# Temperature, Energy and Performance: Addressing Embedded System Challenges through Fast Cache Simulation

by

Josef Lionel Schneider

A Thesis

Submitted in Accordance with the Requirements

for the Degree of

Doctor of Philosophy

School of Computer Science and Engineering

The University of New South Wales

May 2015

# Statement of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

_____

Josef Schneider

May 2015

# Copyright Statement



_____

Josef Schneider

May 2015

# Authenticity Statement

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.

_____

Josef Schneider

May 2015

# List of Publications

- J. Schneider, J. Peddersen and S. Parameswaran. Speeding Up Single Pass Simulation of PLRUt Caches. In *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2015.

- J. Schneider, J. Peddersen and S. Parameswaran. MASH{fifo}: A Hardware-Based Multiple Cache Simulator for Rapid FIFO Cache Analysis. In *Proceedings of the Design Automation Conference*, June 2014.

- I. Nawinne, J. Schneider and S. Parameswaran. Hardware-based fast exploration of cache hierarchies in application specific MPSoCs. In *Proceedings of the Design, Automation and Test in Europe Conference*, March 2014.

- J. Schneider, J. Peddersen and S. Parameswaran. A Scorchingly Fast FPGA-Based Precise L1 LRU Cache Simulator. In *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2014.

- H. Amrouch, T. Ebi, J. Schneider, S. Parameswaran and J. Henkel. Analyzing the thermal hotspots in FPGA-based embedded systems. In *Proceedings of the Field Programmable Logic and Applications Conference*, September 2013.

- J. Schneider and S. Parameswaran. An Extremely Compact JPEG Encoder for Adaptive Embedded Systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, March 2013.

## Contributions of this Thesis

- Extremely compact JPEG encoder for adaptive embedded systems capable of varying the application quality at the DCT level.
- A thermal study of FPGAs produced in collaboration with Hussam Amrouch and Thomas Ebi.
- A series of precise cache simulators which are shown to be the fastest in their categories:
  - MASH{lru}, a hardware-based simulator capable of concurrently simulating multiple LRU caches.
  - MASH{fifo}, which simulated multiple FIFO caches in hardware by making use of newly-discovered FIFO cache inclusion properties.
  - MASS{plrut}, a software-based cache simulator of PLRUt caches.
- The concept of in-system cache simulation where the task of evaluating multiple caches is performed from within an embedded system in real-time.
- A trace compression algorithm that sets itself apart with its extremely fast encoding and decoding times.

# Abstract

Temperature, energy and performance are essential design considerations during the conception of modern digital systems. Static and dynamic methods can be used to overcome the limitations imposed by energy and power consumption, and the resulting increase in temperature. Static methods target digital circuit aspects such as cache configuration, bus width and component layout which, once fixed, can generally not be changed after the circuit has been manufactured. On the other hand, dynamic methods including dynamic voltage and frequency scaling, application adaptation and task re-mapping on a Network-on-Chip make modifications to the system while the device is deployed. The work presented in this thesis focusses on three aspects in the realm of temperature, energy and performance.

First an evaluation of the suitability of the dynamic application adaptation method is researched with the aim of using it to control the temperature of an FPGA device. Despite the use of an extremely adaptive custom JPEG encoder it was determined that application adaptation alone is ineffective in an FPGA for thermal management. However, when implemented alongside other dynamic methods such as frequency scaling and clock gating, a temperature difference of 6°C could be achieved.

Next, a study is performed which aims to assess which components are principally responsible for the rise in temperatures in FPGAs. A novel thermal measurement system based on an infrared camera was used to determine that the biggest contributor to elevated FPGA temperatures is the external memory interface. Having found that a lower external memory access rate lowers the FPGA temperature, a model is proposed linking cache miss rate with device temperature.

The third and main aspect covered in this dissertation is that of CPU cache simulation. Depending on the application executed, the cache configuration can have a drastic impact not only on temperature but also on system performance and energy consumption. Precise cache simulation is often used to evaluate different cache configurations, yet current simulation tools are notorious for their slow execution times, especially when analysing

the trace of large and complex applications. Three Multiple cAche Simulators in Hardware (MASH) or in Software (MASS) are proposed for three cache replacement policies: MASH{lru} for the Least Recently Used (LRU) cache algorithm, MASH{fifo} for First In First Out (FIFO) and MASS{plrut} for Pseudo Least Recently Used tree (PLRUt). The former two are novel in that they are implemented in hardware and are respectively 53x and 11.10x faster than software counterparts. The PLRUt simulator presents for the first time an optimised hash table-based algorithm yielding a speedup of 1.93x over an unoptimised solution. All cache simulators employ cache properties specific to their replacement policies to improve simulator characteristics. For the hardware-based simulators, these cache properties are exploited to minimise the resource usage of a simulator instance.

Additionally, it is shown that the hardware (or MASH) simulators can be implemented in-system alongside an embedded system, allowing for the direct trace extraction and cache simulation from within an FPGA. Using in-system simulation, large speedups can be achieved as trace generation and multiple cache simulation happen at the same time at high frequencies.

Finally, a cache compression algorithm is discussed that is characterised by extremely fast encoding and decoding times with the additional benefit of being easy to implement in hardware.

# Acknowledgements

Significant achievements do not occur in isolation, and I would like to thank a number of people for their contributions and help along the way. First and foremost I would like to thank my supervisor, Professor Sri Parameswaran, whose expertise, patience and trust in my abilities were invaluable over the course of my cadidature. I am also very grateful for Jorgen Peddersen's friendship and assistance over the years, and doubt that many of the findings presented in this dissertation would have materialised if it hadn't been for those many multi-hour discussions we had over a whiteboard, bouncing ideas back and forth.

Many thanks to Hussam Amrouch and Thomas Edi for being a pleasure to collaborate with. To colleagues Jude Angelo Ambrose, Isuru Nawinne, Haseeb Bokhari and Liang Tang amongst others, who were always ready to listen to new ideas and to lend a helping hand. To my immediate family, Sarah Bisley, Wolfgang Schneider and Anthony Schneider who have consistently been there for me and whose enthusiasm for my work has been a real driving force. Last but not least I would like to express my gratitude towards my wonderful partner, Charlotte Fetherston, for the unwavering love and support I have received during this journey.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The world of computing has advanced in leaps and bounds since the invention of the transistor in the middle of the 20th century. A transistor, the fundamental building block of electronics and digital circuits, was combined with other transistors on a die to make an Integrated Circuit (IC) a little over ten years thereafter, paving the way for electronics that could do a lot more at a cheaper cost while requiring less power and energy. As the process of manufacturing integrated circuits was refined, the transistors could be made smaller allowing for additional, faster IC functionality and lower power consumption. The trend of miniaturisation is famously observed by Moore's Law, which states that the number of transistors in an integrated circuit will double approximately every two years [1]. While we are currently experiencing the end of Moore's law as we reach the physical limits of the materials used [2], Moore's prediction has been surprisingly accurate for over fifty years.

Traditionally, the primary aim of digital integrated circuit designers has been to maximise the computational throughput of their chips. Until recently this could be achieved in two ways: by increasing the complexity of the chip or by increasing the operating frequency. These options conflict with each other, as increasing the complexity of a chip generally results in a lower maximum operating frequency. Recently however, the energy consumption and the resulting increase in die temperature have become the main factors limiting the processing power of integrated circuits. A transistor switching at a

higher frequency consumes more energy: smaller and faster transistors lead to higher power densities. Nowadays, extremely complex circuits could be clocked at higher frequencies if it were not for the high temperatures produced. The consequences of elevated temperatures in ICs can be devastating. As the aging of silicon is accelerated at higher temperatures, overheating integrated circuits are prone to premature failures in the form of transient or permanent faults [3]. Sudden and localised temperature increases lead to large temporal and spacial thermal gradients which can cause cracks in the die. To counteract these problems, an obvious need has arisen for IC designers to seriously consider the thermal aspect of their circuits at design time.

With a surging market for mobile devices, another facet of electronic design has become prominent: energy and power consumption. The vast majority of mobile devices are powered by batteries which can only store a limited amount of energy. Additionally, batteries are adversely affected by fluctuating power consumption levels [4]. It has become important for manufacturers to design electronic circuits that present high processing power and offer acceptable battery life in order to remain competitive.

## 1.1 Energy, Power and Temperature

Electronics are powered by electric energy that circuits can convert into many other forms of energy such as heat, light and kinetic energy. Digital electronics are also powered by electric energy of course, yet it is not immediately obvious why this electric energy is required. After all, the output of purely digital circuits is information in the form of ones and zeroes which, by themselves, contain little to no energy. And yet computers require non-negligible amounts of energy (in the case of mobile phones for example) to vast amounts of energy (for server farms) to function. In digital circuits, the energy is consumed by the transistors that compute the output. This energy is dissipated within the wires of the IC in the form of heat. The temperature and energy consumption of a digital IC are therefore very closely related. Power is the rate of energy consumption with respect to time. The power consumption of a circuit greatly impacts the battery lifetime

of mobile devices and is an important consideration for power supply design.

CMOS (Complementary Metal-Oxide-Semiconductor) ICs are by far the most common digital transistor technology, but are plagued by certain electrical characteristics that arise from the way they are constructed and the materials they are made from. Firstly, a transistor in a CMOS chip will always exhibit a certain amount of capacitance at a number of locations within the transistor. A capacitor stores an increasing amount of energy when the differential voltage across the capacitor increases. When the voltage difference decreases, that energy is fed back into the driving circuit. In the case of a CMOS transistor, there is capacitance between the input (gate) and the ground of the circuit. When the gate is '1', or high, a differential is applied across the capacitance which causes it to store energy. If afterwards the gate is switched to '0', or low, the energy stored in the capacitance will be dissipated in the circuit driving this transistor. The capacitance of our transistor therefore consumes and stores energy when the gate goes from '0' to '1', and releases it when the gate goes from '1' to '0' by dissipating it within the wires of the CMOS IC in the form of heat.

The energy consumption due to transistor capacitance only manifests itself when the transistor switches and is therefore called 'dynamic energy consumption'. The amount of energy ($E_{cap}$) stored in a capacitor of capacitance $C_{eff}$ is given by the following equation [5]:

$$E_{cap} = \frac{1}{2} * C_{eff} * V_{dd}^2 \tag{1.1}$$

Thus the dynamic power consumed by a transistor switching at frequency $f$ can be described as follows [6]:

$$P_{dynamic} = C_{eff} * f * V_{dd}^2 \tag{1.2}$$

In other words, by reducing the frequency at which the device is driven, the power consumption is also reduced in a linear manner. As energy/power are proportional to the square of the voltage, it is possible to reduce the energy consumption of a switching

transistor by reducing the power supply voltage. Lowering the voltage, however, lowers the speed at which the transistor operates: a lower voltage potential at the gate leads to a longer switching time caused by the gate capacitance, as it takes longer to charge. Additionally, slow switching times are the cause of short circuit energy dissipation, which is also a form dynamic energy consumption. Each wire within an IC needs to be driven by at least two transistors, where one drives the wire high and the other drives it low. Transitioning from one state to the other takes a finite amount of time in which both transistors allow an electric current to flow through them, effectively short–circuiting the supply voltage to the ground.

Static power consumption is the power an IC consumes when it is not switching and is primarily comprised of tunnel and subthreshold leakage currents. The former have been exacerbated by the shrinking of the transistor size as more electrons are likely to tunnel across the oxide insulation layer as it gets thinner. The latter have been on the rise due to the lowering of the supply voltage, and the need it created to lower the subthreshold voltage accordingly. Subthreshold leakage currents are also strongly temperature dependent as the subthreshold voltage decreases with temperature [7]. As a result, static power consumption, which has been insignificant for a large part of the transistor's history, has become a dominant factor in recent years [8].

In summary, with the advancements in IC fabrication technology, transistors have become increasingly more compact, thereby reducing their dynamic power consumption and making them faster. This has been accompanied with lower supply voltages and higher static power consumption. Overall, the energy consumed by transistors has been decreasing linearly, while their density has been increasing exponentially. This has led to an increase in energy density causing higher temperatures which, in turn, cause even higher static power consumption. It should be clear by now that there are many factors that closely affect each other in the functioning of an integrated circuit. The designers of modern digital chips are presented with a number of quantities that are often conflicting:

1. processing throughput, to ensure the IC is capable of performing the tasks it is made for,

2. energy and temperature, as excessive temperatures can lead to transient and permanent faults, reduced lifetime and greater power consumption,

3. power consumption, which affects the cost of the electronics used to power the chip and the battery life in mobile devices

4. chip size and complexity, which has an impact on the processing throughput, the power consumption of an IC and also cost, as larger dies are more expensive.

As it happens, CPU caches profoundly affect each of these quantities, as we shall see in Section 1.2. However the first part of this dissertation focuses on temperature, one of the main bottlenecks in recent digital circuit design. More precisely, the impact that a method called Application Adaptation (Chapter 2) can have on the temperature of a Field Programmable Gate Array (FPGA, Section 1.3) is investigated. The self-heating characteristics of FPGAs is looked into in Chapter 4.

## 1.2 CPU Caches

The advancements in the fields of digital circuit design are met with ever increasing software complexity and memory requirements [9], 'consuming' the increased performance that is provided. The need for larger memories has outpaced the rate at which memory performance increases, resulting in memory speed that is now lagging far behind the performance of the processor it is connected to. Unfortunately, a processor, or Central Processing Unit (CPU), heavily relies on memory to read the instructions it needs to execute and to load and store data. This means that the CPU, while physically capable of greater processing throughput, has to spend much time waiting for the memory to catch up. This is commonly known as the "Processor Memory Gap" or the "Memory Wall" [10], the magnitude of which is shown in Figure 1.1.

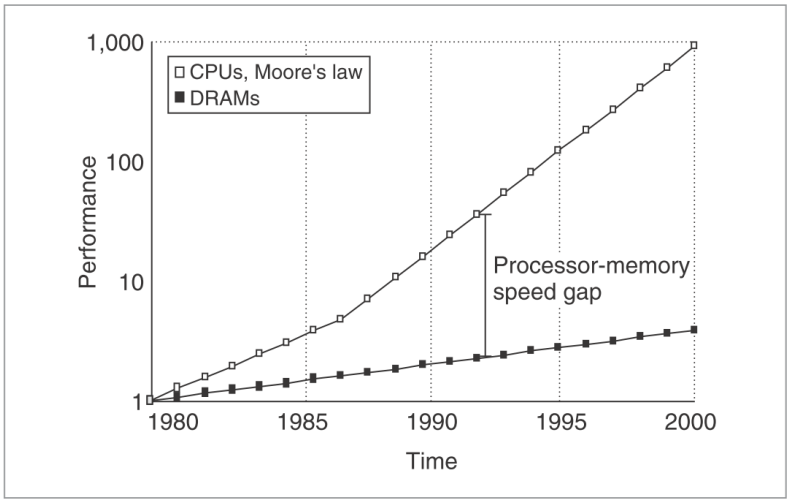



**Figure 1.1:** The processor memory gap is growing at an exponential rate. Figure by Jacob et al. [11]

For years this bottleneck has largely been overcome with the use of CPU caches [6, 12]. The main memory is so slow because it is so large, and a smaller memory is faster but cannot hold as much data. However, during the execution of a program, the locations accessed in memory exhibit a large amount of spatial and temporal proximity: an item of data that was recently accessed and its neighbouring items of data are very likely to be accessed in the near future.

A simple analogy for this situation would be the role of an accountant who has to work their way through many documents located in a filing cabinet. Each document is uniquely identified, and this identifier bears a relation to the order in which the documents are stored. For example, documents could be identified and ordered by filing date. It is very probable that a document obtained from the filing cabinet will be looked at a number of times before it is returned. Given the order in which files are stored, there is also a high chance that the neighbouring files will be used shortly thereafter. Alas, accountants, much like computer engineers, are famed for their lack of physical vigor, and it takes a certain amount of time and effort to retrieve a document from the filing cabinet. To make the trip worthwhile the accountant therefore collects the targeted document together with the neighbouring files, and puts them on their desk for quick subsequent access.

A cache acts very much like the accountant's desk in that it is a small, fast-access data

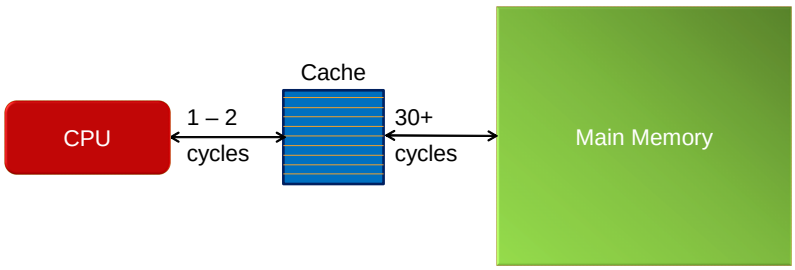



**Figure 1.2:** A cache is the small fast memory, close to the CPU, that acts as intermediate storage between the fast CPU and the slow main memory. The clock cycles given are only a rough indication as to how long it would take to access these memories.

repository located close to the unit that requires the data, and that it stores the recently used and neighbouring data. The document identification, in this case, is much like the memory address of the data. While accessing the main memory can take many tens of clock cycles, a cache is accessed in a matter of clock cycles. If the data is not present in the cache we have a *cache miss* and the data has to be obtained from the slow main memory. Accessing data that is cached results in a *cache hit* and the data can be obtained much more quickly. This is shown in Figure 1.2. Returning to the desk analogy, choosing a desk is not always straight-forward. The accountant could buy a desk that is very large to temporarily store more documents, but this desk would cost more and finding a document amongst other documents on it could take longer. In fact, the desk could be so large that finding a document on it could take longer than getting it from the filing cabinet. Caches face the exact same issues: larger caches display greater hit rates but are slower and more expensive. But cache size is not a simple linear quantity as caches are configured by a number of parameters, notably:

- block size, or line size ($ls$), defining how many bytes are stored in a cache line,
- set size ($ss$), defining how many sets are stored in a cache,
- associativity ($assoc$), the number of different locations where an item of data could be stored within a set,
- replacement policy ($rep$), which determines which item of data will be evicted if a cache set is full.

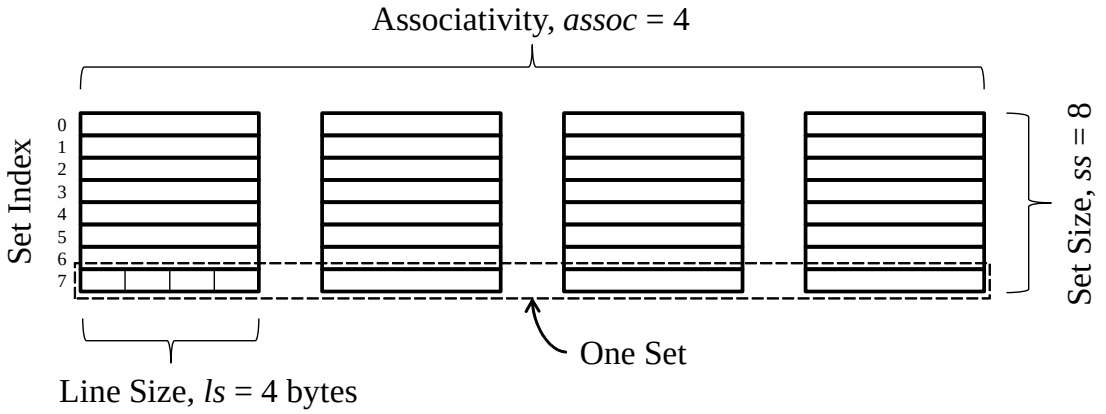



**Figure 1.3:** A demonstration of how line size, set size and associativity affect cache size and layout.

Confusingly, the quantity *set size* does not define the size of sets but the number of sets in a cache. This follows the convention established by previous authors in the field of cache simulation (Janapsatya [13], Haque [14] and Tawada [15]).

On one extreme there are caches with an associativity of one, also known as direct mapped caches, where a cached item of data can only be stored in one single location. For example, the desk system could be set up so that every document whose identifier ends with a '6' can only be stored in one location of the desk. If however more than one different document with an identifier ending in '6' are accessed in an alternating manner, every document is evicted before it is reused. The occurrence of continuous storing and eviction of data from a cache line is called *cache thrashing* and needs to be avoided as it removes any benefit of having a cache. Although it is far from ideal, a direct mapped cache is fast, as it only needs to look at a single location to check if a data item is there or not. Increasing the number of locations in which a data item could be stored amounts to increasing the associativity. If a cache has an associativity of four it means that data at a certain address could be cached at four different locations. A depiction of a cache with these specifications can be seen in Figure 1.3. The set index is like the last digit of our document identifier, and each set can hold four cache lines. If a data item needs to be stored in a set but all four lines are filled, the *cache replacement policy*

determines which item of data to evict to make room for the new data. The most common replacement policies are Least Recently Used (LRU), First In First Out (FIFO), Pseudo Least Recently Used (PLRU) and Random replacement. We will be looking into their precise functionality in future sections, but just as an example: an LRU replacement policy keeps track of how recently items of data were used and orders them accordingly. When time comes to evict an item, the least recently used cache line is selected for replacement.

The other extreme of cache configurations is a fully associative cache which is equivalent to a cache with a set size of one and associativity one or larger. When it comes to CPU caches, fully associative caches are hardly ever used due to their high hardware resource cost (one comparator for every tag) and low speed (at every access the entire cache is searched). Size of the cache is determined by the product $ls \times ss \times assoc$ bytes.

### 1.2.1 The Impact Caches have on Systems

In rough terms, the larger a cache is the slower it is, the more power it consumes and the more it costs. But a CPU cache does not work in isolation: a cache miss causes a memory request to the main memory whose interface can consume a large amount of energy. A cache miss also forces the CPU to stall for a certain number of clock cycles, thereby *reducing* the overall CPU power consumption and performance. The complex relations between CPU, cache and memory closely relate to one quantity, which is the cache hit rate. The cache hit rate, in turn, relies on the cache configuration and the order of the memory accesses which depends on the application being executed. This means that for any application, certain cache configurations will be more optimal in terms of power and performance than other cache configurations.

Figure 1.4 plots the hit rates of many cache configurations against execution time (system performance) and energy consumption for a G.721 encoding application. It is easy to see that the configuration yielding the maximum hit count is far from being the configuration with minimum execution time or minimum energy consumption.

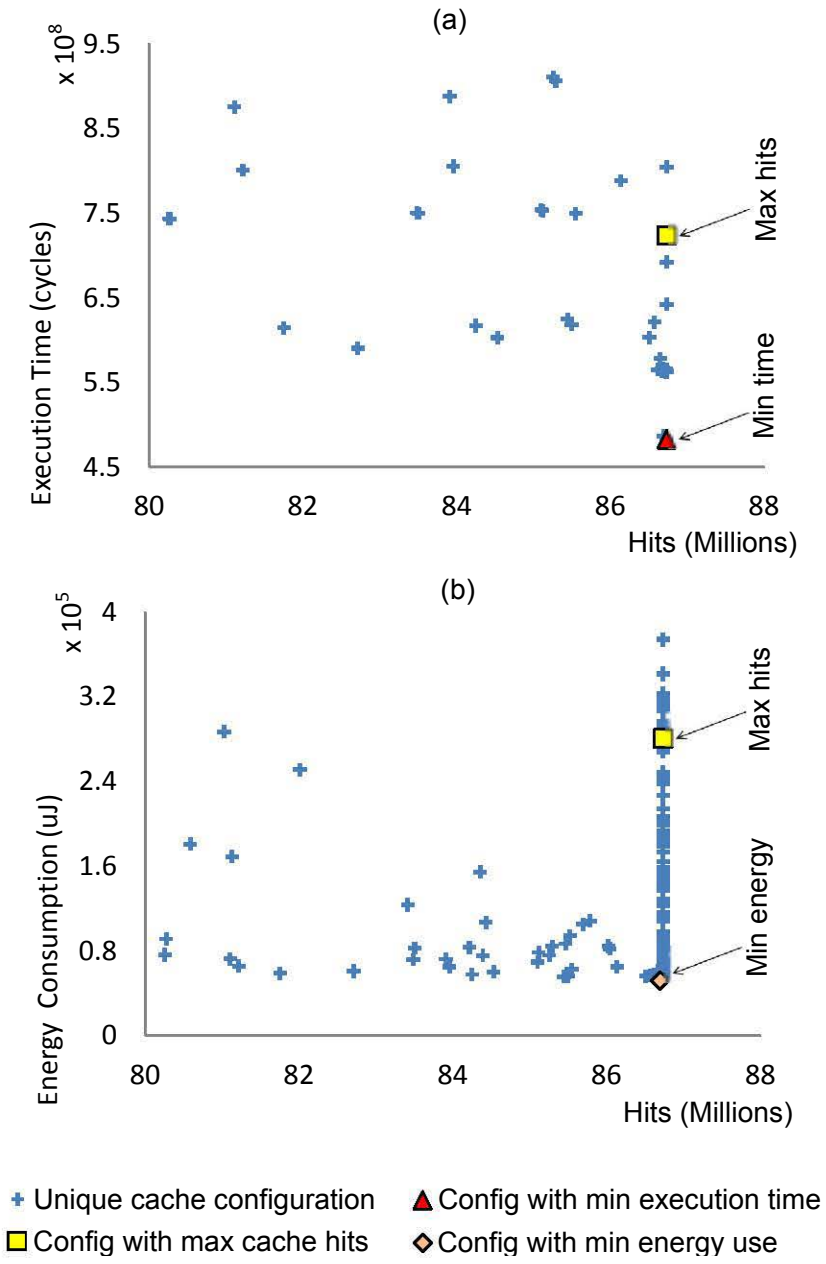



**Figure 1.4:** Plots of the hit rates of different cache configurations vs. execution time (a) and energy consumption (b) for a G.721 encoding application.

### 1.2.2 Choosing the Right Cache through Cache Simulation

Many modern processor cores such as NIOS II [16], ARM [17] and Tensilica Xtensa [18] provide the designer with the ability to configure the system cache or caches. Designers therefore need to know which cache configuration (set size, associativity and block size) would be optimal depending on the application they are running on the given processor. To find the cache configuration yielding the minimum execution time and/or power consumption, the cache hit and miss counts need to be determined for each unique cache configuration available. The obtained hit and miss values can directly be used to determine the timing and energy consumption of caches and even memory systems comprised of a cache and external memory device (e.g., a DRAM chip) by using equations such as those presented by Janapsatya et al. [13]. The equations require cache performance metrics in the form of cache hit and miss rates and a number of cache parameters, such as die area, timing and energy values that can easily be obtained using the free CACTI tool [19]. Having calculated the performance and memory consumption of the memory system for each cache configuration, designers can then choose the best cache configuration for their purposes based on their design constraints.

Cache performance metrics can be precisely determined through precise cache simulation. A precise cache simulator analyses the order of memory accesses, also known as the memory trace, and computes the hit and miss values of caches of different configurations. Some simulators only keep track of one cache configuration at a time [20] and are therefore known as 'multi-pass' simulators, as $x$ simulation runs need to be performed to simulate $x$ configurations. The downside of this approach is that it can take an exceedingly long time to explore the design space given a representative range of inputs, especially for modern and complex applications that perform hundreds of millions of memory accesses per second. For example, it was found that encoding 24 low resolution images into an MPEG2 video using a software encoder (i.e., without hardware acceleration) produces a trace file containing over 11 billion memory accesses. Precise cache simulation can be sped up through the use of 'single-pass' simulators, a technique known as 'multiple cache

simulation' where a memory trace is only read once but many different caches are simulated. Single-pass cache simulators lend themselves well to the creation of optimised algorithms and data structures specially tailored for rapid cache evaluation.

Unfortunately, despite all the research effort in the field, single pass simulators are still very slow. In a test conducted on one of the fastest single pass simulators, processing the MPEG2 encoding trace file mentioned above took over an hour. The hit rates of only 40 caches out of hundreds of possible configurations were predicted, and the simulation run took this long even though the output of the application was only *one second* worth of low-resolution video (see Chapter 5). In an environment where design turnaround time is paramount, the speeding up of cache simulation is vital.

Increasing the performance of single-pass cache simulators is the main focus of this thesis. Methods are explored whereby simulating multiple caches can be sped up by making use of cache properties.

## 1.3 Field Programmable Gate Arrays

This thesis dissertation researches a number of aspects relating to temperature, performance and the power consumption of digital devices, with a strong focus on Field Programmable Gate Arrays, or FPGAs. The functionality of a digital chip relies on the binary state of its transistors which can be '1' or '0', also often described as 'high' or 'low', 'set' or 'cleared' respectively. Transistors are combined to make logic gates, the very basic functional units of digital circuits. Logic gates perform simple operations such as outputting high if all input signals are high (AND gate), or outputting high if any input is high (OR gate). Of special interest is the NAND (Negated-AND) gate which only outputs low if all input signals are high, and outputs high otherwise. Any digital circuit can be implemented with a combination of NAND gates as it exhibits a property called functional completeness (a property also exhibited by the NOR gate). Most digital ICs are Application Specific Integrated Circuits (ASICs) as the functionality is hardwired into the chip for a clearly defined purpose. For example, a memory chip is an ASIC that is designed

and wired to act as a memory chip and could hardly be used for any other purpose, as one would expect.

An FPGA, on the other hand, is not application specific as it can be configured to mimic any digital circuit. An analogy for an FPGA would be a very large pile of NAND gates that can be connected together any way the designer pleases. The way in which the logic is connected is called the FPGA 'configuration' and is defined in software. The designer is then able to load the configuration, or design, onto the FPGA, following which the device behaves like a fully functional digital IC. In reality, an FPGA is not made up of NAND gates but of Look-Up Tables (LUTs) and Registers that are connected together by configuration logic and interconnect. The size of the FPGA, which is determined by the number of LUTs and registers, defines the maximum size or complexity of the circuits that can be implemented.

Naturally, there are upsides and downsides to using FPGAs instead of ASICs. The latter are faster and require less power to perform the same task as an FPGA contains a large amount of configuration and interconnect logic that both slow it down and consume additional power. However, the fabrication of ASICs is characterised by very high non-recurring engineering costs that make it unsuitable for low production volumes. On the other hand, the flexibility of FPGAs means that the same chip can be used for many different purposes, and that it can even be reconfigured in the field. For these reasons FPGAs have become increasingly popular as reconfigurable accelerators, development platforms and as complex ICs in low-volume production designs. Despite consuming more power than ASICs for the same functionality, it is generally considered that FPGAs do not suffer from overheating, as the logic is subject to lower switching densities and frequencies. While this holds true for older devices and simpler designs, the recent progress in semiconductor manufacturing processes and the appearance of FPGA ICs with vast logic resources have made thermal issues a concern, even for FPGA designs, as will be explored in Chapter 2.

## 1.4 Thesis Overview

The aim of the research presented in this dissertation is to supplement the current knowledge on addressing the pertinent issues facing embedded and FPGA-based systems today. As previously mentioned, one of the principal concerns is self-heating which severely limits the processing throughput of digital circuits and lowers their long term reliability. Even in the field of FPGA research, temperature has been identified as a limiting factor. Jones et al. [21], for example, documented a case where the self-heating of an FPGA caused the destruction of their development board. Jones's solution was mostly based on frequency scaling, a technique that has long been used to reduce the power and energy consumption of digital electronics.

The decision was made to evaluate the effectiveness of software-based Application Adaptation as a means to dynamically lower the temperature within FPGAs. Using an extremely adaptive custom JPEG encoder, the impact of application adaptation was tested on two Altera FPGAs built on older (90nm) and more recent (40nm) technology. Those experiments are covered in Chapter 3 which concludes that application adaptation on an FPGA is only effective when combined with conventional clock gating and frequency scaling. The observation is also made that a greater understanding of FPGA thermal characteristics is required to better mitigate self-heating issues while maximising system performance.

Chapter 4 describes a thermal analysis of reconfigurable logic. Insight into the self-heating of FPGAs was obtained through accurate measurements using an infrared thermal camera pointing at an exposed FPGA die. This thermal evaluation determined that the hotspot in a conventional embedded system implemented on an FPGA is caused by the memory controller. Memory controllers are often large and complex pieces of hardware running at high frequencies, especially for modern memory devices such as DDR3 and QDR SRAM. Additionally it was discovered that minimising the number of accesses to the external memory can significantly lower the hotspot temperature of an FPGA device. As processor caches reduce the number of external memory accesses, it was found that

cache configuration impacts device temperature, and a model is proposed linking device hotspot temperature with cache hit and miss rates.

Cache evaluation is a commonly performed task that enables designers to select caches that meet certain performance and energy criteria. As mentioned in Chapter 2, the main difficulty with cache simulation is that determining precise cache metrics is a slow process which increases the design time, especially for embedded systems running complex, process-intensive software with long traces. Having established that cache evaluation can also help minimising the temperature of System-on-Chips, methods to speed-up cache simulation were investigated.

The first attempt towards this goal targeted the rapid simulation of caches obeying the LRU replacement policy. The simulator called Multiple cAche Simulator in Hardware for LRU replacement policy (MASH{lru}) is covered in Chapter 5. It is implemented as a hardware simulator that runs considerably faster than any software alternatives. A variant of the simulator, MASH{fifo}, is proposed in Chapter 6 which is capable of evaluating caches with the FIFO replacement policy. As the proposed simulators are fully implemented in hardware it is also possible to implement them within a real-world embedded system, an aspect also covered in Chapter 6. A PLRUt cache simulator, MASS{plrut}, is presented in Chapter 7, though this tool was built in software as the optimisations presented are ill-suited for hardware implementation. Finally, Chapter 8 covers a trace compression algorithm that can be employed to significantly speed up trace-based cache simulation.

# Chapter 2

# Literature Survey

This thesis focuses on three main areas of research:

- minimising the detrimental effects of temperature in digital circuits through application adaptation,
- a study of temperature within FPGA-based systems, evaluating which functional components contribute the most towards FPGA self-heating, and,
- most importantly, the study of cache simulators and methods to improve them for greater performance and flexibility.

The related work of each of these points is analysed throughout this chapter.

## 2.1 IC Self-Heating and What To Do About It

In order to limit the overheating of digital electronic circuits numerous techniques have been developed and researched. The resulting improvements are situated either in the field of chemical materials and IC technology or actual system design. While the former are understandably reserved to chemical materials and manufacturing engineers, the latter involves attention to circuit routing and hardware/software design.

The cause of the self-heating of the CMOS transistor within a digital circuit was covered in Chapter 1: temperature increase is proportional to power consumption which is made up of a static power and a dynamic power component. Some methods used to lower power consumption can therefore also be used to target temperature control and temperature-aware circuit design. This section covers dynamic voltage and frequency scaling and application adaptation though other methods are also mentioned at the end.

### 2.1.1 Dynamic Voltage and Dynamic Frequency Scaling

Dynamic power consumption was the main cause of self-heating throughout most of the integrated circuit's history. Even as late as 1998, Yeap [5] stated that "dynamic power dissipation is several million times larger than leakage current". Research therefore focused on dynamic power consumption, and especially the capacitor power equation (Equation 1.1 presented in Section 1.1). Dynamic voltage and dynamic frequency scaling, or a combination of both (known as Dynamic Voltage and Frequency scaling, or DVFS) became a common way to lower the power consumption of a processor at the cost of processing power. Lowering the voltage decreases the amount of charge stored in the transistor capacitance and also lowers the short-circuit current, thereby decreasing the energy consumption. The frequency at which these capacitors charge/discharge and at which short circuits are caused depends on the frequency at which the circuit is driven. Lowering the frequency will therefore also have the obvious consequence of lowering the dynamic power consumption.

There is a limit however as to the extent that these factors can be changed: transistors require a minimum voltage in order to function correctly, lower voltages induce greater leakage power and reduce the maximum switching speed of the transistors. It is the latter that is very important, as it sets the maximum frequency at which an IC can be clocked. Many DVFS schemes lower voltage and frequency together, thereby lowering processing capabilities linearly while decreasing power consumption exponentially.

Dynamic voltage and/or dynamic frequency scaling has been extensively researched [21–

34]. In [21], Jones et al. witnessed the catastrophic failure of one of their FPGA boards caused by the self-heating of their chip. They managed to resolve the issue by adjusting the frequency at which the system was running depending on the temperature of the device. A similar technique was employed in later work by Jones, targeting a multi-core pattern recognition system on FPGA [22] and including a new feature: if overheating still occurred after the lowest frequency had been reached, a rough form of Application Adaptation (see section 3.3) disabled low-priority processing cores to lower the dynamic power consumption.

Choi et al. in [23] present a DVFS scheme that uses predictions on upcoming workload in an MPEG decoding application. Using these predictions, frequency and voltage levels are dynamically selected allowing the application to meet its deadlines. This is a little similar to [24] except that Aydin et al. statically compute the workload of periodic tasks instead of predicting them. The main difference between these two approaches lies in the nature of the application: Choi et al. aim for applications that display a large amount of variation in workload (and therefore also extremely large worst-case workloads that depend on the input data), while the tasks employed by Aydin et al. have comparatively equal workloads during their operation.

Other research by Choi et al. [25] uses a technique named 'workload decomposition' which is a different form of application controlled DVFS. The ratio of the on and off-chip workloads is collected by a performance monitoring unit, and is used to determine the frequencies assigned to the CPU, the internal bus and the external bus, i.e., higher clock frequencies are allocated to the hardware components that are doing more work.

Chow et al. [26] discuss a dynamic voltage scaling method that uses feedback from an on-chip thermal sensor. As the delay of transistors is both voltage and temperature dependent, the DVS scheme proposed runs the IC at the lowest possible voltage given a certain temperature. The main inconvenience of this approach is that extensive tests need to be run for every digital design and technology to determine the temperature thresholds at which to switch voltages.

The progressive move from single-processor to Multi-Processor Systems on Chip

(MPSoC) and the emergence of Networks on Chip (NoC) has made space for new methods to control IC temperature. Martinez and Atienza [34] describe such a form of temperature management. The temperature of each processing element (PE) is measured and sent to a central thermal management unit using the NoC interconnect. This temperature measurement is used together with local communication statistics and workload prediction to set the voltage and frequency levels of the individual cores.

### 2.1.2 Application Adaptation

The mechanism behind Application Adaptation is extremely simple: processes within an application vary their resource demand by varying their Quality of Service (QoS) in order to meet resource constraints. These resources may be connectivity bandwidth, processing power, memory, temperature etc.

The Odyssey tool [35–38] provides an API intended to simplify the communication between the high-level application software and the low level resource information. The application lets the API know which resources it is interested in. Odyssey then monitors that resource and notifies the application asynchronously when given tolerance bounds are reached. [35] gives an example use of Odyssey with a video and mapping application.

[36] demonstrates the use of Odyssey in mobile applications for a video player, a web browser and a speech recogniser. The paper also delves into the finer control details of the application adaptation, simulating bandwidth perturbations in order to estimate the effectiveness of the API.

A rough form of application adaption for thermal control is presented by Jones et al. in [39] and is combined with DVFS in [22]. The system is implemented in an FPGA and contains four parallel processing elements running an image recognition task. Each processing element contains two masks that store a feature they are meant to recognise. All masks are ordered by feature priority, before being assigned to processing elements in pairs. If the FPGA becomes exceptionally hot and DVFS is insufficient to lower the chip temperature, low priority feature masks are disabled one by one to lower the power

consumption and temperature. If both masks allocated to a processor are disabled, the associated processor is also disabled to further reduce the temperature. The change in QoS that the application performs affects the number of features that the system can recognise.

Application adaptation is also at the centre of the research performed by Peddersen et al. [40]. The resource under constraint is power, as the application tries to keep the current drawn from a battery constant to maximise its life expectancy. The applications performed are JPEG, MPEG and LAME MP3 encoding, the power consumption of which are strongly dependent on the raw media data. The power is measured using performance counters. Application QoS parameters are modified during operation such as lossiness, DCT algorithm quality and colour depth for JPEG encoding to meet the given power target. For the MPEG encoding application, adaptation is performed on the search space used for backward and forward prediction and the amount of motion prediction. Huffman coding, quantisation, noise shaping, psychoacoustics and side stereo coding are adapted for LAME encoding.

Peddersen et al. go into great detail describing the low-level programming techniques that can be used to perform application adaptation in the C programming language. Another point taken into account is the rate at which the QoS is changed. The adaptation algorithm employed attempts to minimise the quality jumps, even when the processor is faced with large jumps in processing requirements, to give the user a smooth media experience.

### 2.1.3 Other Methods to Lower IC Temperature

Only the most relevant research on lowering the temperature in ICs has been covered so far. For completeness, task remapping and temperature aware routing are quickly covered here.

With the rise of multiprocessor systems (and MPSoCs), Task Remapping has become a popular area of research to minimise the impact of self-heating [41–48]. The principal idea behind it is that tasks are moved from one processor to another in order to distribute

the power consumed over a larger area of the die. As a consequence individual cores can be clocked at higher frequencies; when a thermal threshold is reached a processor's tasks are remapped to another core to allow it to cool down. This results in an overall higher throughput and an even temperature distribution across the die.

Temperature aware routing is a step that can be performed at design time or in the field for reconfigurable systems and is the process by which hardware block and signal wire placement are taken into account during IC design to minimise self-heating. Many different types of algorithms have been developed over the years to optimise these placements [49–54]. Two steps are generally required for temperature aware routing which are evaluating the temperature distribution on the die (in thermal simulation or hardware) and then routing the system depending on these measurements.

## 2.2 FPGA Thermal Sensors and Characteristics

The ability to study the thermal characteristics of ICs and to perform temperature management and control strongly depends on the ability to measure temperatures accurately. Though many ICs have a built-in temperature sensor, it is often important to measure the temperature at many different locations within an FPGA for more fine grained thermal management. Sensor accuracy is of importance: it was found that a 1% error margin in sensor accuracy introduced slowdowns of 6-13%, and up to 75% performance loss in the dynamic thermal management system presented by Skadron et al. [30]. Below is a description of the different known means by which temperature can be measured.

### 2.2.1 Temperature-Sensing Diode

As previously mentioned, numerous ICs contain a thermally sensitive diode for silicon temperature measurement. The devices themselves are cheap to implement though additional circuitry is required for clocking, control and analogue-to-digital conversion (ADC). They also present the advantage of being fairly resistant to power supply voltage

fluctuations. High-end Xilinx® and Altera® FPGAs contain one such circuit [55]. For ideal FPGA thermal measurement many such sensors would need to be deployed within the die, yet such devices do not currently exist. Other methods for temperature sensing therefore need to be investigated.

### 2.2.2 Clamping Diode

One approach mentioned by Lopez et al. in [56] suggests measuring the junction forward voltage of a clamping diode. These are present at every input/output (I/O) pad of an FPGA in order to protect the IC from low-power voltages that are outside the chip's range. As most modern FPGAs are fabricated in a Ball Grid Array (BGA) package, with some models containing over a thousand I/O pins, it is possible to place a temperature sensor almost anywhere on the IC, assuming that there is an I/O pad at the right location that is not being used by the design.

Lopez et al. point out that such an approach would also be very resistant to power supply voltage fluctuations, though they also state that diodes are less linear than ring oscillators. The main disadvantage is also clear: this setup requires a significant amount of external circuitry to provide a constant negative bias voltage or current, a means to measure a voltage with high-precision (as the relation between voltage and temperature at a constant current of 5mA is 1.14mV/C) together with circuitry to communicate the measured value to the FPGA digitally. However, given the amount of process variation (and therefore, voltage supply variation) in the latest technology ICs of today, using the clamping diode for thermal measurement may still be advantageous.

### 2.2.3 Delay Line

Measuring the temperature with a delay line is possible as IC delay is strongly affected by temperature in a mostly linear way. Theoretically it is therefore possible to measure the temperature by timing the responsiveness of a circuit. The main difficulties presented by this approach are accurate timing and strong dependence on power supply voltage.

There are however clear advantages: this circuit can be configured anywhere on an FPGA at minimal hardware cost, the implementation is executed in the configuration software (no external components required) and the output is in digital format without having to perform analogue to digital conversion. One of the few research groups to have used delay lines are Chen et al. In [57] such a temperature sensor using 140 FPGA logic elements is described with a minimum error of -0.7 to 0.9C following a second order curvature correction of the measured values. In [58] the sensor design only uses 48 FPGA logic elements, and its performance is compared to that of a ring-oscillator. Their findings show that ring oscillators are more accurate than delay lines. It must be taken into account that the FPGA used for the experiments is low-tech, and that power supply voltage fluctuations are therefore much less likely to occur.

A delay line is also implemented by Chow et al. [26], as was described in Section 2.1.1. The purpose of the sensor in this case is not to measure the temperature but the overall maximum speed achievable given the current voltage/temperature of the IC.

### 2.2.4 Ring Oscillator

Ring Oscillators (ROs) have been widely used as the most popular way of measuring the temperature within a digital circuit, be it in FPGA hardware or simulation [3,22,30,41,44, 48,56,58–62] and are patented by Xilinx [63]. Their functioning relies on the same physical phenomena as delay lines. A small number of buffers/inverters are chained together in a ring configuration containing an uneven number of inverters. This ring oscillates at the highest frequency possible given the thermal and voltage conditions. Assuming a steady supply voltage, one can therefore estimate the temperature by measuring the oscillating frequency of this setup.

The advantages and disadvantages are identical to those presented by delay lines. ROs are characterised by great flexibility and low hardware cost while being plagued with sensitivity to supply voltage levels. This disadvantage was pointed out as early as the year 2000 by Lopez Buedo et al. [56]. Jones et al. [22] also noted this effect as different

applications with different processing requirements caused the supply voltage to fluctuate, making the ring oscillator readings unusable for temperature measurement. To avoid this issue they implemented a pause phase where the processor halted for a certain amount of time to let the transients dissipate in the hopes that the supply voltages stabilise. Needless to say this pause in processor operation incurred a performance penalty, the magnitude of which was 1% in the case of Jones et al. who performed a temperature measurement every 50ms.

Zick et al. [64] pointed out that the greatest problem faced by ROs today is the advancement in technology. As the transistor size decreases, the effect of temperature on delay becomes much less significant. Worse still, lower supply voltages and process variation dramatically increase the noise ratio on the power supply lines. Accurately measuring the temperature using an RO is difficult at 65nm, and is predicted to be near-impossible at 28nm.

