Real-time processor reconfiguration: from high-performance to low power consumption, MSc Thesis.

João Pedro Soares Joyce

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Júri
Presidente: Doutor Nuno Cavaco Gomes Horta
Orientador: Doutor Pedro Filipe Zeferino Tomás
Co-orientador: Doutor Nuno Filipe Valentim Roma
Vogais: Doutor José Carlos Alves Pereira Monteiro

Outubro de 2011
Acknowledgments

First of all I would like to thank Professor Pedro Tomás and Professor Nuno Roma for their helpful advice that enabled me to surpass the roadblocks encountered along the way. Since my main research area during the course years was not Computers I know that their decision of assigning me this work involved a certain amount of risk. For that I am really thankful and I hope the risk has paid off.

I would like to thank my mother Paula, my father António, my brother Pedro and his wife Ana for cheering me up along the way and putting up with me in these last stressful days. I know it ain’t easy, believe me, I have to do it every day.

Also, in these last days, Bia has been hearing me talking to myself about tunable caches. If her first words happen to be “Memory-Hierarchy” you will know who to blame.

Special thanks go to Joaninha for all her support and for being present in most of the hours that this work has taken to accomplish. This work is as much yours as it is mine.

Thank you!
Abstract

Previous works have shown that providing the option to tune a cache to the program being executed can have an impact in energy efficiency and performance.

In this work a new cache design is proposed with the ability to adapt to the running program by adjusting block size and associativity. The selection is made through control signals that can be controlled by a special processor instruction or the processor itself running a self-tuning algorithm. On this particular design tuning can be achieved by changing the control signals at any given time of execution while attempting to re-utilize already filled cache lines.

The model was implemented in a DLX superscalar processor for testing purposes and synthesized for an FPGA.

The designed cache was shown to provide tuning while adding a small overhead in terms of area and propagation delay when compared to the equivalent fixed cache.

Keywords

Associativity configuration, Block size configuration, Dynamic and static cache tuning.

This thesis was performed in the scope of the research project “HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis”, with reference PTDC/EEA-ELC/113999/2009.
Resumo

Trabalhos anteriores demonstraram que ajustar as características de uma cache a um determinado programa em execução pode ter ganhos em termos de eficiência energética e performance.

Para este trabalho foi desenvolvida uma cache com a capacidade de ajustar a sua associatividade e tamanho do bloco ao programa em execução. A seleção do tamanho do bloco e associatividade é feita através de sinais de controlo que podem ser controlados por instruções adicionadas ao processador ou por um algoritmo de reconfiguração dinâmica. Esta reconfiguração pode ser alcançada em qualquer momento de execução.

Este modelo de cache foi implementado num processador DLX super-escalar para testes e foi também sintetizado para uma FPGA.

Foi demonstrado que a cache desenvolvida pode ser reconfigurada sem que seja adicionado custo adicional significativo em termos de área e tempos de propagação quando comparada com uma cache fixa de iguais características.

Palavras Chave

Associatividade, reconfiguração de caches dinâmica e estática.
# Contents

1 Introduction  
  1.1 Motivation .................................................. 3  
  1.2 Main contributions ........................................ 4  
  1.3 Dissertation outline ...................................... 5  

2 Background on Memory-Hierarchy  
  2.1 Pipeline ....................................................... 8  
  2.2 Memory Hierarchy ........................................... 9  
  2.3 Cache design techniques  
    2.3.1 Cache size .............................................. 12  
    2.3.2 Block size .............................................. 14  
    2.3.3 Associativity ........................................... 15  
    2.3.4 Block replacing policies ............................... 16  
    2.3.5 Block writing policies ................................. 18  
    2.3.6 Block allocation policies ............................... 18  
    2.3.7 Virtual Memory ......................................... 19  
  2.4 Cache Tuning ................................................ 21  
  2.5 Summary ....................................................... 22  

3 Designing a tunable cache  
  3.1 Scope and Objectives ...................................... 24  
  3.2 Initial considerations and definitions ..................... 25  
  3.3 Reading data from the cache  
    3.3.1 Datapath ................................................ 27  
    3.3.2 Associativity .......................................... 27  
    3.3.3 Associativity Configuration ........................... 28  
    3.3.4 Block Size ............................................. 32  
  3.4 Writing data to the cache  
    3.4.1 Write and Substitution Policy ......................... 32  
    3.4.2 Associativity .......................................... 33
Contents

3.4.3 Block Size .................................................. 34
3.5 Storing Performance Indicators ................................. 35
3.6 Run-time Configuration ......................................... 37
   3.6.1 Static ................................................... 38
   3.6.2 Dynamic ................................................ 38
3.7 Summary ....................................................... 40

4 Cache integration in a CPU architecture ......................... 41
   4.1 The DLX processor ........................................... 42
      4.1.1 Datapath and Tomasulo’s algorithm .................. 44
      4.1.2 Instruction Fetch Unit .................................. 45
      4.1.3 Dispatcher ............................................... 45
      4.1.4 Branch Target Buffer ................................... 46
      4.1.5 ALU ..................................................... 47
      4.1.6 MDU .................................................... 47
      4.1.7 Reorder Buffer .......................................... 48
      4.1.8 Load/Store Unit ........................................ 49
      4.1.9 Write Buffer ............................................ 49
      4.1.10 Cache and Translation Lookaside buffer ........... 50
   4.2 Configurable Cache Integration .............................. 51
   4.3 Summary ..................................................... 53

5 Results ........................................................... 55
   5.1 Testing framework ........................................... 56
      5.1.1 Compiling and Assembling .............................. 56
   5.2 Software Simulation ......................................... 57
   5.3 VHDL Simulation .............................................. 60
      5.3.1 Tuning cache in run-time .............................. 60
   5.4 Synthesis of the VHDL description in an FPGA ............ 62

6 Conclusions ....................................................... 65
   6.1 Future work .................................................. 66

A Program used for testing ....................................... 69
## List of Figures

1.1 Moore's law. ................................................. 2
1.2 Evolution of processor and memory performance. .............. 3
1.3 Miss rate versus associativity of different block sizes. ........ 4
1.4 Miss rate versus associativity of different block sizes. ........ 4

2.1 The DLX processor pipeline. This pipeline is composed of 5 different stages: Instruction Fetch, Instruction Decode, Execution, Memory and Writeback. The pipeline registers are represented as gray boxes between the stages. ........ 8
2.2 Memory hierarchy. ........................................ 9
2.3 Representation of a 8-line directly mapped cache. ............ 11
2.4 Directly mapped instruction cache with 2 lines. ............... 13
2.5 Directly mapped instruction cache with 4 lines. ............... 13
2.6 Directly mapped instruction cache with 8 lines. ............... 14
2.7 Directly mapped cache with blocks. ........................ 15
2.8 Block size. .............................................. 15
2.9 Cache Associativity. ...................................... 16
2.10 A 2-way set-associative cache with victim buffer. .......... 17
2.11 Virtual Memory. ......................................... 19

3.1 Address word field division .................................. 25
3.2 A single cache Way ....................................... 26
3.3 Cache Datapath. .......................................... 27
3.4 Associativity Selector. ..................................... 28
3.5 Directly Mapped Cache .................................... 29
3.6 Directly Mapped Cache datapath. .......................... 29
3.7 Virtual Mapping for the Directly Mapped cache ................ 30
3.8 A 2-way set associative cache. ............................ 30
3.9 The 2-way set associative cache datapath. .................... 30
3.10 Virtual Mapping for the 2-way set associative cache ......... 31
3.11 A 4-way set associative cache. ............................ 31
List of Figures

3.12 The 4-way set associative cache datapath. ........................................... 31
3.13 Virtual Mapping for the 4-way set associative cache. .......................... 32
3.14 Associativity selection. The Tag/Index is represented at the top of each Way. The Index appears in black while the remaining bits appear in grey. ................. 34
3.15 Block substitution in a 2-way set associative configuration .................... 35
3.16 When there is a hit on one cache way, the respective hit count register is incremented. For all the other ways the miss count register is incremented. .......... 36
3.17 When a line leaves the cache the Used and Fetched bits are combined to increment or decrement the block size performance indicator register. ....................... 37
3.18 Block size markov chain for the configurable cache. .............................. 39
4.1 Top level DLX model. ................................................................................. 43
4.2 Branch Target Buffer. ............................................................................. 46
4.3 ALU reservation station. The number of bits used for each field are represented and the total size in bits of the reservation station is calculated ..................... 47
4.4 MDU reservation station. .......................................................................... 48
4.5 Reorder buffer. ......................................................................................... 48
4.6 Load Store Unit reservation station. .......................................................... 49
4.7 Write buffer. ............................................................................................. 50
4.8 Data cache. ............................................................................................... 50
4.9 Address Word for the realized DLX cache .................................................. 52
4.10 C-Type Instruction class .......................................................................... 53
5.1 Miss rate of the matrix operations program according to associativity and block size for a 256 B cache ................................................................. 58
5.2 Miss rate of the matrix operations program according to associativity and block size for a 512 B cache ................................................................. 58
5.3 Miss classification in percentage for a 512 B cache with 32 B blocks. ........... 59
5.4 Miss classification in percentage for a 512 B cache with 64 B blocks. ........... 59
5.5 Miss rate in program execution with a blocksize of 2 lines. Time is presented in ms. The different program operations are identified at the top of the figure: 1) Variable initialization. 2) Matrix multiplication 3) Matrix summation ......................................................... 61
5.6 Miss rate in program execution with a blocksize of 8 lines. Time is presented in ms ................................................................. 61
5.7 Miss rate in program execution with a blocksize changing from the 8 line to 2 line. Time is presented in ms. ................................................................. 62
List of Tables

2.1 Simple program example where the third instruction depends on the correct execution of the first two. ................................................................. 9
2.2 Representation of pipelined execution of three instructions in the DLX processor ................................................................................. 9
2.3 Fibonacci progression program. Consider that the initial conditions are $R_{i} = 10$, $M[R_{m}] = 0$ and $M[R_{m} + 4] = 1$ ........................................................................................................ 12
2.4 Division of the CPU address ......................................................................................................................................................................... 14
3.1 Performance indicator for block size ......................................................................................................................................................... 36
3.2 Changing and reading configuration instructions ........................................................................................................................... 38
3.3 Instructions for reading and resetting performance indicators ............................................................................................................ 38
3.4 Changing tuning interval instructions ..................................................................................................................................................... 39
4.1 Instructions added to the instruction set .................................................................................................................................................. 53
5.1 Execution times in milliseconds for the Superscalar DLX with a 256 B tunable cache using initial conditions .............................................................................. 60
5.2 Execution times in milliseconds for the Superscalar DLX with a 512 B tunable cache using initial conditions .............................................................................. 60
5.3 List of synthesized logic for a 256 B cache with 64 bit lines, 4 associativity ways with 8 lines each ................................................................................. 63
5.4 Size and timing for different 128 B caches .................................................................................................................................................... 63
5.5 Size and timing for different 256 B caches .................................................................................................................................................... 63
5.6 Size and timing for different 512 B caches .................................................................................................................................................... 64
# Introduction

## Contents

1.1 Motivation .................................................. 3
1.2 Main contributions ......................................... 4
1.3 Dissertation outline ......................................... 5
1. Introduction

In computer design there has been nothing more certain than Moore’s Law. This law states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. This trend has resisted 40 years of computer design and it is expected to hold true for some years to come.

Figure 1.1: Moore’s law.

Albeit its consistency, partly because it has been used as a guideline for setting goals in the semiconductor industry, there’s a high degree of consensus that Moore’s Law will reach its end rather sooner than later. These limitations come from the fact that the number of transistors has grown by decreasing its size and eventually the atom size would come as an unsuppressible barrier. However, we should note that the number of transistors does not automatically translate into performance gains and that better design can be used to mitigate some of the problems faced by the end of Moore’s Law.

But designers are facing other problems and barriers to performance increase. The limitations in Instruction Level Parallelism (the number of instructions that can be performed simultaneously) has been a major point of concern in recent years and as rendered the increase in CPU frequency useless because of the penalties involved in the excessive use of pipelined architectures.

This has been amplified by the disparity between the performance of processors and memories, a phenomenon known as the processor-memory gap or memory wall which made the design of better memory-hierarchy one of the major concerns in the recent years.
1.1 Motivation

Figure 1.2: Evolution of processor and memory performance.

With CPUs already hitting the memory wall the focus of computer architecture designers shifted from clock speed to multi-core architectures and parallelization of software.

As a last major concern it should be referred that with the advent of mobile communication there is an increasing demand for mobile devices (laptops, netbooks, mobile phones, tablets...) which require some degree of autonomy from an external power supply. The need for power-efficient processors comes in opposition to the trend of trading power for speed.

