured to estimate the energy consumption, time consumption, and leakage power of specific cache memories using datasheet values.

- MainMemoryPhysicalModel - A class made to be used in physical/combined memory hierarchy simulations as a container of the energy consumption, time consumption, and leakage power of main memory models.

## A.3 Simulation File

The simulation file of MPCache are located in the **simulation.py** file in the folder named **simulation_file**. The file contains the simulation file used to run simulations that utilize the functional and physical models of MPCache.

# Appendix B

# Functional Verification

This Appendix contains the files used for the verification of the functional cache model described in Section 5.2. These files are separated into the cache configurations and the access traces, and can be found in the ZIP-file attached to this thesis (**Appendices.zip**).

## B.1 Cache Configurations

The cache configuration files used in the verification of the functional cache model are located in the folder named **functional_verification_configs**. There is one file for every test case listed in Table 5.3, including the duplicate test cases. The configuration files are named after its test case number and test parameter, e.g., **test_case_1_associativit_0.ini**.

## B.2 Memory Access Traces

The trace files used in the verification of the functional cache model are located in the folder named functional_verification_trace_files. The trace files are listed below with a comment about their function.

- **general_trace.txt** - A trace made to highlight the different cache structures by use of small and large loops, and a loop with large jumps.
- **write_hit_trace.txt** - A trace made to highlight the differences between various write-hit policies.
- **write_miss_trace.txt** - A trace made to highlight the differences between various write-miss policies.

- **replacement_trace.txt** - A trace made to highlight the differences between various replacement policies.