### 2.2.5 Power Meters and Event Counters

As the temperature rise in a circuit is proportional to its power consumption, it should theoretically be possible to measure the temperature of hardware blocks by measuring the power they consume, assuming that initial temperature conditions are known. Power measurement techniques have been researched at all levels of design [65–71]. The power consumption of different instructions is determined by Tiwari et al. [65] by isolating the processor so that the power consumption of software can be optimised to draw a minimum of current. Nikolaidis et al. [67] use the same approach, taking their CPU measurements with a high-speed and high-accuracy automated test-setup.

In [70], integrated performance counters are used in an Intel XScale processor. These hardware counters present the advantage that they are already integrated into the system. Having estimated the power consumption of the hardware blocks under different parameters, count values can be collected during operation and computed to estimate the current drawn. This technique was found to be accurate within 4%.

Peddersen et al. [71] strategically placed their own performance counters in hardware by running simulations to analyse events. Separate hardware modules were analysed and ranked by their power variation. To minimise the amount of hardware used by the counters, only the modules displaying the largest amount of power variation were monitored. This setup yielded a power estimation error of 2% and an energy estimation error of 1.5% requiring a 4.9% increase in chip area and, on average, 3% more power.

A rare instance where activity monitors are used for temperature measurement is found in [72]. This paper presents an emulation framework with an MPSoC implementation on an FPGA. Statistics are extracted by hardware sniffers during operation and sent through a standard Ethernet connection to a host PC running software for thermal modelling. The computed temperature is returned to the FPGA to test different temperature management strategies. The main disadvantage of this setup is the processing needed for the thermal modelling, requiring an external PC to be computed.

On the whole, power meters are an inadequate method to measure on-chip temperature. As pointed out in [16], localised heating can only be detected when statistics are extracted at the granularity of on-chip blocks, and the thermal modelling needs to take into account spatial distribution, block coupling, heat sinks etc.

### 2.2.6 Thermal Simulation

Obtaining accurate temperature measurements that are not affected by external factors (fluctuating ambient temperature or power supply voltages) is difficult, which is why some researchers turn to thermal simulation to get an idea of how their chip is heating up. IC thermal modeling software tools go to great lengths to create thermal models of all the different parts that an IC is made of (e.g., die, pins, packaging, heat sink, leakage power). Each model has its margin of error, and these errors are cumulative. The error is aggravated when observing transient temperatures as errors of thermal capacitors are made all the more important. On top of that, the power consumption of a block which is used as an input to the model is provided by software tools, such as Wattch [73], which contain a

certain amount of error [74]. The extent of the overall error is very difficult to estimate, as instances of published comparisons between static and transient simulation results and real-life die implementations do not exist. HotSpot [75] was validated by comparing it with finite element analysis tools such as ANSYS [76] and FreeFEM3d(FF3d) [77]. This proved that the results given by HotSpot, in comparison to those given by ANSYS, were accurate within approximately $-2°$C for a given system with a good thermal interface material, and within $+4°$C and $-1°$C for the same system with a bad thermal interface material. It must be remembered, however, that ANSYS is also but a simulation tool, containing a certain margin of error, and that the power data for both tools was extracted using Wattch or similar.

Other software developed to model temperature include 3D Thermal-ADI [78] by Wang et al., a 3D transient thermal simulator based on the alternating direction implicit method. Zhan et al. [79] proposed observing temperature effects by using the green function method. Neither of these methods present any form of temperature validation.

Atienza et al. [80] introduced an interesting approach that emulates a system in an FPGA where performance counters provide power consumption estimates to a PC running a thermal model in real time. The clear advantage of such an emulation framework is speed. What is not so clear is how accurate the thermal model really is, as it was calibrated against a 3D finite element analysis tool, and validated against HotSpot with less than 3°C error difference in the best cases.

### 2.2.7 Thermal Camera

Recent thermal research has seen the employment of thermal cameras directed at the exposed die of FPGAs [81, 82]. This allows for a fine-grained and precise measurement of the temperature at any location within the die. The approach also has obvious downsides: temperature measurement cannot be performed in the field as the IC needs to be 'opened' for the die to be exposed and the equipment required comes at a considerable cost. It has however proven to be very useful to study the temperature of digital circuits

in a lab setting. In Chapter 4 a thermal camera is also used for the study of FPGA thermal characteristics.

### 2.2.8 FPGA Thermal Characteristics

An early study on the manner in which FPGA fabric heats up was conducted by Sundararajan et al. [83]. Relying on the results obtained from the HS3D thermal simulator they established the amount of heat generated by the different types of resources that can be instantiated in an FPGA design. As this study was based on temperature simulation the accuracy was limited. Huang et al. [62] went so far as to validate their HotSpot thermal simulation software with a $0.13\mu$m FPGA system. They proposed to spread many ROs across the FPGA die to measure the temperature of different operating blocks. Thermal variations of up to $0.7^{\circ}$C were measured, and the simulated and measured values correlated with errors within $0.2^{\circ}$C. However, the limited range of these results may suggest that this correlation is mainly caused by the heating trend and are insufficient to validate the HotSpot thermal simulator.

Long before FPGA self-heating became a key concern, Boemo et al. [60] proposed ROs as temperature sensors in reconfigurable logic. An array of such sensors was later used to observe the thermal behavior of a $0.22\mu$m FPGA [59] configured with two *soft-core* processors. This showed that a change in processing throughput resulted in a change in temperature though these variations were very small (within $-0.8^{\circ}$C and $+0.7^{\circ}$C from the mean temperature). Similar work was performed many years later by Zick et al. [53] who also configured an array of ring oscillators on a modern 65nm FPGA to analyze process variation and to track the aging of the chip's fabric.

Aided by a thermal camera pointing at an FPGA chip, Cochran et al. [81] presented a methodology to convert the captured images to estimate power patterns and demonstrated the low-pass filter effect the die has on temperature. Nowroz et al. [82] also used a similar thermal camera setup demonstrating that thermal gradients could reach $9^{\circ}$C in their setup. Happe et al. [84] researched the precise design of micro-heaters and their transient

effect on temperature based on the components they are made up from (LUTs and FFs) at different frequencies. None of these papers analyzed the temperature of FPGA-based embedded processors to clarify the responsible parts for the chip's temperature increase, despite the fact that it is considered a key usage of FPGAs [85].

## 2.3 Cache Simulation

CPU caches profoundly impact a computer system's performance and energy consumption as was described in Chapter 1.2.1. Designers of time critical systems need to ensure that their Worst Case Execution Time (WCET) is within acceptable levels. Judiciously selecting the optimal cache configuration can also provide a significant improvement to the functioning of the system, especially if it is designed under stringent performance or energy limitations. Cache behaviour depends on two factors which are cache configuration and the application executed. Two general steps are therefore required for cache simulation: the execution of the application resulting in a certain order of memory accesses and the subsequent cache analysis. An intermediate step is often employed where the trace is stored to a file for later processing by cache simulators.

Often cache simulators are categorised depending on how they are 'driven', with terms such as 'trace-driven' ('off-line'), 'execution-driven' ('on-line') and 'binary instrumentation' being used in the literature. One could argue that such categorisation is confusing as all cache simulators are effectively *trace-driven* and that these categories are therefore unhelpful at differentiating how these simulators function. From this point onwards, cache simulators will be classified as follows:

- cycle-accurate, or cache emulation, where processor and cache interactions are accurately simulated or emulated, allowing for cache statistics and timing information to be reported (formerly described as 'excution-driven'),

- entire-trace, or analytical cache simulation, where the whole trace is required before the cache simulator can perform its task (formerly 'trace-driven'),

- heuristic cache selection, where the successive selection of different caches is used to converge on the optimal cache configuration (formerly 'execution-driven'), and
- continuous, or precise cache simulation, where addresses from the trace are processed one-by-one in the order in which they were accessed (formerly 'execution-driven', 'trace-driven' or 'binary instrumentation' depending on sources).

These distinctions having been made, this section will first look into the approaches used to generate traces before covering the methods employed to simulate caches.

### 2.3.1 Trace Generation

To obtain the trace from an application running on a processor the application needs to be executed in some form or other. As the processor executes the application the successive memory addresses that are accessed need to be sent to a cache simulator or saved to a trace file depending on the type of simulation employed. For continuous and analytic cache simulation the data flow, in the form of the memory access trace, proceeds in one direction only, from the CPU to the simulator. For cycle-accurate cache simulation a cache model needs to be employed where processor and cache interact with each other in a cycle-accurate manner. As a consequence, cycle-accurate cache simulation is very involved and slow but has distinct benefits. For one, the exact timing information can be obtained. On top of that, cache timing is important when using out-of-order execution processors as the order of memory accesses is affected by the delays caused by having to fetch data from memory. As such, the only way to obtain precise cache hit and miss values when using an out-of-order processor is to use cycle-accurate cache simulation.

The Pin tool provided by Intel [86] is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures. It allows the execution of unmodified executables on common multi-core Intel processors while extracting all kinds of data about what is going on within the CPUs, including the instructions executed and the addresses that are accessed. This is achieved by making modifications to the application after it is loaded into memory adding new instructions to perform analysis from within

the program, and then recompiling code with a just-in-time compiler. Other popular but older instrumentation tools include ATOM [87] and Dynist [88]. Based on the Pin tool, CMP$im by Jaleel et al. [89] was able to collect the performance statistics of multiple caches by intercepting the addresses accessed by each CPU core. The Sniper Simulator by Carlson et al. [90] is also based on the Pin tool, giving insight into complex systems containing DVFS, branch prediction, shared and private caches etc. Using the sniper tool it is easy to obtain memory access traces at any point in the memory hierarchy.

Ideally, designers would extract traces directly from a real, hardware-based processor as this would be fast and also accurately reflect program execution in the real world. The tools provided by Lauterbach [91] allow for such fast in-system extraction with the downside that the required hardware comes at a considerable cost. The Altera Nios II [16] provides the option to implement a trace-collecting unit within the processor core. The size of the trace it can store is limited by the amount of internal block RAM allocated to it. The Nios II core can therefore only sample snippets of the trace, having to send the trace data to a host computer on a very regular basis.

One of the most popular methods to obtain the memory trace, especially for Application Specific Integrated Processors (ASIPs), is to simulate the target processor. Often during the creation of ASIPs hardware will not be available at the time when cache configuration needs to be chosen. As simulated processors are implemented in software they are very flexible, and the system can be analysed at the level of detail required by the designer. The processor can be simulated at the most superficial level in an instruction set simulator, a cycle-accurate simulator covering a microarchitecture on a step-by-step basis, or a fully fledged full-system simulator. From the plethora of existing processor simulators, only the most relevant are covered here.

The late nineties saw the development of SimOS by Rosenblum et al. [92] supporting the full hardware simulation of MIPS processor variants. By using a combination of high-speed machine emulation and accurate machine simulation, SimOS was able to run significantly faster than its competitors at the time. At around the same time, the SimpleScalar tool was presented by Burger and Austin [93], capable of simulating computer

architectures at a more or less intricate level, from fast functional simulators down to a more accurate processor model. Since its initial development, the SimpleScalar tool has managed to keep up to date with modern processor architectures with the implementation of ARM and x86 instruction sets, amongst others. Other well-known simulators capable of generating traces are SimICS [94] and the GEMS toolset which is based on SimICS and which targets multiprocessor systems used in databases and web servers [95]. Finally, the Xtensa processor by Tensilica [18] is a customisable processor frequently used in embedded systems research and implemented in modern embedded systems. The simulator provides many tools for in-depth system analysis, custom instructions and Digital Signal Processor (DSP) integration.

### 2.3.2 Cache Emulation

Many cycle-accurate simulators and emulators track the behaviour of an entire system and include cache models. As the cache behaviour is fully mimicked in these models it is easy to add counters to them, counting the number of hits and misses registered. Generally these cycle-accurate simulators are built to study the system as a whole and produce the cache performance metrics as a useful byproduct. The Xtensa simulator [18] processor can produce cache hit and miss values after application execution, as can other simulators such as SniperSim [90] and SimICS [94]. In fact, advanced simulation tools are not even required. It would be possible to download the hardware description for the open source 32-bit OpenRISC [96] processor for example, add cache hit counters (which is trivial), and then simulate the processor at the RTL level using software such as Icarus [97] or Modelsim [98].

Hardware emulation platforms are an interesting alternative as they implement most if not all of the target circuit in hardware. Taking advantage of the inherent parallelism of FPGAs, hardware platforms can run much faster than their software counterparts, especially when researching large, complex circuits. ProtoFlex [99] is one such simulator: running an UltraSPARC processor at a frequency of up to 100 MHz complete with L1 and

L2 caches, complex and infrequent behaviours such as I/O devices are performed in software on a connected computer. FAST by Chiou et al. [100] is a cycle-accurate model for entire computer systems and is capable of running the x86 instruction set. RAMP Gold is another FPGA based architecture simulator where functionality and timing are modeled separately for 64 SPARC CPUs [101]. Ravishankar et al. presented pCache [102], a specialised component designed to take a number of measurements related to the cache performance characteristics. Data memory accesses (i.e., the data trace) was routed through this component which collected information such as read and write hits and misses, average memory access times etc.

The downside of cache emulation as a means to evaluate cache performance is that only one single cache configuration can be analysed at any one time. Given that hundreds of cache configuration combinations are possible, exploring the entire design space would take a considerable amount of time. For a comprehensive cache analysis other types of cache simulators are generally preferred.

### 2.3.3 Analytic Cache Simulators

Analytic cache simulators make estimates as to the hit and miss rates by analysing the memory trace as a whole. Cache performance is predicted by determining memory access locality and data reuse patterns using mathematical models. In other words, general memory access patterns are detected and then used to make an educated guess as to what the cache performance could be. The main advantage of analytic methods is that they are extremely fast with the downside that they are not fully accurate. In fact, accuracy drops considerably once applications are executed that do not follow a regular program execution flow; this applies mainly to applications whose program flow is strongly affected by the application input such as video encoding or data compression. A principal aim of analytic cache simulator designers has therefore been to make the simulator as accurate as possible while keep the simulation execution acceptably fast.

Agarwal et al. [103] presented a hybrid cache model that derived cache miss rates by

categorising conditions that lead to cache misses. This included start-up effects (initial burst of misses caused by an empty cache), nonstationary behaviour (performing a set of identical operations but on a different data set), intrinsic interference (when useful data is discarded due to lack of cache space), and extrinsic interference (cache or program flow are changed due to external events such as I/O or another thread accessing data). A model was made for each of these miss categories which were combined to give an overall miss rate for different caches. The model was not very accurate though, with errors as large as 39% for caches of associativity 2 and line length 16 bytes.

In a similar way, Harper et al. [104] also categorises the types of hits and misses that can occur and models them separately. Performance is extremely fast with errors in the region of 15 to 20%. The ability of the simulator to handle different types of program flow was tested by four different kinds of computations, namely matrix multiply, 'Stencil' operation, two-dimensional Jacobi loop and blocked matrix multiply. A different approach was taken by Pieper et al. [105] where the original source code is annotated before the application is run in an instruction set simulator. As a consequence, this is one of the rare cache simulators that is not trace-driven as the memory trace is not used as the simulator input. Instead the cache behaviour of program fragments is obtained, based on which cache configurations can quickly and effectively be determined. The downside is that code annotation is slow (as slow as "normal address trace based simulation"), but it only needs to be performed once. Li et al. [106] exploited the properties of the lossless Sequitur compression algorithm. The Sequitur algorithm [107] detects data patterns in an input string and represents it as a context free grammar. A memory trace compressed in that manner will have the memory access order efficiently represented as sets of patterns and repetitions. All the simulator needs to do is calculate the cache hits for these patterns or snippets and then determine the order in which these patterns occur to calculate the total hit count. What is not mentioned in this work is the time taken to compress the trace in the first place, a process that effectively does much of the heavy lifting this simulator relies on.

The simulator by Ghosh et al. [108] does not perform exhaustive cache evaluation

but instead uses the desired cache hit rates as an input and determines cache configurations that satisfy the desired hit and miss rates. Guo and Solihin appear to be the first researchers to take into account the cache replacement policy in an analytical simulator [109]. Statistical properties of application access patterns were determined and combined with replacement probability functions which are capable of approximating a number of different replacement policies. An average simulation error of 1.41% was reported which is surprisingly accurate, though it must be noted that the simulation time is not given. A probabilistic approach was taken by Liang et al. [110] who also use a generalised binomial tree presented by Sugumar et al. [111]. This research is some of the only research in analytical cache simulation to make a direct comparison with fast precise simulators. Another point of interest is that apart from Guo and Solihin, the vast majority of analytical cache simulators assume the use of caches employing the LRU replacement policy. The LRU replacement policy however is rarely implemented in embedded systems due to its high hardware cost (See Section 6.1).

### 2.3.4 Heuristics-Based Cache Optimisation

Heuristics-based cache simulation is also an estimation method, and is used to determine a cache configuration that, though not optimal, will be good enough for the current design. Fornaciari et al. [112] presented such an approach where an instruction set simulator is connected to a hierarchy of reconfigurable caches and evaluators measuring cache performance and energy consumption. After a one-off sensitivity analysis the framework learns the 'tuning information', the ways in which a change in cache parameters affects system performance. As the application executes, performance statistics are collected which are then combined with performance targets and tuning information to iteratively produce a more optimal cache configuration. After successive guesses a suitable cache hierarchy is found without exhaustively exploring the design space which yields enormous speedups.

The research surrounding dynamically-reconfigurable caches, i.e., caches that can change configuration while executing an application in the field, has heavily focussed

on heuristics as a means for determining the best cache at runtime. Such an approach is often described as 'cache tuning'. Zhang et al. [113] presented a self-tuning cache architecture: energy is evaluated based on cache performance metrics following which a hardware-based tuning heuristic algorithm suggests a better-suited cache configuration. Similarly, Gordon et al. [114] also changed cache configuration on the fly but instead of using consecutive guesses designed a 'one-shot' tuner that could select an adequate cache configuration in one single iteration. It is not based on heuristics as such but the general problem approach is identical.

Tony Givargis [115] developed a heuristic algorithm that analyses cache accesses at design time to establish optimal cache indexing for a given application. In other words, the address bits used for cache index selection are connected in such a way that the cache suffers the fewest misses. If, for example, an extremely simple application consecutively accesses addresses 0x01 (1) and 0x71 (113), a conventional direct-mapped cache would require to have a minimum size of 128 bytes to avoid cache thrashing. If however the seventh bit is used as an index (which is '0' for 0x01 and '1' for 0x71) instead of the lowest 7 bits of the address, a cache of only 2 bytes would be sufficient. By optimising cache indexing, performance can be improved at no additional cost.

### 2.3.5 Precise Cache Simulators

All cache simulators presented until now are either slow (cache emulation) or make only estimates of the cache hit rates. Slow cache simulation is a big problem when simulating complex applications as exploring the design space can take an extremely long time. It is also necessary for measurements to be precise as even a small deviation in hit rate can significantly alter the effectiveness of a cache [116]. These two requirements have spawned the creation of precise cache simulators which take into account every single memory access to calculate the hit and miss rates. Exhaustively analysing the effect of individual memory accesses requires a certain amount of computing power which is why the field of precise cache simulation has seen many proposed optimisation methods.

One such method is to simulate multiple caches at the same time resulting in single-pass simulators (also known as multiple cache simulators) as the trace file is only read once as opposed to the multi-pass simulators who need to re-read the trace for every distinct cache configuration.

Mattson et al. [6] were one of the first to look into storage size and hierarchy evaluation, presenting methods to determine the 'success function' of multi-level storage devices. To do this they presented a stack algorithm where the two-level storage hierarchy is depicted as a stack. Any replacement policy can be simulated as long as the contents of smaller storage devices is contained in larger storage devices, a characteristic defined as an 'inclusion property'. In modern terms (the article was written in 1970), the storage hierarchy described is a two level fully associative cache with constant line length. By adding up the stack distance of the accessed items it is possible to easily calculate how effective different storage size combinations are.

Hill and Smith [117] proposed the forest simulation algorithm and use the all-associativity algorithm (a stack algorithm generalised to arbitrary set-mapping functions) specifically targeting direct-mapped, set-associative and fully-associative CPU caches (see Section 1.2). The precise simulators presented until now mainly focus on the LRU replacement policy and make use of LRU cache properties, though these cache properties have different names depending on the source. In this thesis, these properties are called the *LRU cache inclusion properties* and are formally defined in Sections 5.3.1 and 5.3.2 and are summarised as such:

- Given two caches of same set size and line length, the data of the cache of smaller associativity will always be a subset of the cache of larger associativity.
- Given two caches of same associativity and line length, the data of the cache of smaller set size will always be a subset of the cache of larger set size.

Sugumar et al. [111] proposed what they call a Generalized Binomial Tree, a data structure that very efficiently stores the cached tags in such a way that they can quickly be scanned. It heavily relies on the LRU inclusion properties, as each data tag that may

be held in any of the caches simulated is only stored once. Caches are searched from the smallest to the largest. As soon as a matching entry has been found the larger caches no longer need to be simulated as they also contain that entry. The algorithm was implemented in the Cheetah cache simulator which is available for free as a piece of open source software. DineroIV [20] is another open source cache simulator capable of simulating LRU, FIFO and Random replacement policies. The simulator can analyse 2-level instruction and data caches and is easy to configure. The downside of DineroIV is that it is a multi-pass simulator and that it is therefore excessively slow for design space exploration purposes. Due to its free availability it is often employed as a reference point when assessing the performance of cache simulators.

Janapsatya et al. [13] also employed LRU inclusion properties in their forest of trees data structure, simulating many LRU caches at the same time. One of the main contributions of this paper is that it provides equations to calculate energy and performance metrics based on cache hit rates. Subsequent LRU cache research ( [14, 118]) considered Janapsatya's algorithm to be the fastest at the time (in 2006) though this assertion is questionable as no comparison was made with respect to the Cheetah simulator from 1995. Tojo et al. developed the CRCB algorithm [118] which encompasses Janapsatya's implementation and augments it with two simple optimisations, named CRCB1 and CRCB2. Had Tojo et al. inspected the Cheetah source code more closely they may have come to the realisation that both CRCB1 and CRCB2 had effectively already been implemented in Cheetah in 1995. Haque et al. [14] took a different approach for their SuSeSim LRU simulator in that the binomial tree is searched from the leaf node to the root node (bottom to top) instead of from the root to the leaf like previous simulators. The simulator was sped up through the use of contrapositions of the LRU cache inclusion properties, i.e., the definition of cases where data would definitely *not* be present in smaller caches. Impressive speedup results were given with respect to the CRCB algorithm though comparisons with Cheetah were, again, lacking.

Up until 2010 the effect of cache replacement policy had mostly been ignored in the design of single pass simulators. This can be explained by the fact that only the LRU

replacement policy contains obvious inclusion properties making LRU caches an easy target for optimisation. Haque et al. presented a number of FIFO cache simulators, first DEW [119], then SCUD [120] and finally CIPARSim [121], with each improving on the former. DEW represented the caches in a binary tree and used wave pointers to keep track of where the data for a given address is stored in different caches. SCUD dropped the wave pointers and instead used a central look up table in combination with a binomial tree. It is CIPARSim however that is the most effective and interesting of the simulators as it makes use of a number of FIFO cache intersection properties. The intersection properties are presented as a list of certainties (for example, data block is present in larger cache) that hold true if a number of conditions are met. Tawada et al. [15, 122] also dabbled in the conception of FIFO and PLRU cache simulators by applying the CRCB algorithm to these replacement policies together with a priority queue, making recently accessed data (which is more likely to be re-accessed) quicker to find. The optimisations presented could be considered trivial, and no performance comparison was provided with respect to the work by Haque et al.

A hash table-based implementation was presented by Chen et al. [123] with their LRU simulator HC-Sim. They claim their simulator is 2.56 times faster than SuSeSim which is questionable as the memory trace is obtained from an instrumentation tool, Pin [86], meaning that the timings reported in their results section hardly reflect cache simulator performance as the creation of the trace is the bottleneck of this setup. In his thesis, Mohammad Haque [116] proposes PSAICO, a PLRU cache simulator which improves on the PRLU simulator by Tawada et al. [15] by stating that the *two* most recently used tags will also be stored in all caches of larger associativity and set size assuming a constant line length. Zang et al. [124] developed a space-efficient stack-based algorithm capable of simulating both level one and level two caches simultaneously. It is one of the few precise simulators to evaluate multi-level caches.

### 2.3.6 Parallelised Cache Simulators

The rise in multi-processor systems gave birth to another category of cache simulators that leverage the ability to execute code on multiple processors at the same time. As early as 1990, Philip Heidelberger [125] sped up cache simulation by splitting the trace into non-overlapping partitions and simulating them separately on different processors. A problem that then arises is that the state of the cache at the beginning of a trace snippet is unknown, leading to inaccuracies. Heidelberger largely overcame this issue by performing short 're-simulations' at the end of the main simulation runs. For every trace partition, the re-simulation takes into account the state of the cache at the end of previous trace snippet.

Han et al. [126] made use of the abundance of CUDA cores in general purpose Graphics Processing Units (GPUs) to speed up the simulation of multiple caches. Their GPU simulator relies on the fact that given a set in a cache, a small subset of the entries in a trace will affect that set, and that set only. By mapping sets to GPU cores simulation was sped up by a factor of 2.76 with respect to single-pass simulation. Ma et al. [127] also employed GPUs yet their approach was more similar to that by Han et al. [126] in that the trace was split up and distributed amongst the CUDA cores resulting in a speedup of only 1.91 times.

### 2.3.7 Trace Reduction and Compression

Any simulator that bases its calculations on a trace obtained from a file will be spending a considerable amount of time accessing the file from the harddrive. This affects cache simulation time but also makes optimised cache simulation algorithms look less impressive: if two cache simulators, one normal *sn* and one optimised *so*, take different amounts of time to process a given trace assuming zero disk access time, $t_{sn}$ and $t_{so}$ where $t_{sn} < t_{so}$, then the simulation speedup due to improvements in the optimised algorithm are equal to $t_{sn}/t_{so}$. However, if trace file access takes a constant time $t_{fa}$ then the real duration of the cache simulation runs will be $t_{sn} + t_{fa}$ and $t_{so} + t_{fa}$ resulting in a lower overall speedup. It is therefore in the interest of developers of cache simulation algorithms to reduce $t_{fa}$ as

much as possible.

Reducing the disk access time can be done in two ways: by employing faster trace memory (such as Solid State Drives, known as SSDs) or by making the trace file smaller through trace reduction or compression. Alan Smith [128] reduced the size of the memory trace to increase the performance of certain paging algorithm simulators. For example, some paging algorithms ignore any accesses that result in a hit in the cache. The trace could therefore be traversed and all the memory accesses that cause a cache hit could be deleted from the trace. Wang et al. [129] also determined a method by which a trace could be made shorter if misses are counted. Assuming a direct mapped cache and a constant line length, the misses of a smaller cache are a superset of the misses of a larger cache. In a first simulation run of the smallest target cache, only the memory accesses causing a miss are written to the new trace file. As a result, simulating the larger caches can be done on a trace that is up to two magnitudes smaller while still calculating exact cache miss rates. By the same token, Wu et al. [130] also stripped the memory trace after an initial simulation run, removing entries that had no impact on the result of subsequent simulations.

Though trace stripping is an effective way to reduce the trace size, it can only be applied to direct mapped or LRU caches, and only if all simulated caches have constant line length. Trace compression retains the information of all memory accesses but finds a way to represent them in a more compact manner. Memory addresses are large (at least 4 bytes are needed to represent a 32-bit memory address) but often occur consecutively or in patterns. Johnson et al. [131, 132] took advantage of these facts to encode the offset, or difference, from one memory access to the other which is usually much smaller. Luo et al. [133] also encoded address offsets using data types of different sizes depending on the size of the address difference.

Zhang et al. [134] labelled a static program representation with profile information to obtain a program control flow graph. Each item in the graph is a block of memory accesses which then only needs to be represented once instead of being repeated at each occurrence. The work by Li et al. [106] which was covered in Section 2.3.3 used the

Sequitur compression algorithm where the compressed trace is efficiently represented as a series of patterns in a context free grammar. However little mention is made of how long the trace compression process actually takes. Janapsatya et al. [135] used a similar approach with the difference that symbols could appear more than once in a single grammar. This results in a less efficient compression algorithm than Sequitur but which is much quicker to execute. This brings up the important notion that trace compression is only beneficial if the process of compressing the trace does not take too long. Aleksandar and Milena Milenkovic [136] took compression and decompression time into account in their single-pass trace compression algorithm. The general program flow is detected and split into streams, trying to detect blocks of memory accesses that start and stop with the same address. These streams are then compactly represented.

# Chapter 3

# Application Adaptation on FPGAs

## 3.1 Introduction

The need for temperature control within FPGAs has become increasingly apparent, highlighted by the 2006 study by Jones et al. documenting the catastrophic failure of a development board that was caused by FPGA self-heating [21]. In this particular case the self-heating of the FPGA caused the deformation of the circuit board it was placed on, leading to short-circuits and subsequent component failure. Jones et al. were capable of mitigating the issue of over-heating by making their design temperature aware. In their design the operating frequency of the circuit is lowered as the temperature increases, a method commonly referred to as Dynamic Frequency Scaling (DFS). The same authors also researched a rough form of application adaptation [22] where the Quality of Service (QoS) of the application is changed in order to lower device temperature. Their approach is as follows: if the temperature needs to be lowered in a system performing multi-core pattern-matching, the lowest priority processor is switched off.

This chapter focuses on temperature control within FPGAs that perform JPEG image encoding. For these purposes, a special adaptive JPEG encoding implementation was designed for resource-constrained embedded systems (Section 3.2). That same JPEG

encoder was presented in 2013 at the Design Automation and Test in Europe conference [137]. After initial experiments on an older generation FPGA (covered in Section 3.4), JPEG application adaptation was then tested on a multicore SoC instantiated within a modern FPGA (Section 3.5). The contributions are:

- a highly adaptive JPEG encoder with the following advantages:
  - it has a very small footprint, requiring 20 to 27kB of ROM and a minimum of 5 to 9kB of RAM, depending on the processors tested,
  - it can easily adapt its QoS between frames, and
  - the adaptation significantly alters the processing requirements of the DCT algorithm. This is done by combining different luma and chroma subsampling ratios, switching between a fast yet inaccurate, and a slow accurate DCT alogrithm and performing downsampling by averaging, or directly through a 16x16 DCT.
- the observation that application adaptation alone has little effect on the temperature of an FPGA, and
- that application adaptation can be effective at controlling the temperature of an FPGA when combined with frequency scaling and clock gating.

But first, a little background on JPEG encoders.

### Background on JPEG Encoders

Digital image compression is often essential as it reduces the image data size by at least an order of magnitude with very little loss in quality. This comes at the expense of computational and memory requirements as the Discrete Cosine Transform (DCT) employed in JPEG encoding performs many multiplications and memory accesses. As a result, executing JPEG encoding on a small embedded processor with varying power, bandwidth or throughput constraints can be problematic. Tailoring a system for the worst

case constraints is not a good solution as the processor will spend most of the time under-performing.

Peddersen et al. [40] were capable of meeting power targets, in simulation, through application adaptation in a number of applications including JPEG encoding. The source code used was developed by the Independent JPEG Group [138]. This software, though extremely flexible, instantiates data memory in the order of megabytes and also requires a large amount of ROM. This makes it unsuitable for very small embedded systems. Other available open source JPEG encoders, such as the Embedded JPEG Codec Library [139], jpec [140] and Jpegant [141], are not capable of varying the output quality in any way. It is also important to note that what is conventionally referred to as JPEG *quality* is in fact a reference to the amount of data compression performed. This changes the size of the quantisation factors which has a direct impact on the low-power Huffman encoding stage and the file size. Independent of the compression chosen, typical implementations do not change the process-intensive DCT stage which requires a fixed, large amount of computation. For this reason, the focus here is on quality variation at the DCT level which can easily be combined with the variation of compression. The most suitable candidate for our purposes from available open-source software is "jpeg-compressor" [142] as both the compression and DCT levels are adaptable. It is important to note that one of its shortcomings is the use of dynamically allocated memory, which is often undesirable in small embedded systems. Also, the options for quality variation of a colour image on the DCT level are limited, as it relies solely on differing chroma subsampling ratios (i.e., three quality levels 1x1, 2x1, 2x2).

## 3.2 JPEG Encoding Application

The stages of JPEG encoding can be seen in Figure 3.1. Variation of the application is primarily achieved by selective downsampling of the YCbCr components (by a factor of 2 in both horizontal and vertical directions) and changing the DCT algorithm used. In the JPEG encoding process, an image is first converted from the RGB into the YCbCr

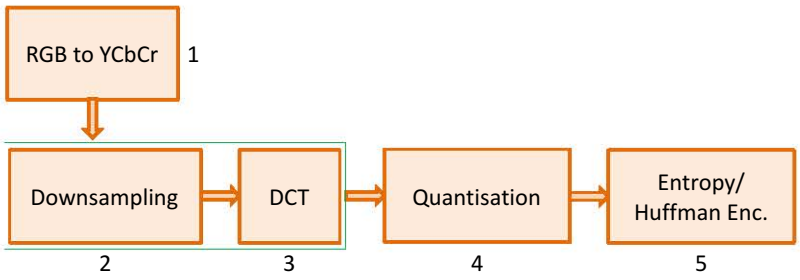



**Figure 3.1:** Different Stages of JPEG encoding.

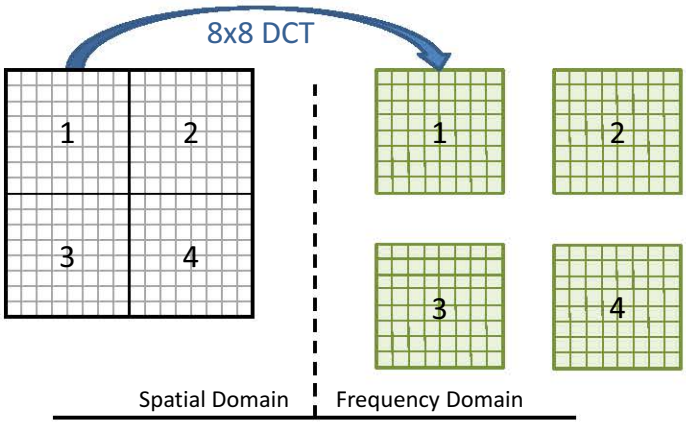



**Figure 3.2:** Discrete Cosine Transform of an MCU block that is not downsampled.

colour space (stage 1) before being subdivided into Minimum Coded Unit (MCU) blocks. Only a 16x16 MCU block size is used in the new JPEG encoder. If a component is not downsampled (Figure 3.2), the MCU of that component is divided into four 8x8 blocks. Four 8x8 DCTs are then performed producing four 8x8 arrays of DCT coefficients.

Quality variation of the 8x8 DCT is accomplished by switching between a fast yet inaccurate, and a slow accurate algorithm. The slow algorithm is characterised by using 12 multiplications per pass, whereas the fast algorithm uses 5 multiplications. Note that 16 passes are computed for each 8x8 block. Downsampling, the process of reducing the DCT output of an MCU to one single 8x8 array, is performed in two ways. Either the entire MCU is computed by a 16x16 DCT algorithm (Figure 3.3) or the MCU is first averaged to obtain an 8x8 array before executing an 8x8 DCT (Figgure 3.4). The former produces the better quality output, though it requires 28 multiplications per pass, and 24 passes.

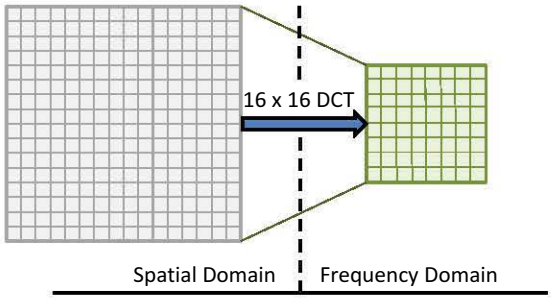



**Figure 3.3:** Downsampling through the use of a 16x16 DCT.

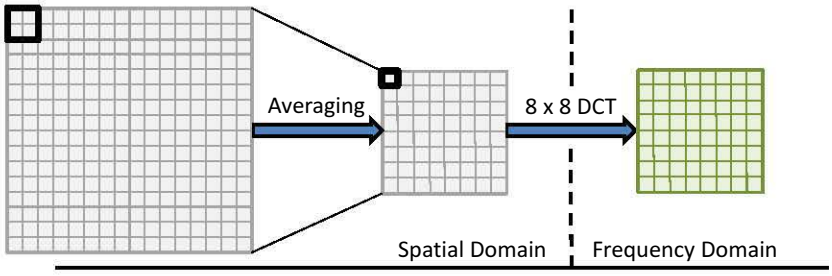



**Figure 3.4:** Downsampling by averaging.

The number of multiplications required for each process mentioned can be seen in Table 3.1. Though the computational requirements of an algorithm are not solely defined by the multiplication count, this table gives a rough idea of the processing requirements of the different approaches. Additionally, the JPEG encoding format allows for different downsampling rates for the different components of the YCbCr colour space. The most common configuration contains no downsampling of the luminance component (Y), while both chrominance components (Cb and Cr) are downsampled by a factor of 4; this is known as the 4:2:0 ratio. A 4:4:4 ratio describes a configuration where no components are downsampled. On the other hand, downsampling all of the components effectively lowers the image resolution by a factor of four, i.e., a 640x480 pixel image is transformed into a 320x240 pixel image with a 4:4:4 ratio. Combining the different luminance and chrominance downsampling ratios and conversion methods, 9 quality levels were created. This can be seen in Table 3.2. By changing the quality level, the application can adapt its processing requirements.

**Table 3.1:** Multiplications required for different MCU transform methods.

| Conversion Method | 8x8 DCT speed | Mults. |
|---|---|---|
| Four 8x8 DCTs | Slow | 768 |
| Four 8x8 DCTs | Fast | 320 |
| Downsampled by 16x16 DCT | N/A | 672 |
| Downsampled by averaging | Slow | 196 |
| Downsampled by averaging | Fast | 80 |

**Table 3.2:** Definition of the 9 quality levels.

| Lvl | Output Resolution | Ratio | Downsampling | 8x8 DCT |
|---|---|---|---|---|
| 1 | 640x480 | 4:4:4 | N/A | Slow |
| 2 | 640x480 | 4:4:4 | N/A | Fast |
| 3 | 640x480 | 4:2:0 | 16x16 DCT | Slow |
| 4 | 640x480 | 4:2:0 | 16x16 DCT | Fast |
| 5 | 640x480 | 4:2:0 | Averaging | Slow |
| 6 | 640x480 | 4:2:0 | Averaging | Fast |
| 7 | 320x240 | 4:4:4 | 16x16 DCT | N/A |
| 8 | 320x240 | 4:4:4 | Averaging | Slow |
| 9 | 320x240 | 4:4:4 | Averaging | Fast |

## 3.3 Performance

An input image size of 640x480 pixels was used for the experiments. From Table 3.3 it can be seen that the compressed file size mostly depends on the chosen YCbCr downsampling rates. The JPEG encoding application was implemented on three different embedded processors: an Atmel AVR ATmega1280 (8 bit), a Microchip PIC24FJ256GB110 (16 bit) and an Altera NIOS II softcore processor (32 bit) running only from on-chip memory with 8kB instruction and 8kB data caches. For fair comparison, the timing values are scaled to represent operation at 16MHz. The source code was compiled with the respective *gcc* compilers optimising for size (-Os). Memory usage can be seen in Table 3.4 while Figure 3.5 shows the timing at each quality level. A compression level of Q=90 was used at each run.

**Table 3.3:** Output File size for the different quality levels

| Quality Level | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Compressed File Size (kB) | 114 | 114 | 96 | 96 | 96 | 96 | 38 | 37 | 37 |

**Table 3.4:** Memory requirements for the different processors (in Bytes)

| | AVR | PIC24 | NIOS II |
|---|---|---|---|
| ROM | 20078 | 26412 | 21924 |
| RAM | 6154 | 4994 | 8236 |

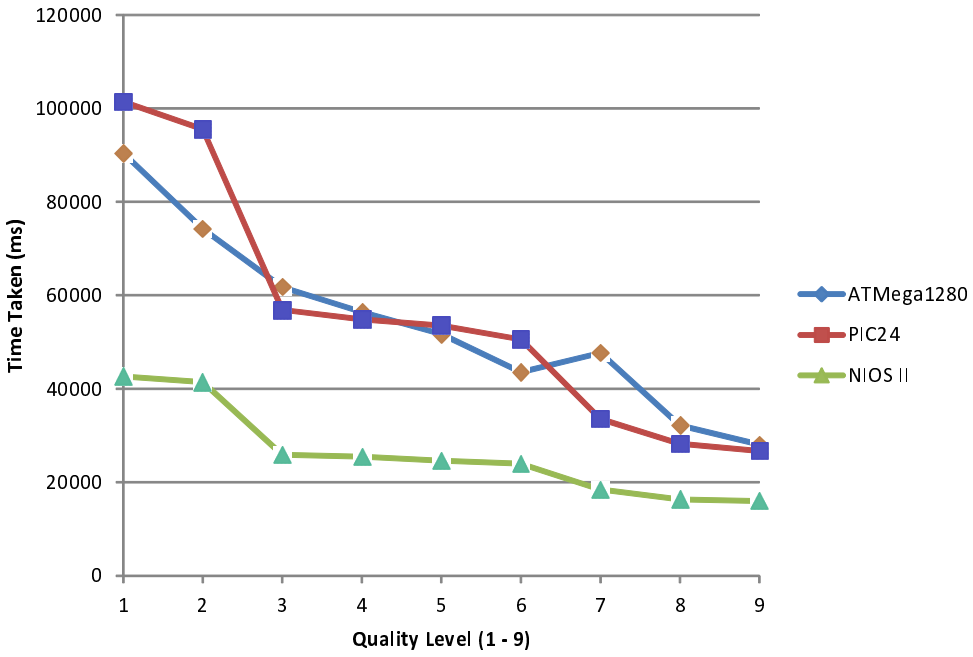



**Figure 3.5:** Time it takes to encode a frame on the different processors. The values were scaled to display operation at 16MHz.

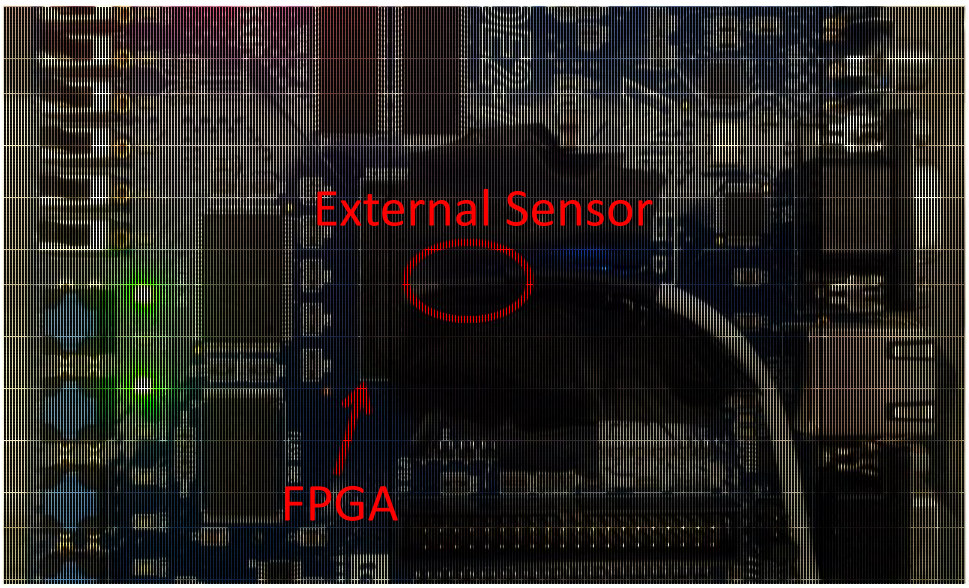



**Figure 3.6:** Photo of the Altera Cyclone 2 experimental setup with temperature sensor.

## 3.4 Application Adaptation on Cyclone 2 FPGA

The first attempt at implementing the adaptive JPEG encoder on an FPGA was performed on a DE2-70 development board, depicted in Figure 3.6, which is centered around an older technology Altera 90nm Cyclone 2 FPGA. As this FPGA does not contain any internal temperature-sensing capabilities a temperature sensor was attached to the outside of the device. Thermal paste was applied to the surface between the sensor and the external housing for better heat conduction. A soft-core System-on-Chip (SoC) was instantiated within the FPGA supporting external SRAM and DDR-RAM memories, a 5 megapixel digital camera sensor and an SD storage card. The temperature sensor was of type DS18B20 from Maxim Integrated [143], capable of temperature readings between $-55°$C to $+125°$C with a 12-bit ($0.044°$C) resolution.

Much like Peddersen et al. [40] the application was adapted by varying the compression quality (the quantisation) and switching between a slow and fast 8x8 DCT. Additionally, subsampling was either done through 16x16 DCT or averaging. In other words, the adaptive JPEG encoder was used at quality levels 3 to 6 and quantisation levels of 1, 10 and all increments of 10 up until 90. The sample JPEG image was encoded 100 times

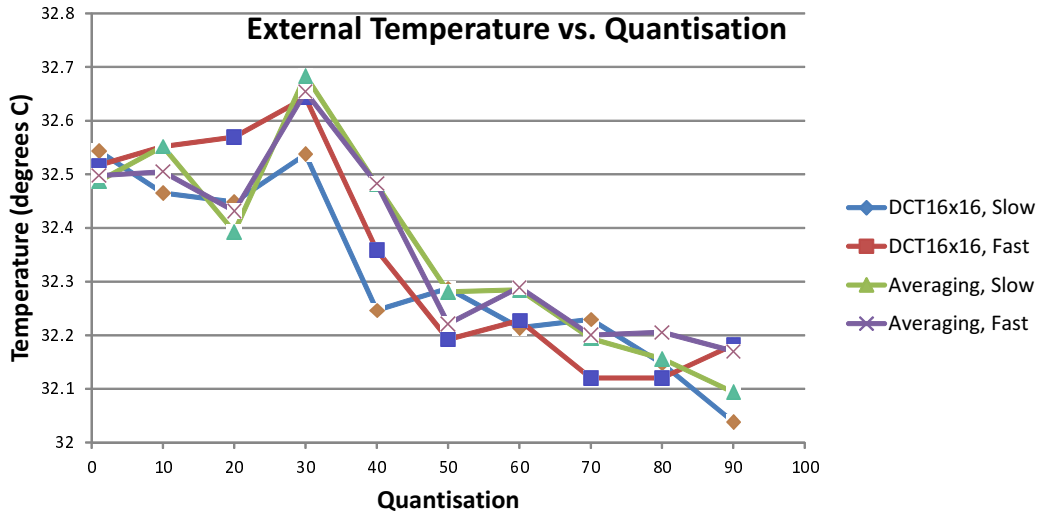



**Figure 3.7:** Temperature of FPGA given successive image encoding. Slow/Fast indicates the speed of the 8x8 DCT employed.

at every quality level. No measurements were taken for the first 50 iterations to allow the temperature to stabilise. The following 50 iterations were then averaged to obtain the temperature for that quality level. Note that the measurements were conducted in a general-purpose office with air conditioning.