1.1 Motivation

Usually a CPU has a cache with fixed total size, block size and associativity. These three options are chosen as a trade-off for some design considerations such as cost, area, power consumption, number of misses, miss time, CPU clock frequency and task being performed.

The relative weight given to different design considerations has been changing over the years. With the penalties involved due to the processor-memory gap there has been a shift in focus from CPU frequency to cache performance. To study the influence of association and block size on cache misses the program DineroIV was used to generate miss rate statistics from a memory trace file. This statistics were plotted for comparison.

A typical plot for miss rate versus association for different block sizes can be seen in figure 1.3.

In this plot there can be seen a decrease in miss rate with the increase of associativity. This is not surprising because with the increase in associativity there are more places in which a specific line of memory data can be stored in cache, reducing the miss rate. This could be amplified by the effect of temporal locality. Another visible phenomenon is that an increase in block size has the effect of reducing the miss rate (a consequence of prefetching data and spatial locality). Although this is often true there are cases where there’s benefits in lower associativity and lower block size.
1. Introduction

In the case shown in figure 1.4 block sizes of 128 bytes and 64 bytes clearly have greater miss rates than smaller blocks. And it is also visible an increase in miss rate with the increase of associativity for 32 and 64 bytes blocks. These findings serve as starting point for a discussion on real-time cache reconfiguration. Caches capable of changing their behavior according to different factors (operator desire, source code, battery level) to meet different goals (namely high-performance and low power consumption).

1.2 Main contributions

In this work a new cache design is proposed with the ability to adapt to the running program by adjusting block size and associativity. The selection is made through control signals that can be controlled by a special processor instruction or the processor itself running a self-tuning algorithm.

A VHDL model was designed so that a simulation could be implemented in an already existent
model of a processor and, furthermore, tested in an FPGA board. The VHDL was carefully designed in order to be easily adjustable, by mean of a set of variables, to the maximum size, maximum associativity and maximum block desired.

The model was implemented in a DLX superscalar processor. A DLX simulator was altered in order to produce memory traces that could be evaluated in a cache simulator. This simulations served as a basis for the understanding of the trade-off between different cache configurations.

1.3 Dissertation outline

The remaining of this dissertation is organized as follows: First the fundamental concepts behind cache design in a chapter called Background are reviewed. These concepts will be presented with clear and simple examples to illustrate the trade-offs involved in computer architecture projects and will serve as the foundation for the understanding of the design decisions made. An introduction to cache tuning will follow with reference to the work made by researchers in this particular field. Following, in chapter 3 the cache model is presented in a chapter called Designing a reconfigurable cache containing a thorough discussion of the decisions and trade-offs involved followed by the self-tuning algorithm used to control the configuration of the cache. The DLX processor used as the test environment for the cache is described in chapter 4, with focus on the adaptations made to fit the new cache. The differences between the new cache and the pre-existent one are contrasted.

The test framework is then presented with a short discussion of the compiling and assembling process and the Results of benchmarking presented leading to the last chapter where Conclusions are drawn.
1. Introduction
2

Background on Memory-Hierarchy

Contents

2.1 Pipeline .................................................. 8
2.2 Memory Hierarchy ......................................... 9
2.3 Cache design techniques ................................. 11
2.4 Cache Tuning ............................................... 21
2.5 Summary ................................................... 22
2. Background on Memory-Hierarchy

The **Central Processing Unit** (CPU) is the main element of a computer system. It executes a program by reading instructions and data from memory (or other storage device), executes the instruction and stores the result back to memory.

The group of instructions known to a processor is called the **Instruction Set**. Most of these instructions are very simple operations (*e.g.* data transfer, arithmetic and logic operations) but performed synchronously at very high speeds. The instructions are represented internally in the form of an **instruction word**, a fixed size group of bits. The bits in an instruction word are divided in fields, each field with its own meaning to the processor.

### 2.1 Pipeline

Most CPUs divide their operation into a number of simple stages each one taking a certain number of clock cycles to perform, a technique known as **pipeline**.

![Figure 2.1: The DLX processor pipeline. This pipeline is composed of 5 different stages: Instruction Fetch, Instruction Decode, Execution, Memory and Writeback. The pipeline registers are represented as gray boxes between the stages.](image)

The DLX pipeline shown in figure [2.1] has five stages: the **fetch stage**, where the instruction word is read from memory, the **decode stage**, where the instruction is decoded and the values needed are read from registers, the **execute stage** to perform the operation, the memory stage to read/write values from memory and a **writeback stage** to store the results. These stages are separated by registers, to store the intermediate level data needed by the subsequent stages.

The advantage of a pipelined architecture comes from having multiple instructions performing simultaneously in the CPU but in different stages; it trades resources and the total execution time of one single instruction for a higher clock frequency and greater throughput by taking advantage
of Instruction Level Parallelism - the number of instructions that can be performed simultaneously by a CPU.

One of the limitations of this technique is in the nature and order of the instructions performed by the CPU. For instance, in the sequence present in table 2.1, the third instruction depends on the correct execution of the first and second instruction creating a data dependency between them.

\[
\begin{align*}
\text{addi} & \quad r2, \quad r0, \quad 0x7777 \quad (r2 \leftarrow r0 + 0x7777) \\
\text{addi} & \quad r3, \quad r0, \quad 0x1111 \quad (r3 \leftarrow r0 + 0x1111) \\
\text{sub} & \quad r4, \quad r3, \quad r2 \quad (r4 \leftarrow r3 - r2)
\end{align*}
\]

Table 2.1: Simple program example where the third instruction depends on the correct execution of the first two.

In the pipelined presented in figure 2.1, neither of those has reached the write-back stage when the third instruction gets to the Instruction Decode stage, as can be seen in table 2.1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi ( r2, r0, 0x7777 )</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>addi ( r3, r0, 0x1111 )</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>sub ( r4, r3, r2 )</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Representation of pipelined execution of three instructions in the DLX processor

These problems can be reduced by using forwarding techniques or register renaming. [Hennessy and Patterson, 2003]

## 2.2 Memory Hierarchy

Making large amounts of fast memory is technologically difficult and economically not viable. It thus makes the slower memories a bottleneck to CPU performance has seen in figure 1.2. In practice, to solve this problem, computer systems use a memory sub-system which is a hierarchy of storage devices with different cost, capacity and access times. The smaller and faster memories are closer to the CPU and the bigger and slower are further from the CPU.

![Figure 2.2: Memory hierarchy.](image-url)
2. Background on Memory-Hierarchy

The idea for memory-hierarchy comes from a phenomenon called locality, which states that computer programs do not access all code or data uniformly \cite{Hennessy2003}. There are two types of locality:

**Spatial locality** if a certain memory location is referenced than it is likely that nearby locations will also be referenced.

**Temporal locality** if a certain memory location is referenced than it is likely for it to be referenced again in the near future.

These locality concepts are the foundation for the existence of cache. Cache is a term widely used at many levels of Computer Systems to designate the temporary storage of commonly occurring items, so as to reuse them in the near future thereby increasing performance.

An example of cache known to most computer users is the caching made in Internet browsers to avoid retransmission of web pages recently viewed. This has advantages for the Client, Server and overall Internet by lowering the number of requests, the delay between request and response and the number of bytes travelling through all infrastructure by taking advantage of temporal locality.

Examples of spatial locality can also be found in browsers for instance when dealing with media contents. If a video is being watched by a computer user, usually the browser pre-fetches data from the remaining of the video since it is likely to be accessed in the near future.

In the context of Computer Architecture a cache is a type of fast and small memory residing close to the CPU, which should store the data most likely needed in the future so as to avoid the longer access times from main memory (or other types of memory down the hierarchy).

As seen by the CPU, this operation should be transparent, meaning that the CPU should function independently of cache structure or even existence.

Usually, a CPU has 2 distinct caches: One for instructions (the program) and another for data (Loads and Stores). As the cache dimension is by definition small, care must be taken in order to decide what data should reside in cache at any given time. Furthermore, because of the proximity between cache and CPU these, decisions should be easy to implement in hardware to avoid stalling the CPU.

Some of the problems that need to be balanced in order to accomplish a good cache design are:

- The best way to map main memory addressing space to cache.
- How many data should be fetched to cache every time the main memory is accessed.
- How is the replacement of data handled.
- What happens on a write.
To further understand these problems a general description of cache architecture techniques follows.

2.3 Cache design techniques

As caches are, by definition, smaller than main memory only a fraction of the total memory content can be cached. Therefore, caches need to store not only the data but the addresses where that data reside. In this sense, whenever data is read from memory, the following steps will occur. First, the processor generates the address it wants to read. Following, the address is searched within the cache to check if data is present. If the cache controller finds the address in cache it immediately retrieves the data. If it does not find the data in the cache, the controller then requests the data from the main memory.

Each address correspond to a minimum amount of data that the CPU can access. Usually, each memory address define a Byte of data and the memory hierarchy is said to be byte-addressable.

The simplest cache architecture is called a **directly mapped cache**. It is composed of two sets of registers one to store memory addresses and another for the data content corresponding to that specific addresses. In this case, any given memory address can occupy only one specific cache line. This line is chosen by the **Index**, a group of bits from the address. To the remaining Least Significant Bits (LSB) of the memory address, the name **Offset** is given. This field is used to access individual bytes (for a byte-addressable memory) from each cache line. To the remaining bits of the address, the name **Tag** is given.

![Figure 2.3: Representation of a 8-line directly mapped cache.](image)
According to this structure, the Index field unambiguously define only one specific line in the cache. Therefore, there is no need to store the Index bits in the cache for address comparison. Only the Tag must be stored, as can be seen in figure 2.3.

When the CPU asks for a given address, the cache line pointed by the index field is checked first by comparing the cache’s Tag field to the Tag provided by the CPU. If a match exists (cache hit) the data from cache is directly sent to the CPU avoiding longer access times. If there is no match (cache miss) the data must be fetched from the next memory device down in the hierarchy, usually a second level of cache.

In addition to storing addresses and data, the cache also needs to store a validation bit for each block. If this bit is unset, a cache hit can never occur.

To analyse the influence of caches in CPU performance, the following simple program will be used as an example. This program finds the first 12 Fibonacci numbers \(^1\) and stores them as 32 bit integers consecutively in memory starting at the value stored in \(R_m\). To simplify the code analysis it is assumed that the first two numbers (0 and 1) are already in memory \((M[R_m] = 0\) and \(M[R_m + 4] = 1\)) and a register \(R_l = 10\) is considered. Each instruction word has 32 bits with a byte addressable memory.

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Code</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000</td>
<td>Loop: LW R1, 0(R_m)</td>
<td>(R1 \leftarrow M[R_m + 0])</td>
</tr>
<tr>
<td>0000000100</td>
<td>LW R2, 4(R_m)</td>
<td>(R2 \leftarrow M[R_m + 4])</td>
</tr>
<tr>
<td>0000010000</td>
<td>ADD R3, R1, R2</td>
<td>(R3 \leftarrow R1 + R2)</td>
</tr>
<tr>
<td>0000011000</td>
<td>ADDI R_m, R_m, #4</td>
<td>(R_m \leftarrow R_m + #4)</td>
</tr>
<tr>
<td>0000100000</td>
<td>SW R3, 4(R_m)</td>
<td>(M[R_m + 4] \leftarrow R3)</td>
</tr>
<tr>
<td>0000101000</td>
<td>SUBI R_l, R_l, #1</td>
<td>(R_l \leftarrow R_l - #1)</td>
</tr>
<tr>
<td>0000110000</td>
<td>BNEZ R_l, Loop</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Fibonacci progression program. Consider that the initial conditions are \(R_l = 10\), \(M[R_m] = 0\) and \(M[R_m + 4] = 1\)

To analyse the impact of cache size on performance, it will be considered that every cache-miss has a penalty of 5 cycles, while a cache-hit takes just one cycle. These values will be used in the following examples.

### 2.3.1 Cache size

An instruction cache with only two lines of 4 bytes each will be first considered. In this cache, the 10 bit address represented in figure 2.3 is divided in two offset bits to address the individual bytes in each 32 bit instruction word (32 bit = 4 bytes = \(2^2\) bytes), one index bit (2 cache lines = \(2^1\) cache lines) and seven tag bits. The mapping of the memory addresses to memory cache lines is represented in figure 2.4. In this case it is easy to notice that there will be no cache hits.

---

\(^1\)The Fibonacci numbers are integer numbers starting with 0 and 1 where every subsequent number is the sum of the previous two. The first 12 Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 and 89
2.3 Cache design techniques

Figure 2.4: Directly mapped instruction cache with 2 lines.