The results of this experiment can be seen in Figure 3.7. Four observations can be made from this graph:

1. the range of the temperatures measured is very small, within 1°C,

2. the average temperature is not especially stable from one quantisation level to the next,

3. the different subsampling and DCT methods appear to have had very little effect on the temperature as they are drowned out by noise, and

4. the temperature is the highest when encoding at the lowest quality level, whereas higher quality levels yield the coldest temperatures.

The limited temperature range is due to the low utilisation of the FPGA logic. The

sequential software application causes few signal transitions compared to complex hardware blocks that are highly parallelised. Points two and three can mostly be explained by the fact that the temperature range is so small, resulting in a lot of noise. As a consequence, finer temperature changes caused by a change in quality or quantisation cannot be measured. The last point is of great interest as it brings another variable into the JPEG QoS definition, the frequency at which images are encoded. The output file at the lowest quantisation level (1) is significantly smaller than for the highest quantisation level (90). With higher quantisation levels the file becomes larger and the Huffman encoding stage requires more time to process. Accordingly, more data needs to be written to the SD card through Input/Output (I/O) routines. It appears that the final encoding stages (lower energy routines, Huffman encoding and I/O) require much less energy than the rest of the JPEG encoding process (higher energy routines, RGB to YCbCr conversion, downsampling and DCT). At lower quantisaton the ratio of time spent in high energy to low energy routines is much higher than at high levels of quantisation.

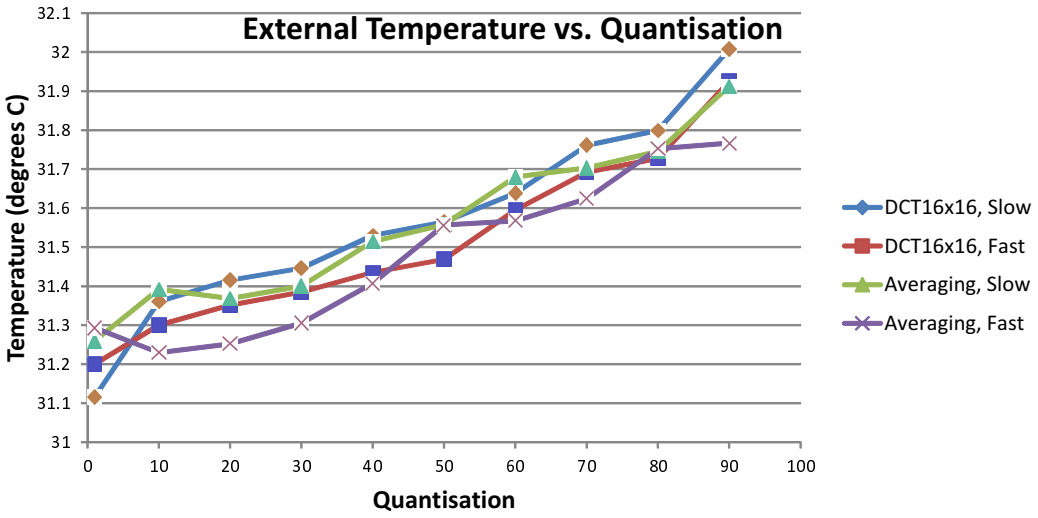



**Figure 3.8:** Temperature of FPGA given constant time image encoding.

Lower quantisation leads to higher encoding frequency, leading to higher temperatures. Figure 3.8 shows the temperatures measured when the frame encoding rate is regulated. After each encoding operation the processor is put to sleep for the remainder of

the period, ensuring that each image is given the same amount of time to encode the picture. This time the general temperature trend matches the quantisation level corresponding more to what was expected. The range of temperatures is still very small though, and the effect of using different downsampling and DCT methods is still drowned out by noise.

## 3.5 Application Adaptation on a Modern FPGA

In an attempt to overcome the limitations of the Cyclone 2 setup, subsequent application adaptation experiments were run on a modern FPGA built on more recent technology. A 40nm Altera Stratix IV FPGA was employed to estimate the impact of temperature control on a modern FPGA. Apart from being a faster device it is also much larger than the Cyclone II FPGA, is capable of implementing circuits that are significantly more complex, and also contains an internal temperature-sensing diode that can be used for accurate temperature measurement from within the device. The only downside is that the internal temperature sensor is not highly precise, as it can only take measurements at a resolution of 1°C. In order to observe a greater variation in temperature, all heatsinks and cooling fans were removed from the chip.

Moving the experiment to a larger and later technology FPGA came with the drawback that a single core processor does not consume as much energy and that it occupies a much smaller portion of the available resources. Therefore, an experiment was designed involving multiple processors performing JPEG encoding at the same quality level at the same time, increasing the overall device temperature. On this device, the maximum number of 32-bit NIOS 2 processors with sufficient amounts of local RAM and ROM that could be instantiated was 15. The JPEG encoding application was stripped of the Huffman encoding stage and the I/O interaction that comes with it. The application was therefore not usable as it processed static batches of artificial image data, but was representative of the effects that application adaptation can have on such a system. A mechanism was devised to synchronise all of the processors where one master CPU ensures that its operation and that of the slave CPUs happen in step with each other. A ‘start’ signal going from the

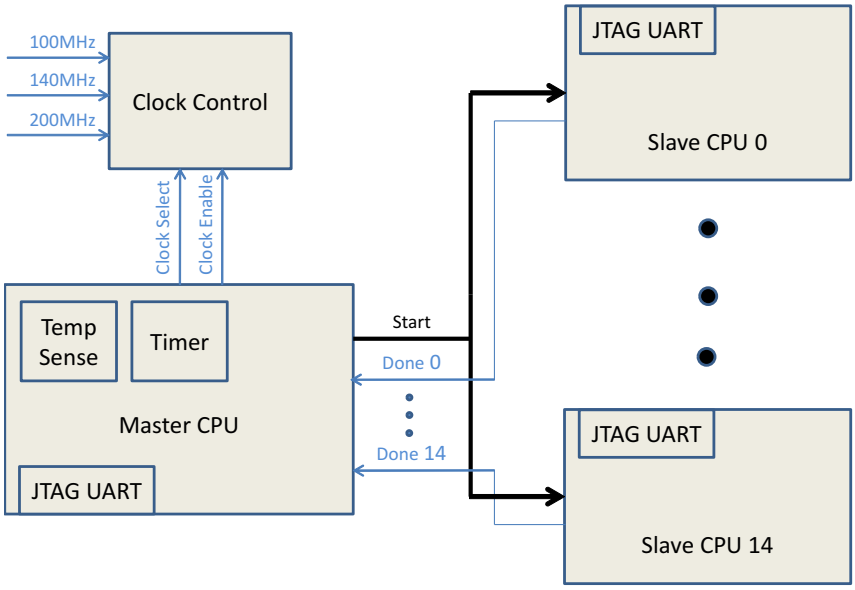



**Figure 3.9:** Synchronous multicore experimental setup with clock control module for frequency scaling and clock gating.

master CPU to all the slave CPUs would indicate when to encode the next frame, while every slave CPU was able to let the master CPU know if it was finished through its 'done' signal. This configuration can be seen in Figure 3.9.

Furthermore, a clock control module was designed to test the effect of frequency scaling and clock gating. This module was also controlled by the master CPU. JPEG encoding was performed at a constant frequency of one frame every 2.2 seconds as this is the time it takes to encode an image at the highest quality and highest clock rate. Three temperature control modes were tested:

1. application adaptation (AA) on its own where the processor would execute a low-power loop after encoding a frame,

2. application adaptation and clock gating (AACG) where the clock is gated for the remaining time at the end of the period, and

3. application adaptation, frequency scaling and clock gating (AAFSCG) where the clock is slowed down to the lowest frequency possible to ensure the desired frame rate. After encoding a frame the processor is then clock gated.

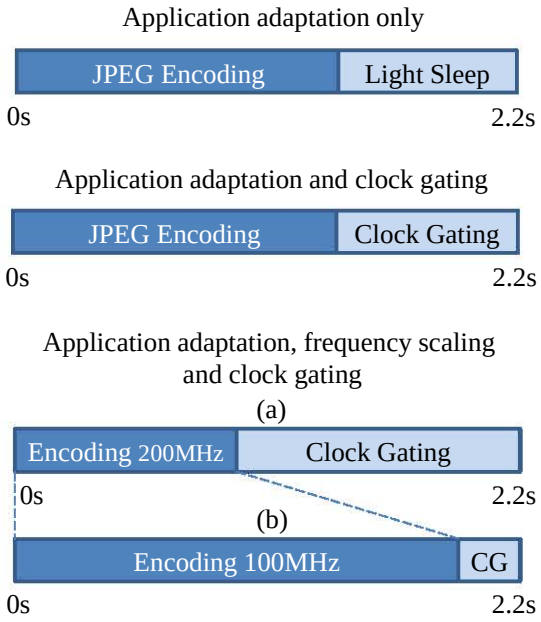



**Figure 3.10:** Demonstration of how the processor spends its time under the different temperature control modes.

Figure 3.10 visualises how the CPU spends its time under these three different modes. The same data was successively encoded at quality levels 1 to 9 (as seen in Table 3.2) with 100 frames encoded at each quality level. The resulting temperature measurements are shown in Figure 3.11. Here it can be seen that application adaptation, all by itself, has very little impact on the device temperature with the temperature, varying mostly within 1°C. This is despite the fact that the amount of time spent in high energy routines is much higher at quality level 1 than at quality level 9. By comparison, when the processors are clock gated (in AACG) at the end of the frame, the temperature drops sensibly with the worsening of quality although the effect is delayed by the high heat capacity of the FPGA die. Within the time span of the experiment the temperature dropped from 81°C down to 75°C. A similar temperature range was also measured when using AAFSCG, although the temperature drops earlier when compared to AACG. On average, AAFSCG is 0.9222°C cooler than AACG even though they both effectively execute the same instructions. The only difference between the two is that AACG performs all these instructions as quickly as it can and then goes to sleep, while AAFSCG tries to distribute them evenly across the

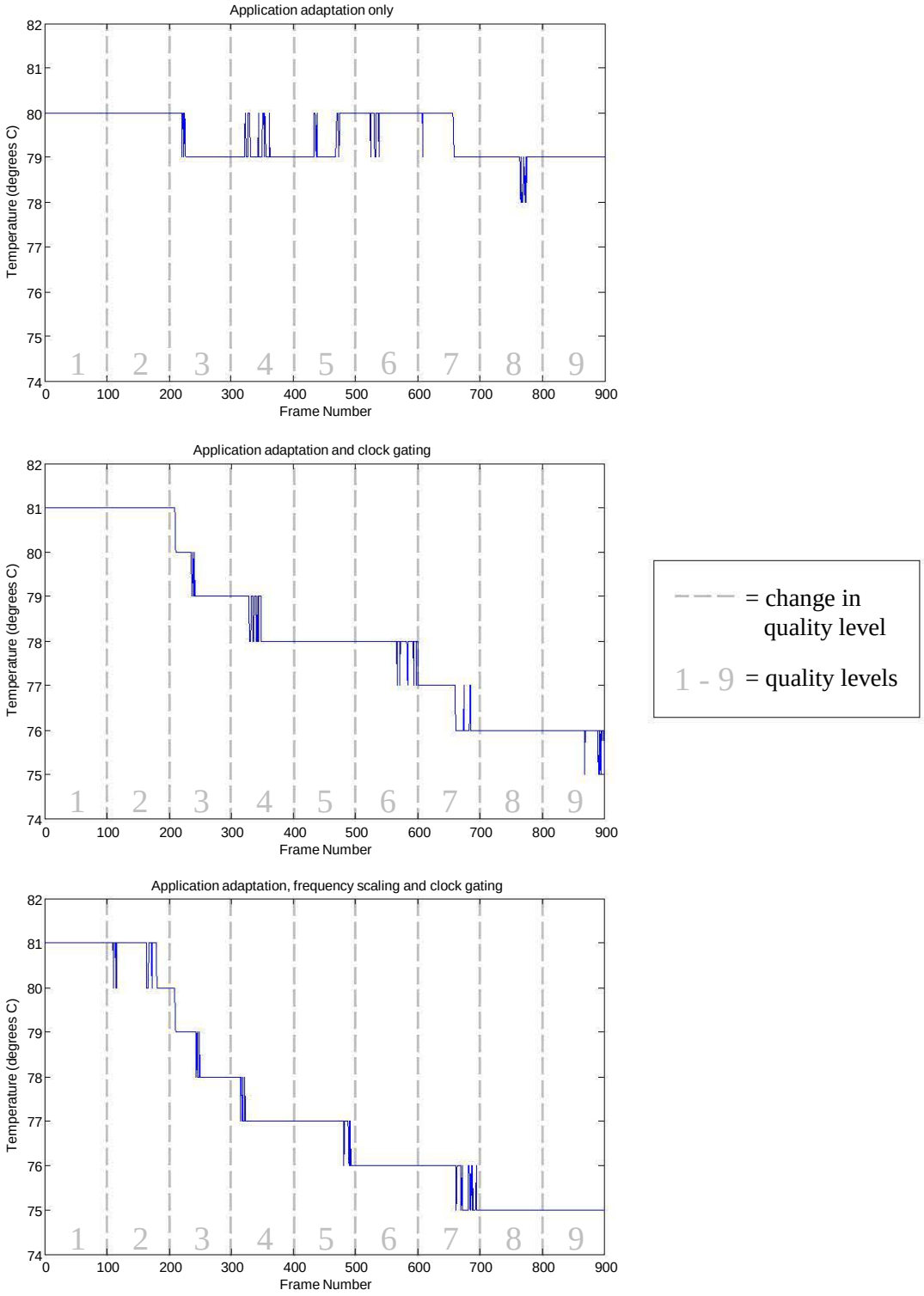



**Figure 3.11:** Temperatures measured as JPEG quality changes over time at 100 frames for each quality level.

frame period.

One hypothesis that can explain the difference in temperature between AACG and AAFSCG is that although the dynamic energy consumption of both modes is identical, the temperature is averaged out in AAFSCG whereas it spikes and troughs for AACG. As the leakage current increases exponentially with temperature [144] it is possible that a temporary temperature that is more elevated than the average leads to overall greater static current consumption. This hypothesis is further endorsed by the fact that FPGAs consume a considerable amount of static power due to their large die area. All the redundant resources of the FPGA (configuration bits, vias, unused LUTs and Registers) are always powered, independent of whether they are used or not. To give a rough indication of the amount of idle power consumption: when the FPGA used for these experiments is left in an un-configured state (i.e., consuming barely any dynamic power), a temperature of 47°C was reached in an air-conditioned space of a little over 20°C. This drastic temperature increase was caused nearly exclusively by static power consumption.

## 3.6 Summary

The ability to perform JPEG application adaptation on an FPGA was studied. For these purposes, a specialised embedded JPEG encoder was implemented that is characterised by improved adaptability over most JPEG encoding implementations and is processor independent. It enables embedded system designers to change application QoS depending on system constraints by dynamically switching between nine discrete quality levels. Its extremely small footprint also sets it apart and makes it ideal for use in embedded systems. This is unlike usual JPEG encoders with limited or no adaptability and which often require a large amount of memory.

Two successive experiments were executed based on one older generation and one modern FPGA. The conclusion was reached that software application adaptation, by itself, is insufficient to control the temperature of the FPGA in a significant way. It yields a temperature variation of less than 1°C in most cases on the given setups. The second

experiment demonstrated that the processor idle time caused by application adaptation could be used to implement other temperature control mechanisms such as clock gating and frequency scaling, enabling the FPGA to cool down by as much as 6°C. Following the study of the experimental results, it became apparent that a greater understanding is required of how and where FPGA fabric heats up. This issue was subsequently researched and is covered in Chapter 4.

# Chapter 4

# Thermal Characteristics of FPGAs

## 4.1 Introduction

In recent years, there has been an increasing focus on FPGA-based systems as they offer an interesting alternative to ASICs due to their high level of functionality, flexibility and advantageous low-volume production costs. The issue of overheating is already severe in ASIC design where the dynamic workload is often responsible for thermal hotspots. FPGAs present additional thermal problems as they are capable of being reconfigured at run time [85]. Due to the increase in deployment of such run time reconfigurable hardware in FPGAs, it is no longer clear when and where thermal hotspots evolve as not all scenarios can be predicted at design time. This has spurred a significant amount of research in the field including Ring Oscillator-based (RO) thermal sensor design [53, 59, 60] and thermal simulation [62, 83]. Not only is there a need to accurately understand the thermal characteristics of modern FPGAs, it is also essential to define the parts responsible for the temperature increase in FPGA-based embedded systems to avoid undesirable elevated temperatures during operation.

The work presented in this chapter is mostly the result of a collaboration between the Embedded Systems Group at the University of New South Wales and the Department of Computer Science at the Karlsruhe Institute of Technology (KIT). Any images produced

by the latter are marked as such. A large portion of the findings that resulted from this collaboration were published at the Conference on Field Programmable Logic in 2013 [145]. The principal contributions are:

1. A case-study of the most important thermal characteristics of FPGAs such as spatial/temporal thermal gradients and peak temperatures. This includes a discussion about the stability of ring oscillator-based thermal sensors and the accuracy of simulation-based temperature evaluation techniques.

2. An analysis of the thermal behavior of a conventional embedded system implemented on an FPGA using soft as well as hard embedded processors and a demonstration that thermal hotspots are frequently located on the memory interface (e.g. DDR controller).

3. An investigation of the thermal impact of a cache component, showing that it plays a substantial role in the thermal behavior of embedded systems. A model is proposed linking cache configurations with peak system temperature.

There is no existing research linking cache configuration with FPGA temperature. Moreover, unlike most previous thermal studies of FPGAs, this work employs a thermal camera to precisely capture the infrared emissions from the silicon wafer of the FPGA.

## 4.2 Measuring Temperatures

While built-in thermal diode sensors are becoming ubiquitous in high-end FPGAs, these are limited by the fact that there is usually only one of them. Moreover, the fixed placement of such a sensor often makes it unable to capture the peak temperature, particularly when the thermal hotspot is generated far away from the sensor. To overcome this limitation, most previous research relies on spreading an array of RO-based thermal sensors across the FPGA die to obtain the spatial temperature distribution. Another method is to employ thermal simulations. The shortcomings of both of these methods is discussed together with a justification for the use of a new setup based on an infrared thermal camera.

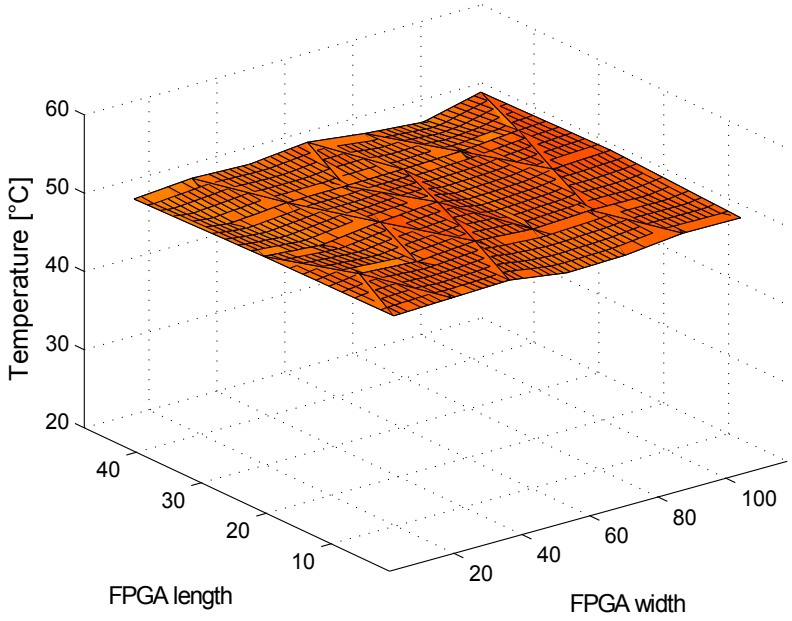



**Figure 4.1:** 3D thermal profile of a design with a low logic activity resulting in smooth temperature changes, measured using ROs.

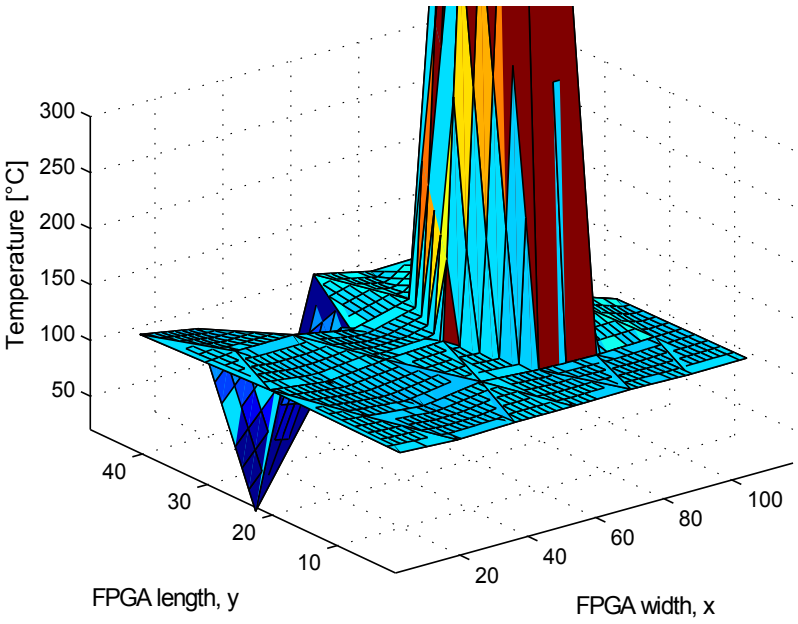



**Figure 4.2:** Measured 3D thermal profile of an FPGA die in the case of intense logic activity leading to severe RO instability.

### 4.2.1 RO-based Thermal Sensor

The ease of implementation of ROs has made them the temperature sensor of choice for reconfigurable logic [22, 44, 53, 59]. Though many designs and optimizations have been proposed, their accuracy is plagued by their dependence on a stable supply voltage. This is made all the more problematic by technology scaling, as smaller process technology is characterised by an increase in voltage supply noise. Zick et al. [53] mention this issue and tried to circumvent it by measuring the chip voltage and including it in their temperature calculations, but even this does not make up for local voltage fluctuations. To examine the issue, an experiment was run where a mesh of 33 ROs was configured to cover half a 40nm Altera Stratix IV FPGA. ROs were found to be surprisingly accurate as temperature sensors when the FPGA is configured with a design with low amounts of switching activity. The result is smooth temperature changes and an even supply voltage distribution, as observed in Figure 4.1 which shows the thermal distribution once the temperature had stabilised.

On the other hand, configuring a micro-heater [81, 84] in the top corner of the FPGA causing intense logic activity in the designated area resulted in severe instability of the RO readings. The 3D plot showing the measured temperatures derived from the RO frequency at one instance in time can be seen in Figure 4.2. It must be noted that it is representative of the locations where the ROs were picking up a large amount of noise. The figure demonstrates that the introduction of intense logic activity in the FPGA fabric leads to sharp changes in temperature readings from the RO-based sensors. This made them unusable as temperature sensors as measured errors reached a maximum of 10,000% with respect to the internal thermal diode sensor. And yet, with the exception of the one problematic sensor (recognisable by the negative peak), the erroneous behavior was contained within the area in which the micro-heater was placed. All other sensors displayed steady, accurate readings leading to the conclusion that supply voltage fluctuations are localised. As the aim is to measure the temperature of precisely those areas of intense activity, RO-based thermal sensors are inadequate for extreme FPGA thermal analysis.

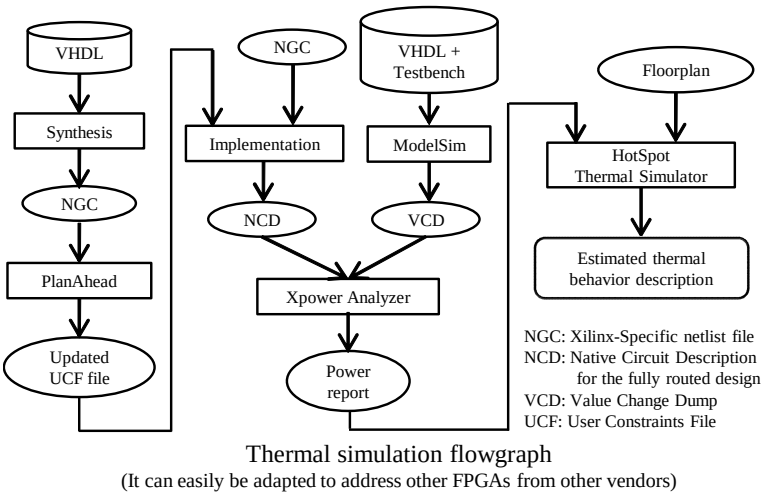



**Figure 4.3:** Steps performed for FPGA thermal simulation.

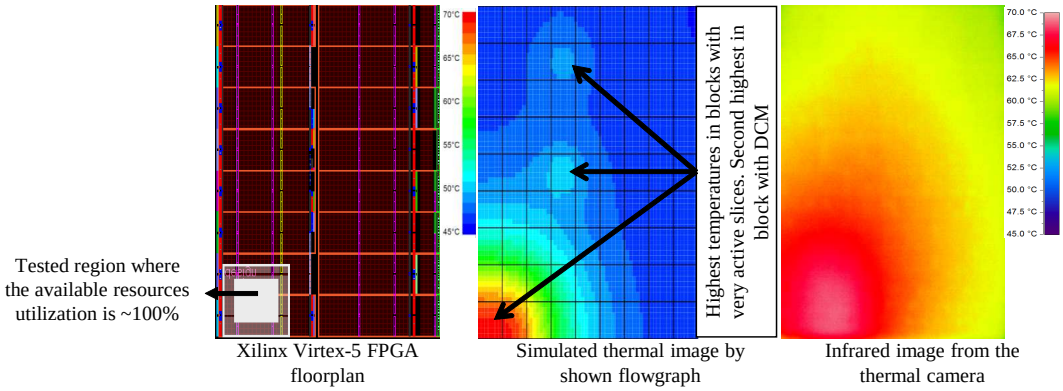



**Figure 4.4:** Comparison between simulated and infrared thermal images show that the simulated measurements display larger amounts of thermal variation. Images by KIT.

### 4.2.2 Thermal Simulation

Estimating the temperature of a design based on thermal simulation always starts with having to estimate the power consumption trace in different blocks in that design.[1] Then, the power information is used by a thermal simulator such as HotSpot [62] which evaluates an RC model to obtain the thermal profile. To estimate the consumed power in the targeted FPGA, the XPower Analyzer tool from Xilinx [146] was used because it gives more realistic quiescent values than the device power spreadsheet [146] which has been

[1] In this scope, a block is a rectangular region of the FPGA which simplifies thermal simulation through abstraction.

utilised in other studies [83]. The XPower Analyzer takes as input the corresponding Value Change Dump (VCD) and the fully routed Native Circuit Description (NCD) file that represents a physical circuit description of the tested design as applied to a specific FPGA device. The Xilinx PlanAhead tool can localise the routing of the design. It takes the netlist file that describes the synthesised design and updates the User Constraints File (UCF) to guarantee that the design will be routed in the desired location. The XPower analyser can also be used to estimate temperature but it assumes that the consumed power is uniformly distributed across the die. As the thermal distribution within the die is targeted, this measurement could not be used. Instead the HotSpot thermal simulator was employed, as it generates an estimated thermal profile which can easily be compared to the images obtained from the infrared camera. HotSpot takes the FPGA flooplan and the generated power distributions within different blocks across the chip to build the thermal image. To draw the required floorplan the area of the target FPGA was divided into equally sized blocks. It is worth noting that Xilinx does not provide the actual size of the FPGA die and that an approximation of these dimensions had to be measured after removing the packaging of the chip. The flowgraph that describes the phases required to obtain the simulated thermal image is presented in Figure 4.3.

The results for the thermal simulation taken from the HotSpot simulator can be seen in Figure 4.4. Since the HotSpot thermal model always includes a heat sink and heat spreader, their thickness was set as low as possible in order to make their effect on temperature and temperature distribution negligible. In this experiment these specifications were set at 100nm because simulation fails at smaller sizes. This allowed for easy comparison between the simulation results and the temperatures captured by the infrared camera. It was found that the maximum temperature of the simulation is comparable to the measured temperatures, but the maximum thermal variation over the chip is significantly higher (a difference of 25°C for simulation and 12°C for the thermal camera). This is mainly due to the inaccurate simulation of leakage power in blocks where there are no switching components. Additionally the amount of detail in the temperature distribution is considerably lower since power values are used at the block abstraction level, i.e. each

block has exactly one summed power value and power variations within a block are not considered. The same holds for temporal power variations since XPower only provides one averaged power value over time. While this is sufficient for some designs, it can lead to inaccurate power estimation in highly dynamic designs.

Following these experiments, the conclusion was reached that thermal simulation may be too imprecise to study the thermal behavior of the FPGA chip. Additionally, RO-based setups may suffer from instability. To overcome these issues a novel setup employing an infrared camera is recommended for design time temperature exploration. This technique can be used to obtain real thermal images containing the detailed thermal distribution of FPGA-based systems.

## 4.3 Experimental Setup

In the experimental thermal camera setup a 65nm Xilinx Virtex-5 FPGA [146] was considered. On-chip temperatures were directly measured using a DIAS pyroview 380L compact infrared thermal camera capable of precisely capturing temperatures with an accuracy of $\pm 1^{\circ}$C and a spatial resolution of 50 $\mu$m [147]. Figure 4.5 illustrates the experimental setup with the required equipment to take the thermal measurements. Once the readings have been obtained the camera sends them at a frame rate of 50Hz to a PC which analyses them to build the corresponding thermal image of the tested chip.

**Emissivity Aspect:** An important aspect to consider when performing infrared measurements is the *emissivity* of the material. This property states what percentage of heat is emitted from the material in the infrared spectrum. Ideal measurements can be obtained from a so-called *black body* with an emissivity of 1.0. Other materials, such as polished metal, have a very low emissivity of around 0.01. Figure 4.6 shows an emissivity test of an FPGA chip with and without metal packaging. The left half of the Altera chip appears cool due to the low emissivity of the metal. The actual temperature, however, is much closer to the right coated half of the chip. For this purpose, masking tape (with an emissivity of 0.92 [148]) was applied to the material's surface, increasing its emissivity

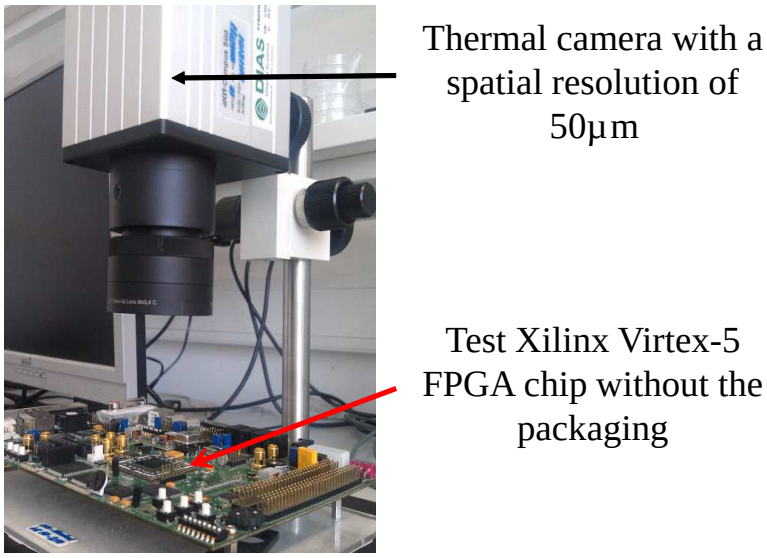



**Figure 4.5:** The experimental setup used for thermal measurement. Image by KIT.

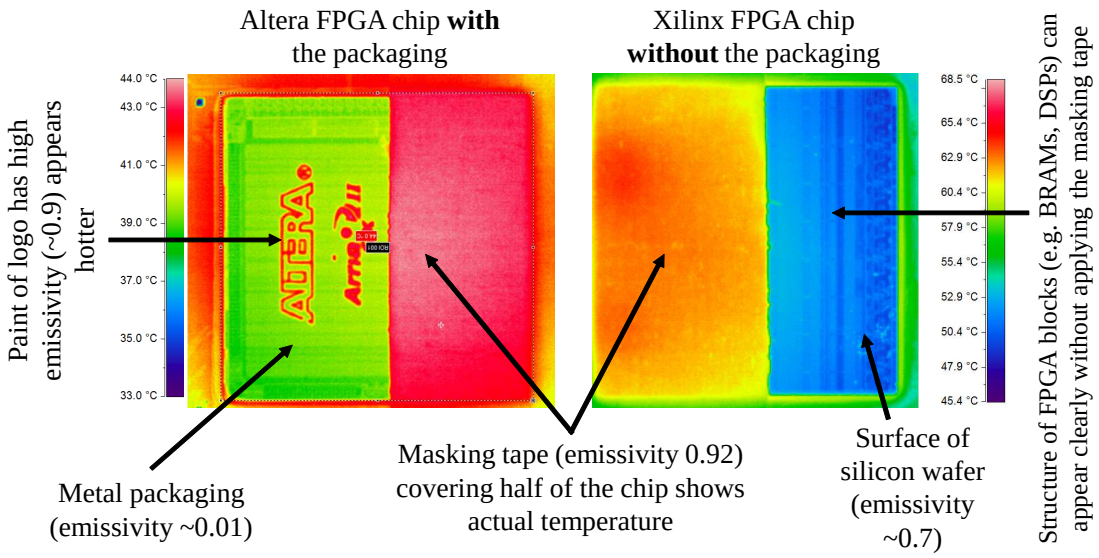



**Figure 4.6:** Emissivity tests. Measurements of materials with low emissivity are very inaccurate as most of the heat measured is reflected from the surroundings. Images by KIT.

in the infrared spectrum of the camera ($8-14\ \mu$m). Masking tape was chosen due to its relatively high emissivity and its non-intrusiveness (it can easily be removed) compared to, for instance, black paint. The uniform distribution of the temperature measurement (shown in the uncoated half) is the result of the properties of the packaging which acts as a heat spreader and uniformly distributes the generated heat. To accurately determine local temperatures and hotspots, the packaging had to be removed from the chip to expose the silicon wafer. The silicon die (in the right half) has an emissivity between 0.75-0.9. Masking tape was still applied to the silicon wafer to improve accuracy. Since the emissivity is known the camera software can internally compensate for the missing 8% temperature. As noted, the variation in temperature between the covered and uncovered halves in this example falsely appear to be up to 20°C.

## 4.4 FPGA Thermal Characteristics

### 4.4.1 High-Stress Scenario

To investigate the thermal characteristics of the targeted Xilinx FPGA platforms under a high-stress scenario, regions were selected on the tested FPGA die and subjected to high-stress conditions while capturing and analysing the infrared images emitted. A special design was synthesised and implemented with the intention of pushing all the available resources (such as LUTs, FFs and BRAM/DSP blocks) to consume as much dynamic power as possible in the considered region by continuously toggling them. Registered parts of the design were operated at 600 MHz, the maximum rated frequency of the target FPGA. This test design is similar to the micro-heaters used in papers mentioned earlier [81, 84] but is different in that they seek stable and reliable heat generation. The aim of this experiment is to reach maximum power consumption by taking into account all types of available FPGA resources. As seen in the captured thermal image shown in Figure 4.7a, the peak temperature of the FPGA chip under this worst-case, high-stress scenario can reach a critical level (124°C) which can disturb the functionality of the configured system

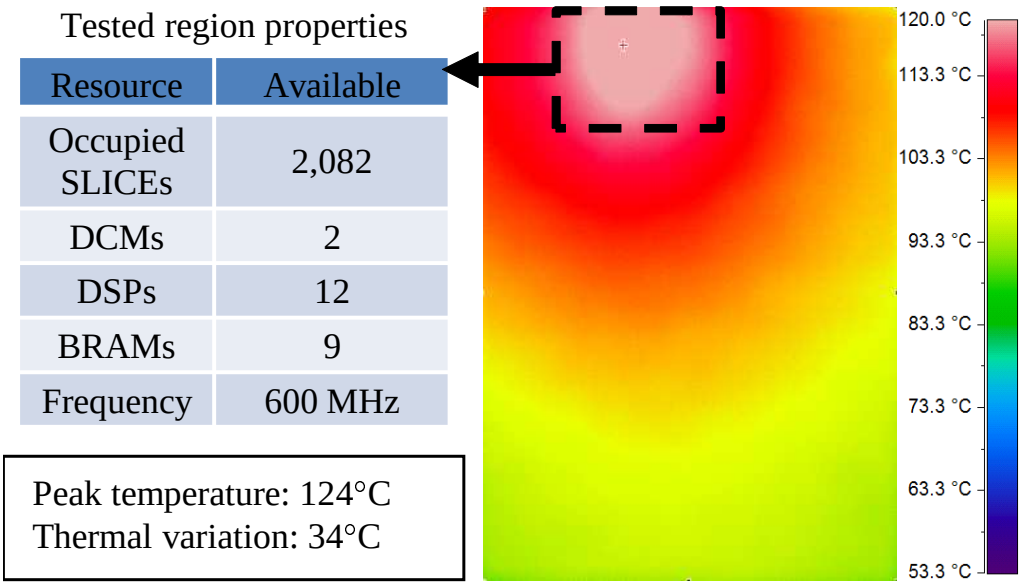
on the FPGA.

More importantly, the on-chip thermal diode sensor that comes built into state of the art FPGAs and is permanently placed at the center of the chip will most likely fail to capture the maximum temperature because the thermal hotspot is located far away from it. It was observed that the thermal variation between where the hotspot is generated and the center of chip reaches around 20°C. The maximum thermal variation (defined as the temperature difference between the hottest and coldest sites across the FPGA's die) reached 34°C. This, in turn, can negatively affect the timing constraints of the running system. Additionally, the aspect of spatial and temporal thermal gradients are also considered one of the key reliability concerns. It is therefore crucial to analyze potential thermal gradients during FPGA operation as well as the peak temperature. The spatial thermal gradient is defined as the temperature variation over a distance, while the temporal thermal gradient is known as the temperature variation in a predetermined location over time. Figure 4.7b presents an example of measured spatial/temporal thermal gradients.[2] Measured peak spatial and temporal thermal gradients reached 32°C/*mm* and 10°C/*s* respectively under high-stress scenarios.

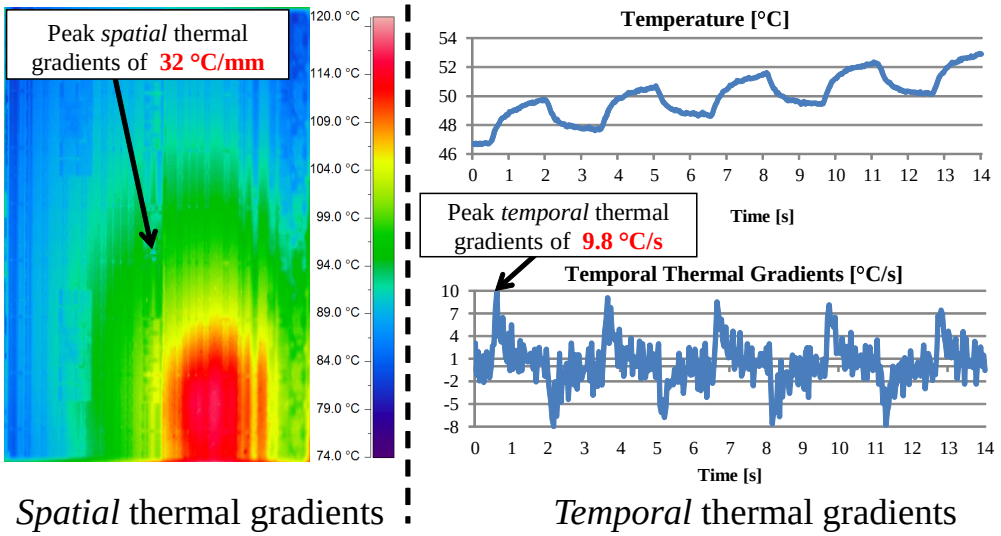
### 4.4.2 FPGA-based Embedded Systems

In this section commonly used FPGA-based embedded processors are examined. The soft-core MicroBlaze processor from Xilinx [146] and LEON3 from Aeroflex Gaisler [149] were targeted as well as the hard-core PowerPC-440 microprocessor from IBM [150] for these purposes. A soft-core CPU is a synthesisable VHDL model of a processor that can be built using the FPGA resources whereas the hard-core processor is an ASIC core permanently embedded in the FPGA die. The MicroBlaze system is built by combining blocks of Xilinx IP cores to end up with a 32-bit RISC-based DLX architecture optimised for implementation in Xilinx FPGA reconfigurable logic. It is capable of single-cycle

[2] In the experiment capturing the spatial thermal gradient shown in Figure 4.7b, the masking tape (further details in Section 4.3) was not applied to cover the FPGA die in order to obtain a more detailed observation. Measurements were adjusted to make up for the lower emissivity of silicon.




**(a)** The infrared image of a Xilinx FPGA under high-stress scenario.




**(b)** Examples of the measured *spatial*/*temporal* thermal gradients.

**Figure 4.7:** FPGA thermal characteristics. Images by KIT.

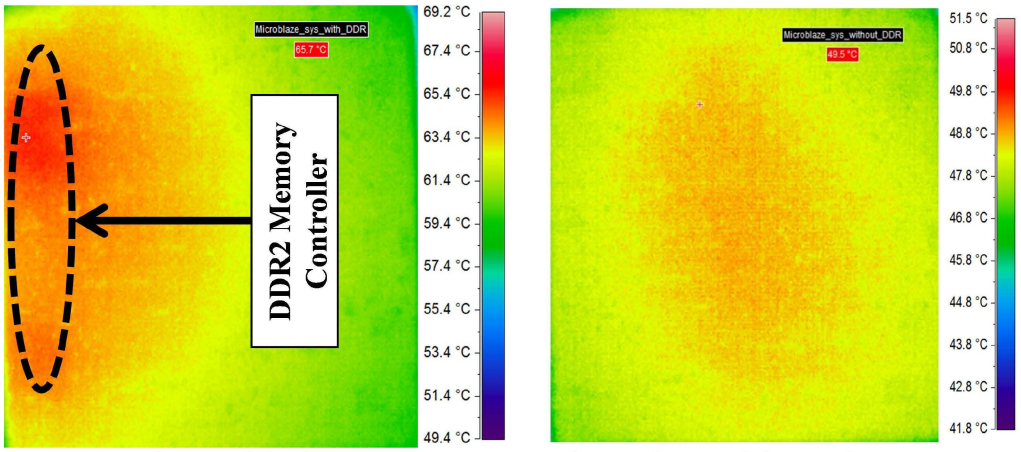



**Figure 4.8:** Infrared thermal images of FPGA-based embedded processors. It can be seen that the thermal hotspot is located on the chip's border due to the dominant role of the memory interface on FPGA temperature. Images by KIT.

throughput under most circumstances and can be configured with a 3- or 5-stage pipeline. The LEON3 is also a 32-bit soft-core microprocessor based on the SPARC-V8 instruction set architecture and implements a 7-stage pipeline. On the other hand, the hard-core PowerPC-440 is a 32-bit high-performance superscalar embedded RISC processor consisting of a 7-stage pipeline. Unlike the soft-core CPUs that can only operate at relatively low frequencies (between 100 - 200MHz), the hard-core PowerPC-440 CPU can run at higher frequencies (up to 550MHz). For fair comparison, the same FPGA family (Xilinx Virtex-5) was used throughout study to evaluate the different embedded target CPUs.

The key observation from running different applications from the cBench [151] (an updated version of Mibench [152]) Benchmark Suite on the aforementioned FPGA-based processors is that the peak temperature of the FPGA chip when running only from on-chip memory is always in a relatively low temperature range with an average temperature of 55°C.[3] On the other hand a drastic increase in the chip's temperature was measured, exceeding 70°C, when a memory interface (e.g. DDR controller) is part of the system. The infrared thermal images in Figure 4.8 show that the thermal hotspot is located on the border of the FPGA chip where the DDR memory interface is implemented[4]. This observation can be explained by the fact that the generated temperature is directly related to the consumed power per area. As discussed in Section 1.1, the consumed dynamic power is equal to:

$$P_{dynamic} = C_{eff} * f * V_{dd}^2 \tag{4.1}$$

where $C_{eff}$, $f$, and $V_{dd}$ are the load capacitance, frequency, and supply voltage respectively. The load capacitance associated with the input and output pins of the DDR memory interface is significant and the switching frequency is high at 400MHz (transfers at double-data rate with a 200MHz clock). Moreover, the supply voltage of the pins

[3] In these experiments the soft-core and hard-core CPUs were run at the highest possible frequencies of 200 MHz and 550 MHz respectively.

[4] It was found that the DDR controller in Xilinx FPGAs is implemented as a hard-macro which is permanently placed on the side of the FPGA's die adjacent to the DDR memory chip.

($2.5V$/$1.8V$) is considerably higher than that of the rest of the FPGA fabric ($1.0V$). Combined with the fact that a memory interface is made up of many pins (e.g. 113, 115 and 117 pins for the DDR2 memory controller in the MicroBlaze, PowerPC-440, and LEON3 CPUs respectively) it can confidently be presumed that the inclusion of the DDR memory interface in an FPGA-based embedded system results in thermal hotspots in that location. A similar observation was made when interfacing the system with an SRAM memory chip.

## 4.5 Modelling the Thermal Impact of Cache

Having determined that frequently accessing the DDR pins significantly increases the temperature of the FPGA chip, the impact of memory configuration on temperature was investigated. More specifically, the aim was to establish whether the inclusion of a cache in a system impacts the temperature of an FPGA in a meaningful way. A cache miss will force the system to retrieve data from the external memory, while a cache hit means that the CPU can retrieve the data locally from the cache. For the same cache associativity and line size, a larger cache generally means fewer misses which leads to fewer DDR memory accesses. It is therefore likely that a lower miss rate has a positive impact on FPGA die temperature. Another factor that comes into play is cache configuration: the combination of cache associativity, line size and cache size directly influences cache miss rates depending on the memory access patterns of the application. As such, a large cache with a suboptimal configuration for a given application will not perform as well as a smaller cache with an optimal configuration. For this reason, both temperature and cache configuration are analysed for different applications in this section.

Figure 4.9 shows the temperature results obtained when varying the cache size of a MicroBlaze embedded system running a JPEG encoding application, together with both read and write DDR accesses registered with each cache size. The evaluation here raises the observation that the temperature of an FPGA-based embedded system strongly correlates with the number of DDR accesses. As a consequence, increasing the cache size

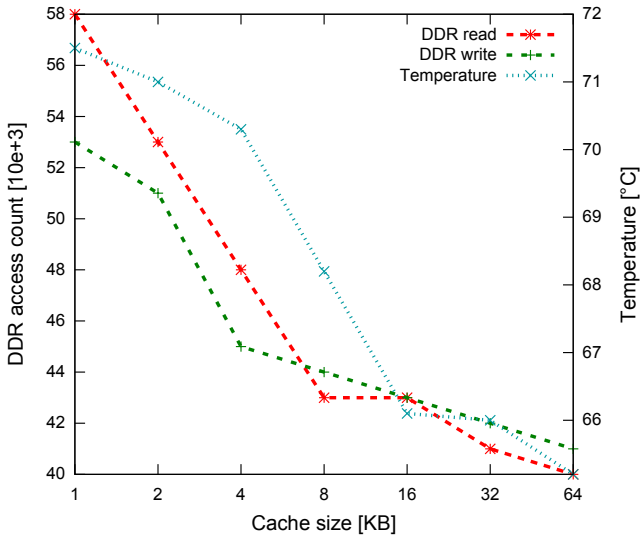



**Figure 4.9:** The influence of cache size on the FPGA temperature as well as the DDR memory read/write accesses for a direct-mapped cache with a line size of 4 words.

can have a positive effect on FPGA temperature as it may result in reducing the need to access the DDR memory due to a higher cache hit rate. This is contrary to existing cache power impact research [73] which usually focuses on ASIC cache implementations. In this case, the low switching density of the FPGA fabric that the cache is built upon, combined with the high power consumption of DDR pins means that lower miss rates are always synonymous with lower power consumption and die temperature.

### 4.5.1 Proposed Thermal Cache Model

Adding a memory interface alone adds a base offset in temperature compared to a system where a processor uses purely on-chip memory, largely due to signals such as the DDR clock. While the latter alone had an average temperature of $T_{\mathrm{BASE}} = 55°\mathrm{C}$, adding the memory interface raises this to $T_{\mathrm{BASE}} + T_{\mathrm{MEM}} = 63°\mathrm{C}$. Afterwards, it was observed that the rise in peak temperature can be estimated by the number of accesses per time interval, the access rate $r$.

$$T = \alpha \cdot e^{-1/\beta r} + T_{\mathrm{MEM}} + T_{\mathrm{BASE}} \tag{4.2}$$

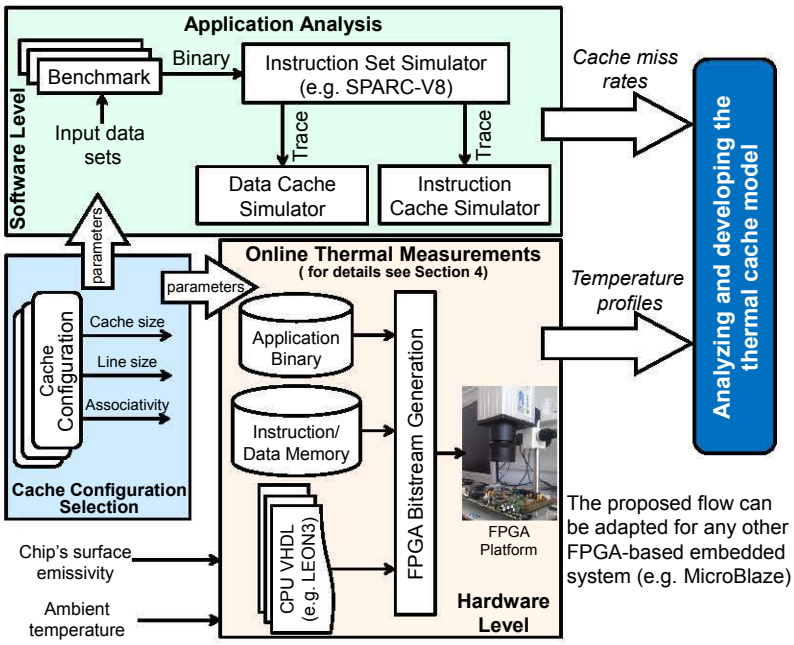



**Figure 4.10:** Evaluating and Modelling the impact of cache.

$\alpha$, $\beta$, $T_{\text{MEM}}$, and $T_{\text{BASE}}$ can all be obtained experimentally using select temperature profiles obtained as shown in Figure 4.10 and are constant for a particular FPGA platform assuming that the ambient temperature is constant. It can be noted from Equation 4.2 that the temperature increase for an initial increase in $r$ is more significant than a change where $r$ is already large, mainly due to heat conduction.

In order to model the effect of cache on temperature, measurements were taken using a number of different cache configurations using the LEON3 processor whose cache is more configurable than the cache available for the MicroBlaze processor. As the cache behavior is largely application dependent, various applications were examined using the DineroIV [20] cache simulator. This was used to calculate their cache miss rates as a metric for memory accesses. These results were then stored in $N \times M$ matrices of rates for $N$ different line sizes and $M$ different associativities for each cache size $k$. The estimated peak temperature is thus given as an extension of Equation 4.2.

$$T_k = \alpha \cdot f(I_k + D_k) + T_{\text{MEM}} + T_{\text{BASE}} \tag{4.3}$$

| $\alpha$ | $\beta$ | $T_{\text{MEM}}$ | $T_{\text{BASE}}$ |
|---|---|---|---|
| 4.01 | 0.52 | 7°C | 55°C |

| | |
|---|---|
| Associativities | 1, 2, 4 |
| Line sizes (B) | 16, 32 |
| Cache sizes (kB) | 0, 4, 8, 16, 32, 64 |

**Table 4.1:** Model parameters in the target Xilinx FPGA platform.

where $f$ is a function defined as

$$\begin{aligned} f : \mathbf{R}^{N\times M} &\rightarrow \mathbf{R}^{N\times M} \\ x_{n,m} &\rightarrow e^{-1/\beta x_{n,m}} \end{aligned} \tag{4.4}$$

and $I$ and $D$ are the cache miss rate matrices for instruction and data cache, respectively. For $k = 0$, the cache configuration is irrelevant meaning that $\forall i_{n.m} \in I_0, i_{n,m} = r_I$ and $\forall d_{n,m} \in D_0, d_{n,m} = r_D$ with $r_I$ and $r_D$ being the maximum memory access rates for instruction and data memory, respectively, of the examined application.

The model was analysed using various benchmark applications (ADPCM, Matrix Multiply, QSORT and Random) by comparing the peak temperatures estimated by the model with those obtained using the infrared thermal camera. In the experiments the parameters were set as shown in Table 4.1, and $r$ was taken as memory accesses per $10^3$ cycles. As can be seen in Table 4.2 the maximum estimation error remains low and is within the accuracy granularity of the camera ($\pm 1$°C) for all examined applications, bar one exception. The exception was QSORT and was partly due to an overestimation of the average $r$ in the instruction memory through various short peaks during application execution, but also due to a slightly higher ambient temperature resulting in a higher $T_{\text{BASE}}$.

The Random application is comprised of random accesses to memory that allowed for the examination of cases where the data cache hit rate remains low regardless of cache size. Since these accesses are performed in a small loop, small instruction cache sizes are already sufficient to prevent cache misses. As a consequence, the maximum estimation error of the given model for the Random application is the lowest, and it is also the application with the smallest range in temperature at the memory interface. It must be noted that the temperature measurements were made by members of KIT, that the DineroIV

| ADPCM | Matrix Multipication | QSORT | Random | Average |
|---|---|---|---|---|
| 0.53°C | 0.64°C | 1.2°C | 0.2°C | 0.64°C |

**Table 4.2:** Maximum estimation error between the given model and infrared camera measurements for different benchmarks

cache simulation was executed by the author while the cache temperature model was proposed by Thomas Ebi.

## 4.6 Summary

Technology scaling has made temperature concerns one of the major challenges that designers face as higher-temperatures have a negatively impact on reliability. Accurately analyzing the thermal properties of an FPGA chip is crucial to avoid violating thermal constraints during operation. The error of thermal simulation methods was quantified through the use of a temperature-measurement setup employing an infrared camera. It was also shown that the dominant part of thermal hotspots comes from the memory interface when targeting FPGA-based processors. Based on the observation that different cache configurations result in different memory access rates affecting the thermal behavior, a thermal cache model was proposed linking cache miss rates with FPGA die temperature. This model exhibits an average maximum error of 0.64°C.

The link between cache configuration and temperature combined with the presented thermal model opens the doors to early-stage thermal evaluation in simulation that does not require hardware implementation. In other words, the thermal impact of many different cache configurations can be analysed in software. However, such thermal evaluation relies on cache simulation which, depending on the complexity of the application, can take a very long time to compute. Such cache analysis comes with the additional benefit that designers are then able to chose the cache configuration displaying optimal performance and/or energy consumption. It is therefore of great interest for designers to simulate caches as quickly as possible.

# Chapter 5

# MASH{lru}: Hardware-Based LRU Cache Simulation

## 5.1 Introduction

Much research has focused on simulating Least Recently Used (LRU) cache behaviour under certain applications in order to find the optimal cache configuration. Precise cache simulators keep track of every single memory access and determine precisely when a hit or miss occurs in a cache of given configuration. Despite many advancements, these cache simulators can still be prohibitively slow, especially when simulating the latest high-power applications. As mentioned in Chapter 2, a number of different approaches have been proposed for the simulation of LRU caches by taking advantage of LRU inclusion properties. Most notably, Cheetah by Sugumar et al. [111] and SuSeSim by Haque et al. [14] employ specialised data structures and algorithms with the goal of speeding up cache simulation. Even GPGPU methods have been proposed. However, no existing simulator makes use of reconfigurable logic to accelerate the exploration of LRU caches.

**The contribution of this chapter is:** a presentation of the first precise multiple cache simulator that is exclusively based on hardware and is implemented on an FPGA.

This design is a Multiple cAche Simulator in Hardware (MASH) for caches with the

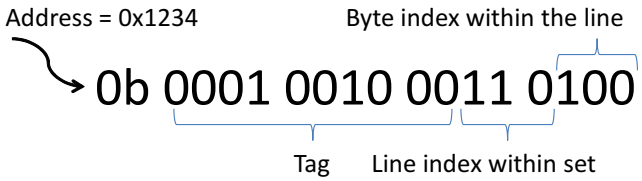



**Figure 5.1:** Breakdown of a 16 bit address for a cache of line length 8 bytes and a set size of 8 lines.

LRU replacement policy. By exploiting the inherent parallelism found in FPGA fabric many cache configurations can be analysed at high speed. The Multiple cAche Simulator in Hardware (MASH) makes use of the LRU cache inclusion properties to minimise the FPGA resource usage. The MASH{lru} simulator was presented at the Asia South Pacific Design Automation Conference in January 2014 [153].

## 5.2 Cache Organisation

Although cache functionality and organisation has already mostly been covered in Chapter 1, it is briefly reiterated here.

The three main configuration parameters that govern the manner in which data is stored in a cache are line length ($ls$), set size ($ss$) and associativity ($assoc$). The line length defines how many bytes are stored in one cache line and the set size defines how many sets are stored in the cache. When an address is accessed, the $log_2(ls)$ lowest bits determine the desired byte, while the next lowest $log_2(ss)$ bits point to the set index within the cache. The rest of the address bits are the Tag, which is held in the cache together with the associated data to determine whether a specific line is stored or not. Figure 5.1 shows how a 16 bit address access is broken down in a cache where $ls = 8$ bytes and $ss = 8$ sets, i.e., the lowest three bits are the byte index while the next lowest three bits are the set index.

Associativity defines the number of locations in which memory lines with the same set index can be stored. In a cache with associativity one (i.e., a direct mapped cache) all addresses with set index $0b111$ (7 in decimal) could only reside in one location in

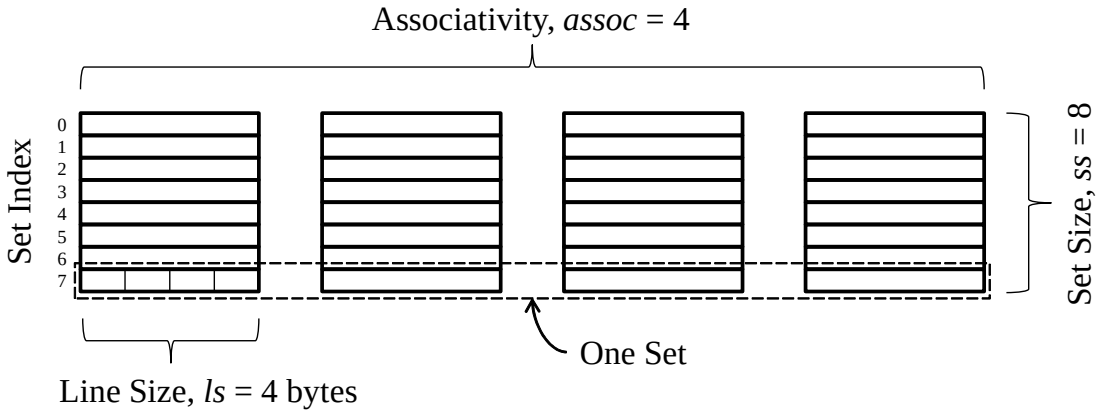



**Figure 5.2:** Depiction of the data organisation within a cache with line length 8 bytes, set size 8 lines and associativity 4.

the cache. On the off-chance that the software contains an access pattern where different lines with the same set index are frequently being accessed alternatively, we will get *cache thrashing*, leading to an excessive amount of cache misses. A set associative cache lowers the risk of such an occurrence as data can be stored in *assoc* different locations with other data of same set index. The data organisation of a 4-way associative cache can be seen in Figure 5.2. In the event that all four ways of a set are occupied and a new line needs to be stored, one of the older lines needs to be discarded to make room for the new line. The line to be discarded is determined by the cache replacement policy. In this chapter only the LRU cache replacement policy is investigated; a policy which keeps track of the order in which lines were accessed and discards the one that was accessed the longest time ago.

## 5.3 Minimising Hardware Resources

Different hardware implementations of the LRU replacement policy exist, but for the purposes of this work it is convenient to represent a cache set with all levels of associativity as shift registers. This is similar to the linked list representation used in many software cache simulators [13, 14]. In real cache implementations each entry would hold the address tag, the line data and a valid bit. As the cache is simulated and not emulated, only

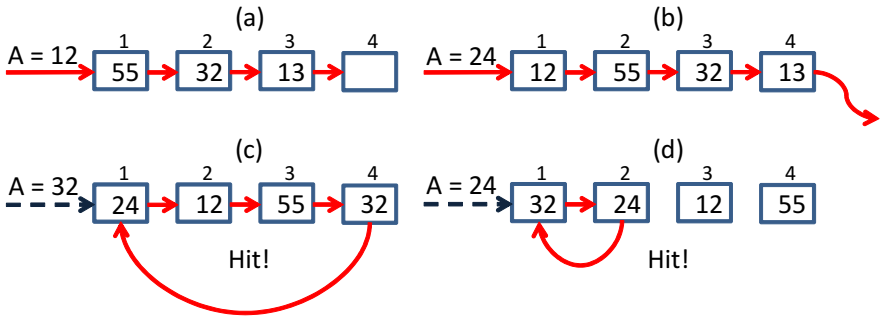



**Figure 5.3:** A cache set with 4 levels of associativity implemented with shift registers and multiplexers. The solid arrows indicate in which direction data will be shifting at the next clock tick.

the address tag and the valid bit are needed. In four steps Figure 5.3 shows how different address accesses are processed by a cache set of a four way set associative cache:

(a) The address accessed, 12, is shifted into the cache set. After this, all four ways are full.

(b) Another address token, 24, is shifted in. As all four ways are full the least recently used address token, 13, is discarded.

(c) The accesses address matches the oldest address token, 32, moving it to the front of the queue.

(d) Again a hit occurs as address 24 is already stored in the cache. The older address tokens, 12 and 55, do not move.

### 5.3.1 Inclusion Property 1: Associativity

This is where the first LRU cache property is used to minimise the amount of hardware required: a cache of set size *ss*, line length *ls* and associativity *assoc* is a subset of a cache of set size *ss*, line length *ls* and associativity larger than *assoc*. This means that for multiple caches of identical set size, only the cache of largest associativity needs to be simulated in order to also simulate all lower levels of associativity with the same hardware.

Figure 5.4 depicts the design of a 4-way set associative cache (i.e., with four lines) containing only one set. If a hit occurs in the second line, the En_2, En_3 and En_4 signals

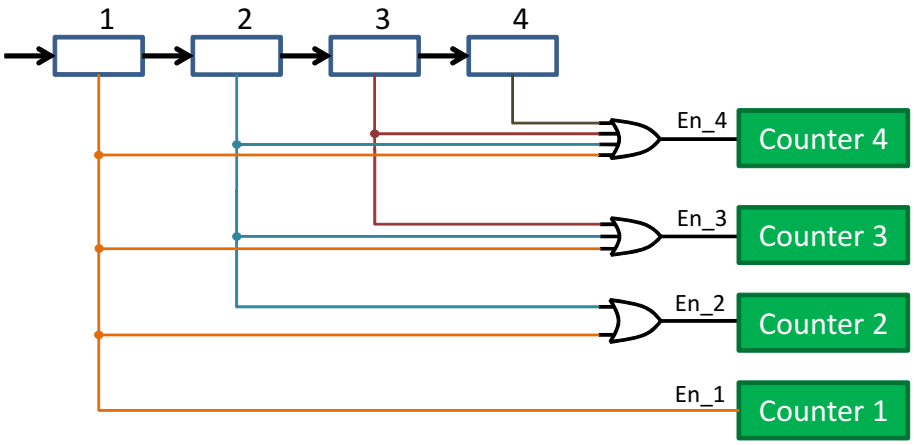



**Figure 5.4:** One cache set of associativity 4 is all that is required to also simulate caches of same set size of associativity 3, 2 and 1. The signal coming out of each level of associativity (or line) indicates whether a hit has occurred at this level. The EN_x signals indicate when the counter for that level of associativity should increment.

would enable the hit counters 2, 3 and 4 to increment. We would know that caches with associativity 2, 3 and 4 would have registered a hit, while a cache of associativity 1 would have suffered a cache miss. Caches usually have more than one set, in which case all the hit signals for a given level of associativity are 'or'-ed to drive the counters and to create a group of signals called *assoc. hits*, as can be seen in Figure 5.5. This setup can be extended to simulate any group of caches of same set size *ss* and associativity *assoc* and smaller, within the limits of the available FPGA resources.

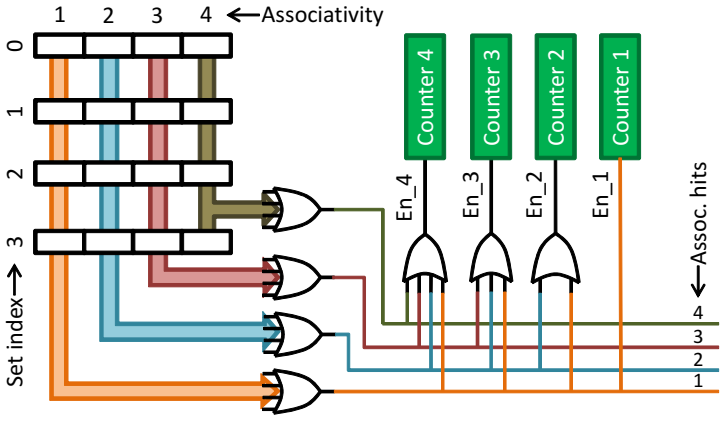



**Figure 5.5:** Cache simulator for caches of set size $s = 4$ and associativity $a \leq 4$.

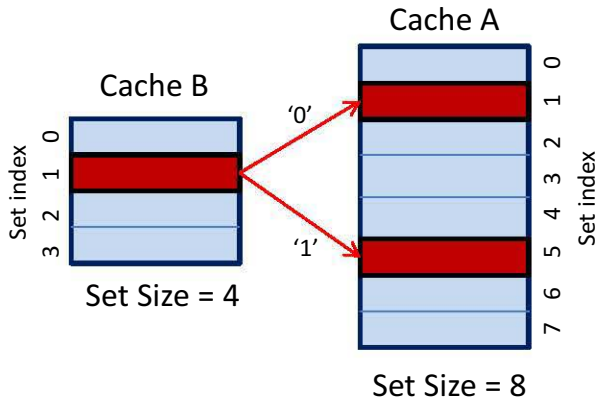



**Figure 5.6:** Two caches of identical associativity and line size. Cached data stored at set index 1 in the smaller cache B would also always be stored at either of two locations at set indices 1 or 5 in cache A of double the set size. These two locations are denoted by '0', or *left* and '1', or *right*

To simulate multiple caches with different cache sizes it would be possible to implement multiple instances of this design. Yet such an approach would not be efficient in terms of resource usage. The current design utilises 71 LookUp Tables (LUTs) and 84 Registers for a cache set of associativity 4. These numbers hold true for a cache simulator capable of consuming a trace with an address width of 32 bits, but even with narrower address spans the resources required are very high. To overcome this issue, a second cache property is exploited.

### 5.3.2 Inclusion Property 2: Set Size

Caches with LRU replacement policy contain another important characteristic that can be used to reduce hardware utilisation of the cache simulator. A cache of line length *ls*, associativity *assoc* and set size *ss* is always a subset of a cache of line length *ls*, associativity *assoc* and set size larger than *ss*. We can further refine this property as the data from a cache of given set size would be located in either of two sets of a cache of twice the set size. This can be seen in Figure 5.6, where data stored at set index 1 in a cache of set size 4 would also always be stored at either set index 1 or 5 in a cache of set size 8, given the same trace. *This means that caches of smaller set size can be simulated by keeping track of where their data is stored within a simulated cache of larger set size.*

For this purpose, specialised hardware components called *sub-sets* were designed. Although their functionality is considerably more complex than the shift registers presented in Section 5.3.1, they do not need to manipulate, hold and compare as many data bits. As a result, one sub-set only requires 32 LUTs and 16 registers for an associativity of 4. To more clearly demonstrate the task of a sub-set, let us look at the example given in Figure 5.7. This shows the state the highlighted sets from Figure 5.6 could be in if the caches A and B had been subjected to the same trace. It can be observed that:

1. all entries of the set in cache B are contained within the two given sets of cache A,
2. if the set from cache B contains $x$ similar entries from one of the sets from cache A, then the matching entries will be the $x$ entries of lowest associativity of that set from cache A, and
3. the order of the last four accesses was first 61, then 12 and 23, and finally 5.

Figure 5.8 shows the same cache set state as mentioned, only that the set of cache B is depicted by an extremely small shift register. Instead of containing a duplicate of all the address tokens held in Cache A, only two bits are shifted, a *source* bit and a *valid* bit. The valid bit is set when the associated source bit is not empty. The source bit determines if the address token it is keeping track of is located at set index 1 or at set index 5 of cache A, which is determined by a value of '0' or '1' respectively. In the example, the source bit order "1001" means that the first token (5) is stored at set index 5, the second (23) and third (12) tokens are stored at set index 1, and the last token (61) is stored at set index 5 of cache A.

The sub-set is based on such a shift register but also includes two *mask registers* as seen in Figure 5.9. These registers indicate whether the address token at a certain level of associativity in cache A is held within cache B. In the current scenario, tokens 23 and 12, and 5 and 61 are tracked by the sub-set, which explains why the first two bits in Mask 0 and Mask 1 associated with those values are set. The purpose of these masks is to easily determine whether a hit has occurred in the sub-set, simplifying future operation. The finer implementation details are given in Appendix A.5.

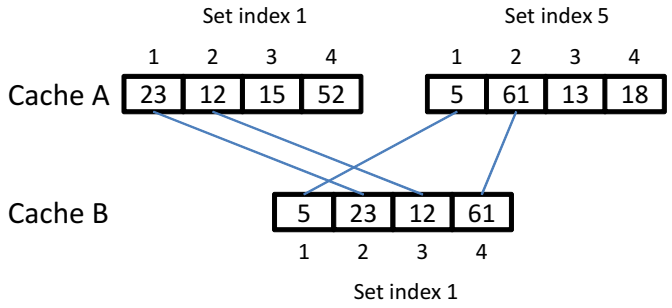



**Figure 5.7:** Example of addresses that could be stored in the highlighted sets of Figure 5.6 if both caches A and B were administered the same trace. Each set is of associativity 4.

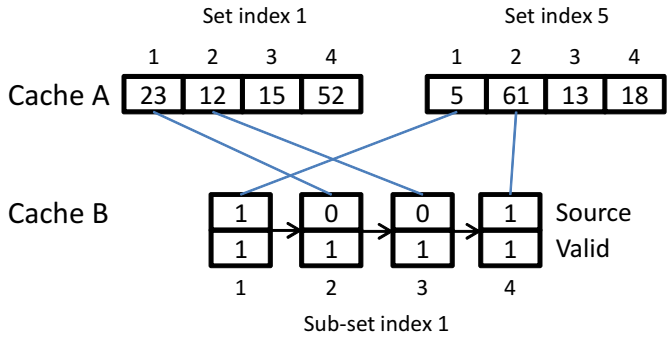



**Figure 5.8:** Same example from Figure 5.7 with the set from cache B represented as shift registers.

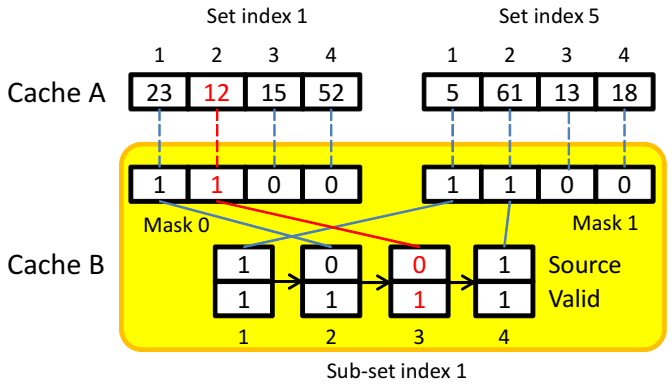



**Figure 5.9:** One subset and its registers. If a hit occurs with address token 12 in cache A, set index 1, we would immediately be able to tell if a hit has also occurred in the subset thanks to Mask 0.

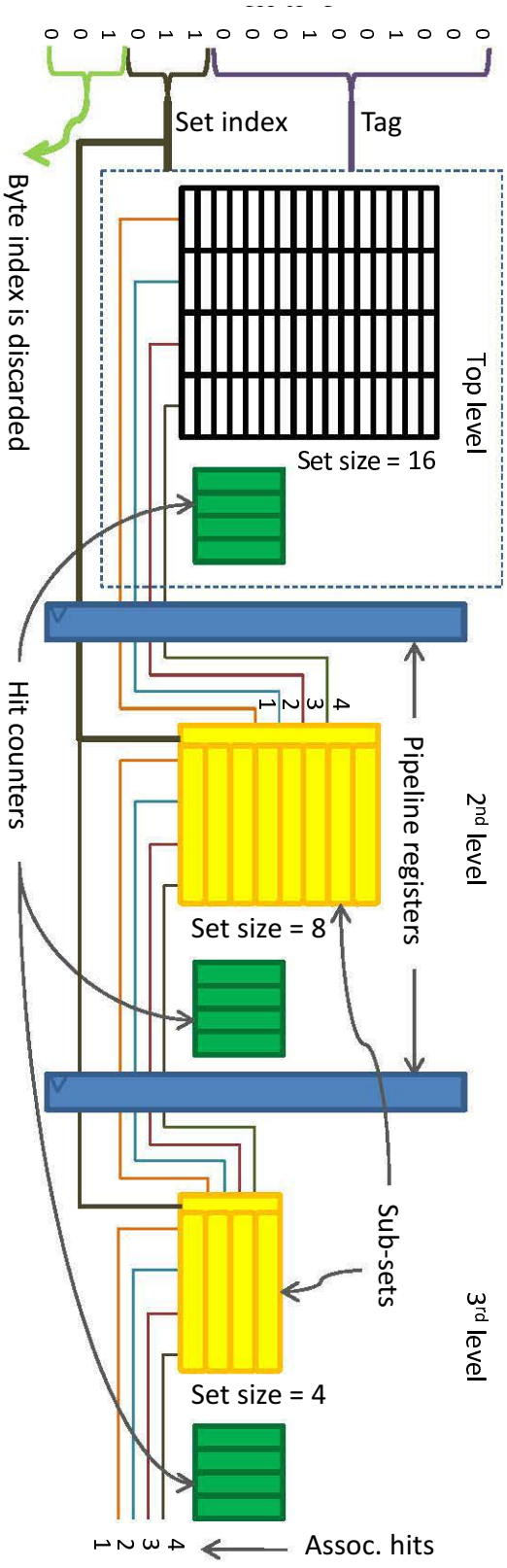



**Figure 5.10:** Cache simulator for caches of set size $s = 16$, $s = 8$ and $s = 4$ and $a \leq 4$. Note that the top level of the cache simulator is identical to the simulator presented in Figure 5.5 except that it has a set size of 16, not 4. The address from Figure 5.1 is reused to show how it is broken down. The assoc. hits and set index are passed down to the lower levels through registers.

At this point the cache simulator can be divided into *levels* where each level represents the hardware used to simulate caches of a given set size. The biggest set size ($s^{top}$) is simulated by the *top level*, like in the design depicted in Figure 5.5, and directly manipulates address tags. The associativity hit vector (assoc. hits) of the top level is then passed down to the second level of the simulator which simulates caches of set size $s^{2nd} = s^{top}/2$, and is made up of $s^{2nd}$ sub-sets. The three signal groups used to control the sub-sets are: an enable signal (determined from the set index), an address bit (the highest set index bit from the level above) and the assoc. hits vector from the level above. Each level of sub-sets also has its own assoc. hits output which is used to count the hits for this level and which can also be passed down to a lower level of the cache simulator. Figure 5.10 shows most of these signals, and more importantly, how they are connected within the cache simulator. Registers are introduced between the levels of the cache simulator to increase the maximum operating frequency (i.e., address token consumption rate), with each level increasing the latency by one clock cycle.

The entire cache simulator design is described in VHDL, most of which can be viewed in Appendix A. The parameters of the simulating hardware determining the maximum and minimum set size and the largest associativity to be simulated are defined in one location (Appendix A.1). Lower levels are instantiated recursively, halving the set sizes at each level, until the smallest set size has been reached. The maximum address token width can also be modified to minimise the resources required by the top level if the range of the addresses is limited.

### 5.3.3 Line Length

For now the cache simulator can simulate different levels of associativity and different set sizes. Simulating different line lengths is trivial: when breaking down the address token the size of the byte index is set to $log_2 ls$ bits (where $ls$ is the line length in bytes), the set index is of constant size but is shifted so it is placed next to the byte index, and the remaining bits are used for the tag. This is demonstrated in Figure 5.11. However, an

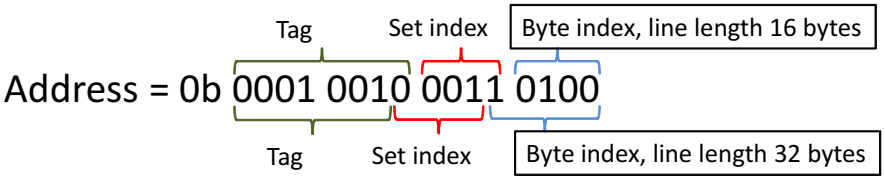



**Figure 5.11:** Breakdown of the 16-bit address of value 0x1234 for the line lengths of 16 and 32 bytes and set size $s = 16$.

instance of the cache simulator can only simulate one line length at any one time. This means that simulating a different line length requires a rerun of the simulator. Alternatively, one could instantiate another cache simulator instance, in parallel, configured to simulate that line length.

## 5.4 FPGA Implementation and Performance

### 5.4.1 Resource Usage

The amount of resources required in the form of LookUp Tables (LUTs) and Registers (Regs) by a hardware cache simulator instance is an important aspect to consider to make sure the design will fit on a given FPGA. This is entirely dependent on the size of the largest cache simulated. Three main cache simulator parameters defined at design time come into play: the address token width ($addr_w$), the maximum set count ($set\_count_{max}$) and the maximum associativity simulated ($a_{max}$). Table 5.1 shows the required resources of a cache simulator with its CPU interface depending on these parameters. Some parameter combinations could not be compiled as the resource usage of the resulting configuration would have been too large to fit the target FPGA. The numbers show that the size of the hardware instance grows linearly with the parameter values.

### 5.4.2 Performance

The cache simulator was tested on an Altera Stratix IV GX FPGA containing 230 thousand logic elements depicted in Figure 5.12. Given the amount of resources available, it

| $addr_w$ | $a_{max}$ | $set\_count_{max}$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 32 | | 256 | | 1024 | |
| | | LUTs | Regs | LUTs | Regs | LUTs | Regs |
| 17 | 4 | 4932 | 3702 | 25085 | 15326 | 79431 | 46077 |
| | 8 | 10529 | 7163 | 56859 | 29935 | | |
| | 16 | 21434 | 14084 | 112098 | 59152 | | |
| 22 | 4 | 5517 | 4342 | 26541 | 20446 | 87626 | 66557 |
| | 8 | 11644 | 8443 | 60321 | 40175 | | |
| | 16 | 23672 | 16644 | 127876 | 79632 | | |
| 27 | 4 | 5997 | 4982 | 31990 | 25566 | 105037 | 87037 |
| | 8 | 12655 | 9723 | 70216 | 50415 | | |
| | 16 | 26203 | 19204 | 147526 | 100112 | | |
| 32 | 4 | 6410 | 5622 | 35313 | 30686 | 126235 | 107517 |
| | 8 | 13500 | 11003 | 77732 | 60655 | | |
| | 16 | 27829 | 21764 | | | | |

**Table 5.1:** Resource requirements (in LUTs and Registers) depending on the size of the largest cache simulated.

was possible to instantiate a simulator capable of simulating caches with four different levels of associativity ($a$ = 1, 2, 3 and 4), eleven different set sizes ($s$ = 1 to 1024) and seven different line lengths ($l$ = 4 to 256). Within these parameters 308 different cache configurations can be simulated, while 44 configurations can be simulated concurrently as the line length has to remain constant for each simulation run.

On the target FPGA, the resulting cache simulating hardware instance is capable of running at 100 MHz with a latency of eleven clock cycles; it is capable of consuming an address trace at 100 million address tokens per second. As the hardware implementation is the first of its kind, it was not possible to compare it with existing cache simulators at the time of conception and evaluation. Static trace simulation requires fast access to a very large trace file. At the time of experimentation it was not possible to transfer such a large trace file to the FPGA cache simulator, as a high-speed data interface was yet to be implemented.

The correct functionality of the cache simulator was tested in Modelsim [98] and with a hardware implementation where static traces were provided through a JTAG UART connected to a PC.

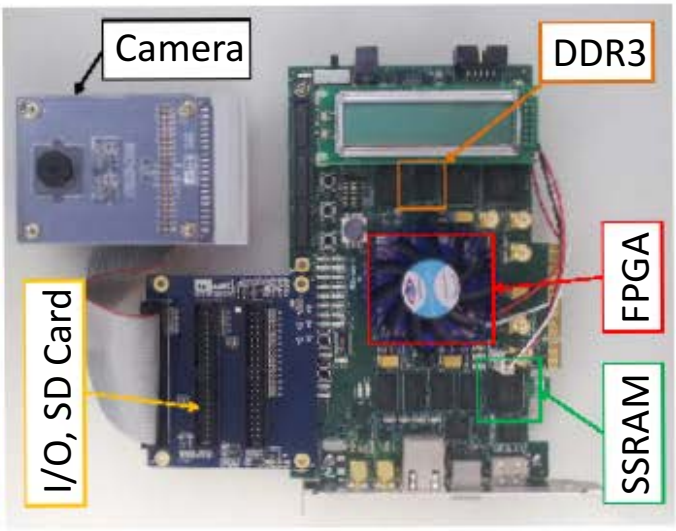



**Figure 5.12:** FPGA hardware used to test the cache simulator. The devices outside the FPGA (Camera, I/O etc.) are used in future chapters.

Due to the extremely slow bandwidth of the JTAG UART interface it was not used for the performance assessment. Instead, timing results of software-based cache simulators were compared with the *throughput* of the cache simulating hardware. The traces simulated were obtained from both the SimpleScalar computer architecture simulator [93] and the Tensilica Xtensa processor simulator [18]. The traces of four Mediabench [154] applications and five SPEC CPU2000 [155] benchmarks were extracted using these two simulators. Every trace was fed to three different software-based cache simulators, namely DineroIV [20], Cheetah [111] and SuSeSim [14]. The simulations were performed on the high-end Intel system described in Table 5.2. The exact 44 configurations that the hardware is capable of simulating were also simulated on Cheetah and SuSeSim. This was not possible with DineroIV as it does not allow for associativities that are not a power of two. As a consequence, only 33 cache configurations were simulated with this tool. The line length simulated was 8 bytes and the traces were purely made up of instruction memory accesses in order to simulate the instruction cache only.

Given the trace consumption rate that the FPGA implementation is capable of, the cache simulation times are given in Table 5.3 while the speedup with respect to software-based simulators can be seen in Figure 5.13. In Table 5.3 the last row indicates the speedup

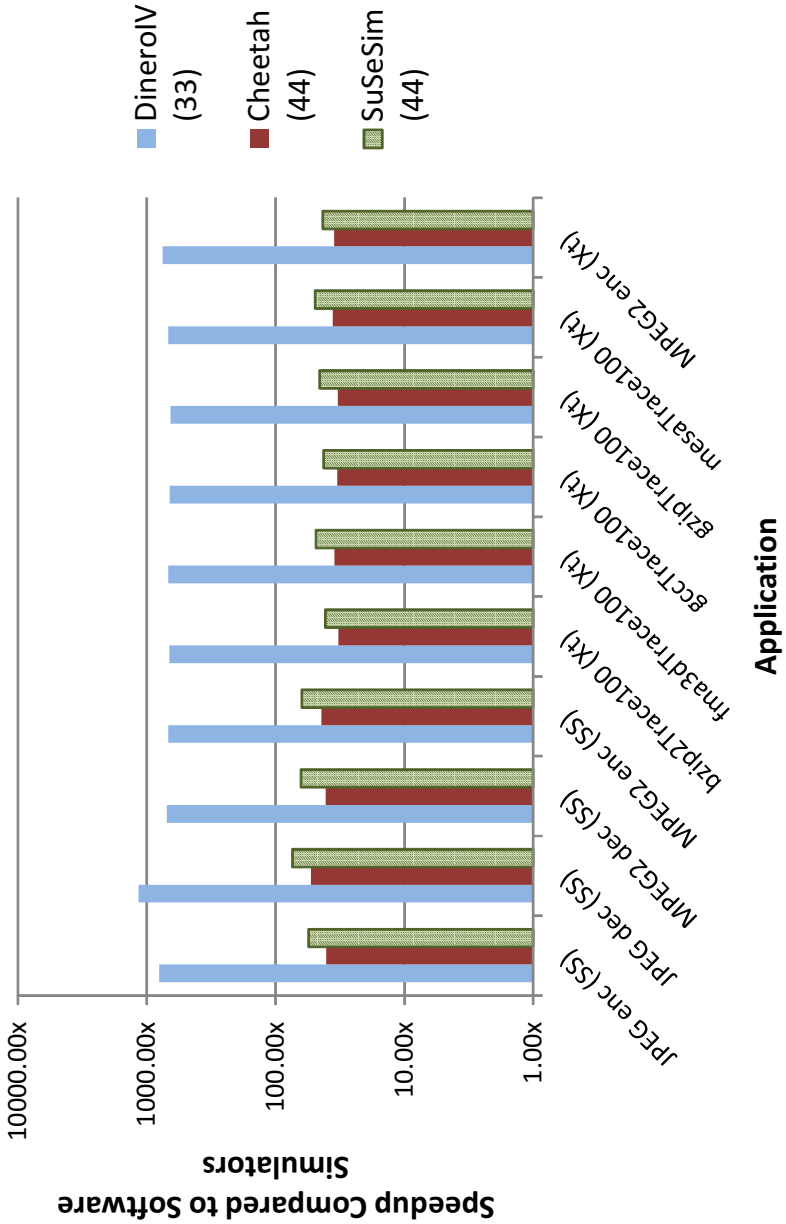



**Figure 5.13:** Speedup of the FPGA-based cache simulator core when compared to existing software-based implementations. The application trace was extracted by either the SimpleScalar (SS) or the Tensilica Xtensa (Xt) simulator.

| | |
|---|---|
| Processing core count | 32 |
| Processor | Intel Xeon X7560 @ 2.27GHz |
| System memory | 256 GB |

**Table 5.2:** Specifications of the setup used for software simulating purposes.

that could be achieved by a MASH{lru} instance as it can consume 100 million address tokens per second. Compared with Cheetah, the hardware core displayed a speedup of 32x to 53x, with an average speedup of 38x. When set against SuSeSim, MASH{lru} was between 41x and 74x times faster, averaging 52x faster. DineroIV, despite only simulating 33 configurations, was still up to three orders of magnitude slower than the FPGA-based setup simulating 44 configurations.

## 5.5 Summary

This chapter presented MASH{lru}, the first multiple cache simulator implemented in hardware. Its ability to process address tokens at 100MHz, concurrently simulating 44 different cache configurations, make it up to 53x faster than the fastest software-based cache simulators for a set of benchmarks. Cache inclusion properties were exploited to optimise the amount of FPGA hardware resources required. As a result, an implementation of a cache simulator capable of simulating 308 different cache configurations requires over half the resources of a 230 thousand logic element Altera Stratix IV FPGA.

The simulator, its optimisations and hardware design are very specific to the LRU cache replacement policy due to the handy inclusion properties. The general concept and approach, however, have paved the way for extremely fast cache simulators for other replacement policies.

| Simulator | SimpleScalar | | | | Tensilica Xtensa | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Application | JPEG encode | JPEG decode | MPEG2 decode | MPEG2 encode | bzip2 100 | fma3d 100 | gcc 100 | gzip 100 | mesa 100 | MPEG2 encode |
| Trace token count | 11.51M | 2.84M | 937.32M | 2.36B | 333.61M | 312.60M | 320.10M | 331.70M | 317.08M | 11.15B |
| DineroIV (33) | 92.12s | 32.99s | 6544s | 16045s | 2233s | 2135s | 2130s | 2174s | 2159s | 83922s |
| SuSeSim (44) | 6.37s | 2.09s | 593.92s | 1476s | 136.51s | 151.71s | 135.44s | 150.31s | 155.85s | 3913s |
| Cheetah (44) | 4.65s | 1.51s | 382.41s | 1039s | 108.34s | 109.21s | 105.92s | 109.05s | 113.87s | 4794s |
| MASH{lru} (44) | 0.12s | 0.03s | 9.37s | 23.59s | 3.34s | 3.13s | 3.20s | 3.32s | 3.17s | 111.51s |
| Speedup w.r.t. Cheetah | 40x | 53x | 41x | 44x | 32x | 35x | 33x | 33x | 36x | 35x |

**Table 5.3:** Instruction cache simulation trace lengths and timings of 44 cache configurations (33 for DineroIV) based on ten application traces.

# Chapter 6

# MASH{fifo}: FIFO Cache Simulation in Hardware

## 6.1 Introduction

This chapter examines the FIFO cache replacement policy which is frequently implemented on embedded processors (Tensilica Xtensa [18], ARM9 and ARM11 [17] etc.) due to its low hardware complexity, especially when compared to the LRU cache replacement policy [156]. MASH{fifo} is presented here which improves upon existing research with the following contributions:

- for the first time inclusion properties are presented that are applicable to the FIFO cache replacement policy,
- based on the FIFO inclusion properties, a design is presented supporting hardware-based multiple cache simulation for caches employing the FIFO replacement policy and
- it is demonstrated how such a cache simulator instance can be configured in-system, on an FPGA, together with a CPU and memory. Leveraging the high throughput of MASH{fifo}, multiple cache simulation is performed in real-time.

As discussed in Section 2.3.5 it is much more difficult to simulate many FIFO caches compared to simulating LRU caches as the FIFO replacement policy does not obey the LRU inclusion properties that are easy to exploit. As a consequence, multiple cache simulators supporting the FIFO replacement policy have only recently been proposed. Notably Haque et al. presented CIPARSim [121] which is considered one of the fastest FIFO cache simulators. Tawada et al. published two works on FIFO cache simulators [15] [122] with the shortcoming that their proposed optimisations (CRCB and priority queue) are trivial. In comparison with previous work, MASH{fifo} distinguishes itself with very high simulation speeds of multiple caches that employ the FIFO replacement policy. Most of the work in this chapter was published at the Design Automation Conference (DAC) in 2014.