To analyse the total time for these memory accesses the average memory access time expression is presented where $p_H$ and $p_M$ are, respectively, hit and miss rate, $t_H$ and $t_M$ are hit and miss time while $t_P$ equals to miss penalty:

$$
\begin{align*}
    t &= p_H \cdot t_H + p_M \cdot t_M \\
    t_M &= t_H + t_P
\end{align*}
$$

The total number of cycles required for these memory accesses is:

$$
    t = \text{InstructionCount} \cdot (t_H + p_M \cdot t_P) = 70 \cdot (1 + 6/7 \cdot 5) = 420
$$

Considering a larger cache supporting 4 lines of code, the index field will have 2 bits ($2^2 = 4$) and the tag field will have the remaining six bits. With this cache there will be one hit for every loop iteration excluding the first pass. The instruction that will hit in every iteration is the one with the address 0000001100 (ADDI $Rm$, $Rm$, #4). This happens because no other instruction has an address with the same index bits (11), so there will be no collision and this instruction will never be replaced.

Figure 2.5: Directly mapped instruction cache with 4 lines.

In this case, the total number of cycles is:

$$
    t = \text{InstructionCount} \cdot (t_H + p_M \cdot t_P) = 70 \cdot (1 + 6/7 \cdot 5) = 370
$$

In a directly mapped cache with 8 lines it is possible to see that all the code fits in the cache and, therefore, the only misses will be in the first pass. Then, every subsequent pass will lead to a cache hit. The index field for a directly mapped cache with 8 lines is composed of 3 bits ($2^3 = 8$).
2. Background on Memory-Hierarchy

The total number of cycles is:

\[ t = \text{InstructionCount} \cdot \left( t_H + p_M \cdot t_p \right) = 70 \cdot \left( 1 + 1/10 \cdot 5 \right) = 105 \]

With this simple example the advantages of larger caches are evident.

2.3.2 Block size

To take advantage of spatial locality when the cache fetches data from the lower hierarchy levels, it usually takes not only the required word but a block of adjacent words.

This fact divides the address coming from the CPU in two parts: The Block Offset and the Block Address. The block offset is composed of \( n \) Least Significant Bits (LSB) from the memory address, which define \( 2^n \) words in a block. The remaining bits in the address form the block address, which is further divided in the Tag and Index fields.

This has the positive side effect of reducing the number of bits the cache needs to compare in order to find out if there is a cache hit. When the tag field of the Block Address is stored in cache the CPU knows that the whole block is present and that it only needs to use the Offset bits to find out the relevant data for execution. This is shown in figure 2.7.

Building up from the example in figure 2.3, it will be considered a cache with blocks of four 32 bit words. In the previous example it was shown that this cache took 370 cycles to complete.

Considering that the cache takes the same 5 cycles to fetch the first word from the lower level memory, but the subsequent words take only 1 cycle each, the total number of cycles needed to complete this loop iteration is:

\[ t = \#\text{Iterations} \cdot \left[ t_{\text{firstblock}} + t_{\text{secondblock}} \right] = 10 \cdot \left[ (5 + 1 + 1 + 1) + (5 + 1 + 1) \right] = 150 \]
Another interesting fact is that the address 0000011100 is now loaded from memory although it does not have an active part on the code. This is a consequence of fetching the whole block when there is a cache miss, which can lead to inefficiency problems when large blocks are fetched from memory just to use a small fraction of the block content, with serious impact to performance.

Figure 2.8: Block size.

2.3.3 Associativity

An \textbf{m-way set associative} cache defines a cache where any particular memory tag can be stored in one of \textit{m} possible sub-caches. A cache where there is only one place for each memory tag is a \textbf{direct-mapped} cache. A \textbf{fully-associative} cache is the case where any memory tag can occupy any cache block. Fully-associative and direct-mapped are only particular cases of set-associative caches, the first one being one-way set associativity and the other one, for a cache with \textit{n} blocks, an \textit{n}-way set associative cache.

The number of sets in a cache can be calculated by dividing the cache size (in blocks) by its set associativity. The number of existing sets define the number of bits needed to index each
2. Background on Memory-Hierarchy

Figure 2.9: Cache Associativity.

set. These bits are taken from the Least Significant Bits of the block-address, as previously seen. More generally the following equation is used to find out how many bits are needed in the index field:

$$\text{index} = \lceil \log_2 \left\{ \frac{\text{Cachesize}}{\text{BlockSize} \times \text{Setassociativity}} \right\} \rceil \quad (2.1)$$

Searching the cache for a hit is usually made in parallel through the several sets to reduce the time taken. This means that the cache controller needs as many comparators as the number of searched associativity ways for each request, placing the constrains of increasing cache associativity in silicon area (and, consequently, cost) instead of time.

2.3.4 Block replacing policies

When there is a cache miss, the cache controller must find out which block to dispose, in order to make room for new data.

In a direct-mapped cache this decision is simple because there is only one slot where the block to be fetched from main memory can be placed. However, this is not true for set-associative caches. As a consequence, a replacement algorithm is needed in order to make this decision. The used replacement algorithm should take in consideration the obtained gains in terms of cache hits versus cache misses and the cost of implementing such solution in hardware.
For instance, according to temporal locality, a good choice would be to dispose the \textbf{Least Recently Used} (LRU) block. This is usually done by storing \textit{age bits} associated with each block. Every time a particular block is accessed, the age bits of all other blocks are incremented. Although this is generally regarded as a good replacement algorithm, the involved hardware costs can be a barrier to its implementation.

Other common option would be the \textbf{Least-Frequently Used} algorithm (LFU), where each block has a counter that is incremented whenever that particular block is accessed. Then, when there is a miss, the block with the smaller count is the one replaced and the counters are reset.

There are also some \textit{naive} replacement approaches that can be good options in what concerns reducing involved costs. For instance, a \textbf{queue} algorithm where the Least Recently Fetched block is replaced can be very efficient, due to its simple design, requiring only the routing of blocks around the set domain.

To reduce some of the consequences of conflicts, some caches also use a \textbf{victim buffer}. This is a tiny fully associative cache, faster then lower-level memory, implemented just like a queue, where the blocks that were replaced from cache are stored before they are removed (Figure 2.10). Then, if a request is made for that same block again a miss occurs but the block is fetched from the faster victim buffer instead of lower-level memory.

![Figure 2.10: A 2-way set-associative cache with victim buffer.](image-url)
2. Background on Memory-Hierarchy

2.3.5 Block writing policies

In the case of a write operation, important decisions have to be made. There are basically two options for the write policy: Write Through and Write Back. In a Write Through cache the information is directly written to lower-level memory. In a Write Back cache the information is written only in the cache. The modified block is written in memory when that particular block is replaced. In this case an extra bit is stored to indicate if the data in cache needs to be written in memory when it is disposed. This bit is usually called the dirty bit.

The advantages of a Write Back cache is the fact that writes occur at the speed of cache writes. There is also a reduction in bandwidth, between cache and lower-level memory, since not every write goes to lower-level memory.

In the case of a Write Through cache the advantages are the simplicity of implementation and the fact that read misses never result in writes to memory\(^2\) which is not the case in a Write Back policy.

The coherency between cache and lower-level memory is also a very important feature for I/O and multiprocessors. For instance using multiprocessors to analyse the same data, if a processor has modified fetched data to its Write Back cache the other one can only get coherency by getting data directly from that cache or by waiting for the writing of that cache block to a coherent shared memory. This will guarantee that the data read and written will be the most recent data but this technique can lead to data stalls in CPU. The obvious disadvantage in a Write Through policy is that the CPU must wait for writes to slower lower-level memory, a fact known as write stall. These consequences can be reduced (but not eliminated) by having a write buffer, allowing the processor to continue to operate while data is being written to memory. However, even then the CPU may stall when the buffer is full.

2.3.6 Block allocation policies

There are also two common options to solve a write miss upon a write operation (when the CPU needs to write data to memory but the data is not currently cached) known as Write Allocate and No-Write Allocate. In the Write Allocate strategy, the data is allocated to cache on a write miss. In a No-Write Allocate strategy, the block is directly modified in the lower-level and not allocated in cache. Although both strategies could be used for the two Write policies (Write Through and Write Back) presented earlier, generally the Write Through caches use a No-Write Allocate and Write Back caches use a Write Allocate policy.

It is important to notice that read operations dominate the memory access \cite{Hennessy2003}. Usually more than 75% of memory traffic are reads. This fact can attenuate some of the discussed problems faced by writing to memory in cached architectures.

\(^2\)In a Write Back configuration a read miss may cause a dirty block to be replaced from cache. In that case a, read miss also leads to a lower memory write.
2.3 Cache design techniques

2.3.7 Virtual Memory

Virtual memory is a technique that allows for better memory management in multitasking systems. This is accomplished by giving a virtual address space for each computer process, to execute as though there is only one memory device and the process has, by default, the sole access to that device. The virtual address space of each individual process is then divided in parts and each part is allocated by the Operating System to physical memory space.

Virtual Memory also defines an abstraction layer that frees the programmers from the burden of knowing the physical memory bank, as well as its implementation on every computer system.

![Virtual Memory](image)

Figure 2.11: Virtual Memory.

In spite of virtual memory replacement being managed by the Operating Systems, the CPU has to support address translation from virtual to physical addresses and the implementation of this address translation scheme can influence the performance of the CPU itself. Also, since the CPU only works with virtual addresses, the size of virtual memory is limited by the width of the CPU address bus.

Virtual memory systems can be categorized by the way the memory addresses are divided. Fixed-size blocks are called pages, whereas variable size blocks are known as segments. Paging is the type of virtual memory that most CPUs use, mainly because of its simplicity when replacing a page, since all pages are of the same size. In the segmentation case, the memory controller would have to check the physical memory for the existence of a contiguous space large enough to fit that segment.

Another advantage of virtual memory is that it simplifies loading programs for execution with a technique called relocation. This means that individual pages of virtual memory can be placed anywhere in physical memory or disk, by just changing the mapping between them. In the case of
2. Background on Memory-Hierarchy

paging systems this map between virtual and physical addresses takes the form of a page table.

Since the page table usually require a significant amount of storage space, they are usually paged too, which means that an extra memory access would have to be made to translate the virtual address to a physical address. A common hardware technique to accelerate this translation is a dedicated cache called Translation Look-aside Buffer (TLB).

A TLB can be seen as a small cache with a certain number of slots that map between virtual and physical addresses. When the CPU asks for an address this table is searched. Only in case of a TLB miss the CPU needs to search the page tables in a time expensive process known as page walk.

With the advent of virtual memory a decision has to be made about the type of memory addresses (physical or virtual) used to address the CPU cache. This defines three different cache types.

**Physically indexed, physically tagged (PIPT)** In this case the virtual address needs to be translated before searching the cache. This can be a slow process in case of a TLB miss.

**Virtually indexed, virtually tagged (VIVT)** By avoiding the address translation this results in very fast lookups. There are, however, some coherency problems in this scheme because, since every program uses its own address space, the same virtual address can be assigned to different physical addresses (by two different processes). To solve this problem the CPU would have to flush all cache in a context switch or use Process Identification (PID) numbers. This has the advantage of, instead of flushing all data from cache, eventually reusing some of its data previously stored even if different processes are scheduled in between, by checking the PID for every cache access.

**Virtually indexed, physically tagged (VIPT)** This is a middle-point between the other two options. As this cache is physically tagged, it does not have coherency problems like in the VIVT case. Being virtually indexed means that the cache block can be searched in parallel with the Translation Lookaside Buffer therefore having less latency than PIPT caches.

The hardware options for managing virtual memory are greatly simplified by the significant delays involved in accessing lower level storage (magnetic disks can take millions of clock cycles to access). Therefore, any program page can reside in any block of physical memory and no indexing (or set associativity) is used. Also, contrary to what happens with caches, the used replacement algorithm is usually LRU, trading the simplicity of implementation for the reduction in page faults. Upon a write, a Write Back algorithm is always used, to avoid the long delays involved in lower level memory accesses.
2.4 Cache Tuning

With the increasing disparity between processor and memory speeds, there has been a research effort for the introduction of new techniques for better cache allocation and management.

This research effort has gained further momentum with the tendency for designing dedicated cores, whether stand alone or part of heterogeneous multi-core architectures, in opposition to the general-purpose trend that was seen in the last decades. Furthermore, the recent boom of mobile technologies has been a drive force for the research of energy-efficiency technologies, in the quest for improvements in the autonomy of battery powered devices. In this particular field, studies have shown that the memory hierarchy of a system can consume up to 50% of microprocessor system power [Malik et al., 2000].