## 6.2 Cache Simulator Design

A naive approach towards hardware-based cache simulation would be the parallel instantiation of many caches of targeted configurations. If caches of sizes $ss_{max}$ and smaller and associativities $assoc_{max}$ and smaller were to be simulated, the cache simulator would have to manipulate and compare $(2ss_{max}-1)*(2assoc_{max}-1)$ tags. As storing and comparing tags of up to 32 bits requires many hardware resources, reducing the number of tags the hardware simulator has to hold can also reduce the size of the simulator instance. For MASH{lru}, LRU cache inclusion properties were exploited to minimise hardware utilisation as the state of smaller LRU caches can easily be deducted by observing the state of larger caches. In this section, inclusion properties for the FIFO replacement policy are defined which can be used to significantly reduce the size of a MASH{fifo} instance.

### 6.2.1 Inclusion Property 1

The first inclusion property is as follows: *an LRU cache set of associativity* $2*assoc-1$ *will always hold the data blocks present in a FIFO cache set of associativity assoc and*

*smaller*. To develop the inclusion property for the FIFO cache we evaluate the longest possible lifetime of an address tag in a cache line. Let us suppose that at time $t_0$ a FIFO cache line of associativity *assoc* is subject to a miss following which address tag ($addr1$) is inserted in the front of the FIFO queue. The first inclusion property relies on the fact that there is a limited number of unique addresses that can be accessed before $addr1$ is evicted. The instant after $t_0$, $addr1$ is pushed onto the FIFO and is held alongside $assoc-1$ addresses that were stored previous to $t_0$. Any of these $assoc-1$ addresses or $addr1$ can be accessed without changing the state of the FIFO set. If an address is accessed that is not equal to $addr1$ or any of the old $assoc-1$ addresses the cache set signals a miss and a new item is pushed onto the FIFO. This brings $addr1$ one step closer to being discarded. After $assoc-1$ misses $addr1$ is pushed to the end of the FIFO queue, preceded by $assoc-1$ newer addresses. At the $assoc^{\text{th}}$ miss, $addr1$ is discarded from the cache line. As a consequence, we know that at most:

$$1_{(addr1)} + (assoc-1)_{(old)} + (assoc-1)_{(new)} = 2*assoc-1 \tag{6.1}$$

unique addresses will be accessed before $addr1$ is discarded. In other words, as soon as more than $2*assoc-1$ unique addresses that are not $addr1$ have been accessed after $t_0$, we can be certain that $addr1$ will no longer be in the cache.

To simulate a FIFO cache set of associativity *assoc* or smaller an abstract data type (or 'container') can be used which is certain to hold at least all the address tags of the cache line. It will also have to follow the given specification: store a data item $d$ until at least $2*assoc-1$ unique data tokens that are not equal to $d$ have been accessed or stored. An LRU cache line of associativity $assoc_{LRU} = 2*assoc-1$ presents precisely these properties: by prioritising recently used data it only discards data once $assoc_{LRU}$ other data tokens have been accessed.

### 6.2.2 Inclusion Property 2

The second FIFO inclusion property is based on the second LRU cache inclusion property from Section 5.3.2: *given an identical line size ls and associativity assoc, an LRU cache of set size ss will always contain the data stored in an LRU cache of set size smaller than ss*.

As the data of each FIFO set of associativity *assoc* is also held in an LRU set of associativity $2 * assoc - 1$, both inclusion properties can be combined to more clearly match our purposes: *given a line size ls, an LRU cache of associativity* $2 * assoc - 1$ *and set size ss will always hold all the data blocks present in a FIFO cache of line size ls, associativity assoc and smaller and set size ss and smaller*.

### 6.2.3 Hardware Design

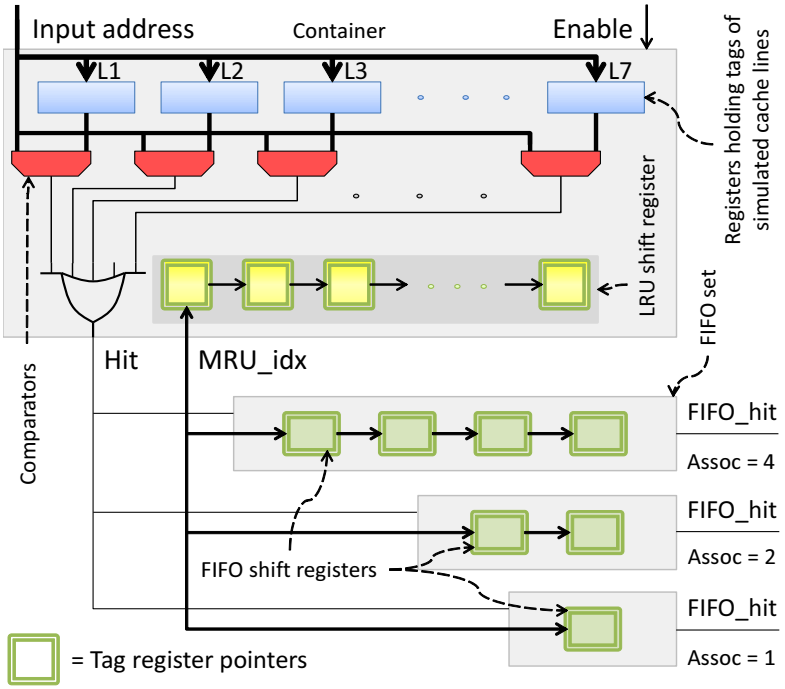



**Figure 6.1:** Top level FIFO set simulating associativities 1, 2 and 4. A container holds and compares the tags, and the tag to be discarded upon a miss is determined by the LRU shift register. Based on the signals output by the container, the state of the FIFO sets can be deduced by the FIFO shift registers.

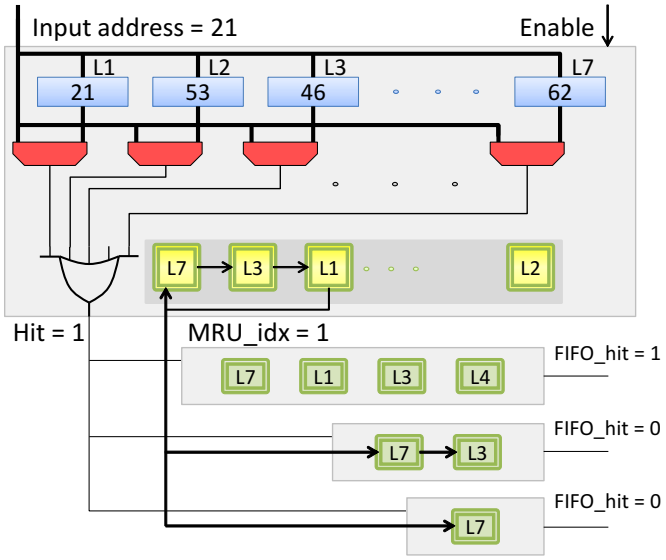



**Figure 6.2:** Top level set at time $t_0$. The input address (21) matches the tag stored at location L1. L1 is therefore the most recently used index and is pushed to the front of the LRU shift register. The FIFO set of assoc. 4 is currently tracking L1 and therefore signals a hit while the other two FIFO sets of assoc. 2 and 1 signal a miss as they are not tracking L1. Arrows indicate the direction in which data will be latched at the next clock tick.

Figure 6.1 shows the design of the top level set which makes use of the first inclusion property. Just like the LRU simulator only the tag and valid bit are stored and not the entire cache line. Each top level set holds a container capable of storing $2 * assoc - 1$ tags at fixed locations (L1, L2, L3, etc.) and compares their value to the input address. If the input address matches one of the stored tags, the $Hit$ signal goes high. The LRU and FIFO shift registers manipulate *tag register pointers* encoded on $log_2(2 * assoc)$ bits, and are therefore very compact. An LRU shift register is employed to determine which tag was least recently used, effectively making the container behave like an LRU set of associativity $2 * assoc - 1$. Upon container hit (as indicated by $Hit = 1$), each FIFO set checks whether the hit location (given by $MRU\_idx$) is stored in its FIFO shift register. If the given tag pointer is present, the FIFO set signals a hit. If the FIFO set does not contain the tag pointer $MRU\_idx$ or if the container signals a miss ($Hit = 0$), the FIFO set signals a miss and the FIFO shift register is updated with the new value.

Figures 6.2 to 6.4 demonstrate the inner workings of a top level set which is also described by the code given in Appendix B.1. Currently only a single set is simulated.

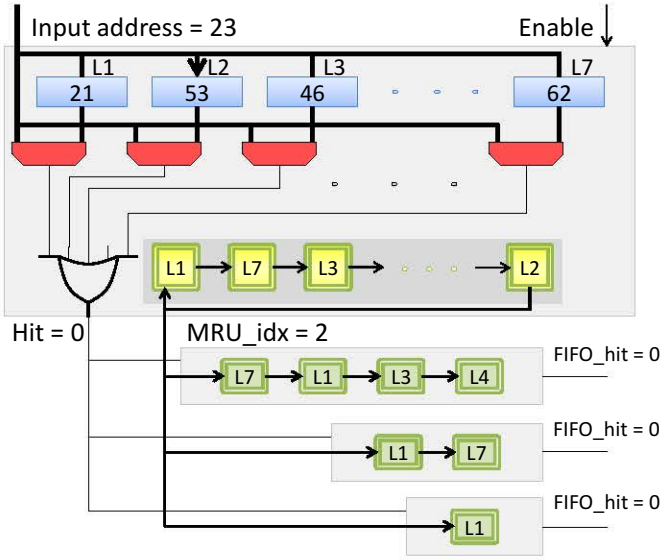



**Figure 6.3:** Top level set at time $t_1$. The address input (23) does not match any of the stored tags. It will therefore be stored in the register containing the oldest data which is pointed to by the last pointer in the LRU shift register, L2. As the container signals a miss, all FIFO sets also signal a miss. All shift registers and the new tag at location L2 are updated.

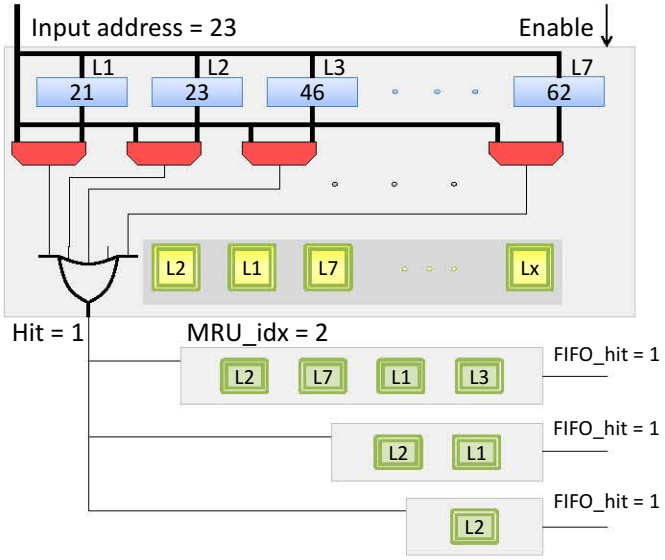



**Figure 6.4:** Top level set at time $t_2$. The address given is stored in the container at location L2. L2 is the most recently used location in the LRU shift register and is being tracked by all the FIFO sets. At the next clock tick the set will not change. Lx indicates the second to last location at time $t_1$ which was not depicted due to space constraints.

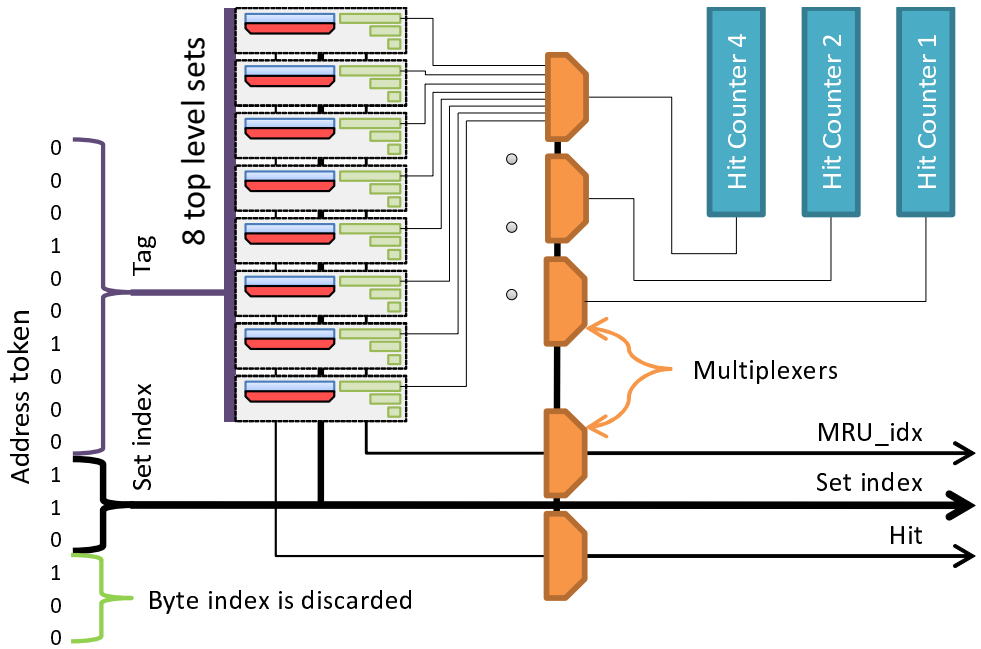



**Figure 6.5:** Top level of the FIFO cache simulator. The multiplexers ensure that only the outputs from the set selected by *Set index* are used.

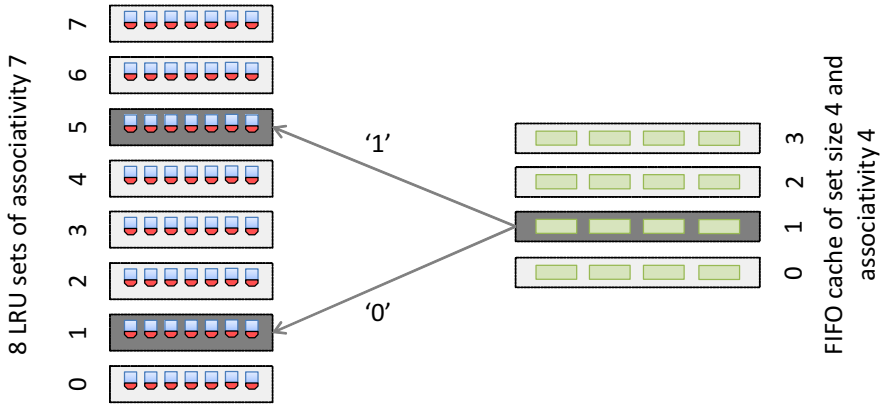



**Figure 6.6:** The data in set 1 of a FIFO cache of set size 4 and associativity $a$ will always also be stored in sets 1 or 5 of a container of set size 8 and associativity $2a - 1$.

To simulate a cache of set size *ss*, *ss* top level sets need to be instantiated. The $FIFO\_hit$ signals of each level of associativity *assoc* can then be multiplexed to determine whether a hit would have occurred in a cache of set size *ss* and associativity *assoc*. Each level of associativity has a hit counter which is enabled by the multiplexed hit output. Figure 6.5 shows a top level design capable of simulating caches of set size 8 and associativities 4 and smaller.

Figure 6.6 demonstrates the two inclusion properties combined. The data stored in set 1 of a FIFO cache of set size 4 has to be present in either set 1 (set index 1 of the lower half) or 5 (set index 1 of the upper half) of the container of twice the set size. A tag register pointer in the given lower level FIFO set will be encoded on $log_2(2 * assoc) + 1$ bits as it also needs to include a set pointer encoded on one bit which is equal to '0' if the tag is present in container 1 and a '1' if the tag is present in container 5. By storing the tags in a container of set size *ss* and associativity *assoc* at the top level, it is possible to simulate FIFO caches of set sizes *ss* and smaller and associativities *assoc* and smaller. The components are combined as shown in Figure 6.7. The lower levels simulate smaller set sizes and are governed by the registered $MRU\_idx$, *Set index* and $Hit$ signals, effectively pipelining the design and increasing the overall operating frequency. By using the two inclusion properties, the number of tags the cache simulator has to store has been reduced to $ss_{max} * (2 * assoc_{max} - 1)$, which is nearly half as many tags than would have been needed for an unoptimised implementation as described in the beginning of this section.

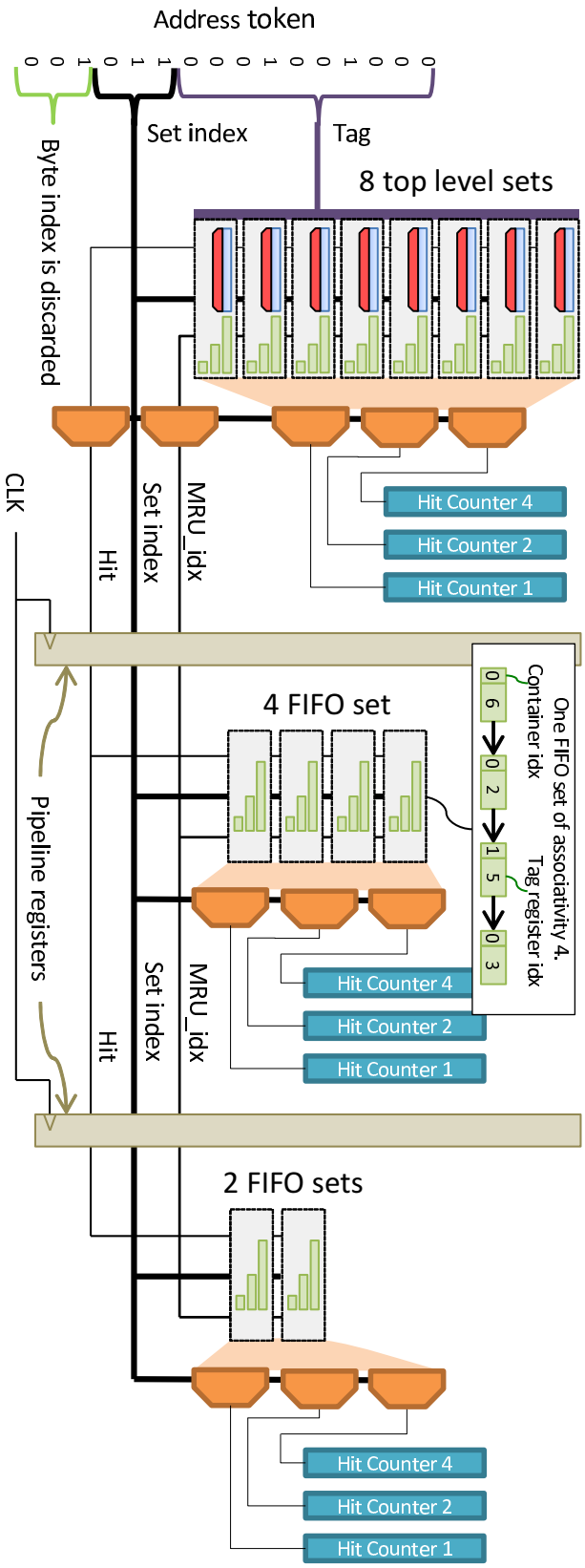



**Figure 6.7:** FIFO cache simulator for caches of set sizes 8, 4 and 2 and associativities 4, 2 and 1. An example 16-bit input address of value 0x1234 is also shown.

| $addr_w$ | $assoc_{max}$ | $ss_{max}$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 32 | | 128 | | 512 | |
| | | LUTs | Regs | LUTs | Regs | LUTs | Regs |
| 22 | 4 | 7763 | 8545 | 26769 | 28405 | 95818 | 100713 |
| | 8 | 15564 | 18271 | 58826 | 65104 | | |
| | 16 | 33260 | 39132 | 122643 | 142122 | | |
| 27 | 4 | 8117 | 9665 | 27595 | 32986 | 106578 | 118572 |
| | 8 | 16412 | 20847 | 61084 | 73812 | | |
| | 16 | 41562 | 46946 | | | | |
| 32 | 4 | 8897 | 10852 | 30415 | 37523 | 111604 | 135458 |
| | 8 | 17834 | 23156 | 65936 | 83428 | | |
| | 16 | 40347 | 49779 | | | | |

**Table 6.1:** MASH{fifo} resource usage measured in terms of Altera Look Up Tables (LUTs) and Registers (Regs) depending on the largest cache simulated.

## 6.3 Implementation

The resulting cache simulator is a self-contained hardware instance written in VHDL. The important parts of the source code can be seen in Appendix B. The same centralised settings file is used as for the MASH{lru} simulator (Appendix A.1) making the design extremely parametrisable based on three main settings:

- $addr_w$, the width of the address tags stored at the top level,
- $assoc_{max}$, the maximum associativity simulated and
- $ss_{max}$, the largest set count simulated.

The settings define the size of the largest cache to be simulated and the hardware to simulate all the smaller caches is instantiated automatically. The line length $ls$ to be simulated is set at run-time with a range between 4 and 128 bytes as only one line length can be simulated at any one time. Using the Altera Quartus [157] software the design was compiled for the same Stratix IV FPGA with 230 thousand logic elements that was used for the MASH{lru} simulator. Despite the modest size of the FPGA when compared to the latest offerings by FPGA vendors, it was possible to compile a cache simulator for a maximum associativity of 4 and set size of 512, i.e., the largest cache that can be simulated

has a size of 256 kB. The relationship between simulator parameters and hardware usage can be seen in Table 6.1 in which $s_{max}$ defines the largest set count, $a_{max}$ the largest associativity and $addr_w$ the address token width simulated. A blank field indicates that the resource requirements would have been too large for the target FPGA.

In terms of throughput, the simulator instance can be clocked at up to 75 MHz on the target device, i.e., the design can analyse an address trace at a rate of 75 million addresses per second. This is $3/4$ of the throughput of the MASH{lru} simulator which is faster as its combinational logic is simpler. However, higher frequencies may be achieved on devices built on more recent technology (e.g., Altera Stratix V). Two slave buses allow for easy integration and usage: one port acts as the address input and the other controls the simulator and is used to obtain the cache statistics at the end of a simulation run. The correct functioning of the simulator was verified in Modelsim [98].

## 6.4 Static Trace Simulation

Conventional cache simulators usually obtain their input from a very large trace file stored on the hard drive of a computer. To use MASH{fifo} to simulate large static traces a storage medium could be connected directly to the FPGA, but for the purposes of fair comparison the FPGA was mounted within the same test rig on which the software-based simulations were run. The target computer contained a quad core Intel Core 2 processor with 3 GB of memory. In order to minimise hard drive read times for the large traces a 256 GB Solid State Drive (SSD) was employed with a tested throughput of 280 MB/s.

A high-speed PCIe data interface was used to transfer the trace file from the test rig to the FPGA. An application had to be programmed that would read the trace file, perform a simple trace-compression algorithm before sending the data to the PCIe interface. Alternatively, the trace could be pre-compacted for faster hard drive read times. On the hardware side, the PCIe interface would send the data to a hardware instance that performs trace decompression before passing the trace on to the cache simulator instance. The different steps involved can be seen in Figure 6.8. The trace compression algorithm

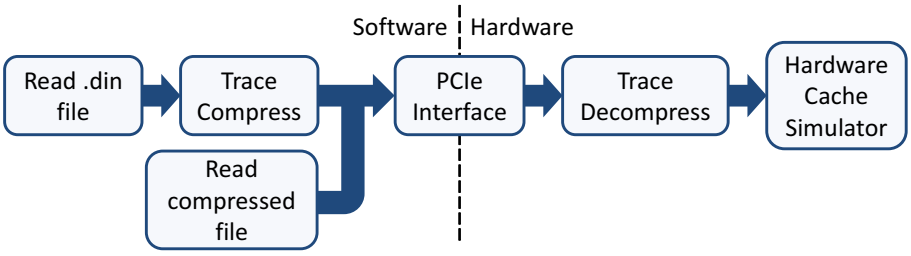



**Figure 6.8:** Steps that are executed in software and hardware allowing MASH{fifo} to perform simulation based on a large static trace. The input trace can be either in the Dinero '.din' format or pre-compressed.

employed is described in Chapter 8. The algorithm was tailored for extremely fast performance when compressing the trace, and the ability to easily decompress the trace in hardware.

The performance of MASH{fifo} was compared with DineroIV [20] and also CIPARSim [121] which is widely regarded as one of the fastest FIFO cache simulators. Separate data and instruction traces of a number of applications were used as input and were encoded in the Dinero '.din' file format. Care was taken that each cache simulator covered the same design space with set sizes of 1 to 512, associativities of 1 to 4 (2 to 4 for CIPARSim) and line lengths of 4 to 64 bytes. A total of 150 (100 for CIPARSim) distinct cache configurations were simulated by each simulator. The traces were obtained by previously running a number of embedded media applications and benchmarks in an Xtensa simulator. For each application cache simulation was performed for the instruction and data trace separately.

The resulting timings and speedups of the experiment can be viewed in Table 6.2. It can be seen that MASH{fifo} is between 7.38x and 11.10x faster than CIPARSim and 180x to 255x faster than DineroIV.

## 6.5 In-System Cache Simulation

Due to the hardware-only nature of the simulator it is possible to instantiate a MASH{fifo} instance within an actual embedded system. Such an in-system simulator would allow a

| Application | | autocorrelate | | convolution | | fir | | idct | | JPEG encoder | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Trace Type | | Inst | Data | Inst | Data | Inst | Data | Inst | Data | Inst | Data |
| Memory Accesses | | 406M | 700M | 71.2M | 71.5M | 58.8M | 98.2M | 15.6M | 26.6M | 161M | 89.8M |
| Simulation time (s) | CIPARSim | 472.61 | 1153 | 96.21 | 84.81 | 70.70 | 133.37 | 23.48 | 36.10 | 199.68 | 149.13 |
| | DineroIV | 12200 | 25084 | 2154 | 2415 | 1749 | 3435 | 477 | 925 | 4849 | 3200 |
| | MASH{fifo} | 64.05 | 134.26 | 8.68 | 9.45 | 7.13 | 14.18 | 2.55 | 4.39 | 19.87 | 13.43 |
| MASH Speedup w.r.t. CIPARSim | | 7.38x | 8.59x | 11.09x | 8.97x | 9.91x | 9.40x | 9.19x | 8.23x | 10.05x | 11.10x |
| MASH Speedup w.r.t. DineroIV | | 190.48x | 186.83x | 248.32x | 255.50x | 245.26x | 242.14x | 186.56x | 210.93x | 244.04x | 238.28x |
| Application | | mandelbrot | | matrix | | MP3 decoder | | MP3 encoder | | MPEG2 encoder | |
| Trace Type | | Inst | Data | Inst | Data | Inst | Data | Inst | Data | Inst | Data |
| Memory Accesses | | 620M | 245M | 1.16B | 1.66B | 831M | 302M | 10.7B | 2.21B | 11.1B | 1.48B |
| Simulation time (s) | CIPARSim | 845.23 | 340.76 | 1522 | 2471 | 1239 | 516.72 | 17897 | 3646 | 16202 | 2232 |
| | DineroIV | 20348 | 8975 | 35771 | 58606 | 26724 | 11259 | 355630 | 82711 | 364884 | 51647 |
| | MASH{fifo} | 105.92 | 44.02 | 184.30 | 318.87 | 148.06 | 55.81 | 1836 | 443.27 | 1859 | 286.22 |
| MASH Speedup w.r.t. CIPARSim | | 7.98x | 7.74x | 8.26x | 7.75x | 8.37x | 9.26x | 9.74x | 8.23x | 8.71x | 7.80x |
| MASH Speedup w.r.t. DineroIV | | 192.11x | 203.89x | 194.09x | 183.79x | 180.50x | 201.73x | 193.62x | 186.59x | 196.25x | 180.45x |

**Table 6.2:** Timing comparison between cache simulators for a number of applications. Each application trace was split into an instruction trace and data trace to simulate instruction and data caches separately.

designer to perform multiple cache simulation, in real time, on an embedded system running an application that is subjected to real-world inputs. Notably, the advantages of such an approach are:

- Fast multiple cache simulation. Given the high throughput of MASH{fifo}, many cache configurations can be simulated quickly.

- Fast trace generation. As mentioned previously, generating a trace from a processor simulator can take a long time especially when running high-performance applications for a wide range of inputs. An in-system MASH approach is significantly quicker as the processor runs in hardware at high frequencies.

- The trace is governed by real-world inputs. If the application input strongly affects program flow (e.g., video encoding, file compression or any application with a user interface) a designer may want to analyse the cache design space given a range of realistic inputs. In contrast, when an application is run on a simulated processor inputs tend to be hard-coded and contrived.

An overview of the embedded system with integrated cache simulator can be seen in Figure 6.9. A NIOS II/f CPU clocked at 200 MHz was employed that uses external DDR3 memory as data memory. MASH{fifo} unobtrusively monitors all the accesses to the DDR3 memory by snooping the data bus thereby obtaining the data trace. On the given NIOS II/f CPU only data caches can be simulated as it is not possible to disable the internal instruction cache as the core is closed source, i.e., instruction accesses by the application may be cached internally in which case they will not be visible on the external instruction bus. If a different CPU had been employed that is open source (e.g., OpenRISC) or where the internal instruction bus is accessible from the outside (e.g., NIOS II/e), instruction caches could also be simulated.

The same applications and benchmarks evaluated in Section 6.4 were implemented on the embedded system. An SD card was used to provide input and output capabilities to the applications. Other input or output devices, such as a camera, could be added and

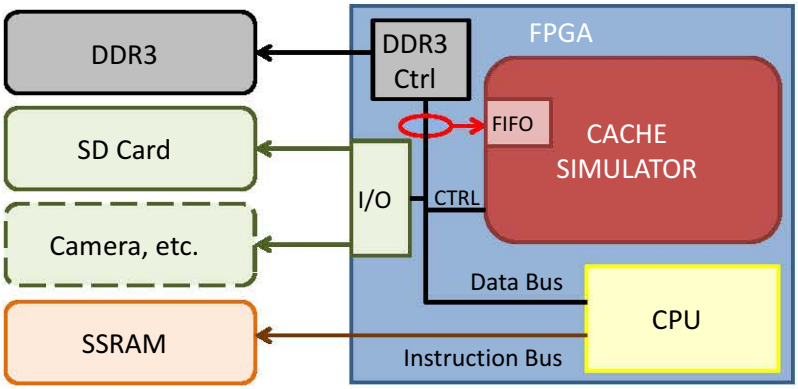



**Figure 6.9:** Overview of the embedded system which includes in-system cache simulation.

are limited only by the available resources remaining on the FPGA. The same FPGA development board was used as in the previous chapter, a depiction of which can be seen in Figure 5.12 highlighting the main components of the system.

The in-system run time was compared with the time it would have taken to create a trace for the given application on a simulated Xtensa system. The Xtensa simulator was run as a single thread on a computer containing 32 Intel Xeon X7560 cores running at 2.27 GHz and 256 GB of system memory. As cache simulation is executed at the same time as application execution, run times are also compared to CIPARSim data cache simulation times from Table 6.2 for those same applications. *It is only a rough comparison for obvious reasons: the processors are built on different architectures, the applications are compiled with different compilers and the input and output interfaces are vastly different. The aim of this experiment is to generally demonstrate the magnitude of the speedup that can be achieved with the hardware-only approach.*

The resulting timings, trace lengths and speedups are given in Table 6.3. The table shows that the number of data memory accesses for the benchmarks (autocorrelate, convolution, fir, idct, mandelbrot and matrix) are very similar for the Xtensa and the in-system implementations. On the other hand, memory access counts varied greatly between the Xtensa and in-system experiments for embedded applications such as the JPEG, MP3 and MPEG2 encoders and MP3 decoder as they perform many I/O operations. On average

| Application | Xtensa | | In-System | | Speedup | |
|---|---|---|---|---|---|---|
| | Time (s) | Trace length | Time (s) | Trace length | Trace gen. | CIPAR -Sim |
| autocorrelate | 24792 | 700M | 35.57 | 657M | 697x | 8.57x |
| convolution | 3616 | 71.5M | 3.47 | 69.5M | 1042x | 5.45x |
| fir | 3590 | 98.2M | 4.61 | 91.5M | 779x | 6.38x |
| idct | 962 | 26.6M | 1.05 | 21.7M | 917x | 7.32x |
| JPEG enc. | 5730 | 89.8M | 16.01 | 277M | 358x | 7.51x |
| mandelbrot | 15115 | 245M | 11.45 | 249M | 1320x | 7.41x |
| matrix | 66941 | 1.66B | 90.69 | 1.39B | 738x | 5.51x |
| MP3 dec. | 22633 | 302M | 28.86 | 594M | 784x | 10.55x |
| MP3 enc. | 322894 | 2.21B | 818.11 | 14.9B | 395x | 7.74x |
| MPEG2 enc. | 255397 | 1.48B | 957.73 | 3.20B | 267x | 1.37x |

**Table 6.3:** A rough comparison between the time taken to run an application on a simulated Xtensa processor and on an embedded system with in-system cache simulator. The speedup shown with respect to CIPARSim is in terms of *throughput* as the trace lengths differ.

the NIOS II processor generated the trace 730x faster than the Xtensa simulator with minimum and maximum speedups of 267x and 1320x respectively. When compared with CIPARSim throughput (trace length divided by time), the throughput of the in-system cache simulator was on average 6.78x times faster while covering the same cache design space. One notable outlier is the MPEG2 encoding application, where cache simulation was only 1.37x faster in hardware than in software. The modest speedup can mostly be explained by the fact that the Xtensa processor includes a floating point accelerator whereas the NIOS II processor does not.

Implementing MASH{fifo} in a NIOS II system is trivial as a cache simulator component has been created for the Altera Qsys SoC design software. In fewer than 10 mouse-clicks a designer can include a MASH{fifo} instance in their system, simulating caches for any memory bus by routing it through the simulator.

## 6.6 Summary

MASH{fifo} has been presented which is capable of determining the hits and misses for many caches that employ the FIFO cache replacement policy. FIFO cache inclusion

properties have been discovered which enable the hardware simulator to evaluate 30 cache configurations simultaneously at an address token consumption rate of 75 MHz. In a direct speed comparison, MASH{fifo} was up to 11.10x faster than one of the fastest software based FIFO cache simulators, CIPARSim. In-system implementations of the cache simulator were also investigated where a simulating instance is inserted in a real-world embedded system to perform real-time cache analysis. Traces were generated 730x faster on average when compared to the Xtensa processor simulator, and cache simulation throughput was increased by an average of 6.78x compared to CIPARSim.

# Chapter 7

# MASS{plrut}: Optimised PLRUt Cache Simulation

## 7.1 Introduction

The LRU inclusion property has been shown to be extremely effective with the downside that its hardware design is very complex. As stated earlier, the FIFO replacement policy shows worse performance but is very cheap to implement and hence the FIFO algorithm is frequently used in embedded systems. However, the PLRU replacement policy is advantageous on both facets as it is characterised by low-complexity hardware while being nearly as effective as the LRU algorithm [156]. For this reason it is employed in a range of high-performance Intel and PowerPC processors.

Quickly exploring the design space of PLRU caches is difficult, however, as old data can remain in a cache indefinitely for some cache configurations given certain access patterns. For this reason, different PLRU caches are generally simulated separately with little room for optimisation. In this chapter an alternative approach towards PLRU cache simulation is presented, specifically targeting the PLRU–tree (PLRUt) algorithm. Another facet of cache simulation is explored which is rarely mentioned in the literature which is the disk access performance. The contributions are as follows:

- the first optimised single pass PLRUt cache simulator based on a hash table is presented named Mulitple cAche Simulator in Software, or MASS{plrut},
- for the first time, two PLRU union properties are defined based on which it is demonstrated that the memory usage of an optimised single pass PLRUt cache simulating algorithm has an upper bound significantly below that of a brute–force algorithm, and,
- a fast tree–updating method is proposed based on look–up tables that can be used to quickly point the PLRU tree to the Most Recently Used (MRU) tag and does not require loops.

There are hardly any proposed simulator designs targeting the PLRU replacement policy. Tawada et al. [15] implemented a PLRUt simulator with a CRCB–2 optimisation: if the same address is processed twice in a row all hit counters can be incremented without having to update cache states, a property already implemented in Cheetah for LRU simulation. Performance was boosted by an additional 6.91% on average by using a priority queue (presented as the 'CSG' optimisation). In his thesis, Mohammad Haque [116] proposes PSAICO which improves on CRCB–2 by stating that the *two* most recently used tags will also be stored in all caches of larger associativity and set size assuming a constant line length. Section 7.3 establishes that this property is a subset of the PLRU property presented in this chapter. Additionally, both these simulators compare their results with DineroIV which is a very poor reference point (see Section 7.5). The findings covered in this chapter have been presented at the Asia South Pacific Design Automation Conference in January 2015.

Note: due to the frequent employment of equations in this chapter, cache parameter names have been shortened, from *assoc* to *a* for associativity, *ss* to *s* for set size and *ls* to *l* for line size.

## 7.2 The PLRUt Replacement Policy

The PLRUt replacement policy relies on a tree data structure to determine which item of data should be evicted from the cache upon a miss. Every tree node points towards the most recently used of its two branches. Figure 7.1 shows a PLRUt cache set where $a = 4$.

The manner in which a PLRUt cache set deals with hits and misses is depicted in Figures 7.2 and 7.3, representing a set hit and a set miss respectively. Two types of operations need to be performed on the tree:

- *update*, pointing the tree to the MRU item of data, and
- *get evict*, determining which node should be evicted on the next miss.

If a cache set is subject to a hit an update operation is performed on that tree and the cache line with the matching tag is set as the most recently used item of data. A cache miss, on the other hand, initialises a get evict operation followed by an update, i.e., the cache line to be evicted is determined, the new data is loaded into that new location, and the tree is then updated to mark this new cache line as the MRU.

Both the update and get evict operations can be efficiently implemented in hardware yet software implementations need to loop through each layer of the tree which can be inefficient, especially for caches with large associativities. Updating the tree is performed more frequently, and is purely a matter of ensuring that certain nodes are in certain states while leaving the remainder of the other nodes untouched. If the tree is represented as a row of bits stored in a word variable, it is possible to update the tree by applying a single bitwise 'and' followed by an 'or' operation on this variable with respect to a pair of constants stored in a lookup table, i.e., updating the tree can be performed in two steps as demonstrated in Figure 7.4. The lookup table can be hardcoded and needs to store $2 * assoc$ masking values with a minimum bit-width of $assoc - 1$, where $assoc$ is the maximum associativity simulated, i.e., to simulate associativities of up to 32, an optimised PLRUt tree updating function requires a lookup table of at least sixty-four 31-bit integers. An optimised cache update implementation can be seen in Appendix C.1.

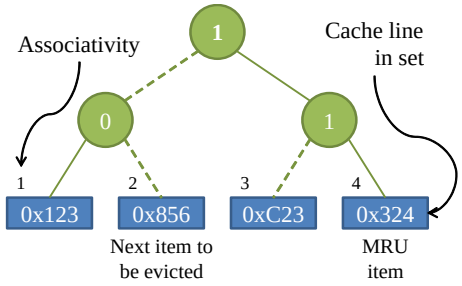



**Figure 7.1:** The PLRUt tree structure for a cache set of associativity 4. Each node indicates whether the least recently used item of data is stored in the left or right branch, denoted by a '0' and a '1' respectively. The numbers in the cache lines indicate the address of the data contained therein.

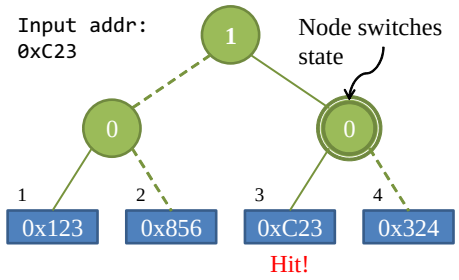



**Figure 7.2:** From the cache set in Figure 7.1 data is accessed at address 0xC23 which is already stored in the cache. The set signals a hit and the tree is updated to point to this least recently used data.

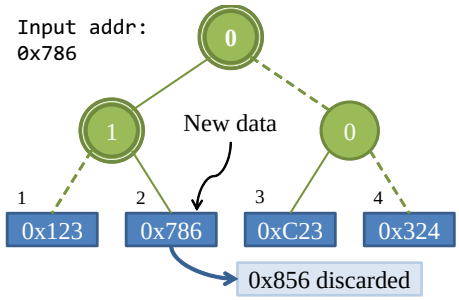



**Figure 7.3:** Data is accessed at address 0x786 which is not stored in the cache. The set signals a miss and the item pointed to for eviction (with address 0x856) is discarded. The top and left node flip states, making the new item of data the least recently used line.

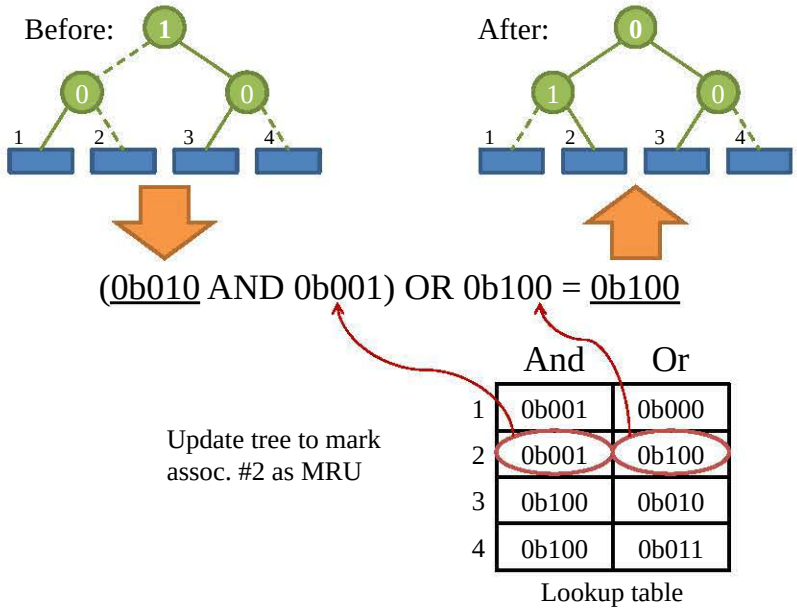



**Figure 7.4:** Example of updating the tree *Before* to point to the data at associativity number 2, resulting in tree *After*. A value pair is obtained from the lookup table at index 2, the *And* value is 'and-ed' with the value of the tree, the result of which is 'or-ed' with the *Or* value.

## 7.3 Cache Overlap

As seen in Chapter 5, LRU caches overlap completely due to their inclusion properties in that all the data stored in a cache will always be present in a cache of larger associativity and/or set size given the same line length. This is not the case with FIFO or PLRUt caches where data can be present in a smaller cache only without being available in a larger cache. For MASH{fifo} inclusion properties were developed for the FIFO replacement policy by taking into account the longest lifetime of an item of data. For the PLRUt algorithm it is possible to prove that there is a minimum of overlap amongst different PLRUt cache configurations (assuming identical line lengths) by observing the *shortest* possible lifetime of cached data.

When an item of data $d_0$ is inserted or accessed in a cache set it is marked as the most recently used data, meaning that all the tree nodes between the target cache line and the root of the tree point towards the target cache line (as seen in both example sets in Figure 7.5). In the case of PLRU set A in Figure 7.5 representing a cache set of

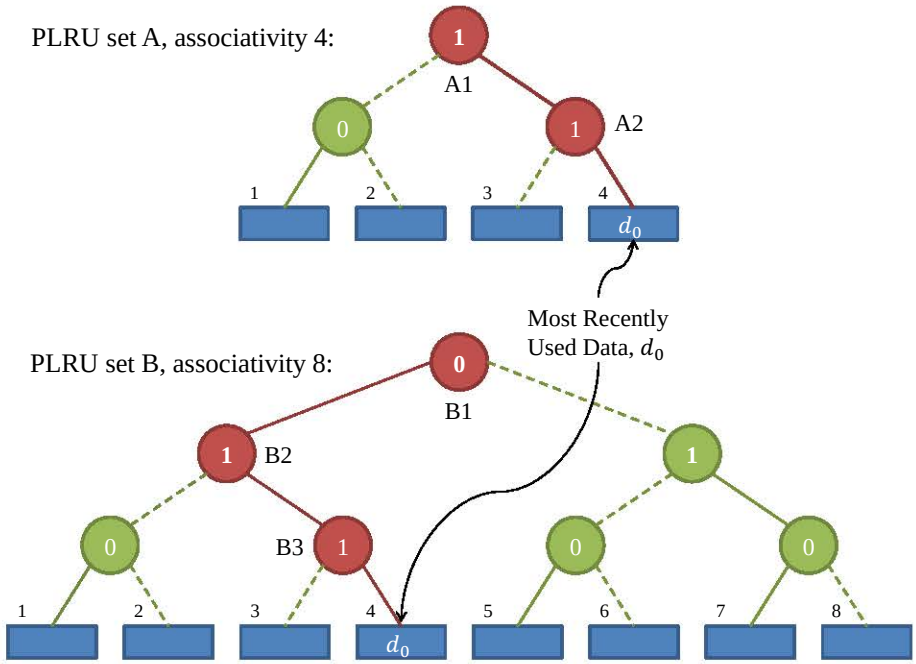



**Figure 7.5:** Possible state of two cache sets A and B that have an associativity of 4 and 8 respectively and are part of two caches of set size 1 and identical line length. Both caches are presented with the same trace and an item of data, $d_0$, has just been accessed. As a consequence, nodes A1 and A2 in set A, and nodes B1, B2 and B3 in set B point towards $d_0$ in the respective cache sets.

associativity 4, nodes A1 and A2 have to change state and a miss has to occur for $d_0$ to be evicted, which means that at least *three* different addresses that are not $d_0$ must be accessed before it is possible for $d_0$ to no longer be in that cache set. The tree in set B of associativity 8 has an additional level of nodes (with a total node count of 7 instead of 3) meaning that at least *four* different addresses need to be accessed before set B may no longer hold $d_0$. This means that as long as we are sure that $d_0$ is stored in set A, we can also be certain that it is stored in set B.

This union property can be described as a minimum certain overlap between sets A and B: at any point in time we can always be sure that at least three of the data items in set A are also stored in set B. The amount of overlap between two cache sets of successive associativity depends on the size of the smaller tree and is equal to the number of levels in this tree plus one, in other words, the amount of minimum overlap between set A and set B is $log_2 a_{setA} + 1 = 3$ which is also equal to $log_2 a_{setB}$. This holds true for every

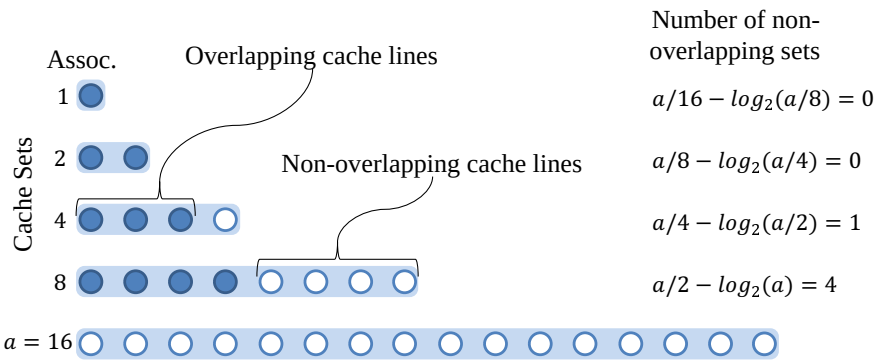



**Figure 7.6:** Demonstration of how many cache lines overlap with the cache lines of the set of larger associativity. The number of non-overlapping cache lines, in this case 21, is the number of maximum data items that will need to be tracked by a cache simulator simulating associativities of 1 to 16. Note: the cache lines previously represented as squares are now shown as circles.

set of consecutive associativity: the overlap between set B and a hypothetical set C of associativity 16 would be $log_2 a_{setC} = 4$.

The increasing amount of overlap between cache sets up to an associativity of 8 can be seen in Figure 7.6. The property can be exploited in that fewer unique items of data need to be tracked during simulation as the contents of the overlapping cache lines are already stored in the set of largest associativity. Figure 7.6 shows that simulating cache sets up to associativity 16 requires keeping track of at most $lcount_{opt} = 16+4+1 = 21$ cache lines at any one time instead of the $16+8+4+2+1 = 31$ required for an un-optimised implementation. Generalising the number of cache lines required to optimally simulate all sets of associativity $a$ and smaller in a cache of set size 1:

$$\begin{aligned} lcount_{topLevelSet} &= a + \frac{a}{2} - log_2 a + \frac{a}{4} - log_2 \frac{a}{2} \ldots \\ &= a + \frac{a}{2} + \frac{a}{4} \ldots - log_2 a - log_2 \frac{a}{2} \ldots \\ &= 2a - 1 - \frac{log_2 a (log_2 a + 1)}{2} \\ &= 2a - 1 - \frac{(log_2 a)^2}{2} - \frac{log_2 a}{2} \end{aligned} \tag{7.1}$$

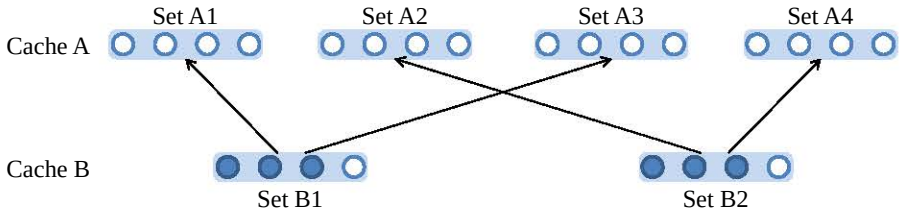



**Figure 7.7:** Two caches A and B of associativity 4 and set sizes 4 and 2 respectively. The three overlapping cache lines of set B1 will be present in sets A1 and A3 and of set B2 in sets A2 and A4 respectively.

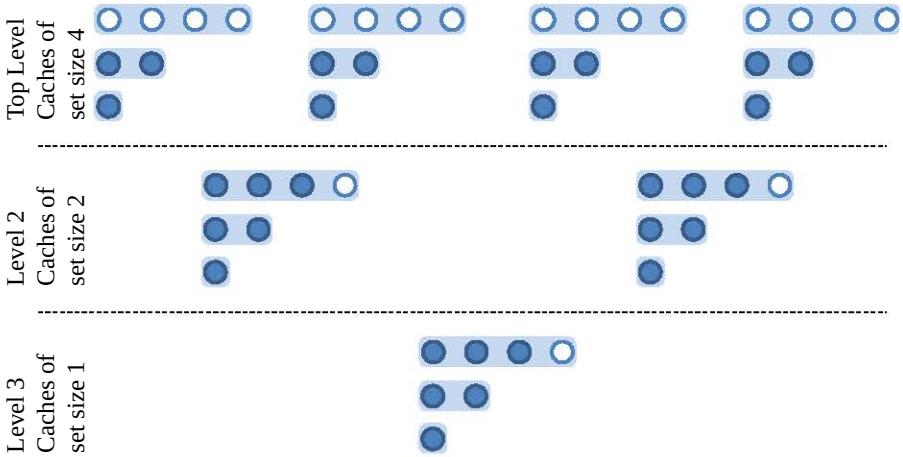



**Figure 7.8:** Cache line overlap between caches of associativities 1 to 4 and set sizes 1 to 4. A total of 19 cache lines do not overlap which means that a maximum of 19 cache lines are required to simulate these nine cache configurations instead of 49 for an un-optimised implementation.

And for set sizes $s$ larger than 1:

$$lcount_{topLevel} = (2a - 1 - \frac{(log_2 a)^2}{2} - \frac{log_2 a}{2}) * s \tag{7.2}$$

The first union property is as follows: *Given PLRUt caches of set size s, line length l and associativities a and smaller, the maximum number of different cache lines that need to be stored is* $(2a - 1 - (log_2 a)^2/2 - log_2 a/2) * s$.

Not only do caches of different associativities overlap, caches of different set sizes do too. Figure 7.7 shows two caches A and B of associativity 4 and constant line length. The set size of cache A is 4 while that of cache B is 2. Given that sets A1 and A3 are subject to the same memory accesses as set B1 and that the shortest lifetime of an item of data is

three for all those sets, we can be sure that three sets of B1 will always be stored in either A1 or A3, i.e., that they will be overlapping. The same applies to set B2 with respect to sets A2 and A4. This results in many overlapping cache lines when simulating caches of varying set sizes and associativities (Figure 7.8).

To calculate the total number of cache lines required to simulate a set of caches of set sizes $s$ and smaller, associativities $a$ and smaller and constant line length, the line count from the caches with the largest set size (given by Equation 7.2) is added to the number of non-overlapping cache lines for the smaller set sizes. The total non-overlapping lines for caches of set size one is equal to the value given by Equation 7.1 minus the overlap for the largest associativity, $log_2 a + 1$:

$$\begin{aligned} lcount_{lowLevelSet} &= 2a - 1 - \frac{(log_2 a)^2}{2} - \frac{log_2 a}{2} - log_2 a - 1 \\ &= 2a - 2 - \frac{(log_2 a)^2}{2} - \frac{3log_2 a}{2} \end{aligned} \tag{7.3}$$

Bringing the total number of cache lines a simulator needs to track to:

$$\begin{aligned} lcount_{total} &= (2a - 1 - \frac{(log_2 a)^2}{2} - \frac{log_2 a}{2}) * s + (2a - 2 - \frac{(log_2 a)^2}{2} - \frac{3log_2 a}{2}) * (s - 1) \\ &= (4a - 3 - (log_2 a)^2 - 2log_2 a) * s - 2a + 2 + \frac{(log_2 a)^2}{2} + \frac{3log_2 a}{2} \end{aligned} \tag{7.4}$$

Which leads to the second union property: *Given PLRUt caches of line length l, set sizes s and smaller and associativities a and smaller, the total number of different cache lines that need to be stored is [Equation 7.4].*

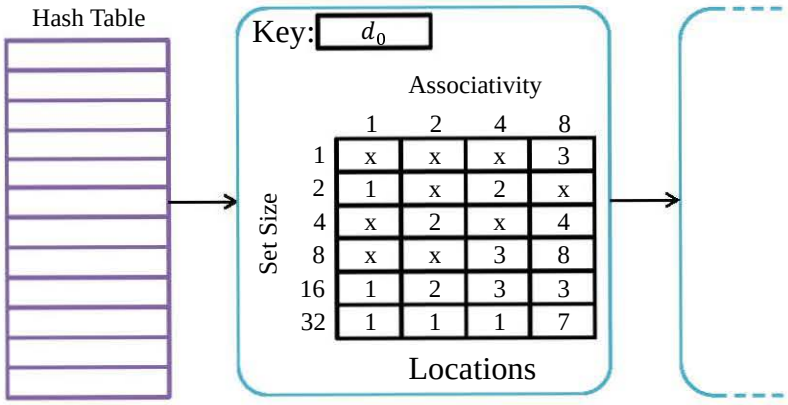



**Figure 7.9:** Hypothetical entry in the hash table for address $d_0$ in a simulator covering cache configurations of set sizes up to 32 and associativities 8 and smaller. Every entry in the Locations table indicates the associativity $d_0$ is stored in in the respective configurations. For example, $d_0$ is stored in cache line 7 in the cache of set size 32 and associativity 8. Every configuration marked with an 'x' no longer contains $d_0$.

## 7.4 Simulator Design

Ideally a hardware based PLRUt simulator would have been implemented in the same vein as MASH{lru} and MASH{fifo}, but as the union properties are not easy to exploit in hardware, the decision was made to implement a Multiple cAche Simulator in Software (hence MASS{plrut}) instead. Much of the source code can be seen in Appendix C.

An effective way to pool all the $lcount_{total}$ cache lines in a fast and accessible manner is to store them in a hash table of size $lcount_{total}$ where the address of the accessed memory is hashed to obtain the table index. As the table may be subject to collisions, addresses yielding the same table index are chained in linked lists. At most $lcount_{total}$ entries will ever be stored in the hash table meaning that these structures can be allocated at initialisation in an 'entry pool', thereby saving on the overhead of dynamic memory allocation and freeing operations. Every entry in the hash table naturally contains the key (i.e., the address) but also a byte-array of locations, one byte for every cache configuration simulated. Each location byte indicates whether this address is stored in that cache configuration and if it is, in which cache line (associativity) it is located. An example hash table and linked list are depicted in Figure 7.9.

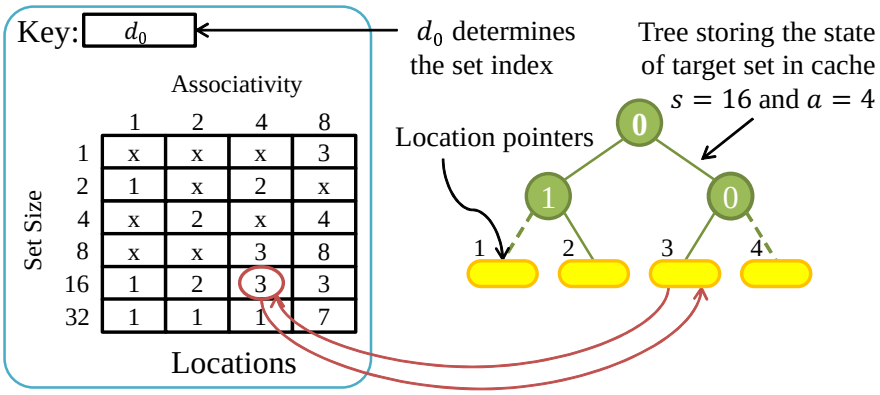



**Figure 7.10:** Hash table entry from Figure 7.9 and its relation to one of the cache lines it is tracking.

A separate data structure is employed to hold all the PLRU trees, one tree for every set in every cache configuration simulated. Each tree additionally stores $a$ byte-pointers, where $a$ is the associativity of the cache configuration the tree is part of. The purpose of these byte pointers is so that every tree can keep track of where the location bytes pointing towards it are stored in the hash table, i.e., the location byte points to the cache line and the cache line points to the location byte (Figure 7.10). If the cache set suffers a miss, the byte pointer of the evicted cache line is used to clear the now obsolete location entry in the hash table (which in the figures is equivalent to writing an 'x' to it).

For every address in the memory trace the cache simulator performs two successive steps:

1. Check whether the input address is stored in the hash table. If it is return the entry, otherwise create a new entry, link it to the hash table and return the new entry instead.

2. Use the returned entry from step 1) to update the target cache set for every cache configuration simulated.

Step 2) is detailed in Algorithm 1 where the update() and getEvict() functions are the ones presented in Section 7.2. The only important step not covered yet is the removal of obsolete entries in the hash table. An entry is obsolete when all of its location bytes

**Algorithm 1** Update cache set states

**Require:** entry, target entry from hash table
**Require:** setStates, stores all set states of simulated caches

```
 1: for all s in set sizes do
 2:   for all a in associativities do
 3:     targetSet = setStates[s][a][entry.addr.setIdx]
 4:     location = &entry.locations[s][a]
 5:     if location is 'x' then
 6:       newLine = getEvict(targetSet)
 7:       *targetSet.lines[newLine]->location = 'x'
 8:       *location = newLine
 9:       targetSet.lines[newLine]->location = location
10:       update(targetSet, newLine)
11:     else
12:       update(targetSet, newLine)
```

no longer track a cache line (in the given examples it means they all contain 'x'). One solution is to scan the entire linked list for a given table index every time the hash table is accessed; this is in the hopes that the linked lists remain short resulting in quicker search times. It is possible, though unlikely, that old entries silently accumulate in the hash table, a situation that can be detected when the entry pool runs out of free entries. This can trigger a garbage collect which scans the entire hash table and removes obsolete entries.

Additionally, the repeat address optimisation which is used in Cheetah [111] and is described as CRCB–2 by Tawada et al. [15] is also implemented.

## 7.5 Experimental Setup

The existing research on PLRUt cache simulators ( [15, 116]) contains a number of drawbacks. Firstly, the optimisations presented are trivial. More importantly though, the main reference point for experimentation is a modified version of DineroIV [20] which is not a single pass simulator. The trace therefore needs to be re-read for every single cache configuration resulting in excessively high disk-access overheads. Additionally, the algorithms and data structures implemented in DineroIV are remarkably ill suited for the

simulation of PLRUt caches, mostly because cache states are stored in linked lists while simple PLRUt implementation requires the tags to remain stationary while storing the set state in a tree.

The MASS{plrut} simulator is therefore compared with two simulator implementations: a DineroIV variant that has been modified to make it obey the PLRUt replacement policy and a CRCB cache simulator implementation that simulates each cache separately and makes use of the repeat address optimisation. It is therefore identical to Tawada's CRCB simulator without the minor CSG optimisation (see Section 7.1). All experiments were run on a computer with a quad core Intel Core 2 processor running at 2.4 GHz with 3 GB of memory. To minimise disk access overhead, traces were read from a 256 GB Slid State Drive (SSD) yielding a measured maximum sequential read speed of 280 MB/s.

The Cheetah source code was used as base for the CRCB and Hash Table-based (HT) simulators as it is a modular, well-programmed and open source piece of software that can easily be extended. The code was also modified for enhanced trace reading speed and parsing of the Dinero trace input format. Additionally a simple trace compression (TC) and decompression algorithm was used to further lower the disk access time of both the CRCB and HT simulators. If the trace needs to be read multiple times the simulator can create a compressed version of the trace in the first run and then employ it in all subsequent simulations. As MASS{plrut} can only simulate one line length at a time, trace compression can be beneficial when consecutively simulating caches of different line lengths. The trace compression algorithm is described in depth in Chapter 8.

The traces of eight benchmarks were used to evaluate the different cache simulators. Caches of set sizes 1 to 1024 and line lengths 4 to 64 were simulated in every simulation run. Associativities of 4 and smaller and 8 and smaller were simulated separately; the larger the maximum associativity simulated the slower the CRCB implementation will be as every miss forces a search through $a$ cache lines while the hash table access is still performed in constant time for the optimised simulator. DineroIV was exclusively run at a maximum associativity of 4 to save time and also to provide a rough estimate of the kind of speedups obtained as a reference point for comparison with previous research.

## 7.6 Results

As the optimised hash table-based cache simulator with trace compression (HT + TC) performs the quickest, the results are displayed as speedup with respect to that simulator. The computation times are compared with DineroIV, a CRCB implementation, a CRCB simulator with trace compression and a Hash Table-based simulator without trace compression. The results of the experiments can be seen in Table 7.1.

The HT + TC simulator was 28.33× faster than DineroIV on average while the CRCB simulator implementation was on average 21.20× faster. Interestingly, Tawada et al. [15] claimed that their CRCB implementation was up to 200× faster than DineroIV. This larger speedup may be explained by a number of factors. Tawada et al. were simulating more cache configurations, 380 instead of our 165, and the disk access overhead of DineroIV is proportional to the number of configurations simulated while for a single pass simulator the disk access overhead remains constant as the trace is only read once. It is also likely that they were not employing solid state storage which is characterised by significantly faster disk access times compared to conventional hard disk drives; if this was the case the disk access overhead would have been significantly and proportionately exacerbated. We omit a comparison with the work by Haque due to questionable results[1].

Speedup with respect to the CRCB cache simulator with trace compression (CRCB + TC) gives an idea of the contribution in speedup as a consequence of using the optimised cache simulator design based on a hash table which we presented in Section 7.4. The maximum and average speedups caused alone by the hash table design are $max = 1.82\ (44.94\%)$ and $1.18\ (15.35\%)$, and $average = 1.58\ (36.51\%)$ and $1.13\ (11.73\%)$ for associativities of 8 and 4 respectively. As explained at the end of Section 7.5, even greater speedups could be achieved at greater associativities. Combining trace compression and the optimised hash table-based algorithm resulted in maximum and average speedups of $max = 1.93\ (48.09\%)$ and $1.42\ (29.82\%)$, and $average = 1.72\ (41.71\%)$

[1]The optimised single-pass simulator by Haque [116] is only four times faster than DineroIV for a line size of four bytes which is extremely unlikely given that he is simulating 75 cache configurations, especially if a standard hard drive was used instead of an SSD.

| Application | | autocorrelate | | | | convolution | | | | fir | | | | JPEG encoder | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Trace Type | | Inst | | Data | | Inst | | Data | | Inst | | Data | | Inst | | Data | |
| Memory Accesses | | 406M | | 700M | | 71.2M | | 71.5M | | 58.8M | | 98.2M | | 161M | | 89.8M | |
| Max. Associativity | | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 |
| Speedup w.r.t. | DineroIV | 28.5 | | 23.0 | | 27.8 | | 31.2 | | 28.8 | | 30.2 | | 28.6 | | 22.5 | |
| | CRCB | 1.30 | 1.61 | 1.19 | 1.93 | 1.35 | 1.59 | 1.24 | 1.72 | 1.31 | 1.63 | 1.23 | 1.91 | 1.34 | 1.76 | 1.25 | 1.68 |
| | CRCB + TC | 1.10 | 1.45 | 1.06 | 1.82 | 1.17 | 1.45 | 1.06 | 1.57 | 1.11 | 1.47 | 1.09 | 1.78 | 1.17 | 1.61 | 1.14 | 1.59 |
| | HT | 1.25 | 1.17 | 1.08 | 1.11 | 1.21 | 1.12 | 1.18 | 1.12 | 1.20 | 1.17 | 1.13 | 1.15 | 1.19 | 1.16 | 1.09 | 1.11 |

| Application | | mandelbrot | | | | MP3 decoder | | | | MP3 encoder | | | | MPEG2 encoder | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Trace Type | | Inst | | Data | | Inst | | Data | | Inst | | Data | | Inst | | Data | |
| Memory Accesses | | 620M | | 245M | | 1.16B | | 1.66B | | 10.7B | | 2.21B | | 11.1B | | 1.48B | |
| Max. Associativity | | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 | 4 | 8 |
| Speedup w.r.t. | DineroIV | 36.8 | | 33.0 | | 27.7 | | 23.6 | | 26.7 | | 23.7 | | 34.6 | | 26.5 | |
| | CRCB | 1.42 | 1.65 | 1.33 | 1.93 | 1.35 | 1.71 | 1.23 | 1.59 | 1.31 | 1.65 | 1.27 | 1.69 | 1.40 | 1.72 | 1.32 | 1.71 |
| | CRCB + TC | 1.15 | 1.43 | 1.13 | 1.75 | 1.18 | 1.57 | 1.09 | 1.48 | 1.15 | 1.52 | 1.17 | 1.60 | 1.17 | 1.53 | 1.17 | 1.58 |
| | HT | 1.31 | 1.21 | 1.21 | 1.16 | 1.16 | 1.14 | 1.08 | 1.07 | 1.16 | 1.10 | 1.14 | 1.06 | 1.24 | 1.20 | 1.15 | 1.12 |

**Table 7.1:** Speedups offered by the optimised hash table-based cache simulator with trace compression with respect to DineroIV (DineroIV), a CRCB implementation (CRCB), a CRCB implementation with trace compression (CRCB + TC) and the Hash Table-based cache simulator without trace compression (HT) for a number of benchmarks. Each trace was split into an instruction trace and data trace to simulate instruction and data caches separately. The speedups achieved when simulating max associativities of 4 and 8 show that the optimised simulators scale better with larger associativities.

and 1.30 (23.28%) for associativities of 8 and 4 respectively. These increased speedups demonstrate the improvements that trace compression can bring and are quantified by the speedup up achieved in comparison to the hash table-based simulator without trace compression (HT) yielding an average improvement of 13.35%.

## 7.7 Summary

For the first time two union properties have been introduced that are applicable to the PLRUt cache replacement policy that sets an upper bound to the memory requirements of a PLRUt single pass simulator. Additionally, a number of optimisations have been proposed which can be employed to speed up the execution of precise PLRUt single-pass simulators. The optimisations include a tree-updating method that can be performed in two simple steps using a look-up table, trace compression, and a hash table-based simulation approach. All have been implemented in a software-based cache simulator called MASS{plrut}. As a result, the optimised simulator is up to 1.93× faster than a non-optimised simulator, averaging a speedup of 1.72× for a maximum simulated associativity of 8.

# Chapter 8

# Trace Compression

## 8.1 Introduction

The benefits of trace compression have been amply demonstrated in the past by authors such as Johnson et al. [131], Luo et al. [133], Janapsatya et al. [135] and Aleksander and Milena Milenkovic [136]. As discussed in Section 2.3.7, trace compression creates a more compact version of a trace file which is quicker to read from a permanent storage device than a raw, un-compressed trace file. A frequently overlooked facet of trace compression is the time it takes to compress and decompress the trace. Trace compression can even *worsen* the performance of cache simulators if the time gained from reading a compacted trace is cancelled out by the time it takes to decompress the file. What is also of interest in the research context of this dissertation is the ability to perform trace decompression *in hardware* to speed up the simulation times of hardware-based simulators such as MASH{lru} and MASH{fifo}. This chapter presents a trace compression algorithm that focuses on simplicity, speed of compression and decompression and the ability to perform decompression in hardware. To simplify the algorithm and minimise hardware complexity the compressed output is byte-aligned so that a minimum of shift operations are required. The proposed algorithm improves on the technique presented by Luo et al. [133] where address offsets are encoded using data types of different sizes.

| Address | Difference |
|---|---|
| 402657512 | |
| 402657514 | +2 |
| 402657516 | +2 |
| 402657516 | +0 |
| 402657518 | +2 |
| 402657520 | +2 |
| 402657514 | -6 |
| 402657516 | +2 |
| 402657518 | +2 |
| 402657520 | +2 |
| 402657514 | -6 |
| 402657516 | +2 |

**Table 8.1:** Address sample.

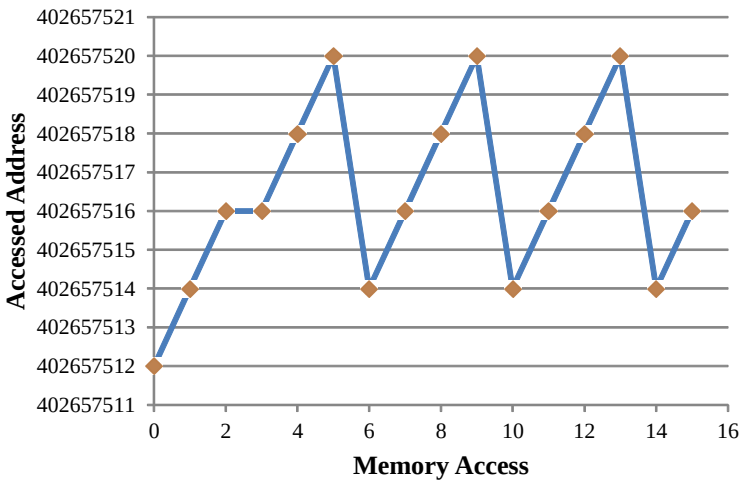



**Figure 8.1:** Address sample with three loop iterations.

The proposed improvements include defining ‘floats’ and detecting repetitions. The trace compression method described here was successfully employed to speed up cache simulation in Chapters 6 and 7.

## 8.2 Address Order Observations

The order of memory accesses of an application will often display predictable behaviour due to the nature of program execution on an in-order processor. Note that it is assumed that the address width is at least 32 bit, and that therefore the lowest two bits of the address are ignored in the current implementation.

### 8.2.1 Small Address Jumps

Though addresses themselves are large, the difference between consecutive instruction addresses is usually very small as one instruction is executed after the other. Other than linear address progression, programs primarily execute loops where a small number of consecutive instructions are executed followed by a jump to an address visited shortly before. Such an address progression can be seen in Figure 8.1 which is a trace sample of a JPEG encoding application running on an Xtensa processor. A snippet of this sample is

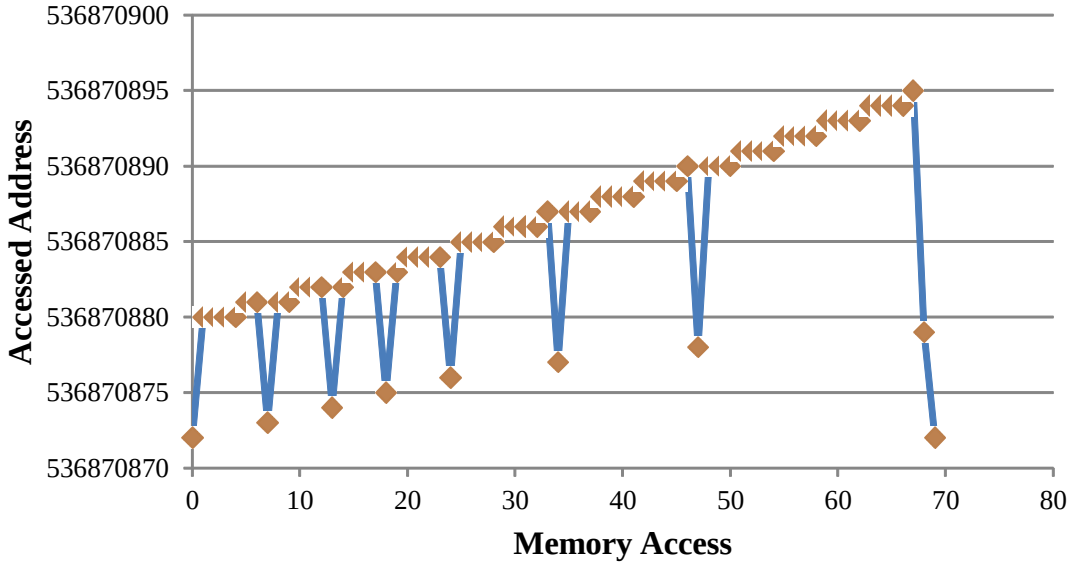
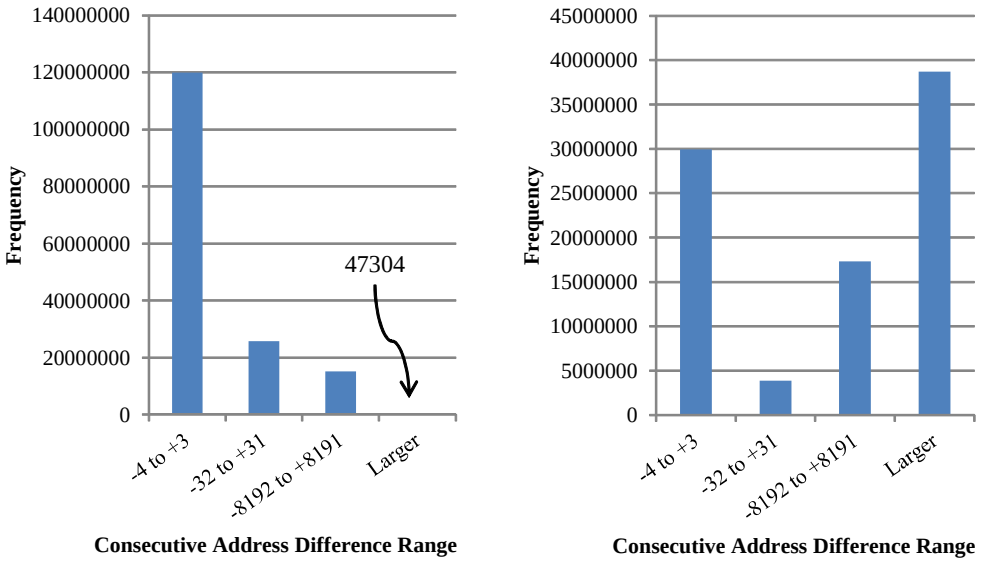



**Figure 8.2:** Sample of a data memory trace.

shown in Table 8.1 showing addresses and difference between the consecutive addresses. Similarly, data memory accesses also display a significant amount of memory access locality, though large jumps are much more common. A sample of data memory accesses for the same JPEG application can be seen in Figure 8.2.

These address differences can be efficiently encoded on data types that are much smaller than the full 32-bit address type. The decision was made to create 3 special data types, one 3-bit for values -4 to 3, one 6-bit for values -32 to 31 and one 14-bit for values -8192 to 8191. The histogram in Figure 8.3a shows how the address differences of an instruction trace fall into these three categories for the same JPEG encoding application. It can be seen that by far the largest category is an address difference in the range of -4 to 3. Data traces (Figure 8.3b) also have many small address differences though on the whole large jumps are performed much more frequently. Encoding address differences on special data types is therefore not as effective for data traces.

The first two bits at the beginning of a data token determine what kind of data type the consecutive bytes contain or whether a command is encoded on this byte. If the difference




**(a)** Instruction trace. **(b)** Data trace.

**Figure 8.3:** Distribution of consecutive instruction and data address differences within the given ranges.

| Bit 7 | Bit 6 | Significance |
|---|---|---|
| 0 | 0 | Command |
| 0 | 1 | 6-bit type |
| 1 | 0 | Two 3-bit types |
| 1 | 1 | 14-bit type |

**Table 8.2:** Data type is determined based on the bits 6 and 7 from the next compressed trace byte.

between two addresses is beyond the largest range of -8191 to 8192 then a command is used to indicate that the next four bytes give the absolute address of the next memory access. This means that an absolute memory address is encoded on 5 bytes, which is larger than the 4 bytes that would be required if the entire trace were made up of absolute addresses. The fair assumption is made that the space savings obtained by using smaller data types outweighs the extra bytes required to represent absolute memory addresses. The byte content for the three data types and an absolute memory address command are depicted in Figure 8.4. The first token in a compressed trace always contains an absolute address command to set the starting point of the memory accesses.

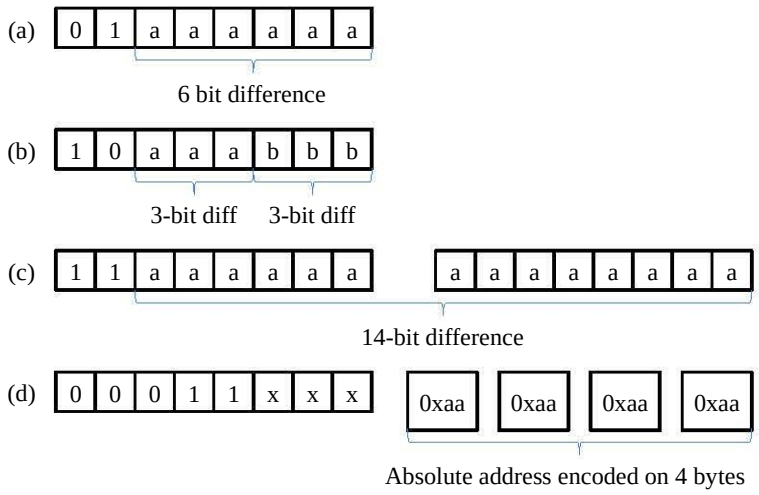



**Figure 8.4:** Example bit encodings for the three supported data types in (a), (b) and (c). (d) shows how an absolute address is loaded for when the address difference is greater than 8191 or less than -8192.


**Figure 8.5:** Bit encoding for a repeat command.

### 8.2.2 Repeat Differences

Very frequently the difference between consecutive memory accesses will be repeated a certain number of times. In the instruction address trace sample from Figure 8.6 it can be seen that a difference of $+2$ is repeated four times and then six times in a row. The compression algorithm detects such repetitions and instead of writing out every single address memory difference sends a command to indicate that the previous difference is to be repeated $r$ times. $r$ is encoded on 5 bits, as shown in Figure 8.5, representing between 1 and 32 repetitions of the previous difference.

### 8.2.3 Address Floats

A number of reasons can cause large jumps in the accessed address. For instruction traces, function calls will often take the program to a location in instruction memory that is far away from the calling code. After the function is executed, the program returns to where

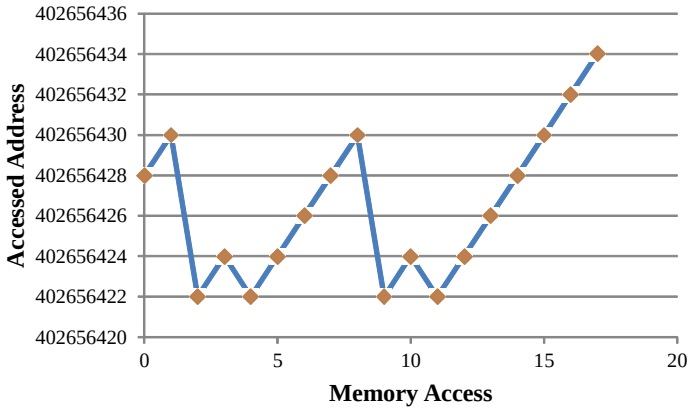



**Figure 8.6:** Instruction trace sample with repetitions.

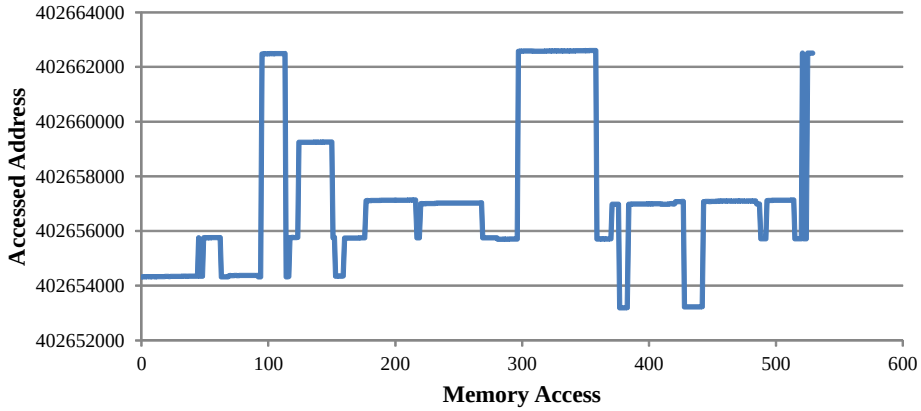



**Figure 8.7:** Instruction trace sample showing jumps between certain address localities due to function calls.

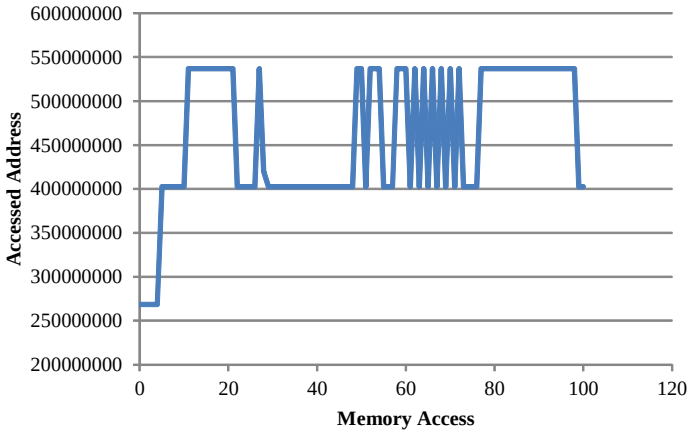



**Figure 8.8:** Data trace sample showing jumps between certain address localities due to stack operations.

| Command | Address | Float 0 | Float 1 |
|---|---|---|---|
| Set float 0 to 28 | 28 | **28** | - |
| +2 | 30 | **30** | - |
| +2 | 32 | **32** | - |
| Set float 1 to 9000 | 9000 | 32 | **9000** |
| +2 | 9002 | 32 | **9002** |
| +4 | 9006 | 32 | **9006** |
| Switch to float 0 | - | **32** | 9006 |
| +2 | 34 | **34** | 9006 |
| +2 | 36 | **36** | 9006 |
| Switch to float 1 | - | 36 | **9006** |
| +4 | 9010 | 36 | **9010** |

**Table 8.3:** Sequence of commands sent and the corresponding addresses produced. The states of two floats are given with the selected float in bold.

it was located before the function was called. Depending on the application, functions may be called a number of times in a row, i.e., large jumps are frequently performed to the same set of locations (Figure 8.7). This observation is even more so valid for data traces where frequent accesses to the stack memory cause large jumps in memory address that keep going back to the same locations.

Given the compression techniques that have been presented so far, each large jump would require a 5 byte absolute memory address. To further reduce the size of the trace, *address floats* are proposed which can keep track of up to eight different localities in memory. Every time an absolute address command is issued it is provided with a float index pointing to the float that is currently used. The given absolute address represents the value of the current float. Subsequent differences between addresses update the location of that float until either another absolute address command or a switch float command are issued. A possible command sequence using floats is given in Table 8.3. Commands in the *Command* column are issued creating the address given in the *address* column. The states of float 0 and float 1 are also given in the right hand columns.

To implement floats the absolute address command was changed to include the float index encoded on three bits. Similarly, the switch float command also uses three bits to indicate which float to switch to. The command bits can be seen in Figure 8.9.

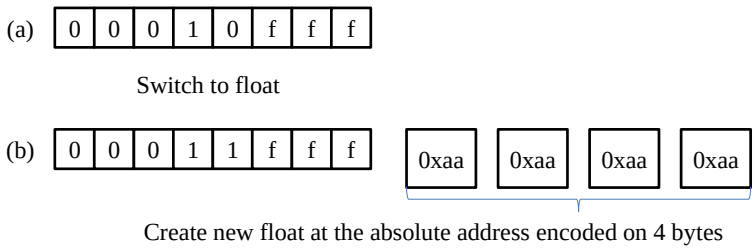



**Figure 8.9:** A switch float (a) and a new float (b) command. The index of the float is encoded on the three 'f' bits.

| | Trace density (bytes/token) | |
|---|---|---|
| Application | Data | Instruction |
| Autocorrelate | 1.33 | 0.38 |
| Convolution | 0.90 | 0.32 |
| FIR | 1.27 | 0.33 |
| Idct | 1.18 | 0.26 |
| JPEG simple encoder | 1.56 | 0.50 |
| Mandelbrot | 1.07 | 0.41 |
| Matrix | 1.48 | 0.20 |
| MP3 decoder | 1.44 | 0.68 |
| MP3 encoder | 1.74 | 0.61 |
| MPEG2 encoder | 1.41 | 0.42 |
| Quantisation | 1.26 | 0.40 |

**Table 8.4:** Trace compression density for data and instruction traces for a number of benchmarks.

## 8.3 Performance

The compression density achieved with the proposed algorithm for a number of benchmarks can be seen in Table 8.4. The large difference between data and instruction trace densities can be explained by a number of factors:

1. Data traces perform large jumps much more frequently, an assertion which is backed by the graphs in Figure 8.3, and every float switch requires an additional byte.

2. Tokens from a data trace require an additional bit to determine whether the memory access is a read or a write. This bit was appended to the end of the address, making it vary ever so slightly more than an address without read/write information.

3. Data traces are not as regular as instruction traces, and thus repeat difference commands are rarely encoded, whereas repeats are frequently found in instruction traces.

The compression and decompression speed of the algorithm was measured for the benchmarks in Table 8.4 on the same computer as was used in Section 7.5, i.e., a quad core Intel Core 2 processor at 2.4 GHz with 3 GB of memory, reading the trace from a 256 GB SSD. The different timings can be seen in Table 8.5. The *Standard .din Parse* column shows how long it takes to read and parse the .din trace file. For the proposed algorithm, it can be seen that encoding and storing the trace takes a little longer than parsing the trace. This observation is exploited in Chapters 6 and 7 where the trace needs to be read a number of consecutive times: in the first simulation iteration the raw .din file is read and a compressed version of the trace is stored with little overhead; any subsequent simulation runs read the compressed trace directly which is significantly quicker, as can be seen in the *Decode time* column for the proposed compressor.

For the sake of comparison, gzip compression was also performed on those same traces. While the size of the compressed traces produced is much smaller than for the proposed algorithm, trace encode and especially decode operations take a lot longer for gzip. Having long trace decode times defeats the purpose of employing trace compression to speed up cache simulation. The proposed algorithm is therefore much better optimised to speed up cache simulation. In terms of hardware implementation, the trace decoder requires 1338 LUTs and 392 registers when implemented on an Altera Stratix IV FPGA. The circuit was successfully tested at 150 MHz but is capable of running at higher frequencies, which is generally limited by the circuits on either side of the hardware decoder.

For cache simulation requiring six simulation runs, the employment of the proposed trace compression algorithm can reduce the trace access time on average by a factor 4.27x for instruction traces and 3.81x for data traces. Additionally, trace compression can reduce the required bandwidth for trace transfer as shown in Chapter 6 where memory access information needs to be sent from a computer to an FPGA through a PCIe port.

| | | | Proposed compression algorithm | | | gzip | | |
|---|---|---|---|---|---|---|---|---|
| Application | Trace Type | Standard .din Parse | Encode time | Decode time | Size Ratio | Encode time | Decode time | Size ratio |
| Autocorrelate | Inst. | 27.61 s | 32.35 s | 1.46 s | 28.92x | 42.88 s | 24.79 s | 223.73x |
| | Data | 34.81 s | 52.71 s | 0.31 s | 12.95x | 91.97 s | 55.05 s | 25.72x |
| Convolution | Inst. | 4.81 s | 5.46 s | 0.43 s | 34.91x | 7.98 s | 3.84 s | 148.42x |
| | Data | 5.08 s | 6.26 s | 0.16 s | 12.26x | 8.46 s | 3.84 s | 89.13x |
| FIR | Inst. | 3.87 s | 4.45 s | 0.14 s | 33.14x | 6.03 s | 3.58 s | 269.43x |
| | Data | 7.24 s | 9.78 s | 0.15 s | 8.67x | 12.67 s | 5.65 s | 37.68x |
| Idct | Inst. | 1.06 s | 1.21 s | 0.13 s | 41.87x | 1.77 s | 0.86 s | 167.88x |
| | Data | 2.15 s | 2.82 s | 0.15 s | 9.30x | 5.04 s | 1.54 s | 33.25x |
| JPEG simple encoder | Inst. | 11.57 s | 13.52 s | 0.53 s | 21.86x | 21.11 s | 9.69 s | 71.38x |
| | Data | 6.88 s | 9.81 s | 0.44 s | 7.07x | 21.63 s | 5.78 s | 20.44x |
| Mandelbrot | Inst. | 35.49 s | 40.66 s | 0.24 s | 38.14x | 72.75 s | 33.08 s | 113.04x |
| | Data | 20.31 s | 26.29 s | 0.47 s | 10.24x | 27.89 s | 13.31 s | 123.58x |
| Matrix | Inst. | 30.25 s | 34.17 s | 0.24 s | 149.66x | 123.29 s | 72.01 s | 229.52x |
| | Data | 33.77 s | 50.63 s | 0.28 s | 28.24x | 203.62 s | 122.36 s | 35.65x |
| MP3 decoder | Inst. | 31.95 s | 39.22 s | 0.24 s | 30.61x | 209.56 s | 73.69 s | 19.11x |
| | Data | 25.12 s | 34.09 s | 0.63 s | 7.66x | 72.95 s | 25.69 s | 19.67x |
| MP3 encoder | Inst. | 32.92 s | 40.69 s | 0.26 s | 394.63x | 1575.08 s | 729.70 s | 45.72x |
| | Data | 35.48 s | 55.16 s | 0.25 s | 31.87x | 423.23 s | 180.87 s | 22.27x |
| MPEG2 encoder | Inst. | 33.25 s | 40.02 s | 0.25 s | 648.22x | 1524.54 s | 694.85 s | 57.83x |
| | Data | 35.95 s | 52.34 s | 0.26 s | 26.65x | 351.46 s | 131.92 s | 17.44x |
| Quantisation | Inst. | 0.79 s | 0.77 s | 0.25 s | 27.47x | 1.03 s | 0.61 s | 233.05x |
| | Data | 0.91 s | 1.30 s | 0.16 s | 8.73x | 1.88 s | 0.69 s | 36.53x |

**Table 8.5:** Algorithm performance for a number of benchmarks. The encode times and decode times of the proposed algorithm and gzip algorithm are given together with the time it took to parse the file conventionally (Standard .din Parse).

## 8.4 Future Work

There is one memory access behaviour which could be exploited, but has not been at this point: the repeated occurrence of patterns in the trace. The loop from Figure 8.1 shows part of a loop which repeats the differences +2, +2, +2 then -6. In the actual trace these differences are repeated 7 times. Instead of encoding this sequence 7 times it would be possible to implement a *start pattern* and an *end pattern* command. The end pattern command could then be followed by a repeat command which indicates how many times the last pattern should be repeated.