To mitigate some of these problems several cache-tuning techniques have been proposed.

Cache tuning is defined as a strategy for adjusting the internal cache characteristics, to achieve particular energy efficiency or performance goals. There are two ways to accomplish this: Statically or dynamically.

In the static approach the CPU Instruction Set provides specific instructions for changing the cache configurations. This gives the programmer, compiler or Operating System the needed tools to make hardware optimization decisions in the software context. While avoiding further hardware complications, the future of static cache tuning applications can be limited by the general trend of designing generic higher-level software and by the problems faced in the implementation of rather complex compiler optimization.

On the other hand, dynamic cache tuning relies on transparent run-time techniques to analyse the code being executed along a determined tuning interval. At the end of the tuning interval, the result of this analysis serves has input for a control system responsible for changing the cache configuration. Some self-tuning caches have been presented that can even vary the tuning interval, resulting in an average energy reduction of as much as 29% [Gordon-Ross and Vahid, 2007].

Both of these approaches have to balance the tuning process overhead against the energy overhead of running in sub optimal cache configurations [Gordon-Ross and Vahid, 2007]. To this regard [Bournoutian and Orailoglu, 2010] noted that the sacrifices in speed for a configurable cache were non-linear related to the amount of power saved, allowing large gains in power savings without significantly disrupting the execution performance. In that study [Bournoutian and Orailoglu, 2010] the authors attributed the problem of excessive static power being consumed to the use of large amounts of L2 and L3 caches that have a sporadic number of memory accesses. This fact was the basis to the approach of temporarily shutting down those idle cache sets.

Although cache tuning can have dramatic effects on energy-efficiency and overall cache performance, studies made with highly configurable caches have shown that this dramatic effects
2. Background on Memory-Hierarchy

are concentrated in a limited number of configurations [Viana et al., 2006], which may indicate that decreasing cache miss rates by increasing complexity of caches can only be beneficial until a certain point, where the delays involved by adding extra hardware for configuration and tuning heuristics start to bottleneck the performance gains [Viana et al., 2006].

2.5 Summary

In this chapter, the basic concepts of cache design were described and the research topic of run-time cache tuning was introduced. In the next chapter the design of a configurable cache with run-time configuration is proposed.
Designing a tunable cache

Contents

3.1 Scope and Objectives .......................................................... 24
3.2 Initial considerations and definitions ................................... 25
3.3 Reading data from the cache .................................................. 26
3.4 Writing data to the cache ...................................................... 32
3.5 Storing Performance Indicators ............................................. 35
3.6 Run-time Configuration ....................................................... 37
3.7 Summary .............................................................................. 40
3. Designing a tunable cache

In this chapter, the proposed cache architecture is described in a top-down approach, starting by an explanation of the scope and objectives, followed by an initial view of the design. Afterwards, a more thoroughly description of all the elements forming the cache is presented, focusing on the important design decisions and mentioning their advantages and drawbacks. Finally the set of configuration possibilities are discussed, considering the static or dynamic runtime features.

The following discussion will focus on the hardware design of the cache and not on the actual VHDL description.

3.1 Scope and Objectives

As reviewed in chapter 2, the most important characteristics in cache design are its size, associativity and block size.

In general, increasing the block size reduces the miss rate by taking advantage of spatial locality. However, when considering a fixed cache capacity, increasing the block size reduces the number of disjunct blocks that can fit in cache and may in fact reduce the cache hit rate by increasing conflict misses. Furthermore, the increase in block size may increase the miss penalty associated with fetching large amounts of memory.

Increasing associativity has also the general effect of reducing miss rate by reducing conflict misses. However, that too can depend on the program executing pattern on that moment. Another important thing to notice is that the effect of increasing associativity has generally a lesser impact for each Set-Associativity increase [Hennessy and Patterson 2003], but with a greater impact in energy consumption and, consequently, on the autonomy of mobile devices [Zhang, 2006; Benitez et al., 2010].

For these reasons, the miss penalty, miss rate and energy consumption should be taken in consideration when setting cache configurations and the dependence of the above characteristics on the executed program or desired user goals (e.g. extended battery life) means that implementing a tunable cache can lead to benefits that might outweigh the costs involved.

For these reasons, on this thesis it is presented the design of a cache where the associativity and block size can be independently configured between pre-established minimum and maximum values.

Furthermore, this cache should be capable of gathering information of some cache performance indicators, to provide some insight for the adequacy of the current configuration to the current code. This information can then be used in one of the following ways:

1. Passed by means of predefined instructions, in order for the programmer to change the hardware configurations during program execution.

2. Used as input for a control system that is capable of identifying changes that should be made to the cache’s configuration, in order to select in run-time the associativity and block size.
size that presumably reflects the best decision for the running code.

The scope of the current work also comprises the control circuit to accomplish this last enumerated item. In this particular field, there have been different approaches. [Zhang et al., 2004] studied the general energy impact of each cache configuration and transition and designed an algorithm to, through active exploration, try to find new configurations that could reduce the energy consumption. This approach was done using a fixed tuning interval and [Gordon-Ross and Vahid, 2007] extended the concept by designing a feedback control system capable of controlling this interval in run-time. To try to reduce the energy used in unnecessary configuration changes [Peng and Wang, 2006] designed a way to find phase changes in code. Decision trees and machine learning algorithms have also been proposed to solve this problem as in [K. Sundararajan, 2011].

Being an active research topic, the circuit designed in the scope of this thesis should not be seen as the ultimate solution for this cache configuration algorithm and further study of the cache design could lead to other equally valid solutions.

3.2 Initial considerations and definitions

The main focus of this work was to design a cache where associativity and block size could be adjusted in run-time. Before the in-depth discussion about the cache elements, and to fully understand this chapter contents, some new concepts have to be defined. These new concepts are built on top of the pre-existent cache models, described in chapter 2 and, therefore, a good understanding on the traditional cache design philosophy and nomenclature are required.

First, the Address Word coming from the CPU is divided into different fields that closely resemble the division made in a traditional (i.e. non-tunable) cache. The new address word division has five fields: Offset, Line Offset, Fixed Index, Tag/Index and Fixed Tag.

![Figure 3.1: Address word field division](image)

Starting from the least significant bits, the Offset field is formed analogous to the Offset of any usual cache and, therefore, used to find the minimum addressable number of bits from the contained data. For instance, if the cache is byte-addressable, each address word can access a single byte of memory. The size of the Offset\( (wo)\) defines the number of bytes (for a byte-addressable CPU) that are contained in a single CPU Word, as \(2^{wo} \) bytes.

The size of the Line Offset field is \(lo\) bits. This field defines the size of a cache line, as \(2^{lo}\) CPU words. A line corresponds to the minimum amount of data that can be fetched for each memory request.
3. Designing a tunable cache

Then the Fixed Index is defined. This field has \( fi \) bits that directly index each cache line of each cache way as in the case of a non-configurable cache. Consequently, the size of the Fixed Index field defines the number of lines present on each cache way, as \( 2^{fi} \) lines.

As the name implies the Tag/Index bits can be used either as tag or index bits. The size of this field in bits \( (ti) \) essentially defines the maximum associativity, as \( 2^{ti} \)-way set associative, with the minimum associativity size being the case of a directly mapped cache. The way the selection of the right bits from the Tag/Index field affect associativity will be described in the following sections with the design of the cache's datapath.

To the remaining bits, the name Fixed Tag is given. These bits are used to find the existence of a cache hit by comparing the stored tags to the CPU address tag. Since the Tag/Index field can be used for indexing or comparison purposes, the cache must always store both; Fixed Tag and Tag/Index. To this merged field the name MaxTag is given.

![Figure 3.2: A single cache Way](image)

The next sections define how these fields were mapped to create a configurable cache.

3.3 Reading data from the cache

To accomplish the design of a cache that can be configured according to selected Block Size and Associativity, together with the characteristics previously defined, the answer to the two following enumerated problems have to be considered:

1. How to read data from cache?
2. How to write data to cache?

The strategies that were applied for solving the first of these two problems are presented with an overview of the cache's datapath, followed by an explanation of how the correct words are routed to the CPU, according to the selected associativity and block size.
3.3 Reading data from the cache

3.3.1 Datapath

The datapath of the proposed design was based on a multi-way cache with $2^k$ ways. The internal structure for each Way is directly mapped and hardwired to the Fixed Index field, with the capacity to store $2^{f_i}$ cache lines, as can be seen in figure 3.3.

Every time an address enters the cache, the Fixed Index bits select a single line from each of the directly-mapped ways. These lines are then routed through a multiplexer. At the end of the multiplexer a single line is selected from the cache data structure. This line passes through the Line Offset multiplexer and from there to the CPU.

This multiplexing structure at the output of the cache ways enables the selection of the data from the desired cache way. This means that by controlling the selection signals of these multiplexers, it is possible to select the line that should be used, according to the desired associativity. A circuit called the Associativity Select was added for this purpose.

![Figure 3.3: Cache Datapath.](image)

3.3.2 Associativity

The selection of the desired associativity level is made through a control circuit called the Associativity Select. This circuit receives as input the desired address MaxTag and the MaxTag
3. Designing a tunable cache

value that is read at each of the directly-mapped ways, as well as the the Associativity Selector control signals. The produced outputs are the controlling signals for the multiplexer structure.

Inside this module there are as many comparators as the number of ways present in cache. Each of the MaxTags coming from the ways are compared to the address MaxTag field. The results are fed to a priority encoder. This encoder always encode the leftmost setted bit coming from the comparators. As it will be seen in the following sections this fact, accompanied by the selected write policy grants the possibility of changing associativity settings without the need to flush all blocks from cache.

The results from the priority encoder and the Tag/Index are multiplexed according to the Associativity Selector bits, enabling the selection of the desired associativity.

The result of the selected associativity is passed to the datapath multiplexers to select a single line from the multi-way structure. In figure 3.4, an Associativity Select block for a 4-way config-

![Figure 3.4: Associativity Selector.](image)

urable cache was drawn. There are the four comparators (one for each way, as previously stated) and the priority encoder. According to the Associativity Select signals, the two output bits that go to the datapath multiplexers can come all from the Tag/Index (the cache is directly mapped), all from the priority encoder (the cache is $2^v$-way set associative), or the least significant bit(s) coming from the Tag/Index and the most significant bit(s) coming from the priority encoder.

3.3.3 Associativity Configuration

To further understand how the Tag/Index bits are combined with the Associativity Selector control signal to provide the desired associativity for the cache an example is now presented.

This example uses two bits for the Variable Index field meaning that it can provide associativity configurations between a directly mapped cache and a 4-way set associative cache. This example will serve as the basis for the discussion and illustration of all the used techniques.
When all the bits in the Tag/Index field are used as Index the resulting cache is a Directly Mapped cache.

This is done by routing the Tag/Index signals through the Associativity Selector block to the output multiplexer structure as it can be seen in figure 3.6. As a result of this routing scheme, for each address only one of the four cache ways can be selected, and the selected way depends exclusively on the Tag/Index bits. In this way, a virtual directly mapped cache is formed.

The virtual mapping of the physical cache ways is drawn in figure 3.7. The bits from the Tag/Index field are written alongside the corresponding letter representing the cache ways from figure 3.6. The reason for not mapping each cache way to the corresponding Tag/Index bits in numeric order as to do with the way the sets are organized for higher-way set associative configurations.

By reducing the number of bits in the Tag/Index field, greater Set-Associative caches are formed. The bits from the Tag/Index field that are not used for the Index (i) are appended to the Fixed Tag field. With this configuration a $2^{v-n-1}$-way set associative cache is formed. The address
3. Designing a tunable cache

Tag/Index

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>C</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Figure 3.7: Virtual Mapping for the Directly Mapped cache

The word field division for this case can be seen in figure 3.8.

The way the bits are mapped define that the first set is composed of the A and B ways, while the second is composed of the C and D.

In this particular case, the least significant bit is directly routed to the multiplexer structure, while the other goes to the comparator and encoder.

Figure 3.8: A 2-way set associative cache.

Figure 3.9: The 2-way set associative cache datapath.
3.3 Reading data from the cache

The virtual mapping of the physical cache ways is represented in figure 3.10. In this case, the Tag/Index least significant bit is the one defining the Set where the data will be placed.

![Virtual Mapping for the 2-way set associative cache](image1)

Figure 3.10: Virtual Mapping for the 2-way set associative cache

The last example corresponds to the case where all the bits in the Tag/Index field are used as Tag. In this particular case, the cache is 4-way set associative.

![A 4-way set associative cache](image2)

Figure 3.11: A 4-way set associative cache.

In this case, both of the Tag/Index bits are compared and encoded to find a hit. The result is then routed to the output multiplexer structure.

![The 4-way set associative cache datapath](image3)

Figure 3.12: The 4-way set associative cache datapath.

The virtual mapping of the physical cache ways is represented in figure 3.13. Here, no bit
3. Designing a tunable cache

from the Tag/Index field is taken in consideration for the way selection and the cache is effectively 4-way set associative.

![Virtual Mapping for the 4-way set associative cache.](image)

3.3.4 Block Size

To solve the problem of varying the block size, the minimum block size was defined as a single line and the maximum size the whole set of lines that can be fitted in a single cache Way ($2^n$).

As the Fixed Index field is hardwired to select only one cache line from each of the directly mapped ways the problem of selecting a single line out of each block is readily solved.

This means that the selection of the right word from the block is done before the associativity is known and that only the candidate cache lines are routed to the multiplexer structure.

3.4 Writing data to the cache

After defining the strategies for reading data from the cache, the next goal will be tackled: how to write data into the cache.

First and foremost attention is given to the write and substitution policies decisions as these two policies essentially define the problems that should be solved in order to accomplish associativity and block size configuration. Only then the actual applied strategies are presented and analysed.

3.4.1 Write and Substitution Policy

A Write Through policy was chosen to handle writes. This decision simplifies the design, with the added advantage of enabling the discard of cache blocks without the need to verify if they should be written to lower level memory.

A Write-Back policy is usually advised when there is a large difference between the delays involved in a write operation to cache and a write operation to the next level of memory in the hierarchy. In this particular case, the proposed cache is being designed to be the first level of memory of the CPU (usually called an L1 cache) and the negative effects of using a Write-Through policy are usually mitigated by using a fast (although slower than L1) L2 cache and the addition of a write buffer.

For the substitution policy a FIFO (First In First Out) queue was used. This was done for three main reasons:
1. A FIFO can be easily implemented in hardware requiring a small number of logic gates and a small critical path. This is a great advantage for a configurable cache that, comprehensively, has an additional overhead (chip area and delay) comparing to more traditional cache designs.

2. As stated in [Hennessy and Patterson, 2003], in general a FIFO can roughly approximate an LRU policy, which is usually the best realizable policy in terms of performance.

3. For the current design, the mutual combination of the Write Through policy, the FIFO queue, the priority encoder (present in the selected associativity circuit) and the decision to always store maximum needed Tag enabled the proposed cache to easily change the block size and associativity configuration without the need to flush the cache contents and therefore to increase the re-utilization of cache blocks between multiple configurations.

This last point can be better understood if it is taken into account what would happen if an increase in block size was made and a bigger block was fetched from memory while a smaller part of this same block was already in cache. This would mean that there would be two versions of the same data in cache and that it should always select the data that is more recent. This is solved with the priority encoder, without the need to write the smaller block to lower level memory.

These design decisions are the foundation for the strategies described in the following sections.

3.4.2 Associativity

In the previous section it was defined that a FIFO ordering would be used for the substitution policy. This decision allowed some simplifications in the decision and, specially, in the routing of the blocks.

To direct the incoming data to the right place, every cache way input is connected to a multiplexer responsible for selecting the data source. The data can come directly from outside (CPU or memory) or from the way immediately to the left, in the case where they both are configured to be part of a larger set. Depending on the selected associativity the individual write bits from each set are selected and the data is routed to the correct cache way.

In figure 3.14 the implementation of the FIFO algorithm can be seen. Whenever there is a write to cache the new data will be placed in the leftmost way of the selected set by shifting the block of data currently there and in all of the cache ways of that particular set one position to the right. The block present in the rightmost cache way of that set will be evicted since it reached the end of the FIFO.

For instance in the 2-way set associative configuration present in figure 3.14 if the CPU needs to fetch a memory block to the first set (way 00 and way 10) the write bits from this set should

---

1 Later in performance indicators it will be seen that the line evicted from the cache can contain important information for the assessment of performance measures that could be used for run-time tuning of the cache.
3. Designing a tunable cache

be enabled and the new block should occupy way 00 pushing the data currently there to way 10. The data formerly residing in way 10 will be evicted from the cache since it reached the end of the FIFO.

![Diagram of cache ways and associativity selection](image)

Figure 3.14: Associativity selection. The Tag/Index is represented at the top of each Way. The Index appears in black while the remaining bits appear in grey.

Another aspect from figure 3.14 is that there is some symmetry that could be used as an advantage in the physical realization of higher maximum associativity configurable caches.

3.4.3 Block Size

For the block size the problem is simpler. Since each way is directly mapped, meaning that each line from memory can only occupy one single position at each way, the only considerations needed to take care in order to accomplish a block size configurable cache is to fetch the right number of lines from memory to the correct way, according to the desired block size.

That is granted by a control circuit that fetches words from memory one by one until the selected block size (defined by the block size selector bits) is reached as seen in figure 3.15.

As the FIFO path is already in place and only depends on the selected associativity, the control circuit only needs to set the common write decoder to point to the first line of the selected block.
Figure 3.15: Block substitution in a 2-way set associative configuration

and signal a write to all the ways in the desired set when the data from lower-level memory is ready. Then adjust the decoder and repeat the process until all the block is written.

3.5 Storing Performance Indicators

Measuring the performance of caches, as in most of the elements for Central Processing Units, is not a trivial task. The most commonly accepted way to solve this problem is to run program excerpts (part of real applications or dummy processing intensive tasks) and assess the performance by means of the total running time required for the task. This technique is called benchmarking.

In this particular case, it is desired to gain some performance information while the program is executing in order to make informed decisions about how to select the presumably most suitable configuration for the cache. Using a benchmarking process for the type of micro-managing involved would be impractical.

To solve this problem, the cache was designed with a method to store performance indicators that, despite their limitation in assessing the overall performance of a system, can be used to aid in the decision process for changing configurations. These performance indicators can be used in dynamically cache tuning, by passing the information to a control system or, in statically cache tuning, by passing the information to the software.

Ideally, this performance indicators should give the most information available with the least overhead involved (time and area).
3. Designing a tunable cache

For this particular cache there are two types of indicators needed:

- Associativity indicators
- Block size indicators

For the **associativity indicators**, a set of registers is implemented for each way. There is one register to count every hit on that particular way and one register to count every miss occurred as shown in figure [3.16](#). This is a commonly used technique to check way performance [Kharbutli and Solihin, 2005](#).

![Figure 3.16: When there is a hit on one cache way, the respective hit count register is incremented. For all the other ways the miss count register is incremented.](#)

The miss and hit counters could be used to justify increasing or decreasing associativity or, if that possibility exists, shutting down some of the cache ways for better power management.

For the block size indicator, a new performance indicator scheme is proposed, where two extra bits are added to each line in cache. These bits are called the **Used** bit and the **Fetched Not Requested** bit. The **Used** bit indicates if the line present in cache was used by the processor. The **Fetched Not Requested** bit is set when a word is fetched from the memory and written to the cache, while not directly requested by the CPU.

These bits are combined to perform the following operation when a word is evicted from the cache:

<table>
<thead>
<tr>
<th>U</th>
<th>F</th>
<th>BSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>−1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>+1</td>
</tr>
</tbody>
</table>

**Table 3.1: Performance indicator for block size**

This operation can be represented by the expression:

$$\text{BlockSizeIndicator} = \#(\text{Used} \land \text{FetchedNotRequested}) − \#(\neg\text{Used} \land \text{FetchedNotRequested})$$

Although involving a subtraction, this operation can be easily implemented online with an Up/-Down counter actuated when a line is evicted from cache (as there is only one line evicted at any given time) avoiding the need for an arithmetic subtracter.
3.6 Run-time Configuration

This counter reflects a balance between the used and unused pre-fetched lines.

This information can be exploited to define statically and/or dynamically strategies, to tune the cache according to the needs of the running code. In the following sections, some possible strategies are presented.

3.6 Run-time Configuration

After designing the cache there is the need to control it. The run-time configuration can be accomplished by selecting the control signals **Associativity Selector** and **Block Size Selector**. Due to the design decisions that were made, these control signals can be changed at any time of the program execution, avoiding some flushing penalties and maximizing the re-utilization of already filled cache lines.

There are two main options for controlling the cache: Statically and Dynamically. For each of these main groups there are multiple algorithms that can be implemented and researched. The following discussion will focus on the algorithms implemented on the testing model during the execution of this work. Consequently, this section should not be seen as the only way to control the configurable cache previously designed, but as an example of the many possibilities enclosed in this type of architecture.
3. Designing a tunable cache

3.6.1 Static

For the static configuration, four instruction words are defined, in order for the programmer to decide on the desired associativity and block size:

\[
\begin{align*}
\text{WAY} & \quad n & \quad \text{Associativity} \leftarrow n \\
\text{BLK} & \quad b & \quad \text{BlockSize} \leftarrow b \\
\text{RWAY} & \quad R & \quad R \leftarrow \text{Associativity} \\
\text{RBLK} & \quad R & \quad R \leftarrow \text{BlockSize}
\end{align*}
\]

Table 3.2: Changing and reading configuration instructions

The value of \( n \) represents the desired \( n \)-way set associativity configuration. The value of \( b \) defines the size of the block size as \( 2^b \).

By using the RWAY and RBLK instruction words, the programmer can read the current configuration to register \( R \).

Instructions are also provided for reading and resetting the Performance Indicator registers.

\[
\begin{align*}
\text{RWAYM} & \quad R \quad n & \quad R \leftarrow \text{ReadMiss}[n] \\
\text{RWAYH} & \quad R \quad n & \quad R \leftarrow \text{ReadHit}[n] \\
\text{RBLKP} & \quad R & \quad R \leftarrow \text{ReadBlockPI} \\
\text{RSTWP} & \quad \text{AssociativityPI} \leftarrow 0 \\
\text{RSTBP} & \quad \text{BlockPI} \leftarrow 0
\end{align*}
\]

Table 3.3: Instructions for reading and resetting performance indicators

In this case the RWAYM and the RWAYH instruction can read respectively the number of misses and the number of hits for the \( n \) cache Way. The RBLKP instruction word reads the Performance Indicator register. For each of the described instructions, the result is stored in register \( R \).

The RSTWP and RSTBP set the value of the associativity and block performance indicators to 0.

These instructions provide a sensor and actuating platform for the programmer (or the compiler) to implement a software mechanism for controlling the cache associativity and block size not only through static code structure analysis, but according to some additional online performance information from the running program.

3.6.2 Dynamic

Although not the main focus of the current work an attempt was made to design a control circuit that would allow for the dynamic tuning of the cache. The solution presented here was partly implemented and should be taken as only one of several possibilities. Finding algorithms to dynamically tuning a cache is an active research topic and this particular cache could serve as a testing environment for new studies on this field.
3.6 Run-time Configuration

By setting a control bit called Dynamic Tuning Bit the programmer could delegate the decision of controlling the associativity and block size of the cache to a controller circuit. Also the programmer can set the tuning interval that will be used.

This can be accomplished by the instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDYN b</td>
<td>DynamicTuningBit ← b</td>
</tr>
<tr>
<td>STINT n</td>
<td>TuningInterval ← n</td>
</tr>
</tbody>
</table>

Table 3.4: Changing tuning interval instructions

In these instructions $b$ should be 1 if dynamic tuning is wanted and 0 if not wanted while $n$ should be an integer representing the number of cycles between cache re-evaluation and configuration change.

The dynamic approach was to implement a Markov chain with defined thresholds that, when compared to the performance indicators coming from the cache, can increase, decrease or maintain the block size or associativity. Figure 3.18 represents an example for the Markov chain implemented for the block size.

![Block size markov chain for the configurable cache.](image)

With this model, machine learning algorithms could be used to train the threshold values by using examples of good configurations from different programs, in a way similar to the one done in [K. Sundararajan, 2011].

It should be noted that the described cache associativity and block size can be changed to any state, not just to the one immediately above or below. In this case, by using a Markov chain the number of choices for configuration are limited. However, the decisions are simplified in comparison to the decision tree approach used in [K. Sundararajan, 2011], by limiting the number of comparisons to only two for each configuration change.

Another possible improvement would be to allow the configuration interval to be automatically tuned, as presented in [Gordon-Ross and Vahid, 2007].
3. Designing a tunable cache

3.7 Summary

In this chapter a new model for a cache was described enabling the configuration of the block size and associativity. In addition performance indicator registers were added that could be used for the static and dynamic configuration of the cache in run time.