To implement such pattern repeat functionality the encoder would need to be made more complex by adding pattern detection capability. The trace decoder implementation would be more straightforward than that, as it would just have to keep a local copy of the pattern to be repeated, and repeat it the given number of times.

## 8.5 Summary

A trace compression algorithm has been presented with a focus on fast trace encoding and decoding, with the capability of decoding the trace in hardware. A number of trace observations were exploited to create this optimised algorithm. Small differences between consecutive addresses are exploited by performing offset encoding using special data types. Repeated differences are compactly represented with a repeat command, and frequent large jumps between localities are improved upon with floats. When performing cache simulation, the amount of time required to read the trace can be sped up by a factor of approximately 4x with the aid of trace compression, if the trace needs to be read 6 consecutive times.

# Chapter 9

# Conclusions

Driven by the constant need for faster and more complex digital devices, designers have to be certain that temperature, energy and performance criteria are met in today's integrated circuit designs. These IC requirements increasingly apply to FPGA-based circuits where the drastic rise in power densities has led to observable thermal problems [29]. Digital circuits have to be optimised at design time using ever more sophisticated tools such as system and thermal simulation, and fast cache evaluation. Despite these efforts, design time optimisations are often insufficient requiring the implementation of run-time methods that adapt the way the device operates in the field depending on up-to-date conditions and requirements.

At first the dynamic application adaptation method was considered through the creation of a specialised JPEG encoding application. The encoder is characterised by unprecedented adaptability at the DCT level while requiring very few resources. This makes it ideal for use in embedded systems. The application was implemented on an FPGA with the goal of lowering the device temperature if the chip became too hot. Following two successive experiments, one on an older 90nm and one on a newer 40nm device, it was found that application adaptation, by itself, is incapable of lowering FPGA temperature unless it is combined with other dynamic thermal management methods such as frequency scaling or clock gating.

Deeper thermal analysis of an embedded system implemented on an FPGA revealed

that the component principally responsible for elevated temperatures is the external memory interface. This large, complex IP block operating at high frequencies is active mostly when the external memory is being accessed. It logically follows that fewer external memory accesses reduce the device temperature. For an embedded system containing a CPU cache this means that fewer cache misses results in fewer external memory accesses and lower memory interface temperature. Based on temperature measurements and miss rates of different caches for a JPEG encoding application, a model was proposed by Thomas Ebi linking cache miss rate $r$ with temperature $T$:

$$T = \alpha \cdot e^{-1/\beta r} + T_{\text{MEM}} + T_{\text{BASE}} \tag{9.1}$$

where $\alpha$, $\beta$, $T_{\text{MEM}}$ and $T_{\text{BASE}}$ need to be determined experimentally. If the miss rate and device thermal characteristics are known the device temperature can be estimated.

Single-pass cache simulation is the tool of choice to accurately evaluate the miss rates of many different cache configurations. Inconveniently, the execution time of precise cache simulation is very slow for an exhaustive analysis of the cache design space. In order to improve a designer's ability to analyse caches and consequently temperature at design time, methods were researched to improve the throughput of single-pass cache simulators.

Initially the simulation of LRU caches was considered with MASH{lru}, a Multiple cAche Simulator in Hardware implemented on an FPGA. The device's resource usage was minimised by making use of two known LRU cache inclusion properties. In rough terms the inclusion properties state that the contents of a cache are a subset of the contents of a larger cache. It follows that only the largest cache needs to be simulated based on which the state of all the smaller caches can be deduced. This design approach requires considerably fewer resources than the creation of a separate cache instance for every distinct cache simulated. The resulting simulator instance can concurrently simulate 44 cache configurations at a trace consumption rate of 100 MHz when instantiated on a medium-sized Altera Stratix IV FPGA. Experiments concluded that MASH{lru} is up to 53x faster than

the fastest software alternatives.

Simulators of caches obeying the FIFO replacement policy are addressed with the proposed MASH{fifo} simulator which is also a hardware-based single-pass cache simulator. To optimise the resource usage of MASH{fifo} two FIFO cache inclusion properties are presented for the first time which can be summarised as follows: the contents of FIFO caches of set sizes *ss* and smaller and associativities *assoc* and smaller will always be a subset of the data held in an LRU cache of set size *ss* and associativity $2 * assoc - 1$. The resulting hardware instance can be clocked at 75 MHz while simulating 30 cache configurations at the same time. Compared to software solutions this makes MASH{fifo} up to 11.10x faster for static trace cache simulation that is read from a disk drive.

A novel 'in-system' cache simulation methodology is explored which is made possible through the hardware-only nature of the MASH simulators. By instantiating a cache simulator instance alongside an embedded system on an FPGA, the memory trace can be obtained by directly observing accesses on the memory bus. The benefits of in-system cache simulation are fast trace generation based on a real-world embedded system combined with concurrent and real time cache simulation. As a consequence the trace is based on an application flow governed by real world inputs and the trace is generated at very high speed. An in-system approach is ideal when designing an embedded system that runs a complex application that is subject to a large range of inputs.

A PLRUt simulator is proposed with MASS{plrut} which is a software based cache simulator. By defining union properties of PLRUt caches the maximum memory requirements of a PLRUt simulator can be determined. Armed with this knowledge an optimised hash table-based simulator was designed that is up to $1.93\times$ faster than an unoptimised solution. The effect of trace compression is also taken into account and a simple trace compression algorithm is presented.

The findings and inventions presented in this dissertation can help embedded designers meet their temperature, energy and performance requirements in a number of ways. The adaptive JPEG encoder can be employed when image compression is required at the highest possible quality level under fluctuating system conditions. LRU, FIFO and

PLRUt caches can now be simulated at unprecedented speeds to determine the cache configurations yielding the best energy consumption and performance. The output of these simulators can also be combined with the thermal cache model from Equation 9.1, allowing for the selection of a cache configuration with suitable thermal characteristics in an embedded system implemented on an FPGA. Finally, the ability to perform in-system cache simulation enables designers to obtain measurements from within a real embedded system, providing fast and realistic cache performance insights like never before.

# Appendix A

# MASH{lru} Implementation

This appendix provides the main code of the different components that the MASH{lru} simulator is made from.

## A.1 MASH configuration

The global.vhd file contains a number of constant and function definitions that are used by the different MASH implementations. It also holds a number of settings that the designer can modify to tailor the simulator to their needs. This includes:

- set_count_mux, the set size of the biggest cache simulated as a power of two.
- min_set_count_mux, the smallest set size simulated, also given as a power of two.
- associativity_mux, the maximum associativity as a power of two.
- address_width, maximum tag size, in bits.
- assoc_inc_width, indicates whether consecutive simulated associativities are incremental or in powers of two.
- counter_width, bit width of the hit counters.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  package global is
6
7  ----> SETTINGS USED TO CONFIGURE THE CACHE SIMULATOR
8    constant set_count_mux       : integer := 10;     -- Determines
       how many sets are at the top level
9    constant min_set_count_mux   : integer := 0;      -- Smallest
       number of sets we want to simulate
```

```
10    constant associativity_mux   : integer := 2;      -- Defines the
          greatest level of associativity simulated
11    constant address_width       : integer := 22;     -- To save on
          tag bits and comparators, make the address width as small
          as possible
12    constant assoc_inc_width     : integer := 2 **
          associativity_mux; -- If the associativity levels
          simulated are incremental (1, 2, 3 etc.) define '2 **
          associativity_mux'.
13                                                      -- For in
                                                            powers of
                                                            two (1, 2,
                                                            4, 8 etc.)
                                                            define '
                                                            associativity_mux
                                                             + 1'
14    constant counter_width       : integer := 40;     -- Bit width
          of the counters to be used
15
16 ----> DO NOT TOUCH <----
17    constant associativity       : integer := 2 **
          associativity_mux;
18    constant tag_length          : integer := address_width -
          set_count_mux - 2;
19    constant counters_count      : integer := 2 * (set_count_mux -
          min_set_count_mux + 1); -- Defines the number of counters
          that will be required. 2x because one is needed for read,
          and one for write.
20    constant bottom_addr_bits    : integer := address_width - 1 -
          tag_length; -- Mainly used in the unpack_addr() function.
          Defines the width of the lower address bits, once the tag
          has been removed
21
22    -- Hit collection bundles together hit results so they can
          easily be passed to lower level sub cache sets.
23    type hit_collection is array (natural range <>) of
          std_logic_vector(associativity - 1 downto 0);
24
25    -- Hit increment collection indicates which counts should be
          incremented.
26    type hit_incr_collection is array (natural range <>) of
          std_logic_vector(assoc_inc_width - 1 downto 0);
27
28    -- Integer values used to store the hit counts. Incremented by
           the hit_counter module.
```

```
29    type hit_counters is array(assoc_inc_width - 1 downto 0) of
          unsigned(counter_width - 1 downto 0);
30
31    -- A blob of hit count values.
32    type hit_counters_blob is array (natural range <>) of
          hit_counters;
33
34    -- A record to store the different parts a requested address
          is made up of.
35    type addr_sections is
36      record
37        tag      : std_logic_vector(tag_length - 1 downto 0);
38        set_idx  : std_logic_vector(set_count_mux - 1 downto 0);
39        new_addr : std_logic_vector(address_width - 1 downto 0);
40      end record;
41
42    function format_hit_list( arg : std_logic_vector(associativity
          - 1 downto 0) ) return std_logic_vector;
43    function unpack_addr( addr : std_logic_vector(address_width -
          1 downto 0); line_len : integer range 0 to 6 ) return
          addr_sections;
44  end;
45
46  package body global is
47
48  ----------------------------------------------------------
49    -- This function basically multiplexes the input. It takes the
           hit list as an input
50    -- and from there indicates at which level of associativity a
          hit has occurred.
51    -- The return value can directly be fed into a hit counter.
52    function format_hit_list( arg : std_logic_vector(associativity
          - 1 downto 0) ) return std_logic_vector is
53      variable result     : std_logic_vector(assoc_inc_width - 1
            downto 0);
54      variable hit_rearr  : std_logic_vector(associativity - 1
            downto 0);
55      variable hit_occur  : std_logic;
56    begin
57      if assoc_inc_width = associativity then
58        -- Associativity sizes simulated are incremental. If a hit
               is detected, set all hits at greater levels of
              associativity.
59        --
60        -- Before:    |0|0|0|1|0|0|0|0|    After:   |1|1|1|1|1|0|0|0|
61        -- Assoc:   |1|2|3|4|5|6|7|8|        |8|7|6|5|4|3|2|1|
```

```
62        --
63        hit_occur := '0';
64
65        for k in assoc_inc_width - 1 downto 0 loop
66          if arg(k) = '1' then
67            hit_occur := '1';
68          end if;
69          result(assoc_inc_width - 1 - k) := hit_occur;
70        end loop;
71      elsif assoc_inc_width = associativity_mux + 1 then
72        -- Associativity sizes simulated are in powers of two
73        --
74        -- First off, reverse the bit order of the hits so this
              whole thing is easier to figure out
75        -- Example, associativity = 8
76        --
77        -- Before:    |0|1|00|0000|    After:   |0000|00|1|0|
78        -- Assoc:   |1|2| 4|    8|       |    8| 4|2|1|
79        --
80        for k in associativity - 1 downto 0 loop
81          hit_rearr(k) := arg(associativity - 1 - k);
82        end loop;
83
84        result := (others => '0');
85        if to_integer(unsigned(hit_rearr)) > 0 then
86          -- Set the MSb if we have any hit at all
87          result(associativity_mux) := '1';
88          for k in associativity_mux - 1 downto 0 loop
89            -- This for loop successively compares the input value
                   to predefined values marking associativity levels
90            -- i.e., for 16 bit input (4 to 0 result), it checks
                   if the input value is smaller than:
91            -- 0b0000000100000000  - result |= 0b01000
92            -- 0b0000000000010000  - result |= 0b00100
93            -- 0b0000000000000100  - result |= 0b00010
94            -- 0b0000000000000010  - result |= 0b00001
95            if (2 ** (2 ** k)) > to_integer(unsigned(hit_rearr(2
                   ** associativity_mux - 1 downto 0))) then
96              result(k) := '1';
97            end if;
98          end loop;
99        end if;
100     else
101       -- If we get here it's because of a bad assoc_inc_width
              setting
```

```
102       assert false report "Bad <assoc_inc_width> setting. Please
              fix constant declaration in global.vhd." severity
              failure;
103     end if;
104
105     return result;
106   end function;
107
108
109 --#########################################################
110   -- unpack_addr takes as an input the currently accessed
        address and chops it up into its different sections:
111   --   - tag, of maximum tag length, with bits cleared depending
         on line_len
112   --   - set_idx, a substring of addr, the location of which
        depends on line_len
113   --   - byte selected, which is actually never used, but there
        for completeness
114   --   - new_addr, the address, but modified with cleared bits
        to take the line length into account
115   function unpack_addr( addr : std_logic_vector(address_width -
        1 downto 0); line_len : integer range 0 to 6 ) return
        addr_sections is
116     variable result : addr_sections;
117     variable addr_0 : std_logic_vector(bottom_addr_bits downto
          0);
118   begin
119     addr_0 := (others => '0');
120     case line_len is
121       when 0 =>
122         result.tag      := addr(address_width - 1 downto
              bottom_addr_bits + 1);
123         result.set_idx  := addr(bottom_addr_bits downto
              bottom_addr_bits - set_count_mux + 1);
124       when 1 =>
125         result.tag      := addr(address_width - 1 downto
              bottom_addr_bits + 2) & '0';
126         result.set_idx  := addr(bottom_addr_bits + 1 downto
              bottom_addr_bits - set_count_mux + 2);
127       when 2 =>
128         result.tag      := addr(address_width - 1 downto
              bottom_addr_bits + 3) & "00";
129         result.set_idx  := addr(bottom_addr_bits + 2 downto
              bottom_addr_bits - set_count_mux + 3);
130       when 3 =>
```

```
131           result.tag       := addr(address_width - 1 downto
                 bottom_addr_bits + 4) & "000";
132           result.set_idx   := addr(bottom_addr_bits + 3 downto
                 bottom_addr_bits - set_count_mux + 4);
133         when 4 =>
134           result.tag       := addr(address_width - 1 downto
                 bottom_addr_bits + 5) & "0000";
135           result.set_idx   := addr(bottom_addr_bits + 4 downto
                 bottom_addr_bits - set_count_mux + 5);
136         when 5 =>
137           result.tag       := addr(address_width - 1 downto
                 bottom_addr_bits + 6) & "00000";
138           result.set_idx   := addr(bottom_addr_bits + 5 downto
                 bottom_addr_bits - set_count_mux + 6);
139         when 6 =>
140           result.tag       := addr(address_width - 1 downto
                 bottom_addr_bits + 7) & "000000";
141           result.set_idx   := addr(bottom_addr_bits + 6 downto
                 bottom_addr_bits - set_count_mux + 7);
142         when others =>
143       end case;
144       result.new_addr := result.tag & addr_0;
145       return result;
146     end function unpack_addr;
147
148 end package body;
```

## A.2 Main LRU Simulator File

This is the top level simulator file, lru_cache_sim.vhd. It instantiates and connects top level and lower level sets together with hit counters.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  library work;
6  use work.global.ALL;
7
8  entity lru_cache_sim is
9    port (
10     clk               : in std_logic;
11     reset_n           : in std_logic;
12     address           : in std_logic_vector(address_width - 1
         downto 0);     -- The address that is being requested
```

```
13      new_addr          : in std_logic;                 -- Like an
            input enable for a new address
14      read_n            : in std_logic;                 -- 0: we are
            reading, 1: we are writing
15      still_computing   : out std_logic;                 -- Set when
            and access is still being computed
16      line_length_mux   : in integer range 0 to 6;          --
            Defines the length of a line for the current run
17      all_hit_counts    : out hit_counters_blob(0 to counters_count
            - 1); -- A blob containing all the hit counts of all
            set sizes and all levels of associativity
18      total_accesses    : out unsigned(counter_width - 1 downto 0);
              -- Contains the total address accesses that have been
            performed
19      read_accesses     : out unsigned(counter_width - 1 downto 0);
              -- Total reads
20      write_accesses    : out unsigned(counter_width - 1 downto 0)
              -- Total write
21    );
22  end lru_cache_sim;
23
24  architecture rtl of lru_cache_sim is
25
26    constant set_count     : integer := 2 ** set_count_mux;
27
28    component lru_set
29      port (
30        clk           : in std_logic;
31        reset_n       : in std_logic;
32        set_select    : in std_logic;
33        address       : in std_logic_vector(address_width - 1
              downto 0);
34        hit_idx       : out std_logic_vector(associativity - 1
              downto 0)
35      );
36    end component;
37
38    component lru_sub_set_collection
39      generic (
40        top_set_count_mux   : integer     -- Let us know how many
              set counts are located at the level above
41      );
42      port (
43        clk           : in std_logic;
44        reset_n       : in std_logic;
```

```
45        set_idx      : in std_logic_vector(top_set_count_mux - 1
              downto 0);
46        set_select   : in std_logic_vector(2 ** top_set_count_mux -
              1 downto 0);
47        hits_above   : in std_logic_vector(associativity - 1 downto
              0);
48        hits_incr    : out hit_incr_collection(set_count_mux -
              min_set_count_mux - 1 downto 0)
49      );
50    end component;
51
52    component hit_counter
53      port (
54        clk          : in std_logic;
55        reset_n      : in std_logic;
56        enable_n     : in std_logic;
57        hit_list     : in std_logic_vector(assoc_inc_width - 1
              downto 0);
58        hit_counts   : out hit_counters
59      );
60    end component;
61
62    -- Determines which set is to be selected for comparison with
        the tag.
63    signal set_select   : std_logic_vector(set_count - 1 downto 0);
64
65    -- The new address to be compared to a tag, and depends on the
        given line length.
66    signal addr_formatted   : std_logic_vector(address_width - 1
        downto 0);
67
68    -- Stores the set index that we are currently accessing
69    signal set_idx      : std_logic_vector(set_count_mux - 1 downto
        0);
70
71    -- Collection of hits in every set.
72    signal hits_in_set    : hit_collection(set_count - 1 downto 0);
73
74    -- Indicates at which associativity level the hit occurred
75    signal assoc_hit   : std_logic_vector(associativity - 1 downto
        0);
76
77    -- Collection that indicates at which levels of associativity
        a hit has occurred.
78    -- This information is contained for every set size simulated.
```

```
79    signal hits_incr   : hit_incr_collection(0 to set_count_mux -
          min_set_count_mux);
80
81    -- Instance that groups all count values together!
82    signal all_counts  : hit_counters_blob(0 to counters_count -
          1);
83
84    -- Counts the total number of accesses
85    signal accesses : unsigned(counter_width - 1 downto 0);
86    signal writes   : unsigned(counter_width - 1 downto 0);
87    signal reads    : unsigned(counter_width - 1 downto 0);
88
89    -- Indicates which levels of the cache simualtor are still
          processing data
90    signal level_processing  : std_logic_vector(0 to set_count_mux
          - min_set_count_mux - 1);
91
92    -- Keeps track of which levels are currently doing a read, and
          which ones are doing a write.
93    signal level_read_n    : std_logic_vector(0 to set_count_mux -
          min_set_count_mux - 1);
94    signal i_level_read_n  : std_logic_vector(0 to set_count_mux -
          min_set_count_mux);
95    signal n_level_read_n  : std_logic_vector(0 to set_count_mux -
          min_set_count_mux);
96
97  begin
98
99    -- Determine which set to access depending on the address
100   process(address, line_length_mux, new_addr)
101     variable v_addr : addr_sections;
102   begin
103     v_addr := unpack_addr(address, line_length_mux);
104     addr_formatted <= v_addr.new_addr;
105     set_idx <= v_addr.set_idx;
106
107     -- Detect which set is currently being accessed. Also make
          sure that new_addr is equal to '1'.
108     -- If it isn't no set is accessed which has the same effect
          as no read or write occurring!
109     for k in set_count - 1 downto 0 loop
110       if v_addr.set_idx = std_logic_vector(to_unsigned(k,
            set_count_mux)) and new_addr = '1' then
111         set_select(k) <= '1';
112       else
113         set_select(k) <= '0';
```

```
114       end if;
115     end loop;
116   end process;
117
118   -- Generates the top level sets
119   GENERATE_SETS:
120   for k in set_count - 1 downto 0 generate
121     lru_set_inst : lru_set
122       port map (
123         clk          => clk,
124         reset_n      => reset_n,
125         set_select   => set_select(k),
126         address      => addr_formatted,
127         hit_idx      => hits_in_set(k)
128       );
129   end generate GENERATE_SETS;
130
131   -- Create sub-set collection.
132   lru_sub_set_col_inst : lru_sub_set_collection
133     generic map (
134       top_set_count_mux  => set_count_mux
135     )
136     port map (
137       clk          => clk,
138       reset_n      => reset_n,
139       set_idx      => set_idx,
140       set_select   => set_select,
141       hits_above   => assoc_hit,
142       hits_incr    => hits_incr(1 to set_count_mux -
            min_set_count_mux)
143     );
144
145   -- This process bundles all the hit values of the highest
        level of the different
146   -- sets so that it can later be determined in which
        associativity block the hit occurred.
147   process(hits_in_set)
148     variable v_assoc_hit  : std_logic;
149   begin
150     -- 'OR' that shit
151     for j in associativity - 1 downto 0 loop
152       v_assoc_hit := '0';
153
154       -- Check if we have any hits in this assoc level
155       for k in set_count - 1 downto 0 loop
156         if hits_in_set(k)(j) = '1' then
```

```
157             v_assoc_hit := '1';
158           end if;
159         end loop;
160
161         -- If there is a hit, set the bit
162         assoc_hit(j) <= v_assoc_hit;
163       end loop;
164     end process;
165
166     -- Turn the raw hit data into an indicator as to which counter
        (s) should be incremented
167     -- Look at global.vhd
168     -- For power of 2 associativities example: associativity = 8,
        assoc_hit = |0|0|01|0000| -> 1100, i.e., increment assoc =
        4 and 8, but not 1 and 2
169     -- For incremental associativities example: associativity = 8,
         assoc_hit = |0|0|0|1|0|0|0|0| -> |1|1|1|1|1|0|0|0|, i.e.,
        increment assoc = 4, 5, 6, 7 and 8, but not 1, 2 and 3
170     --
171     hits_incr(0) <= format_hit_list(assoc_hit);
172
173     -- Connect each hits_incr(k) with a counter!
174     GENERATE_HIT_COUNTERS:
175     for k in 0 to set_count_mux - min_set_count_mux generate
176       counter_inst_read : hit_counter
177         port map (
178           clk          => clk,
179           reset_n      => reset_n,
180           enable_n     => i_level_read_n(k),
181           hit_list     => hits_incr(k),
182           hit_counts   => all_counts(2 * k)
183         );
184       counter_inst_write : hit_counter
185         port map (
186           clk          => clk,
187           reset_n      => reset_n,
188           enable_n     => n_level_read_n(k),
189           hit_list     => hits_incr(k),
190           hit_counts   => all_counts(2 * k + 1)
191         );
192     end generate GENERATE_HIT_COUNTERS;
193
194     all_hit_counts <= all_counts;
195
196     -- Workaround. Modelsim can't handle a shift register where
        one bit is combinational and the rest are registered.
```

```
197   i_level_read_n(0) <= read_n;
198   i_level_read_n(1 to set_count_mux - min_set_count_mux) <=
        level_read_n;
199   n_level_read_n <= not i_level_read_n;
200
201   -- Count the total number of accesses that have been
        registered
202   -- Another function of this process is to pipeline this level
        of the cache simulator
203   -- and to shift the 'level_processing' signal to indicate at
        which levels shit is still happening
204   process(clk, reset_n)
205   begin
206     if reset_n = '0' then
207       accesses <= (others => '0');
208       reads <= (others => '0');
209       writes <= (others => '0');
210       level_processing(0 to set_count_mux - min_set_count_mux -
            1) <= (others => '0');
211     elsif rising_edge(clk) then
212       if new_addr = '1' then
213         if read_n = '0' then
214           reads <= reads + 1;
215         else
216           writes <= writes + 1;
217         end if;
218         accesses <= accesses + 1;
219       end if;
220
221       -- Shift all of the level_processing bits to update where
            data is being computed
222       for k in set_count_mux - min_set_count_mux - 1 downto 1
            loop
223         level_processing(k) <= level_processing(k - 1);
224         level_read_n(k) <= level_read_n(k - 1);
225       end loop;
226
227       level_processing(0) <= new_addr;
228       level_read_n(0) <= read_n;
229     end if;
230   end process;
231
232   process(level_processing, new_addr)
233   begin
234     -- Or all of the signals of level_processing to let the
          higher level know that
```

```
235     -- the cache simulator is still computing values
236     still_computing <= new_addr;
237     for k in 0 to set_count_mux - min_set_count_mux - 1 loop
238       if level_processing(k) = '1' then
239         still_computing <= '1';
240       end if;
241     end loop;
242   end process;
243
244   total_accesses <= accesses;
245   read_accesses <= reads;
246   write_accesses <= writes;
247
248 end rtl;
```

## A.3 LRU Set Definition

LRU set definition from lru_set.vhd. For MASH{lru} the LRU tags are stored in shift registers.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  library work;
7  use work.global.ALL;
8
9  entity lru_set is
10   port (
11     clk           : in std_logic;
12     reset_n       : in std_logic;
13     set_select    : in std_logic;
14     address       : in std_logic_vector(address_width - 1 downto
           0);
15     hit_idx       : out std_logic_vector(associativity - 1 downto
           0)
16   );
17 end lru_set;
18
19 architecture rtl of lru_set is
20   type set_entry is
21     record
22       valid    : std_logic;
23       tag      : std_logic_vector(tag_length - 1 downto 0);
24     end record;
```

```
25    type set_array is array(associativity - 1 downto 0) of
          set_entry;
26    signal set_inst     : set_array;
27
28    signal l_hit_idx    : std_logic_vector(associativity - 1 downto
          0);
29    signal l_hit        : std_logic;
30    signal newest_tag   : std_logic_vector(tag_length - 1 downto 0)
          ;
31    signal comp_tag     : std_logic_vector(tag_length - 1 downto 0)
          ;
32
33  begin
34
35    -- Extract the tag value
36    comp_tag <= address(address_width - 1 downto bottom_addr_bits
          + 1);
37
38    -- Determine whether we have a hit or not, and set the hit
          index
39    process(set_select, address, set_inst, comp_tag)
40      variable v_hit          : std_logic;
41      variable v_newest_tag   : std_logic_vector(tag_length - 1
            downto 0);
42    begin
43      v_newest_tag := (others => '0');
44      if set_select = '1' then
45        v_hit := '0';
46        for k in associativity - 1 downto 0 loop
47          if set_inst(k).valid = '1' and set_inst(k).tag =
                comp_tag then
48            l_hit_idx(k) <= '1';
49            v_hit := '1';
50            v_newest_tag := set_inst(k).tag;
51          else
52            l_hit_idx(k) <= '0';
53          end if;
54        end loop;
55        l_hit <= v_hit;
56
57        if v_hit = '0' then
58          v_newest_tag := comp_tag;
59        end if;
60      else
61        l_hit_idx <= (others => '0');
62        l_hit <= '0';
```

```
63      end if;
64      newest_tag <= v_newest_tag;
65    end process;
66
67    -- Output the hit bit and index
68    hit_idx <= l_hit_idx;
69
70    -- When the clock ticks, update the order in which the tags
        are stored
71    process(clk, reset_n, set_select)
72      variable v_shift  : std_logic;
73    begin
74      if reset_n = '0' then
75        for k in associativity - 1 downto 0 loop
76          set_inst(k).valid <= '0';
77        end loop;
78      elsif rising_edge(clk) and set_select = '1' then
79        if l_hit = '1' then
80          v_shift := '0';
81        else
82          v_shift := '1';
83        end if;
84
85        for k in 1 to associativity - 1 loop
86          if l_hit_idx(k - 1) = '1' then
87            v_shift := '1';
88          end if;
89
90          if v_shift = '1' then
91            set_inst(k - 1) <= set_inst(k);
92          end if;
93        end loop;
94
95        set_inst(associativity - 1) <= (tag => newest_tag, valid
            => '1');
96      end if;
97    end process;
98
99  end rtl;
```

## A.4 Lower Level Definition

Instantiates subsets and connects them with signals from the level above and below. Defined in lru_sub_set_collection.vhd.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  use work.global.ALL;
7
8  entity lru_sub_set_collection is
9    generic (
10     top_set_count_mux  : integer    -- Let us know how many set
           counts are located at the level above
11   );
12   port (
13     clk          : in std_logic;
14     reset_n      : in std_logic;
15     set_idx      : in std_logic_vector(top_set_count_mux - 1
           downto 0);
16     set_select   : in std_logic_vector(2 ** top_set_count_mux - 1
           downto 0);
17     hits_above   : in std_logic_vector(associativity - 1 downto
           0);
18     hits_incr    : out hit_incr_collection(0 to top_set_count_mux
           - min_set_count_mux - 1)
19   );
20 end lru_sub_set_collection;
21
22 architecture rtl of lru_sub_set_collection is
23
24   constant set_count    : integer := 2 ** (top_set_count_mux -
       1);
25
26   component lru_cache_sub_set is
27     port (
28       clk          : in std_logic;
29       reset_n      : in std_logic;
30       set_select   : in std_logic;
31       addr         : in std_logic;
32       in_hit       : in std_logic_vector(associativity - 1 downto
             0);
33       out_hit      : out std_logic_vector(associativity - 1
           downto 0)
34     );
35   end component;
36
37   component lru_sub_set_collection is
38     generic (
```

```
39        top_set_count_mux   : integer     -- Let us know how many
              set counts are located at the level above
40      );
41      port (
42        clk           : in std_logic;
43        reset_n       : in std_logic;
44        set_idx       : in std_logic_vector(top_set_count_mux - 1
              downto 0);
45        set_select    : in std_logic_vector(2 ** top_set_count_mux -
              1 downto 0);
46        hits_above    : in std_logic_vector(associativity - 1 downto
              0);
47        hits_incr     : out hit_incr_collection(0 to
              top_set_count_mux - min_set_count_mux - 1)
48      );
49    end component;
50
51    -- Selects a set by or-ing odd and even set_select bits coming
          from above
52    signal local_set_select     : std_logic_vector(set_count - 1
          downto 0);
53
54    -- A single bit extracted from set_select in order to
          determine if we are accessing the even, '0', or odd, '1'
          upper set
55    signal local_addr       : std_logic;
56
57    -- Contains the hits of this subset collection
58    signal l_out_hits       : hit_collection(set_count - 1 downto
          0);
59
60    -- or-s l_out_hits to contain the hit result at the
          associativity level
61    signal l_assoc_hits       : std_logic_vector(associativity - 1
          downto 0);
62
63    signal reg_hits_above     : std_logic_vector(associativity - 1
          downto 0);
64    signal reg_set_idx        : std_logic_vector(top_set_count_mux
          - 1 downto 0);
65    signal reg_set_select     : std_logic_vector(set_count - 1
          downto 0);
66
67  begin
68
69    -- Use the highest set_select bit to determine the address bit
```

```
70  local_addr <= reg_set_idx(top_set_count_mux - 1);
71
72  -- Creates the sub-set select signal from the top set_select
       signal by or-ing every pair
73  -- of potential storage set select signals.
74  process(set_select)
75  begin
76    for k in set_count - 1 downto 0 loop
77      local_set_select(k) <= set_select(k) or set_select(k +
          set_count);
78    end loop;
79  end process;
80
81  -- Generates the compact sub-set modules that keep track of
       where their data is stored
82  -- in the higher set simulators
83  GENERATE_SUB_SETS:
84  for k in set_count - 1 downto 0 generate
85    lru_cache_sub_set_inst : lru_cache_sub_set
86      port map (
87        clk          => clk,
88        reset_n      => reset_n,
89        set_select   => reg_set_select(k),
90        addr         => local_addr,
91        in_hit       => reg_hits_above,
92        out_hit      => l_out_hits(k)
93      );
94  end generate GENERATE_SUB_SETS;
95
96  -- The sub_set_collection is a recursive module that creates
       instances of itself
97  -- to simulate smaller set-counts. Check if we have reached
       the smallest set count
98  -- that we want to simulate, and if we haven't, instantiate a
       new sub_set_collection
99  GENERATE_LOWER_LEVEL:
100 if min_set_count_mux < top_set_count_mux - 1 generate
101 lru_sub_sub_col_inst : lru_sub_set_collection
102   generic map (
103     top_set_count_mux  => top_set_count_mux - 1
104   )
105   port map (
106     clk          => clk,
107     reset_n      => reset_n,
108     set_idx      => reg_set_idx(top_set_count_mux - 2 downto
          0),
```

```
109         set_select   => reg_set_select,
110         hits_above   => l_assoc_hits,
111         hits_incr    => hits_incr(1 to top_set_count_mux -
                min_set_count_mux - 1)
112       );
113     end generate;
114
115     -- Bundle the hits for different levels of associativity
116     process(l_out_hits)
117       variable v_assoc_hits  : std_logic;
118     begin
119       -- Loop through the different associativity levels
120       for j in associativity - 1 downto 0 loop
121         v_assoc_hits := '0';
122
123         -- Check if we have any hits in this assoc level
124         for k in set_count - 1 downto 0 loop
125           if l_out_hits(k)(j) = '1' then
126             v_assoc_hits := '1';
127           end if;
128         end loop;
129
130         -- If there is a hit, set the bit
131         l_assoc_hits(j) <= v_assoc_hits;
132       end loop;
133     end process;
134
135     hits_incr(0) <= format_hit_list(l_assoc_hits);
136
137     -- Pipeline it!
138     process(clk, reset_n)
139     begin
140       if reset_n = '0' then
141         reg_hits_above <= (others => '0');
142         reg_set_idx <= (others => '0');
143         reg_set_select <= (others => '0');
144       elsif rising_edge(clk) then
145         reg_hits_above <= hits_above;
146         reg_set_idx <= set_idx(top_set_count_mux - 1 downto 0);
147         reg_set_select <= local_set_select;
148       end if;
149     end process;
150   end rtl;
```

## A.5 LRU Subset

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use work.global.ALL;
5
6  entity lru_cache_sub_set is
7    port (
8      clk         : in std_logic;
9      reset_n     : in std_logic;
10     set_select  : in std_logic;
11     addr        : in std_logic;
12     in_hit   : in std_logic_vector(associativity - 1 downto 0);
13     out_hit : out std_logic_vector(associativity - 1 downto 0)
14   );
15 end lru_cache_sub_set;
16
17 architecture rtl of lru_cache_sub_set is
18   type set_entry is
19     record
20       valid   : std_logic;
21       source  : std_logic;
22     end record;
23   type set_array is array(associativity - 1 downto 0) of
       set_entry;
24
25   type store_buffer is array(1 downto 0) of std_logic_vector(
       associativity - 1 downto 0);
26
27   -- Three important signals used to track hits
28   signal valid_buffer    : store_buffer;
29   signal buffer_order    : set_array;
30
31   -- Unimportant signals to store states
32   signal even_hit_result : std_logic_vector(associativity - 1
       downto 0);
33   signal odd_hit_result  : std_logic_vector(associativity - 1
       downto 0);
34   signal l_hit_or        : std_logic_vector(associativity - 1
       downto 0);
35   signal hit             : std_logic;
36   signal hit_idx         : integer range 0 to associativity - 1;
37   signal addr_idx        : integer range 0 to 1;
38   signal not_addr_idx    : integer range 0 to 1;
39   signal shift_until     : integer range 0 to associativity - 1;
```

```
40
41 begin
42   -- Detect the hits we have with half the sets - easy as bro!
43   even_hit_result <= in_hit and valid_buffer(0) when addr = '0'
       else (others => '0');
44   odd_hit_result <= in_hit and valid_buffer(1) when addr = '1'
       else (others => '0');
45   l_hit_or <= even_hit_result or odd_hit_result;
46   hit <= '0' when l_hit_or = (l_hit_or'range => '0') else '1';
47
48   -- Multiplex the hit array to determine the index of the
       element in which the hit occurred.
49   process(l_hit_or)
50     variable v_hit_idx  : integer range 0 to associativity - 1;
51   begin
52     v_hit_idx := 0;
53     for k in associativity - 1 downto 0 loop
54       if l_hit_or(k) = '1' then
55         v_hit_idx := k;
56       end if;
57     end loop;
58     hit_idx <= v_hit_idx;
59   end process;
60
61   addr_idx <= 1 when addr = '1' else 0;
62   not_addr_idx <= 0 when addr = '1' else 1;
63
64   -- This is the tricky part - keeping track of which blocks are
       valid, and where and the order in which they are stored
65   process(clk, reset_n)
66     variable v_nacc_valid  : std_logic_vector(associativity - 1
         downto 0);      -- Valid bits for the address that is NOT
          currently accessed
67     variable v_acc_valid  : std_logic_vector(associativity - 1
         downto 0);      -- Valid bits for the address that is
         currently accessed
68   begin
69     if reset_n = '0' then
70       valid_buffer(0) <= (others => '0');
71       valid_buffer(1) <= (others => '0');
72       for k in associativity - 1 downto 0 loop
73         buffer_order(k).valid <= '0';
74         buffer_order(k).source <= '0';
75       end loop;
76     elsif rising_edge(clk) then
77       if set_select = '1' then
```

```
78          -- Two different situations can occur:
79          if hit = '0' then
80          -- *No hit = shift buffer order. If it overflows from
                the other accessed valid buffer, delete the last '1'
                from it.
81            -- Figure out what to do with any overflow
82            if buffer_order(0).valid = '1' and not(buffer_order(0)
                .source = addr) then
83              v_nacc_valid := valid_buffer(not_addr_idx);
84              for k in 0 to associativity - 1 loop
85                if v_nacc_valid(k) = '1' then
86                  -- We have found the last '1' in the other sub-
                      buffer. Clear it.
87                  v_nacc_valid(k) := '0';
88                  exit;
89                end if;
90              end loop;
91              valid_buffer(not_addr_idx) <= v_nacc_valid;
92            end if;
93
94            -- Only shift the selected valid_buffer if the data
                shifted onto it is not at the end already
95            if not(buffer_order(0).valid = '1' and buffer_order(0)
                .source = addr) then
96              v_acc_valid := valid_buffer(addr_idx);
97              for k in 0 to associativity - 2 loop
98                v_acc_valid(k) := v_acc_valid(k + 1);
99              end loop;
100             v_acc_valid(associativity - 1) := '1'; -- mark the
                  first item as taken
101             valid_buffer(addr_idx) <= v_acc_valid;
102           end if;
103         end if;
104
105         -- Now shift the order buffer
106         for k in 0 to associativity - 2 loop
107           if k >= shift_until then
108             buffer_order(k) <= buffer_order(k + 1);
109           end if;
110         end loop;
111         buffer_order(associativity - 1).valid <= '1'; -- Make
              the first entry valid
112         buffer_order(associativity - 1).source <= addr; --
              Store the 'address' of the entry (in even or odd
              buffer)
113       end if;
```

```
114       end if;
115     end process;
116
117     process(buffer_order, addr, hit_idx, hit, set_select)
118       variable v_counter     : integer range 0 to associativity -
            1;
119       variable v_shift_until  : integer range 0 to associativity -
            1;
120     begin
121       v_shift_until := 0;
122       if hit = '0' or set_select = '0' then
123         out_hit <= (others => '0');
124       else
125       -- *Hit = find location of hit, output the hits for this
            sub_set level, and indicate, through
126       -- shit_until, how many buffer_orders should be shifted at
            the next clock cycle.
127         v_counter := associativity - 1;
128         for k in associativity - 1 downto 0 loop
129           if buffer_order(k).source = addr then
130             if v_counter = hit_idx then
131               -- We have found our set_entry that corresponds to
                    the hit. Exit
132               v_shift_until := k;
133               -- Set the 'hit' bit for this cache sub-set to be
                    connected to cache
134               -- hit counters and lower levels of sub-caches
135               out_hit(k) <= '1';
136             else
137               out_hit(k) <= '0';
138             end if;
139
140             if v_counter > 0 then
141               v_counter := v_counter - 1;
142             end if;
143           else
144             out_hit(k) <= '0';
145           end if;
146         end loop;
147       end if;
148       shift_until <= v_shift_until;
149     end process;
150   end rtl;
```

# Appendix B

# MASH{fifo} Implementation

The main source files for the MASH{fifo} simulator are given here. The configuration file global.vhd is missing as the same file has already been portrayed in Appendix A.1. The main top-level file fifo_cache_sim.vhd and the fifo_sub_set_collection.vhd file are not given either as they are very similar to the ones presented in Appendix A.2 and A.4 respectively.

## B.1 Top Level FIFO Cache Set

Defined in this section is a top level set of the MASH{fifo} cache simulator. It effectively combines the LRU container with the different FIFO shift register chains.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  library work;
6  use work.global.ALL;
7
8  entity fifo_cache_set is
9    port (
10     clk             : in std_logic;
11     reset_n         : in std_logic;
12
13     -- Control inputs
14     set_select      : in std_logic;
15     address         : in std_logic_vector(address_width - 1 downto
           0);
16
17     -- Hit outputs for different levels of associativity of this
           set
18     hits_incr       : out std_logic_vector(0 to assoc_inc_width -
         1);
19
```

```
20      -- Outputs for smaller cache set counts
21      location_ptr   : out std_logic_vector(associativity_mux
           downto 0);
22      hit            : out std_logic
23    );
24  end fifo_cache_set;
25
26  architecture rtl of fifo_cache_set is
27
28    component lru_set is
29      port (
30        clk            : in std_logic;
31        reset_n        : in std_logic;
32        set_select     : in std_logic;
33        address        : in std_logic_vector(address_width - 1
             downto 0);
34
35        location_ptr   : out std_logic_vector(associativity_mux
             downto 0);
36        hit            : out std_logic
37      );
38    end component;
39
40    component fifo_set is
41      generic (
42        tracker_width    : integer;
43        line_count_mux   : integer range 0 to associativity_mux
44      );
45      port (
46        clk            : in std_logic;
47        reset_n        : in std_logic;
48
49        -- Inputs used to control the module
50        set_select     : in std_logic;
51        location_ptr   : in std_logic_vector(tracker_width - 1
             downto 0);
52        hit_in         : in std_logic ;
53
54        -- Output used to detect a hit
55        hit_out        : out std_logic
56      );
57    end component;
58
59    signal l_location_ptr   : std_logic_vector(associativity_mux
        downto 0);
60    signal l_hit            : std_logic;
```

```
61
62  begin
63
64    -- Instantiate the top level lru set of size 2 * associativity
          - 1.
65    -- A 4 way set associative set will need an LRU of 7 ways
66    lru_set_inst : lru_set
67      port map(
68        clk             => clk,
69        reset_n         => reset_n,
70
71        -- Control inputs
72        set_select      => set_select,
73        address         => address,
74
75        -- Status outputs
76        location_ptr    => l_location_ptr,
77        hit             => l_hit
78      );
79
80    hit <= l_hit;
81    location_ptr <= l_location_ptr;
82
83    GENERATE_FIFO_SETS:
84    for k in 0 to associativity_mux generate
85      fifo_set_inst : fifo_set
86        generic map (
87          tracker_width   => associativity_mux + 1,
88          line_count_mux  => k
89        )
90        port map (
91          clk             => clk,
92          reset_n         => reset_n,
93
94          -- Control inputs
95          set_select      => set_select,
96          location_ptr    => l_location_ptr,
97          hit_in          => l_hit,
98
99          -- Output used to detect a hit
100         hit_out         => hits_incr(associativity_mux - k)
101       );
102   end generate;
103
104
105 end rtl;
```

## B.2 LRU Container Definition

LRU container definition from lru_set.vhd for use in the MASH{fifo} simulator. The main difference between this implementation and that of Appendix A.3 is that here the tag data is fixed in a location. Separate LRU shift registers ensure that the tags obey the LRU replacement policy.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  library work;
6  use work.global.ALL;
7
8  entity lru_set is
9    port (
10     clk            : in std_logic;
11     reset_n        : in std_logic;
12     set_select     : in std_logic;
13     address        : in std_logic_vector(address_width - 1 downto
           0);
14
15     location_ptr   : out std_logic_vector(associativity_mux
         downto 0);
16     hit            : out std_logic
17   );
18 end lru_set;
19
20 architecture rtl of lru_set is
21   -- Note that in some situations '(associativity * 2) - 2' was
       not replaced with
22   -- 'least_recently_used': those cases deal with indecies for '
       lru_set_inst' which
23   -- just stores data without any knowledge of how recently a
       block was used.
24   constant least_recently_used  : integer := (associativity * 2)
       - 2;
25
26   -- This array tracks where the data would be stored in an LRU
       cache
27   type lru_set_entry is
28     record
29       valid        : std_logic;
30       source_ptr   : integer range 0 to least_recently_used;
31     end record;
32   type set_tracker is array(0 to least_recently_used) of
       lru_set_entry;
33   signal lru_set_tracker : set_tracker;
```

```
34
35    -- This array of tags holds the tag data itself
36    type lru_set is
37      record
38        valid     : std_logic;
39        tag       : std_logic_vector(tag_length - 1 downto 0);
40      end record;
41    type set_array is array(0 to (2 * associativity) - 2) of
        lru_set;
42    signal lru_set_inst   : set_array;
43
44    -- Signal used initially to provide an index at which to
        initially fill lru_set_inst
45    signal fill_tag_idx : integer range 0 to least_recently_used +
         1 := 0;
46
47    -- Local signals that are exported
48    signal l_hit_idx   : std_logic_vector(0 to least_recently_used)
        ;
49    signal l_hit_ptr   : std_logic_vector(associativity_mux downto
        0);
50    signal l_hit       : std_logic;
51
52    signal i_hit_ptr   : integer range 0 to (2 * associativity) -
        1;
53
54    -- <start_shifting> defines at which location in the LRU shift
         register a hit has occurred.
55    -- If the value is equal to least_recently_used + 1 we are
        pointing to beyond the shift
56    -- register, in which case just shift the entire chain.
57    signal start_shifting   : integer range 0 to
        least_recently_used + 1 := least_recently_used + 1;
58    -- Defines which lru_set_inst index we are writing to.
59    signal write_to_set     : integer range 0 to
        least_recently_used + 1 := 0;
60
61    -- Current address tag
62    signal comp_tag     : std_logic_vector(tag_length - 1 downto 0)
        ;
63
64  begin
65
66    -- Extract the tag value
67    comp_tag <= address(address_width - 1 downto bottom_addr_bits
        + 1);
```