Finally methods for static and dynamic tuning were introduced.
Cache integration in a CPU architecture

Contents

4.1 The DLX processor ................................................. 42
4.2 Configurable Cache Integration ................................. 51
4.3 Summary .......................................................... 53
4. Cache integration in a CPU architecture

To simulate the behaviour of the previously described cache in a real processor environment, a VHDL model of the DLX processor was used. The DLX, designed by David Patterson and John Hennessy and described in the book “Computer Architecture a quantitative approach”, Second Edition, is a RISC processor architecture based on MIPS, designed by the same authors in the First Edition of the book.

The used VHDL description was designed by Joachim Horch and has some key differences from the original design, the most obvious ones being a superscalar architecture and not having floating point capability.

Some of the reasons that led to the use of a DLX VHDL description were:

- Being a well known and studied architecture.
- Public availability of the VHDL source code.
- The use of some interesting architecture design concepts like being Superscalar and using the Tomasulo algorithm to enable out-of-order execution of instructions.
- The existence of publicly available compilers and assemblers.
- The existence of DLX simulators that could be modified to produce memory traces that would be fed to cache simulator programs.

To fully understand the architecture, a top-down description of the DLX will follow. Then, the modifications that were made in order to fit the cache into the DLX processor are stressed.

Furthermore, for clarification purposes an example (which will be used for the overall testing environment, described later in this document) will be presented at the end of this chapter.

4.1 The DLX processor

The DLX model that was used as the testing environment for the proposed tunable cache adopts a Superscalar architecture (can fetch more than one instruction per clock cycle) using the Tomasulo’s algorithm to solve data dependencies and enabling out-of-order execution of instructions.

To offer a first insight into the DLX design a top-level diagram is presented in figure 4.1.

The top-level model of the processor is composed of:

- Instruction Fetch stage capable of fetching 2 instruction words (64 bits) per clock cycle.
- A Branch target Buffer.
- Four execution units: Load / Store, Integer, Multiply and Branch Resolve.
- Reorder Buffer capable of holding the results coming from the execution units before they are committed in program order.
4.1 The DLX processor

- Register file, containing the general purpose registers R0 through R31.
- A Write-Buffer.
- Separated Data and Instruction Caches with Address Translation Buffer.
- A Bus Interface Unit to control the access to the External Bus.
- Commit Unit to control the committing of instructions.
- A Dispatcher, serving as the control unit of the processor

The memory interface can be seen as a *Modified Harvard architecture*, meaning that instructions and data are stored in the same memory, sharing the same bus but having separate caches for instruction and data.

This processor is considered a general purpose load-store architecture having 32 32-bit general purpose registers named R0 to R31. The R0 register can not be written, always holding the value '0'.

The original design had another 32 32-bit single-precision floating point registers named F0 through F31, which could be used in pairs (F0, F2, F4...) forming 16 double-precision floating point registers.

The DLX can internally access data (load or store) of 8-bits (byte), 16-bits (half-word) and 32-bits (word). As the smallest size of memory that the CPU can access is a byte the memory addresses are said to be byte-addressable. This means that each address represents one single byte of data and that, theoretically, the maximum addressable memory space is $2^{32}$ *Bytes* or 4GB.
4. Cache integration in a CPU architecture

A more in-depth description of the characteristics of the VHDL implementation of the Super-scalar DLX processor will follow.

4.1.1 Datapath and Tomasulo’s algorithm

In the presented DLX model, the Tomasulo’s algorithm was used to avoid WAW and WAR pipeline hazards. These hazards are avoided by the use of register renaming and the Common Data Bus.

The register renaming is granted by the use of Reservation Stations which are, essentially, a group of registers that are coupled to each of the execution units: in this particular case the Load Store Unit, Multiply/Divide Unit, Arithmetic and Logic Unit and Branch Resolve Unit, that can be seen in figure 4.1. These reservation stations host the operands of the instructions that are waiting for their turn to execute and can be seen as Virtual Registers that extend the capability of the general-purpose registers.

Another characteristic of the Tomasulo algorithm is the presence of a bus called Common Data Bus (CDB), connected to the output of each of the execution units. By monitoring this single bus, the results of all the executed instructions can be retransmitted to the reservation stations if they are needed.

Because this scheme enables out-of-order execution, a Reorder Buffer (RB) is used to allow speculative execution of instructions. The re-order buffer stores the results of instructions already executed that are awaiting to be written in program order.

The Tomasulo’s algorithm is composed of three stages:

**Issue** The Instruction Fetch unit fetches the instructions in program order (a process called in-order issue) to a free reservation station attached to the execution unit needed to perform the operation. If there are no reservation stations available for the desired execution, the processor stalls with a structural-hazard.

**Execute** While in the Reservation Station, the operands are read from the register file and reorder buffer. If some of the operands are coming from the result of a previously fetched but not committed instruction, the Common Data Bus (CDB) is monitored for the result needed. By monitoring the CDB for the needed operands, RAW hazards are effectively eliminated. When the execution unit is free and all the operands are available, the instruction can then execute.

**Write Results** After the results are computed in the execution units, they are written to the CDB and from there to the re-order buffer and reservation stations awaiting this result as operand for another instruction.

**Commit** The commit process writes the results that are stored in the Reorder Buffer in program order. A process known as in-order commit. This enables the speculative execution of
4.1 The DLX processor

instructions, by allowing instructions coming after a branch to be executed before the result of the branch is known.

4.1.2 Instruction Fetch Unit

Being a superscalar processor the Instruction Fetch unit supports fetching two instruction words ($2 \times 32$ bits) per clock cycle. The address of the next instruction to fetch is held in a register called the Instruction Counter.

Despite the memory being byte addressable, the fetching of instructions is always done aligned to the instruction word, meaning that all the instruction words begin at memory addresses that end with 00 ($2^2B = 32$ bit Instruction Word). Since the Instruction Counter is a 32 bit register, its last two bits are always set to 0.

Each of the fetched instructions are held in two registers, called Stage A and Stage B. Any of the stages can hold a high word (address ending in 000) or a low word (address ending in 100). However, the fetching of two instructions in the same clock cycle can only happen when the Instruction Counter register is double word (8-byte) aligned (This happens when the 3 least significant bits of the Instruction Counter are set to 0) and both stages are free.

If the instruction in Stage A is issued to the processor and Stage B is valid then Stage A will be loaded with the Stage B instruction and Stage B will become invalid and available to fetch another instruction.

If the Instruction Counter is double word aligned and only Stage B is available, the high word is fetched to Stage B and the Instruction Counter is increment by 4, to an address ending in 100.

These procedures ensure that the instruction in Stage A is always the first in program order and that all the instructions are issued in program order (in-order issue), as required by Tomasulo’s algorithm.

4.1.3 Dispatcher

The issuing of the instruction is ordered by the dispatcher circuit, which is the main controlling structure of the Superscalar DLX. This circuit is responsible for decoding the instruction and checking for the existence of a free reservation station in the needed execution unit. Each reservation station is identified with a unique number, which can be thought as a virtual register that will hold the result of the allocated instruction.

If there is no reservation station available the dispatcher should signal the Instruction Fetch unit to stall. On the other hand, if a reservation station is indeed free, then it is allocated. The dispatcher fetches the needed operands from the register file and/or the Reorder Buffer and store them in the reservation station. Since each Reservation Station is identified with a unique number, whenever there are operands that are not yet available, that number can be used to unambiguously identify the instruction that will produce the needed operand. Then, by monitoring the CDB,
the operand can be fetched as soon as it is ready.

4.1.4 Branch Target Buffer

A Branch-Target Buffer is a type of cache that, given the address of previously taken branch instructions, returns the predicted address for the next instruction, in order to diminish the control stalls and the mean number of cycles needed for branching actions.

When an address of a previously known branch instruction comes to the Instruction Fetch, the target address of the last time the instruction was executed is read directly from the Branch Target Buffer into the Instruction Counter Register and the CPU will continue to run instructions speculatively. If this speculation is correct, the execution continues as normal. If it is not, the instructions following the branch must be discarded, the address recalculated and the BTB updated.

Figure 4.2: Branch Target Buffer.

The used buffer is double word (8-byte) aligned and directly mapped, containing four entries indexed by bits 4 down to 3 of the Instruction Counter. The reason behind having the buffer double word aligned with the directly mapped index, at bits 3 and 4, instead of single word aligned with the directly mapped bits index, at bits 2 and 3, is because of the existence of a delayed branch slot, i.e. the instruction following the branch is always executed unless it is another branch. This decision invalidates the presence of two consecutive branch instructions.

A delayed branch slot is an instruction that immediately follows a branch and that should be executed regardless of the branch being taken or not taken.

The reason behind the implementation of a delayed branch is because in the DLX pipeline the decision to take or not take a branch can only happen after the Instruction Decode (ID) stage, meaning that the CPU would need to stall for 1 cycle (the Instruction Fetch stage) or allow the execution of a speculative instruction. The delayed branch tries to solve this problem, by allowing the instruction after a branch to always execute and passing the responsibility to the compiler to allocate a useful instruction to that available slot.

The Branch Target Buffer stores bits 31 down to 5 in the Tag field and additionally uses bit 2
to identify a branch from the high or low word since, as stated, only one of them can represent a branch instruction.

4.1.5 ALU

All Arithmetic Logic Unit operations are register-register, meaning that the result is never directly stored to memory and the operands can never come directly from memory. The operations performed are addition, subtraction, OR, AND, XOR and shift operations. There are two versions of each operation: a version where both operands come from registers and a version where one of the operands is an immediate, coming from the instruction word.

\[
\begin{array}{cccccc}
V & S & Alu\ Inst & V1 & Source\ Data\ 1 & RB\ Data\ 1 \\
<1> & <1> & <5> & <1> & <32> & <5> \\

V1 & Source\ Data\ 2 & RB\ Data\ 2 & RB\ Destination \\
<1> & <32> & <5> & <5> \\
\end{array}
\]

- V – Valid Reservation Station
- S – Speculative Flag
- Alu Inst – Alu instruction to Decode
- V1 – Valid Data on Source Data x
- Source Data x – Source Data for Alu
- RB Data x – Data for Source x will be available on this Reorder Buffer
- RB Destination – Reserved Reorder Buffer block
- ALU Reservation Station size = \((1 \times 1 + 5 \times 5) + (1 \times 32 \times 5) \times 2 = 88\)

Figure 4.3: ALU reservation station. The number of bits used for each field are represented and the total size in bits of the reservation station is calculated

All ALU instructions are performed in one single cycle.

In figure 4.3 a reservation station for the ALU is presented. The leftmost validity bit (V) signal the presence of an ALU instruction to execute while the speculative bit (S) indicate that the instruction present is being executed speculatively. Alongside the operands there are validity bits to indicate the presence of a valid operand or the need to fetch the operand from the reorder buffer entrance pointed by the RB Data field. After execution the results will be stored in the reorder buffer entrance pointed by RB Destination.

4.1.6 MDU

The MDU unit performs signed and unsigned multiplication and division of integers. The multiplication operations takes 4 cycles to complete while the divisions take 16 cycles. These values can be configured by a constant variable value in the VHDL code.

A Multiply-Divide-Overflow exception is raised if the result of a multiplication or division does not fit in the 32 bit register.

In the case of a division, a Divide-by-Zero exception is raised if the divisor is zero.
4. Cache integration in a CPU architecture

4.1.7 Reorder Buffer

The instruction commit is the last step of an instruction execution flow. In this stage, the instruction was already executed and the output value (if any) is ready to be written to the registers.

Because of the out-of-order execution implemented through the Tomasulo algorithm, an instruction can arrive to the reorder buffer before precedent instructions in the original program order. To ensure the instructions leave the processor in program order, instructions can only be committed after all the preceding instructions (in program order) are committed. If there is speculative execution after a branch, the speculated instructions can only be committed if the branch was confirmed as taken. If the speculation was wrong, all the instructions after the branch should be flushed from the reorder buffer.

Another advantage of the reorder buffer is the provided simplicity when dealing with exceptions and interrupts, for instance by I/O and timers, page faults and arithmetic exceptions. By not committing any instruction until they reached the top of the reorder buffer queue, the precise state of the CPU can always be preserved.

To avoid moving every instruction in the reorder buffer queue every-time there is a commit the buffer is implemented as a circular queue with a pointer to the next two instruction to be committed. This processor can commit up to two instructions in program order per clock cycle.

RB size \(= (5 + 32 + 32 + 5) \times 5 = 370\)

Figure 4.4: MDU reservation station.

Figure 4.5: Reorder buffer.

In figure 4.5 the reorder buffer is shown. The instruction address and resulting data are held in registers. The Write back flag is used to indicate the presence of data to store while the exception
flag signals the presence of an exception in that instruction.

### 4.1.8 Load/Store Unit

The Load Store unit is responsible for retrieving and sending data to memory. When there is a load, this unit gets the data directly from the data cache if it is already present. Since the cache is Write-Through, the write buffer should also be checked for valid data before accessing lower level memory.

To guarantee that all the data is consistent, the write buffer can only be written when the data is committed.

#### Figure 4.6: Load Store Unit reservation station.

#### 4.1.9 Write Buffer

As previously described, the memory hierarchy is designed to have smaller but faster memories close to the CPU and progressively bigger, but slower memories, as the data is moved further from the processor.

This means that the writing of data to lower-level memory usually takes additional cycles to complete. To avoid stalling the CPU, a write buffer is present. The write buffer present in the Superscalar DLX is a queue with the capacity to store up to 3 modified cache lines, while awaiting to be written to memory.

All the data present in the write buffer queue has already passed the commit stage and it is ready to be stored. This means that speculative stores should not be written to the queue and that the Write Buffer should always be checked on any data-cache misses before a request is made to lower-level memory.

The Bus Interface Unit is responsible for managing the connection between the CPU and the main memory. Every data coming or leaving the processor goes through the Bus Interface Unit.
4. Cache integration in a CPU architecture

4.1.10 Cache and Translation Lookaside buffer

The original Instruction and Data Caches from Joachim Horch's VHDL description were directly-mapped with 8 lines. Each line had 64 bits (two instruction words). Each byte of the 64 bits could be accessed directly by the processor through the three offset bits 0 through 2. Bits 3 through 5 indexed each line of the cache and the remaining 26 bits from the address formed the tag field. A validation bit was added to the cache to indicate if the data field contained valid data.

A translation lookaside buffer (TLB) with 4 lines was used to translate from virtual to physical addresses. The cache was Virtually Indexed and Physically Tagged (VIPT). This enables the least significant bits to be used to search the cache while the most significant bits were used to search the TLB. This parallel search reduces the side effects of having to always look the TLB for every cache search.

A Process Identification (PID) number field was added to support the execution of concurrent processes to operate without the need to flush all the cache every time there is a context switch.

Both the instruction cache and data cache are write-through no-write allocate meaning that there was no need to store any modification bits to the cache. However the TLB is write back write allocate and a modified bit exists in the data Translation Lookaside Buffer.
4.2 Configurable Cache Integration

Based on the designed cache architecture described in chapter 3, a VHDL model was conceived. This model used generic parametrizations to provide the users the ability to adjust the cache size and structure to their desire. The provided generic variables are:

**AddressableTo**  An integer defining how many bits of data each address word can address. Usually this field will be set to the value 8, signalling the addressability to 1 byte.

**Word**  Defines the size of an address in bits.

**Line_Size**  An integer that defines how many bits are used for the conjunction of the Offset and Line Offset fields. The number of bits used for Offset and Line Offset can then be derived by the size of the Word.

**Block_Max**  Size of the maximum block that can be defined. This is done by the number of bits used for the Fixed Index bits (see figure 4.9).

**Assoc_Max**  Defines the number of ways that are present and, consequently, the maximum associativity level by the number of bits used in the Tag/Index field.

According to the variables provided by the user in the VHDL model the division of the instruction word, the routing of signals and the number of overall internal components is adjusted accordingly. This was done using the declaration of generics in VHDL as can be shown in the entity definition for the cache:

```vhdl
entity Cache is generic (
  Line_Size: integer;
  Block_Max: integer;
  Assoc_Max: integer;
  AddressableTo: integer;
  Word: integer
);
port ( DataIn: IN STD_LOGIC_VECTOR(2**Line_Size*AddressableTo-1 downto 0);
  DataOut: OUT STD_LOGIC_VECTOR(2**Line_Size*AddressableTo-1 downto 0);
  Address: IN STD_LOGIC_VECTOR(Word-1 downto 0);
  Address_W: IN STD_LOGIC_VECTOR(Word-1 downto 0);
  Clk: IN STD_LOGIC;
  Rst: IN STD_LOGIC;
  Hit: OUT STD_LOGIC;
  W: IN STD_LOGIC;
  AssocSel: IN STD_LOGIC_VECTOR(Assoc_Max-1 downto 0);
  BlockSel: IN STD_LOGIC_VECTOR(Block_Max-1 downto 0);
);
end Cache;
```
4. Cache integration in a CPU architecture

The size of the registers for the block size and associativity configuration is also defined by these variables. For a maximum block size of $2^n$ a register with $n$ bits and for a maximum associativity of $2^a$ a register with $a$ bits are created.

When all the bits from these registers are cleared the smallest block size and associativity are selected. For each additional bit set the respective configuration is doubled. For instance, when the associativity register is cleared the cache is configured as a directly-mapped cache. If the two rightmost bits are set to the value ‘1’ the cache is configured to be a 4-way set associative cache.

The implemented cache model for the DLX processor was byte addressable ($AddressableTo = 8$), with 32-bits words ($Word = 32$) and cache lines of 64-bits ($Line.Size = 8$). The maximum block size was 16 ($Block.Max = \log_2{16} = 4$) and the maximum possible associativity was 4 ($Assoc.Max = \log_2{4} = 2$).

With this pre-defined variable settings the DLX address word is automatically divided in the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td>2</td>
<td>with 2 bits (0 to 1), giving 4 bytes for the instruction/data word.</td>
</tr>
<tr>
<td>Line Offset</td>
<td>1</td>
<td>with 1 bits (2), giving 2 words per line. This is the minimum block size.</td>
</tr>
<tr>
<td>Fixed Index</td>
<td>3</td>
<td>with 3 bits (3 to 5) giving 64 bytes for the maximum block size.</td>
</tr>
<tr>
<td>Tag/Index</td>
<td>2</td>
<td>with 2 bits (6 to 7) making direct, 2-way and 4-way associativity possible.</td>
</tr>
<tr>
<td>Fixed Tag</td>
<td></td>
<td>with the remaining bits (8 to 31).</td>
</tr>
</tbody>
</table>

The MaxTag field is composed of bits 6 to 31 and must be always stored in cache independently of the current run-time configuration. The size of the cache data structure can then be computed:

$$\text{Offset} \times \text{Line Size}$$

As previously stated, the cache should be transparent to the CPU, meaning that the CPU should work without the awareness of the existence of the cache. This enables the integration by just removing the old cache, represented in figure 4.8, and rewiring the inputs and outputs to the new cache.
4.3 Summary

As already discussed, the new cache adopts a Write Through writing policy. This was also the case of the DLX cache, which simplified the process by not having to check dirty-bits to know if there is the need to write data that is evicted from the cache.

A new class of instructions was added to the DLX instruction set, in order to support the cache tuning as presented in chapter 3. These instructions were called c-type instruction, for cache-type instruction:

<table>
<thead>
<tr>
<th>c-type</th>
<th>Opcode</th>
<th>R</th>
<th>Immediate</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;6&gt;</td>
<td>&lt;5&gt;</td>
<td>&lt;15&gt;</td>
<td>&lt;6&gt;</td>
</tr>
</tbody>
</table>

Figure 4.10: C-Type Instruction class

All the c-type instructions have the opcode 011111, since this opcode is unused by the Superscalar DLX. The following table represents the function field of the instruction words:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Function field</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAY n</td>
<td>Associativity ← n</td>
<td>000000</td>
</tr>
<tr>
<td>BLK b</td>
<td>BlockSize ← b</td>
<td>000001</td>
</tr>
<tr>
<td>RWAY R</td>
<td>R ← Associativity</td>
<td>000010</td>
</tr>
<tr>
<td>RBLK R</td>
<td>R ← BlockSize</td>
<td>000011</td>
</tr>
<tr>
<td>RWAYM R, n</td>
<td>R ← ReadMiss[n]</td>
<td>000100</td>
</tr>
<tr>
<td>RWAYH R, n</td>
<td>R ← ReadHit[n]</td>
<td>000101</td>
</tr>
<tr>
<td>RBLKP R</td>
<td>R ← ReadBlockPI</td>
<td>000110</td>
</tr>
<tr>
<td>RSTWP</td>
<td>AssociativityPI ← 0</td>
<td>000111</td>
</tr>
<tr>
<td>RSTBP</td>
<td>BlockPI ← 0</td>
<td>001000</td>
</tr>
<tr>
<td>SDYN</td>
<td>DynamicTuningBit ← b</td>
<td>001001</td>
</tr>
<tr>
<td>STINT</td>
<td>TuningInterval ← n</td>
<td>001010</td>
</tr>
</tbody>
</table>

Table 4.1: Instructions added to the instruction set

As there is no support for these instructions in the DLX compiler used the instructions should be manually inserted by the programmer in the assembly code. The assembler was altered to correctly translate these instructions.

4.3 Summary

In this chapter the DLX VHDL description was introduced and the cache was integrated by substitution of the one that was previously present. A set of instruction words were defined to change the cache associativity and block size registers.
4. Cache integration in a CPU architecture
5 Results

Contents

5.1 Testing framework ........................................ 56
5.2 Software Simulation ....................................... 57
5.3 VHDL Simulation ........................................... 60
5.4 Synthesis of the VHDL description in an FPGA .... 62
5. Results

After the integration of the cache into the Superscalar DLX processor some programs can be run in order to evaluate the designed cache.

To gain further insight on the advantages and drawbacks of the cache a set of programs were first simulated using a modified DLX software simulator to produce memory traces that could be fed into a cache simulator. Then the tests run on the VHDL model using a non-configurable cache that served as a control environment. Afterwards cache tuning was applied to the cache and different initial conditions were applied. Then tests were made where the configuration changed at run-time.

5.1 Testing framework

To test the cache model some DLX programs have to be produced. The approach used was to find a publicly available DLX compiler and assembler in order to produce DLX executable code from a higher-level software language. This enables a simplicity in producing tests and the possibility of running benchmark programs for comparison purposes.

5.1.1 Compiling and Assembling

The higher level language used for the production of assembly code was C. Some of the advantages of C language for the current work are:

- The availability of a public version of the compiler.
- Being a popular language used in the writing of many benchmark suits.
- The ability to declare chars (8-bits), short ints (16-bits) and ints (32-bit) enabling the testing of the loads and stores for bytes, half-words and words of the DLX processor.
- Being able to directly declare a pointer to a memory address.

The compiler used was based on the GNU C Compiler (GCC). The current version of GCC does not support the DLX instruction set despite supporting similar architectures like the MIPS so a pre-compiled old release of the GCC DLX port was used. This compiler enabled the production of DLX assembly code.

After having DLX assembly code some modifications had to be made in order to adjust the code produced to the differences in the instruction set between the Superscalar DLX and the original one described in [Hennessy and Patterson, 2003]. The main difference between the two was the absence of floating point operations in the Superscalar DLX. Although most of the problems could be suppressed by not declaring floating points some problems still persisted. In fact, in the original DLX description, integer multiplication and division were always made in floating point by moving the operands from integer registers to floating point, performing the operation...
and moving the results from the floating point registers back to integer as exemplified in the code below:

\[
\text{movi2fp f0, r1} \\
\text{movi2fp f1, r2} \\
\text{mult f0, f0, f1} \\
\text{movfp2i r1, f0}
\]

To solve this problem a program was produced to translate the assembly code coming from the compiler to assembly code that only included instructions from the instruction set of the Superscalar DLX.

Another problem found was that the compiler produced code that referenced some functions that would need to be present for linking. These functions had to be written in order for the programs to execute. One of the functions needed was `bcopy()` that copied bytes from one vector to another.

```c
void bcopy(char *a, char *b, int num) {
    int i;
    for (i=0; i<num; i++) {
        b[i] = a[i];
    }
    return;
}
```

Also because of its usefulness a print function was written. This made use of the previously stated ability to declare a pointer to specific memory locations to copy a defined number of words from memory to special purpose registers that ordered the writing of the output to a file.

```c
void print(int *a, int numpalavras) {
    int *p = (int *)0xffffff00;
    int *number = (int *)0xffffff04;
    int *start = (int *)0xffffff08;
    *p = (int)a;
    *number = numpalavras;
    *start = 0;
    asm("trap 0x104\n");
}
```

### 5.2 Software Simulation

To gain insight for the influence of the cache in the designed test programs a software DLX simulator was used to provide some memory traces that could be fed into the Dinero IV cache simulator. The results from the cache simulator were plotted for comparison. The test program analysed involved matrix multiplication and summation, as presented in Appendix A.
5. Results

The miss rate for a 256 B and 512 B cache are plotted according to different associativity and block sizes.