```
68
69  -- Determine whether we have a hit or not, and set the hit
       index
70  process(set_select, lru_set_inst, comp_tag)
71    variable v_hit      : std_logic;
72  begin
73    if set_select = '1' then
74      v_hit := '0';
75      for k in 0 to (2 * associativity) - 2 loop
76        if lru_set_inst(k).valid = '1' and lru_set_inst(k).tag =
              comp_tag then
77          l_hit_idx(k) <= '1';
78          v_hit := '1';
79        else
80          l_hit_idx(k) <= '0';
81        end if;
82      end loop;
83      l_hit <= v_hit;
84    else
85      l_hit_idx <= (others => '0');
86      l_hit <= '0';
87    end if;
88  end process;
89
90  -- Out of the l_hit_idx vector create a pointer.
91  process(l_hit_idx)
92    variable v_hit_ptr  : std_logic_vector(associativity_mux
         downto 0);
93  begin
94    v_hit_ptr := (others => '0');
95    for k in 0 to (associativity * 2) - 2 loop
96      if l_hit_idx(k) = '1' then
97        v_hit_ptr := std_logic_vector(to_unsigned(k,
             associativity_mux + 1));
98      end if;
99    end loop;
100   l_hit_ptr <= v_hit_ptr;
101 end process;
102
103 -- Assign the hit-related output
104 hit <= l_hit;
105
106 i_hit_ptr <= to_integer(unsigned(l_hit_ptr));
107
108 -- Determine how data should be shifted depending on the
       current hit status
```

```
109   process(l_hit, i_hit_ptr, lru_set_tracker, fill_tag_idx)
110     variable v_start_shifting     : integer range 0 to
          least_recently_used + 1;
111   begin
112     v_start_shifting := least_recently_used + 1;
113
114     if l_hit = '1' then
115       -- We have a hit! Find at which index in our LRU shift
            register this tag is located
116       start_shifting <=  least_recently_used + 1;
117       for k in 0 to least_recently_used loop
118         if lru_set_tracker(k).valid = '1' and lru_set_tracker(k)
              .source_ptr = i_hit_ptr then
119           v_start_shifting := k; -- Determine when in the chain
                we need to stop shifting.
120         end if;
121       end loop;
122       write_to_set <= i_hit_ptr; -- This replaces the hit tag
            value with itself, but is easier to implement.
123     else
124       -- No hit has occurred. Either place new data in an empty
            location, or discard an item of data
125       if fill_tag_idx = least_recently_used + 1 then
126         -- Write to the tag pointed to by the least recently
              used bit of data in the LRU tracker.
127         write_to_set <= lru_set_tracker(least_recently_used).
              source_ptr;
128       else
129         -- We still have some empty tag(s)
130         write_to_set <= fill_tag_idx;
131       end if;
132     end if;
133
134     -- Shift the whole chain
135     start_shifting <= v_start_shifting;
136   end process;
137
138   -- Let the outside world know at which location we are storing
        the next token
139   location_ptr <= std_logic_vector(to_unsigned(write_to_set,
        associativity_mux + 1));
140
141   -- When the clock ticks, update the order in which the tags
        are stored
142   process(clk, reset_n, set_select)
143     variable v_shift  : std_logic := '0';
```

```
144   begin
145     if reset_n = '0' then
146
147       for k in 0 to least_recently_used loop
148         lru_set_inst(k).valid <= '0';       -- Here we have the
              set data itself
149         lru_set_tracker(k).valid <= '0';    -- And here the
              tracker so that the whole thing behaves like an LRU
              cache
150       end loop;
151       fill_tag_idx <= 0;
152
153     elsif rising_edge(clk) and set_select = '1' then
154
155       if l_hit = '0' and fill_tag_idx < least_recently_used + 1
            then
156         fill_tag_idx <= fill_tag_idx + 1;
157       end if;
158
159       if start_shifting = least_recently_used + 1 then
160         v_shift := '1';
161       else
162         v_shift := '0';
163       end if;
164
165       -- Write the tag to the selected set entry
166       lru_set_inst(write_to_set).valid <= '1';
167       lru_set_inst(write_to_set).tag <= comp_tag;
168
169       -- Shift everything smaller or equal to start_shifting
170       for k in least_recently_used downto 1 loop
171         if k <= start_shifting then
172           lru_set_tracker(k) <= lru_set_tracker(k - 1);
173         end if;
174       end loop;
175
176       -- Set the least recently used entry
177       lru_set_tracker(0).valid <= '1';
178       lru_set_tracker(0).source_ptr <= write_to_set;
179     end if;
180   end process;
181 end rtl;
```

## B.3 FIFO Set

fifo_set.vhd is a lightweight shift register that tracks the state of an actual FIFO set by shifting pointers that point towards the corresponding tags in the LRU container.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  library work;
6  use work.global.ALL;
7
8  entity fifo_set is
9    generic (
10     tracker_width    : integer;
11     line_count_mux   : integer range 0 to associativity_mux
12   );
13   port (
14     clk            : in std_logic;
15     reset_n        : in std_logic;
16
17     -- Inputs used to control the module
18     set_select     : in std_logic;
19     location_ptr   : in std_logic_vector(tracker_width - 1 downto
           0);
20     hit_in         : in std_logic ;
21
22     -- Output used to detect a hit
23     hit_out        : out std_logic
24   );
25 end fifo_set;
26
27 architecture rtl of fifo_set is
28
29   constant line_count  : integer := 2 ** line_count_mux;
30
31   -- This array tracks where the data would be stored in a FIFO
       cache
32   type fifo_set_entry is
33     record
34       valid        : std_logic;
35       source_ptr   : integer range 0 to 2 ** tracker_width - 1;
36     end record;
37   type set_tracker is array(0 to line_count - 1) of
       fifo_set_entry;
38   signal fifo_set_tracker : set_tracker;
39
```

```
40   -- External input converted to an integer
41   signal i_location_ptr  : integer range 0 to 2 ** tracker_width
         - 1;
42
43   -- next_entry defines at which location in the FIFO the next
         entry
44   -- will be written to
45   signal next_entry  : integer range 0 to line_count - 1 := 0;
46   -- Local hit
47   signal l_hit     : std_logic;
48
49 begin
50
51   i_location_ptr <= to_integer(unsigned(location_ptr));
52
53   process(i_location_ptr, hit_in, fifo_set_tracker)
54   begin
55     if hit_in = '1' then
56       l_hit <= '0';
57       for k in 0 to line_count - 1 loop
58         if fifo_set_tracker(k).valid = '1' and fifo_set_tracker(
              k).source_ptr = i_location_ptr then
59           l_hit <= '1';
60         end if;
61       end loop;
62     else
63       l_hit <= '0';
64     end if;
65   end process;
66
67   hit_out <= l_hit;
68
69   process(clk, reset_n)
70   begin
71     if reset_n = '0' then
72
73       next_entry <= 0;
74       for k in 0 to line_count - 1 loop
75         fifo_set_tracker(k).valid <= '0';
76       end loop;
77
78     elsif rising_edge(clk) then
79
80       if set_select = '1' and l_hit = '0' then
81         -- Only if there is a miss do we change the state of our
               trackers
```

```
82          fifo_set_tracker(next_entry).valid <= '1';
83          fifo_set_tracker(next_entry).source_ptr <=
               i_location_ptr;
84          if next_entry < line_count - 1 then
85            next_entry <= next_entry + 1;
86          else
87            next_entry <= 0;
88          end if;
89        end if;
90
91      end if;
92    end process;
93
94  end rtl;
```

# Appendix C

# MASS{plrut} Implementation

The main code for the MASS{plrut} simulator is given in this Appendix.

## C.1 PLRU Update and Get Evict Functions

```
// Lookup table for the fast tree update
const uint64_t setPatterns[32] = {
  0x000000007FFF7F74, // 1
  0x000000017FFF7F74, // 2
  0x000000027FFF7F71, // 3
  0x000000067FFF7F71, // 4
  0x000000087FFF7F47, // 5
  0x000000187FFF7F47, // 6
  0x000000287FFF7F17, // 7
  0x000000687FFF7F17, // 8
  0x000000807FFF747F, // 9
  0x000001807FFF747F, // 10
  0x000002807FFF717F, // 11
  0x000006807FFF717F, // 12
  0x000008807FFF477F, // 13
  0x000018807FFF477F, // 14
  0x000028807FFF177F, // 15
  0x000068807FFF177F, // 16
  0x000080007F747FFF, // 17
  0x000180007F747FFF, // 18
  0x000280007F717FFF, // 19
  0x000680007F717FFF, // 20
  0x000880007F477FFF, // 21
  0x001880007F477FFF, // 22
  0x002880007F177FFF, // 23
  0x006880007F177FFF, // 24
  0x00808000747F7FFF, // 25
```

```
28    0x01808000747F7FFF, // 26
29    0x02808000717F7FFF, // 27
30    0x06808000717F7FFF, // 28
31    0x08808000477F7FFF, // 29
32    0x18808000477F7FFF, // 30
33    0x28808000177F7FFF, // 31
34    0x68808000177F7FFF, // 32
35  };
36
37  INLINE uint8_t treeGetEvictIdx(IN PlruTree tree, IN uint8_t n) {
38    register int node;
39    register uint8_t result = 0;
40
41    // Start at node (2 ^ n) / 2
42    for (node = (1 << n) >> 1; node > 0; node >>= 1) {
43      if (!(tree & 1 << (node - 1))) {
44        result |= node;
45        tree >>= node;
46      }
47    }
48
49    return result;
50  }
51
52  INLINE void treeUpdate(IN PlruTree *tree, IN uint8_t idx) {
53    // Ultra-fast tree update with lookup table
54    uint64_t setPattern = setPatterns[idx];
55    *tree = (PlruTree)(((uint64_t)*tree & setPattern) | setPattern
          >> 32);
56  }
```

## C.2 Main MASS{plrut} code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "plru_utils.h"
5
6  #define ONE                  1
7  #define TWO                  2
8  #define LOCATION_EMPTY       0xFF
9  #define LOCATION_EMPTY_64    0xFFFFFFFFFFFFFFFF
10
11 typedef struct SetState_opt {
12   PlruTree tree;
13   uint8_t **trackers;
```

```
14 } SetState_opt;
15
16 typedef struct {
17   SetState_opt *setStates;
18 } CacheState;
19
20 typedef struct Node {
21   uint32_t val;
22   uint8_t *locations;
23   struct Node *nextNode;
24 } Node;
25
26 typedef struct NodeLst {
27   Node *nodes;
28   int nodeCount;
29   int nodesAvailable;
30 } NodeList;
31
32 extern int N;                  // Degree of associativity = 2^N.
33 extern int B;                  // Max set field width.
34 extern int A;                  // Min set field width.
35 extern int L;                  // Line field width.
36 extern int T;                  // Max no of addresses to be
37                                // processed.
38 extern int SAVE_INTERVAL;      // Intervals at which output should
39                                // be saved.
40 extern int P_INTERVAL;         // Intervals at which progress
41                                // output is done.
42
43 extern int(*trace) ();
44 extern FILE *out;
45
46 // Local variables
47 CacheState **cacheStates;      // State of the entire cache.
48 NodeList freeNodes;            // List of empty nodes.
49 Node **nodeHash;               // Hash table containing all the
50                                // nodes stored in all the caches.
51
52 int TWO_PWR_N;                 // 2^N. i.e., Degree of
53                                // associativity .
54 int MAX_DEPTH;                 // B–A, Number of range of sets
55                                // simulated .
56 int TOP_LEVEL_SETS;            // Set size of the largest cache to
57                                // be simulated.
58 int CONTAINER_SIZE;            // Size of a container at top level
    .
```

```
59 int SUBCONTAINER_SIZE;          // Size of a container for the
60                                 // lower levels.
61 int HASH_TABLE_SIZE;            // Size of the hash table.
62 int TOTAL_NODE_COUNT;           // Total number of nodes we will be
63                                 // using.
64 int CONFIG_COUNT;               // Total number of configurations
65                                 // we are simulating.
66 int LOCATIONS_FIELD_SIZE;       // Size of the field containing
67                                 // locations - given in uint64_t.
68
69 uint64_t **hits;                // Hit counters for the different
70                                 // cache configurations.
71 uint64_t hits_all;              // Counts the number of hits due to
72                                 // the previous addr being repeated
73
74 uint64_t accessCount;           // Count of addresses processed.
75
76 // Shared tree functions given in sacplru.c
77 extern INLINE uint8_t treeGetEvictIdx(IN PlruTree tree, IN
       uint8_t n);
78 extern INLINE void treeUpdate(IN PlruTree *tree, IN uint8_t idx)
       ;
79
80 INLINE void clearLocations(IN uint8_t *locations) {
81   memset(locations, 0xFF, LOCATIONS_FIELD_SIZE * sizeof(uint64_t
         ));
82 }
83
84 INLINE uint8_t isTracked(CONST_IN uint8_t *locations) {
85   uint8_t idx;
86   uint64_t *locations64bit = (uint64_t *)locations;
87
88   // Check whether all the given locations are empty, 8 at a
         time.
89   for (idx = 0; idx < LOCATIONS_FIELD_SIZE; idx++) {
90     if (*locations64bit != LOCATION_EMPTY_64) {
91       return TRUE;
92     }
93     locations64bit++;
94   }
95   return FALSE;
96 }
97
98 INLINE void returnFreeNode(IN Node *node) {
99   node->nextNode = freeNodes.nodes;
100  freeNodes.nodes = node;
```

```
101   freeNodes.nodesAvailable++;
102 }
103
104 INLINE void garbageCollect() {
105   int nodeIdx;
106   Node **startNode;
107   register Node *prevNode;
108   register Node *currentNode;
109
110   for (nodeIdx = 0; nodeIdx < HASH_TABLE_SIZE; nodeIdx++) {
111     startNode = &nodeHash[nodeIdx];
112     currentNode = *startNode;
113     prevNode = NULL;
114     while (currentNode != NULL) {
115       if (isTracked(currentNode->locations) == FALSE) {
116         // We have stumbled across an obsolete node, delete it
              from the hash table
117         if (prevNode == NULL) {
118           *startNode = currentNode->nextNode;
119           returnFreeNode(currentNode);
120           currentNode = *startNode;
121         } else {
122           prevNode->nextNode = currentNode->nextNode;
123           returnFreeNode(currentNode);
124           currentNode = prevNode->nextNode;
125         }
126         continue;
127       }
128
129       prevNode = currentNode;
130       currentNode = currentNode->nextNode;
131     }
132   }
133 }
134
135 INLINE Node *getFreeNode() {
136   Node *retNode;
137
138   if (freeNodes.nodesAvailable == 0) {
139     // No free nodes available. Should be a rare occurrence.
140     // Traverse the entire hash table and free any nodes that
141     // can be freed.
142     garbageCollect();
143   }
144
145   retNode = freeNodes.nodes;
```

```
146   freeNodes.nodes = freeNodes.nodes->nextNode;
147   freeNodes.nodesAvailable--;
148
149   return retNode;
150 }
151
152 // Perl's hash function
153 INLINE uint32_t hash_func(CONST_IN void *key) {
154   register size_t i = 4;
155   register uint32_t hv = 0; // can put a seed here instead of 0
156   register const unsigned char *s = (char *)key;
157   while (i--) {
158     hv += *s++;
159     hv += (hv << 10);
160     hv ^= (hv >> 6);
161   }
162   hv += (hv << 3);
163   hv ^= (hv >> 11);
164   hv += (hv << 15);
165
166   return hv;
167 }
168
169 INLINE Node *getNode(IN uint32_t addr) {
170   register Node *currentNode;
171   register Node *prevNode = NULL;
172   Node **startNode = &nodeHash[hash_func(&addr) %
        HASH_TABLE_SIZE];
173   Node *targetNode = NULL;
174
175   currentNode = *startNode;
176
177   while (1) {
178     if (currentNode == NULL) {
179       if (targetNode == NULL) {
180         targetNode = getFreeNode();
181
182         // Initialise this new node
183         targetNode->val = addr;
184         targetNode->nextNode = *startNode;
185         *startNode = targetNode;
186       }
187       break;
188     } else {
189       if (addr == currentNode->val) {
190         // We have found the node we are looking for
```

```
191           targetNode = currentNode;
192         } else if (isTracked(currentNode->locations) == FALSE) {
193           // We have stumbled across an obsolete node, delete it
                 from the hash table
194           if (prevNode == NULL) {
195             *startNode = currentNode->nextNode;
196             returnFreeNode(currentNode);
197             currentNode = *startNode;
198           } else {
199             prevNode->nextNode = currentNode->nextNode;
200             returnFreeNode(currentNode);
201             currentNode = prevNode->nextNode;
202           }
203           continue;
204         }
205         prevNode = currentNode;
206         currentNode = currentNode->nextNode;
207       }
208     }
209
210     return targetNode;
211   }
212
213   INLINE void updateTables(Node *node) {
214
215     uint32_t mask = TOP_LEVEL_SETS - 1;
216     int level;
217     int assoc;
218     uint8_t *locations = node->locations;
219
220     for (level = 0; level < MAX_DEPTH + 1; level++) {
221       for (assoc = 0; assoc < N + 1; assoc++) {
222         SetState_opt *currentSet = &(cacheStates[level][assoc].
              setStates[node->val & mask]);
223
224         if (*locations != LOCATION_EMPTY) {
225           // We have a hit
226           hits[level][assoc]++;
227           treeUpdate(&currentSet->tree, *locations);
228         } else {
229           // We have a miss
230
231           uint8_t newIdx;
232           uint8_t *oldLocation;
233
234           if (assoc > 0) {
```

```
235             newIdx = treeGetEvictIdx(currentSet->tree, assoc);
236           } else {
237             // If our cache only has one set the replcement idx
                   will always be 0
238             newIdx = 0;
239           }
240
241           oldLocation = currentSet->trackers[newIdx];
242           if (oldLocation != NULL) {
243             // The new location is linked to an old address node.
                   Clear it.
244             *oldLocation = LOCATION_EMPTY;
245           }
246           currentSet->trackers[newIdx] = locations;
247           *locations = newIdx;
248           treeUpdate(&currentSet->tree, newIdx);
249         }
250         locations++;
251       }
252       mask >>= 1;
253     }
254 }
255
256 void sacplru_opt() {
257   uint32_t addr;
258   uint32_t prev_addr = 0x80000000;
259
260   uint64_t next_save_time;
261   unsigned l;           // Current line index within the buffer.
262   unsigned *buffer;     // Input buffer.
263   unsigned nr;          // Number of lines read from the last
264                         // file read op.
265
266   next_save_time = SAVE_INTERVAL;
267
268   while (nr = trace(&buffer)) {
269     if (accessCount > next_save_time) {
270       outpr_sacplru();
271       next_save_time += SAVE_INTERVAL;
272     }
273
274     for (l = 0; l < nr; l++) {
275       ++accessCount;
276       if ((accessCount % P_INTERVAL) == 0) {
277         printf("Addresses processed  %llu\n", accessCount);
278       }
```

```
279
280         addr = *(buffer + 1);
281         addr >>= L;
282
283         if (prev_addr == addr) {
284           // The same address has just been repeated. All cache
                configs have a hit.
285           hits_all++;
286         } else {
287           Node *node = getNode(addr);
288           updateTables(node);
289         }
290
291         prev_addr = addr;
292       }
293     }
294 }
295
296 void outpr_sacplru_opt() {
297   int level;
298   int assoc;
299   uint64_t sum;
300
301   fprintf(out, "Addresses processed: %llu\n", accessCount);
302   fprintf(out, "Line size: %d bytes\n", (ONE << L));
303   fprintf(out, "\n");
304   fprintf(out, "Hit Ratios\n");
305   fprintf(out, "----------\n\n");
306   fprintf(out, "\t\tAssociativity\n");
307   fprintf(out, "\t\t");
308   for (assoc = 0; assoc < N + 1; assoc++) {
309     fprintf(out, "%d\t\t", (1 << assoc));
310   }
311   fprintf(out, "\n");
312   fprintf(out, "No. of sets\n");
313   for (level = 0; level <= MAX_DEPTH; level++) {
314     fprintf(out, "%d\t\t", (ONE << (level + A)));
315     for (assoc = 0; assoc < N + 1; assoc++) {
316       sum = hits[MAX_DEPTH - level][assoc] + hits_all;
317       fprintf(out, "%lf\t%llu\t", (1.0 - ((double)sum / (double)
              accessCount)), sum);
318       //                    (t_entries - sum ) );
319     }
320     fprintf(out, "\n");
321   }
322   fprintf(out, "\n\n\n");
```

```
323 }
324
325 void init_sacplru_opt() {
326   int level;
327   int assoc;
328   int set;
329   int node;
330
331   TWO_PWR_N = (ONE << N);
332   MAX_DEPTH = B - A;
333
334   TOP_LEVEL_SETS = (ONE << B);
335   CONTAINER_SIZE = ((1 << N) << 1) - 1 - ((N * (N + 1)) >> 1);
        // 2n - 1 - (N * (N + 1)) / 2
336   SUBCONTAINER_SIZE = CONTAINER_SIZE - N - 1;
337   TOTAL_NODE_COUNT = (CONTAINER_SIZE * TOP_LEVEL_SETS) + (
        SUBCONTAINER_SIZE * (TOP_LEVEL_SETS - 1));
338   HASH_TABLE_SIZE = TOTAL_NODE_COUNT;
339   CONFIG_COUNT = (MAX_DEPTH + 1) * (N + 1);
340
341   LOCATIONS_FIELD_SIZE = CONFIG_COUNT / sizeof(uint64_t);
342   if (CONFIG_COUNT % sizeof(uint64_t) != 0) {
343     LOCATIONS_FIELD_SIZE += 1;
344   }
345
346   // Initialise the hit counters
347   hits = (uint64_t **)malloc((MAX_DEPTH + 1) * sizeof(uint64_t
        *));
348   for (level = 0; level < MAX_DEPTH + 1; level++) {
349     hits[level] = (uint64_t *)calloc(N + 1, sizeof(uint64_t));
350   }
351   hits_all = 0;
352
353   // Initialise the cache states for all the different caches
        simulated
354   cacheStates = (CacheState **)malloc((MAX_DEPTH + 1) * sizeof(
        CacheState *));
355   for (level = 0; level < MAX_DEPTH + 1; level++) {
356     cacheStates[level] = (CacheState *)malloc((N + 1) * sizeof(
          CacheState));
357     for (assoc = 0; assoc < N + 1; assoc++) {
358       SetState_opt *setStates = (SetState_opt *)malloc((
            TOP_LEVEL_SETS >> level) * sizeof(SetState_opt));
359       cacheStates[level][assoc].setStates = setStates;
360       for (set = 0; set < (TOP_LEVEL_SETS >> level); set++) {
361         setStates[set].tree = 0;
```

```
362             setStates[set].trackers = (uint8_t **)calloc((uint64_t)
                    (1 << assoc), sizeof(uint8_t *));
363           }
364         }
365       }
366
367       // Initialise the list of free nodes
368       freeNodes.nodesAvailable = TOTAL_NODE_COUNT;
369       freeNodes.nodeCount = TOTAL_NODE_COUNT;
370       freeNodes.nodes = (Node *)malloc(TOTAL_NODE_COUNT * sizeof(
              Node));
371       for (node = 0; node < TOTAL_NODE_COUNT; node++) {
372         freeNodes.nodes[node].locations = (uint8_t *)malloc(
                LOCATIONS_FIELD_SIZE * sizeof(uint64_t));
373         clearLocations(freeNodes.nodes[node].locations);
374       }
375       // Link all the free nodes together
376       for (node = 0; node < TOTAL_NODE_COUNT - 1; node++) {
377         freeNodes.nodes[node].nextNode = &freeNodes.nodes[node + 1];
378       }
379       freeNodes.nodes[node].nextNode = NULL;
380
381       // Initialise the hash table
382       nodeHash = (Node **)calloc(HASH_TABLE_SIZE, sizeof(Node *));
383     }
```

# Bibliography


[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[2] Solid State Technology, *Moores Law has stopped at 28nm*. http://electroiq.com/blog/2014/03/moores-law-has-stopped-at-28nm/, 2014.

[3] O. Semenov, A. Vassighi, and M. Sachdev, "Impact of self-heating effect on long-term reliability and performance degradation in cmos circuits," *Device and Materials Reliability, IEEE Transactions on*, vol. 6, pp. 17–27, March 2006.

[4] M. Pedram and Q. Wu, "Design considerations for battery-powered electronics," in *Design Automation Conference, 1999. Proceedings. 36th*, pp. 861–866, 1999.

[5] G. K. Yeap, *Practical Low Power Digital VLSI Design*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[6] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[7] A. Vassighi and M. Sachdev, "Thermal runaway in integrated circuits," *Device and Materials Reliability, IEEE Transactions on*, vol. 6, pp. 300–305, June 2006.

[8] P. Nilsson, "Arithmetic reduction of the static power consumption in nanoscale cmos," in *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pp. 656–659, Dec 2006.

[9] G. Gilder, "The coming software shift: Telecosm," *Forbes ASAP*, 1995.

[10] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[11] P. Jacob, O. Erdogan, A. Zia, P. Belemjian, R. Kraft, and J. McDonald, "Predicting the performance of a 3d processor-memory chip stack," *Design Test of Computers, IEEE*, vol. 22, pp. 540–547, Nov 2005.

[12] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, pp. 473–530, Sept. 1982.

[13] A. Janapsatya, A. Ignjatović, and S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, (Piscataway, NJ, USA), pp. 796–801, IEEE Press, 2006.

[14] M. S. Haque, A. Janapsatya, and S. Parameswaran, "SuSeSim: a fast simulation strategy to find optimal L1 cache configuration for embedded systems," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, (New York, NY, USA), pp. 295–304, ACM, 2009.

[15] M. Tawada, M. Yanagisawa, T. Ohtsuki, and N. Togawa, "Exact and fast L1 cache configuration simulation for embedded systems with FIFO/PLRU cache replacement policies," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–4, 2011.

[16] Altera, *Nios II Processor*. http://www.altera.com/devices/processor/nios2/ni2-index.html, 2014.

[17] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, pp. 99–122, Nov. 2007.

[18] Tensilica. http://www.tensilica.com/, 2014.

[19] HP Labs, *CACTI*. http://www.hpl.hp.com/research/cacti/, 2014.

[20] J. Edler and M. D. Hill, *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. http://pages.cs.wisc.edu/ markhill/DineroIV/.

[21] P. Jones, J. Lockwood, and Y. Cho, "A thermal management and profiling method for reconfigurable hardware applications," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pp. 1–7, Aug 2006.

[22] P. Jones, J. Moscola, Y. Cho, and J. Lockwood, "Adaptive thermoregulation for applications on reconfigurable devices," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 246–253, Aug 2007.

[23] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a mpeg decoder," in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pp. 732–737, Nov 2002.

[24] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Power-aware scheduling for periodic real-time tasks," *Computers, IEEE Transactions on*, vol. 53, pp. 584–600, May 2004.

[25] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pp. 174–179, Aug 2004.

[26] C. Chow, L. S. M. Tsui, P.-W. Leong, W. Luk, and S. J. E. Wilton, "Dynamic voltage scaling for commercial fpgas," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 173–180, Dec 2005.

[27] Q. Wu, P. Juang, M. Martonosi, and D. Clark, "Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 178–189, Feb 2005.

[28] A. Portero, G. Talavera, M. Monton, B. Martinez, F. Cathoor, and J. Carabina, "Dynamic voltage scaling for power efficient mpeg4-sp implementation," in *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pp. 257–260, Sept 2006.

[29] P. Jones, Y. Cho, and J. Lockwood, "An adaptive frequency control method using thermal feedback for reconfigurable hardware applications," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pp. 229–236, Dec 2006.

[30] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 2–13, June 2003.

[31] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, pp. 18–28, Jan 2005.

[32] J. Srinivasan and S. V. Adve, "Predictive dynamic thermal management for multimedia applications," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, (New York, NY, USA), pp. 109–120, ACM, 2003.

[33] L. Yan, J. Luo, and N. Jha, "Joint dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, pp. 1030–1041, July 2005.

[34] D. Atienza and E. Martinez, "Inducing thermal-awareness in multicore systems using networks-on-chip," in *VLSI, 2009. ISVLSI '09. IEEE Computer Society Annual Symposium on*, pp. 187–192, May 2009.

[35] B. Noble, M. Price, and M. Satyanarayanan, "A programming interface for application-aware adaptation in mobile computing," in *Computing Systems*, pp. 57–66, 1995.

[36] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," pp. 276–287, 1997.

[37] B. Noble and M. Satyanarayanan, *Experience with adaptive mobile applications in Odyssey*, 1999.

[38] D. Narayanan, J. Flinn, and M. Satyanarayanan, "Using history to improve mobile application adaptation," in *Mobile Computing Systems and Applications, 2000 Third IEEE Workshop on.*, pp. 31–40, 2000.

[39] P. Jones, J. Moscola, Y. Cho, and J. Lockwood, "Changing output quality for thermal management," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 353–354, April 2007.

[40] J. Peddersen and S. Parameswaran, "Energy driven application selfadaptation," in *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pp. 385 –390, jan. 2007.

[41] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pp. 217–222, Aug 2003.

[42] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, "Low cost task migration initiation in a heterogeneous mp-soc," in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 252–253 Vol. 1, March 2005.

[43] E. W. Briao, D. Barcelos, F. Wronski, and F. Wagner, "Impact of task migration in noc-based mpsocs for soft real-time applications," in *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pp. 296–299, Oct 2007.

[44] T. Ebi, M. Faruque, and J. Henkel, "TAPE: Thermal-aware agent-based power econom multi/many-core architectures," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 302–309, Nov 2009.

[45] H. Shen and F. Petrot, "Novel task migration framework on configurable heterogeneous mp-soc platforms," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pp. 733–738, Jan 2009.

[46] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii, "Adaptive task migration policies for thermal control in mpsocs," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pp. 110–115, July 2010.

[47] Y. Ge, P. Malani, and Q. Qiu, "Distributed task migration for thermal management in many-core systems," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pp. 579–584, June 2010.

[48] M. Faruque, J. Jahn, T. Ebi, and J. Henkel, "Runtime thermal management using software agents for multi- and many-core architectures," *Design Test of Computers, IEEE*, vol. 27, pp. 58–68, Nov 2010.

[49] W. Hung, C. Addo-Quaye, T. Theocharides, Y. Xie, N. Vijakrishnan, and M. Irwin, "Thermal-aware ip virtualization and placement for networks-on-chip architecture," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pp. 430–437, Oct 2004.

[50] T. Zhang, Y. Zhan, and S. Sapatnekar, "Temperature-aware routing in 3d ics," in *Design Automation, 2006. Asia and South Pacific Conference on*, pp. 6 pp.–, Jan 2006.

[51] M. Pathak and S.-K. Lim, "Performance and thermal-aware steiner routing for 3-d stacked ics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, pp. 1373–1386, Sept 2009.

[52] K. Lu and D. Pan, "Reliability-aware global routing under thermal considerations," in *Quality Electronic Design, 2009. ASQED 2009. 1st Asia Symposium on*, pp. 313–318, July 2009.

[53] K. M. Zick and J. P. Hayes, "On-line sensing for healthier fpga systems," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, (New York, NY, USA), pp. 239–248, ACM, 2010.

[54] S.-Y. Lin, T.-C. Yin, H.-Y. Wang, and A.-Y. Wu, "Traffic-and thermal-aware routing for throttled three-dimensional network-on-chip systems," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–4, April 2011.

[55] Altera, *Temperature Sensor (ALTTEMP_SENSE) Megafunction User Guide*. http://www.altera.com/literature/ug/ug_alttemp_sense.pdf, 2013.

[56] S. Lopez-Buedo, J. Garrido, and E. Boemo, "Thermal testing on reconfigurable computers," *Design Test of Computers, IEEE*, vol. 17, pp. 84–91, Jan 2000.

[57] P. Chen, M.-C. Shie, Z.-Y. Zheng, Z.-F. Zheng, and C.-Y. Chu, "A fully digital time-domain smart temperature sensor realized with 140 fpga logic elements," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 54, pp. 2661–2668, Dec 2007.

[58] P. Chen, S.-C. Chen, Y.-S. Shen, and Y.-J. Peng, "All-digital time-domain smart temperature sensor with an inter-batch inaccuracy of $-0.7\ ^{\circ}\mathrm{c} - +0.6\ ^{\circ}\mathrm{c}$ after one-point calibration,"

*Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 58, pp. 913–920, May 2011.

[59] S. Lopez-Buedo and E. Boemo, "Making visible the thermal behaviour of embedded microprocessors on fpgas: A progress report," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, (New York, NY, USA), pp. 79–86, ACM, 2004.

[60] E. I. Boemo and S. López-Buedo, "Thermal monitoring on fpgas using ring-oscillators," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, FPL '97, (London, UK, UK), pp. 69–78, Springer-Verlag, 1997.

[61] S. Velusamy, W. Huang, J. Lach, M. Stan, and K. Skadron, "Monitoring temperature in fpga based socs," in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pp. 634–637, Oct 2005.

[62] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotspot: a compact thermal modeling methodology for early-stage vlsi design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, pp. 501–513, May 2006.

[63] R. Conn, "Method and apparatus for measuring localized temperatures on integrated circuits," 2000. US Patent 6,067,508.

[64] K. M. Zick and J. P. Hayes, "Low-cost sensing with ring oscillator arrays for healthier reconfigurable systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, pp. 1:1–1:26, Mar. 2012.

[65] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee, "Instruction level power analysis and optimization of software," in *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pp. 326–328, Jan 1996.

[66] T. Givargis, F. Vahid, and J. Henkel, "A hybrid approach for core-based system-level power modeling," in *Asia and South Pacific Design Automation Conference*, pp. 141–145, 2000.

[67] S. Nikolaidis and T. Laopoulos, "Instruction-level power consumption estimation of embedded processors for low-power applications," *Comput. Stand. Interfaces*, vol. 24, pp. 133–137, June 2002.

[68] A. Raghunathan, S. Dey, and N. Jha, "High-level macro-modeling and estimation techniques for switching activity and power consumption," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, pp. 538–557, Aug 2003.

[69] R. Bergamaschi and Y. Jiang, "State-based power analysis for systems-on-chip," in *Design Automation Conference, 2003. Proceedings*, pp. 638–641, June 2003.

[70] G. Contreras and M. Martonosi, "Power prediction for Intel XScale reg; processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, pp. 221–226, Aug 2005.

[71] J. Peddersen and S. Parameswaran, "Clipper: Counter-based low impact processor power estimation at run-time," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pp. 890–895, Jan 2007.

[72] D. Atienza, P. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and J. Mendias, "A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip," in *Design Automation Conference, 2006 43rd ACM/IEEE*, pp. 618–623, 2006.

[73] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 83 –94, june 2000.

[74] M. S. S. Govindan, S. W. Keckler, and D. Burger, "End-to-end validation of architectural power models," in *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, (New York, NY, USA), pp. 383–388, ACM, 2009.

[75] W. Huang, K. Skadron, S. Gurumurthi, R. Ribando, and M. Stan, "Differentiating the roles of ir measurement and simulation for power and temperature-aware design," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 1 –10, april 2009.

[76] ANSYS. http://www.ansys.com/, 2014.

[77] freeFEM3D. http://www.freefem.org/ff3d/, 20124.

[78] T.-Y. Wang and C. C.-P. Chen, "3-d thermal-adi: a linear-time chip level transient thermal simulator," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 1434 – 1445, dec 2002.

[79] Y. Zhan and S. Sapatnekar, "Fast computation of the temperature distribution in vlsi chips using the discrete cosine transform and table look-up," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, vol. 1, pp. 87 – 92 Vol. 1, jan. 2005.

[80] D. Atienza, P. G. Del Valle, G. Paci, F. Poletti, L. Benini, G. D. Micheli, J. M. Mendias, and R. Hermida, "Hw-sw emulation framework for temperature-aware design in mpsocs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, pp. 26:1–26:26, May 2008.

[81] R. Cochran, A. N. Nowroz, and S. Reda, "Post-silicon power characterization using thermal infrared emissions," *Low-Power Electronics and Design (ISLPED), ACM/IEEE International Symposium on*, pp. 331–336, Aug. 2010.

[82] A. N. Nowroz and S. Reda, "Thermal and power characterization of field-programmable gate arrays," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 111–114, ACM, 2011.

[83] P. Sundararajan, A. Gayasen, N. Vijaykrishnan, and T. Tuan, "Thermal characterization and optimization in platform fpgas," *Computer-Aided Design, IEEE/ACM International Conference on*, pp. 443–447, Nov. 2006.

[84] M. Happe, H. Hangmann, A. Agne, and C. Plessl, "Eight ways to put your fpga on fire - a systematic study of heat generators," *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pp. 1–6, Dec. 2012.

[85] L. Bauer and J. Henkel, *Run-time Adaptation for Reconfigurable Embedded Processors*. 2011.

[86] Intel, *Pin - A Dynamic Binary Instrumentation Tool*. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2014.

[87] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," *SIGPLAN Not.*, vol. 29, pp. 196–205, June 1994.

[88] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 317–329, Nov. 2000.

[89] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A Pin-based on-the-fly multi-core cache simulator,"

[90] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 52:1–52:12, ACM, 2011.

[91] Lauterbach, *Lauterbach Development Tools*. http://www.lauterbach.com/, 2014.

[92] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, pp. 78–103, Jan. 1997.

[93] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, pp. 13–25, June 1997.

[94] P. Magnusson and B. Werner, "Efficient memory simulation in simics," in *Simulation Symposium, 1995., Proceedings of the 28th Annual*, pp. 62–73, Apr 1995.

[95] M. M. K. 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," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, Nov. 2005.

[96] OpenCores. http://opencores.org/or1k/Main_Page, 2014.

[97] Stephen Williams, *Icarus Verilog*. http://iverilog.icarus.com/, 2014.

[98] Mentor Graphics, *ModelSim*. http://model.com/, 2014.

[99] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 15:1–15:32, June 2009.

[100] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 249–261, IEEE Computer Society, 2007.

[101] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP Gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 463–468, ACM, 2010.

[102] P. Ravishankar and S. Abdi, "pcache: An observable l1 data cache model for fpga prototyping of embedded systems," in *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 103–110, Sept 2013.

[103] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Comput. Syst.*, vol. 7, pp. 184–215, May 1989.

[104] J. S. Harper, D. J. Kerbyson, and G. R. Nudd, "Analytical modeling of set-associative cache behavior," *IEEE Trans. Comput.*, vol. 48, pp. 1009–1024, Oct. 1999.

[105] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim, "High level cache simulation for heterogeneous multiprocessors," in *Proceedings of the 41st annual Design Automation Conference*, pp. 287–292, ACM, 2004.

[106] X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury, "Design space exploration of caches using compressed traces," in *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, (New York, NY, USA), pp. 116–125, ACM, 2004.

[107] C. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Data Compression Conference, 1997. DCC '97. Proceedings*, pp. 3–11, Mar 1997.

[108] A. Ghosh and T. Givargis, "Cache optimization for embedded processor cores: An analytical approach," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 9, no. 4, pp. 419–440, 2004.

[109] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, (New York, NY, USA), pp. 228–239, ACM, 2006.

[110] Y. Liang and T. Mitra, "Static analysis for fast and accurate design space exploration of caches," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, (New York, NY, USA), pp. 103–108, ACM, 2008.

[111] R. A. Sugumar and S. G. Abraham, "Set-associative cache simulation using generalized binomial trees," *ACM Trans. Comput. Syst.*, vol. 13, pp. 32–56, Feb. 1995.

[112] W. Fornaciari, D. Sciuto, C. Silvano, and V. Zaccaria, "A design framework to efficiently explore energy-delay tradeoffs," in *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pp. 260–265, 2001.

[113] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 407–425, May 2004.

[114] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros, "A One-Shot Configurable-Cache Tuner for Improved Energy and Performance," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pp. 1 –6, april 2007.

[115] T. Givargis, "Zero cost indexing for improved processor cache performance," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, pp. 3–25, Jan. 2006.

[116] M. S. Haque, *Optimizing single-pass simulation techniques for fast cache memory design space exploration in embedded processors*. PhD thesis, University of New South Wales, 2011.

[117] M. Hill and A. Smith, "Evaluating associativity in CPU caches," *Computers, IEEE Transactions on*, vol. 38, pp. 1612 –1630, dec 1989.

[118] N. Tojo, N. Togawa, M. Yanagisawa, and T. Ohtsuki, "Exact and fast L1 cache simulation for embedded systems," in *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*, pp. 817 –822, jan. 2009.

[119] M. Haque, J. Peddersen, A. Janapsatya, and S. Parameswaran, "DEW: A fast level 1 cache simulation approach for embedded processors with FIFO replacement policy," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 496–501, 2010.

[120] M. S. Haque, J. Peddersen, A. Janapsatya, and S. Parameswaran, "SCUD: a fast single-pass L1 cache simulation approach for embedded processors with round-robin replacement policy," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 356–361, ACM, 2010.

[121] M. Haque, J. Peddersen, and S. Parameswaran, "CIPARSim: Cache intersection property assisted rapid single-pass FIFO cache simulation technique," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pp. 126–133, 2011.

[122] M. Tawada, M. Yanagisawa, and N. Togawa, "Speeding-up exact and fast FIFO-based cache configuration simulation," *IEICE Electronics Express*, vol. 8, no. 14, pp. 1161–1167, 2011.

[123] Y.-T. Chen, J. Cong, and G. Reinman, "HC-Sim: A fast and exact L1 cache simulator with scratchpad memory co-simulation support," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, pp. 295 –304, oct. 2011.

[124] W. Zang and A. Gordon-Ross, "T-spacs – a two-level single-pass cache simulation methodology," *Computers, IEEE Transactions on*, vol. 62, pp. 390–403, Feb 2013.

[125] P. Heidelberger and H. S. Stone, "Parallel trace-driven cache simulation by time partitioning," in *Simulation Conference, 1990. Proceedings., Winter*, pp. 734–737, Dec 1990.

[126] W. Han, G. Xiaopeng, W. Zhiqiang, and L. Yi, "Using gpu to accelerate cache simulation," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pp. 565–570, Aug 2009.

[127] J. Ma, H. Wan, X. Gao, and X. Long, "Gpu-based time parallel cache simulator," in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, pp. 407–410, Nov 2010.

[128] A. Smith, "Two methods for the efficient analysis of memory address trace data," *Software Engineering, IEEE Transactions on*, vol. SE-3, pp. 94–101, Jan 1977.

[129] W.-H. Wang and J.-L. Baer, "Efficient trace-driven simulation methods for cache performance analysis," *ACM Trans. Comput. Syst.*, vol. 9, pp. 222–241, Aug. 1991.

[130] Z. Wu and W. Wolf, "Iterative cache simulation of embedded cpus with trace stripping," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, CODES '99, (New York, NY, USA), pp. 95–99, ACM, 1999.

[131] E. Johnson and J. Ha, "Pdats lossless address trace compression for reducing file size and access time," in *Computers and Communications, 1994., IEEE 13th Annual International Phoenix Conference on*, pp. 213–, Apr 1994.

[132] E. Johnson, J. Ha, and M. Baqar Zaidi, "Lossless trace compression," *Computers, IEEE Transactions on*, vol. 50, pp. 158–173, Feb 2001.

[133] Y. Luo and L. John, "Locality-based online trace compression," *Computers, IEEE Transactions on*, vol. 53, pp. 723–731, June 2004.

[134] X. Zhang and R. Gupta, "Whole execution traces," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 105–116, IEEE Computer Society, 2004.

[135] A. Janapsatya, A. Ignjatovic, S. Parameswaran, and J. Henkel, "Instruction trace compression for rapid instruction cache simulation," in *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, (San Jose, CA, USA), pp. 803–808, EDA Consortium, 2007.

[136] A. Milenković and M. Milenković, "An efficient single-pass trace compression technique utilizing instruction streams," *ACM Trans. Model. Comput. Simul.*, vol. 17, Jan. 2007.

[137] J. Schneider and S. Parameswaran, "An extremely compact jpeg encoder for adaptive embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 1063–1064, March 2013.

[138] Independent JPEG Group. http://www.ijg.org/.

[139] Embedded JPEG Codec Library. http://blaatfabriek.no-ip.com/fpga/.

[140] jpec. https://github.com/Moodstocks/jpec/.

[141] Jpegant. http://developer.berlios.de/projects/jpegant/.

[142] jpeg-compressor. http://code.google.com/p/jpeg-compressor/.

[143] Maxim Integrated, *DS18B20 Programmable Resolution 1-Wire Digital Thermometer*.

[144] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," 2003.

[145] H. Amrouch, T. Ebi, J. Schneider, S. Parameswaran, and J. Henkel, "Analyzing the thermal hotspots in fpga-based embedded systems," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–4, Sept 2013.

[146] *Xilinx Synthesis and Simulation Design Guide*.

[147] *DIAS Infrarred Camera, http://www.dias-infrared.com/pdf/pyroview380lcompact_eng.pdf*.

[148] B. Griffith, D. Trler, and H. Goudey *Infrared Thermography. Encyclopedia of Imaging Science and Technology*, 2002.

[149] *LEON3: A Soft-Core Microprocessor*.

[150] *PowerPC 440: A Hard-Core Microprocessor*.

[151] *Collective Benchmark*.

[152] *Mibench Benchmark*.

[153] J. Schneider, J. Peddersen, and S. Parameswaran, "A scorchingly fast FPGA-based Precise L1 LRU cache simulator," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pp. 412–417, Jan 2014.

[154] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[155] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, vol. 33, pp. 28–35, July 2000.

[156] H. Al-zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *in Proc. 42nd ACM Southeast Conference*, pp. 267–272, ACM Press, 2004.

[157] Altera Corporation. http://www.altera.com/, 2014.