![Figure 5.1](58.png)  
**Figure 5.1:** Miss rate of the matrix operations program according to associativity and block size for a 256 B cache

![Figure 5.2](58.png)  
**Figure 5.2:** Miss rate of the matrix operations program according to associativity and block size for a 512 B cache

For the particular examples in figure 5.1 and figure 5.2 with the mentioned cache sizes associativity increase as a positive influence on miss-rate while the block size increase as a negative influence. To understand the reason behind this the misses were divided according to their classification in capacity, compulsory and conflict misses. An example for the 512 B cache with 32 B and 64 B blocks is presented in figures 5.3 and 5.4.

Here the effect of the associativity increase can be seen in the reduction of the conflict misses and the decrease in performance by increasing block size can be explained by the capacity misses. In fact, over sized blocks in small caches can lead to an increase in conflict and ca-
Figure 5.3: Miss classification in percentage for a 512 B cache with 32 B blocks.

Figure 5.4: Miss classification in percentage for a 512 B cache with 64 B blocks.

pacity misses and eventually to thrashing. This type of behaviour can be mitigated with a tunable cache by decreasing the block size to gain performance without the costs involved in the use of a larger cache.
5. Results

5.3 VHDL Simulation

The VHDL description of the Superscalar DLX was simulated using Modelsim. The same test run for two different tunable caches. The first one was a 256 B cache with 2 associativity ways each containing 16 lines. The second was a 512 B with 4 associativity ways and 16 lines. Each line had 64 bits of data.

The test run in all supported configurations. The processor was configured to have a 100 ns clock. The memory access was configured to take 100 times more than a cache access. The results for the 256 B cache are presented first:

<table>
<thead>
<tr>
<th></th>
<th>1 line</th>
<th>2 lines</th>
<th>4 lines</th>
<th>8 lines</th>
<th>16 lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly mapped</td>
<td>2412.932</td>
<td>2550.098</td>
<td>2792.776</td>
<td>3290.068</td>
<td>5345.748</td>
</tr>
<tr>
<td>2 ways</td>
<td>2342.420</td>
<td>2374.908</td>
<td>2760.128</td>
<td>3449.156</td>
<td>5314.720</td>
</tr>
</tbody>
</table>

Table 5.1: Execution times in milliseconds for the Superscalar DLX with a 256 B tunable cache using initial conditions

<table>
<thead>
<tr>
<th></th>
<th>1 line</th>
<th>2 lines</th>
<th>4 lines</th>
<th>8 lines</th>
<th>16 lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly mapped</td>
<td>1186.425</td>
<td>1180.506</td>
<td>1175.5345</td>
<td>1191.0345</td>
<td>1239.468</td>
</tr>
<tr>
<td>2 ways</td>
<td>1177.50</td>
<td>1168.280</td>
<td>1163.2925</td>
<td>1162.0605</td>
<td>1164.3785</td>
</tr>
<tr>
<td>4 ways</td>
<td>1215.520</td>
<td>1188.440</td>
<td>1206.240</td>
<td>1206.240</td>
<td>1728.470</td>
</tr>
</tbody>
</table>

Table 5.2: Execution times in milliseconds for the Superscalar DLX with a 512 B tunable cache using initial conditions

5.3.1 Tuning cache in run-time

To demonstrate how run-time cache tuning can affect and decrease the execution time, the program presented in appendix A was run for the following cache configurations:

a) a 2-way associative cache, each with 4 blocks of 2 lines per block, totalling 128 bytes in a static configuration;

b) a 2-way associative cache, each with 1 block of 8 lines per block, totalling 128 bytes in a static configuration;

c) a 2-way associative cache, in a dynamic configuration; initially it is configured as the configuration described in b) and then it changes to configuration described in a)

To simulate these configurations a clock of 20 ns (50 MHz) was set. To analyze program execution the cache misses were recorded and plotted using a periodic window of 150us.

Figure 5.5 presents the cache miss plot for case a). The total execution time was 52.033 ms. In the plot 4 different situations are clearly identified. First there is an initial set-up time for variable initialization with reduced number of cache misses. The following cases regard matrix multiplication, matrix summation and again matrix multiplication, respectively.
5.3 VHDL Simulation

Figure 5.5: Miss rate in program execution with a blocksize of 2 lines. Time is presented in ms. The different program operations are identified at the top of the figure: 1) Variable initialization. 2) Matrix multiplication 3) Matrix summation.

Case b), where the block size is of 8 lines can be seen in figure 5.6. The overall miss-rate over time is greater than the case for a 2 line cache, which resulted in a higher execution time: 82.428 ms.

Figure 5.6: Miss rate in program execution with a blocksize of 8 lines. Time is presented in ms.

Figure 5.7 shows case c) where the cache configuration was changed at run-time (t=46ms) from configuration b) to configuration a). To emphasize the advantage of having a configurable cache, the static configuration of case b) is also plotted in background.
5. Results

Analyzing the graphic it is clearly visible that re-configuring the cache can significantly improve the performance, by reducing the execution time from 82.428 ms to 69.304 ms. It should be noticed that this reduction in execution time is also achieved because the proposed tunable cache can re-use data after reconfiguration. For example, when a decrease in block-size occurs, as it happened in this particular case, all the data in the cache can be re-used.

5.4 Synthesis of the VHDL description in an FPGA

In the previous section it was shown that the proposed tunable cache can actually improve processor performance, by decreasing the miss rate. However, to observe the overall impact on the processor, the maximum clock frequency of the cache must also be evaluated. For this the tunable cache was synthesized using the Xilinx ISE 13.1 for a Virtex 4 (xc4vlx25) FPGA.

Before getting to the area and timing results, the list of synthesized logic found by the Xilinx Advanced HDL Synthesis was checked to confirm the existence of the expected logic circuits. The list is presented for a 256 B cache with 64 bit lines, 4 associativity ways with 8 lines each. As shown in Table 5.3, the inferred logic is congruent with the logic defined in chapter 3 with the presence of two 16 bit up counters (hit and miss counters) for each cache way. The 64 bit 4-to-1 multiplexer used for the datapath and the 29 bit comparators, one for each cache way, for the associativity selector circuit. Another fact that should be noticed is that the inferred logic is very similar to the one expected for a fixed cache size with 4 associativity ways with 8-lines.

Since the VHDL code description allowed for the configuration of the maximum level of associativity and maximum block size the timing and size results are shown for a set of possible initial
5.4 Synthesis of the VHDL description in an FPGA

Inferred logic | Quantity
---|---
16-bit up counter | 8
Flip-Flops | 3044
2-bit comparator greatequal | 2
2-bit comparator less | 2
29-bit comparator equal | 4
1-bit 3-to-1 multiplexer | 2
1-bit 4-to-1 multiplexer | 5
1-bit 8-to-1 multiplexer | 760
64-bit 4-to-1 multiplexer | 1

Table 5.3: List of synthesized logic for a 256 B cache with 64 bit lines, 4 associativity ways with 8 lines each.

The cache was synthesized with a fixed line size of 64 bits that could fit 2 DLX instruction words. The total cache capacity was fixed to three possible sizes: 128, 256 and 512 kB. The maximum associativity and number of lines in each way was varied to yield the three chosen total cache sizes.

For the 128 B cache the results are presented.

<table>
<thead>
<tr>
<th>waysize</th>
<th>maxassociativity</th>
<th>delay in [ns (MHz)]</th>
<th>4-input LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>8.879 (112.623)</td>
<td>4438</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6.763 (147.870)</td>
<td>2664</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>5.615 (178.108)</td>
<td>2076</td>
</tr>
</tbody>
</table>

Table 5.4: Size and timing for different 128 B caches

In table 5.4 there can be clearly seen the negative impact of increasing associativity in the frequency and area of the overall circuit. These results demonstrate that, as in a fixed cache, a tunable cache has to be properly chosen for a processor according to design goals in order to avoid bottlenecking the frequency times by selecting high-associative caches while not excessively increasing the cache miss-rate by decreasing associativity.

The 256 B and 512 B caches presented next confirm the previously stated conclusion.

<table>
<thead>
<tr>
<th>waysize</th>
<th>maxassociativity</th>
<th>delay in [ns (MHz)]</th>
<th>4-input LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>9.564 (104.562)</td>
<td>9054</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8.130 (123.003)</td>
<td>6041</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>6.485 (154.193)</td>
<td>4295</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>5.800 (172.408)</td>
<td>3700</td>
</tr>
</tbody>
</table>

Table 5.5: Size and timing for different 256 B caches
### Table 5.6: Size and timing for different 512 B caches

Analyzing these three examples there can be seen that the timing delays are very similar for the same maximum associativity independently of the cache size with the number of LUTs doubling for the doubling of the size of the cache suggesting that the overheads involved in providing a tunable cache can be further neglected for larger caches.
6 Conclusions
6. Conclusions

The scope of this work comprised the design and testing of a tunable cache that could be adapted to different associativity and block size dimensions during program execution. Although there are some previous works in this field the goal of most of the studies is cache energy efficiency. In this particular work the main focus was cache performance. Furthermore, the cache was intended to be attached to a real processor VHDL description to better assess its advantages and drawbacks by running test programs and measuring the execution time. These goals were accomplished.

Also, instructions were added to the the DLX processor for static tuning and an attempt was made to accomplish dynamic cache tuning by means of a control circuit and the design of dynamic performance indicators that can be used to assess the cache configuration adequacy to the running code.

By synthesizing the cache for a Virtex-4 FPGA the timings and area requirements were measured confirming the initial prevision that the option to tune a cache can be provided without a significantly impacting cost or CPU frequency when compared to a fixed cache with the same associativity and size. Furthermore the addition of way-shutdown in this type of design would allow for energy efficiency operation.

In the tests run an increase in performance of 16% was obtained by tuning the cache to a better configuration for the running code.

It is important to notice that advantages can be already obtained when the cache is configured in the beginning of program execution when compared to the fixed caches alternatives used today. In this particular case the cache can be tuned at any time without breaking the program executing and re-utilizing already filled cache lines. In particular, for the cases where the cache is configured for an increase in associativity or a change in block size all the valid data in the cache can be re-utilized. For a decrease in associativity the percentage of data that can be re-utilize depends on the way where each block of data is placed in the moment of reconfiguration.

6.1 Future work

During the execution of this work there were some possibilities discussed that did not reach this final stage but could be the subject of further research study.

Concerning the tunable cache itself it would be interesting to implement and test the cache with the processor in an FPGA board. This could be used to evaluate this type of design in respect to its energy consumption. Then modify it to support energy efficiency strategies, for instance way shutdown.

Another interesting point would be to study the applicability of the proposed cache model to a write-back memory hierarchy.

Also, in the current work an algorithm for dynamic cache tuning was introduced and partly im-
6.1 Future work

implemented but due to timing constraints the results obtained were not included in this work. This topic should be subject to further study on its own due to its specificity and the broad range of possible approaches and algorithms that could be used according to the desired goal of performance or energy efficiency.
Program used for testing
A. Program used for testing

```c
void MatrixMult(int a1[][6], int a2[][6], int a3[][6]) {
    int i = 0;
    int j = 0;
    int k = 0;
    for(i = 0; i < 6; i++)
        for(j = 0; j < 6; j++)
            for(k = 0; k < 6; k++)
                a3[i][j] = a3[i][j] + a1[i][k] ∗ a2[k][j];
}

void MatrixAdd(int a1[][6], int a2[][6], int a3[][6]) {
    int i = 0;
    int j = 0;
    for(i = 0; i < 6; i++)
        for(j = 0; j < 6; j++)
            a3[i][j] = a2[i][j] + a1[i][j];
}

void start() {
                   {12, 35, 34, 65, 23},
                   {12, 35, 34, 65, 23},
                   {12, 35, 34, 65, 23},
                   {12, 35, 34, 65, 23},
                   {12, 35, 34, 65, 23}},
    B[6][6] = {{12, 45, 45, 12, 45, 54},
               {12, 34, 76, 34, 35, 34},
               {35, 43, 87, 24, 87, 34},
               {35, 43, 87, 24, 87, 34},
               {35, 43, 87, 24, 87, 34},
               {35, 43, 87, 24, 87, 34}},
    C[6][6] = {{0, 0, 0, 0, 0, 0},
               {0, 0, 0, 0, 0, 0},
               {0, 0, 0, 0, 0, 0},
               {0, 0, 0, 0, 0, 0},
               {0, 0, 0, 0, 0, 0},
               {0, 0, 0, 0, 0, 0}};

    MatrixMult(A, B, C);
    MatrixMult(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixAdd(A, B, C);
    MatrixMult(A, B, C);
    MatrixMult(A, B, C);
    return;
}
```
Bibliography


Afzal Malik, Bill Moyer, and Dan Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In Proceedings of the 2000 international symposium on Low power electronics and design, ISLPED ’00, pages 241–243, New York, NY,

