MASTER THESIS

Study of Different Cache Line Replacement Algorithms in Embedded Systems

GILLE DAMIEN

08 March 2007

Company:	ARM France SAS Les Cardoulines B2 - Route des Dolines Sophia Antipolis - 06560 Valbonne France
Industrial supervisor:	Frédéric Piry
KTH examiner:	Ingo Sander

ARM

Abstract

The increasing speed gap between processors and memories underlines the criticalness of cache memories. The strong area and power consumption constraints of general purpose embedded systems limit the size of cache memories. With the development of embedded systems provided with an operative system, these constraints are even stronger. The selection of an efficient replacement policy thus appears as critical.

The Least Recently Used (LRU) strategy performs well on most memory patterns but this performance is obtained at the expense of the hardware requirements and of the power consumption. Consequently, new algorithms have been developed and this work is devoted to the evaluation of their performance. The implementation of a cache simulator allowed us to carry out a detailed investigation of the behaviour of the policies, which among others demonstrated the occurrence of Belady's anomaly for a pseudo-LRU replacement algorithm, PLRUm. The replacement strategies that emerged from this study were then integrated in the ARM11 MPCore processor and their performance results were compared with the cache simulator ones.

Our results show that the MRU-based pseudo-LRU replacement policy (PLRUm) approximates the LRU algorithm very closely and can even outperform it with low hardware and power consumption requirements.

Sammanfattning

Det ökande gapet mellan processorer och datorminnen förstärker betydelsen av effektiviteten hos cache minnena. De kraftiga restriktionerna av yta och elkraft begränsar storleken på cache minnena. Med utvecklingen av inbyggda system med operativsystemen blir restriktionerna ännu mer betydande. Så att välja en effektiv ersättningsalgoritm är därför kritiskt.

LRU algoritmen presterar bra på de flesta minnesstrukturer, men det fås till kostnad av hårdvaru specifikationer och elkrafts konsumption. Nya algoritmer har därför utvecklats och den här avhandlingen behandlar evalueringen av deras prestanda. Implementeringen av en cache simulator har tillåtit oss att utföra en detaljerad undersökning av egenskaperna hos algoritmerna, vilket bland annat påvisade Beladys anomali för PLRUm, en ersättningsalgoritm. Algoritmerna som kom fram under analysen integrerades sedan i ARM11 MPCore processorn och deras prestanda jämfördes med cache simulatorns.

Våra resultat visar att PLRUm kommer nära LRU algoritmen och till och med kan prestera bättre än LRU:n med låg hårdvara- och elkrafts krav.

Acknowledgements

This report concludes a six-month internship performed in the premises of ARM France SAS, in Sophia Antipolis. There, people helped me making progress in this interesting and crucial work, either on the theoretical understanding of all the phenomena involved in handling a data cache or about the applied management of the simulations and some C debugging issues. Thus, I take here the opportunity to thank all my colleagues from ARM France. I am particularly grateful to my industrial supervisor, Frédéric Piry, for his abiding support and for his answers to my numerous questions. Finally, I would like to thank KTH and École Polytechnique for having given me the opportunity to spend one year in Sweden as a double degree student.

Table of contents

List of figures ..... **vi**

List of tables ..... **vii**

List of codes ..... **vii**

Chapter 1 Introduction **1**

1. Motivation *1*

2. Methods and plan of the thesis *1*

Chapter 2 Theoretical Background **3**

1. The need of caches *3*
1.1. Memories 3
1.2. Caches 4

2. Generalities about caches *4*
2.1. Definitions 5
2.2. Unified vs. separated caches for data and instruction 5
2.3. Write-through vs. write-back 5
2.4. The question of the replacement policy 6
2.5. The mapping 6
2.6. Different types of misses 9

Chapter 3 Overview of the replacement algorithms **11**

1. The optimal algorithm *11*

2. Principle of locality *11*

3. Usual algorithms *11*
3.1. Random 12
3.2. Least Recently Used (LRU) 12
3.3. First In First Out (FIFO)/ Round Robin 13
3.4. Least Frequently Used (LFU) 13

4. Approximations of the LRU policy *13*
4.1. The 1-bit replacement policy 13
4.2. MRU based Pseudo LRU (PLRUm) 14
4.3. Tree-based Pseudo LRU (PLRUt) 14
4.4. Modified Pseudo LRU (MPLRU) 15
4.5. SIDE algorithm 16
4.6. Comparison of these policies 16

5. Enhanced LRU policies *16*
5.1. 2Q 19
5.2. LRU-*K* 19
5.3. Segmented LRU (SLRU) 19
5.4. Adaptive Replacement Cache (ARC) 19
5.5. Summary of the enhanced LRU policies 20

6. Ideas of improvement *20*
6.1. Cacheable/Non Allocatable 20

6.2. Selective cache way 20

Chapter 4 The current cache implementation 21

1. The ARM11 microarchitecture 21
1.1. Architecture vs. microarchitecture 21
1.2. Memory 21
1.3. System Control Coprocessor 21
1.4. Memory Management Unit (MMU) 22
1.5. Generalities about caches 22
1.6. ARM11 vs. competitor configurations 23

2. The ARM11 MPCore level 1 data side memory system 23
2.1. Slots Unit 23
2.2. Micro Translation Lookaside Buffer 25
2.3. Arbiter 25
2.4. RAMs 25
2.5. Hit stage 26
2.6. Store Buffer (STB) 26
2.7. Line Fill Buffer (LFB) 26
2.8. Eviction Write Buffer (EWB) 26
2.9. Droute 26

Chapter 5 Replacement policies simulation 27

1. Principles 27
1.1. Source files 27
1.2. Choice of the policies 27
1.3. Cache simulator 28
1.4. Benchmarks 29
1.5. Software 30
1.6. Remarks about the simulations 31

2. Simulation results 32
2.1. Benchmarks with usual replacement policies 32
2.2. Non-MRU and Global Round Robin study 33
2.3. Software 34

3. Interpretation 35
3.1. Random 35
3.2. Round Robin 38
3.3. Global Round Robin 39
3.4. 1-bit 39
3.5. Non-MRU 40
3.6. Modbits 40
3.7. Side 41
3.8. LRU and pseudo-LRUs 41

4. Cache set associativity 44
4.1. 2-way vs. 4-way set associative caches 45
4.2. High associativity 45

5. Conclusion: which replacement algorithms will be selected? 46

Chapter 6 The cache implementation 47

1. Replacement policies implementation 47
1.1. Integration of the lockdown feature 47
1.2. Hazards 48
1.3. Updating the status bits 48
1.4. Allocating a way 50

2. Status bits implementation 51
2.1. How to update the status bits 51
2.2. Storage of the status bits 51

3. In the Dirty RAM *52*
3.1. Creation of a status slots module 53
3.2. Actions on memory requests 54
3.3. Obtaining the up-to-date version of the status bits 55
3.4. Sum up 55

4. In a new RAM *56*
4.1. The RAM 56
4.2. Slots module 57
4.3. Actions on memory requests 57
4.4. Sum up 57

5. In a new RAM with a small cache *58*
5.1. Status bit RAM 59
5.2. Status Bit Cache 59
5.3. Read Buffers 63
5.4. Write buffer 66
5.5. Updater 66
5.6. Optimizations 66
5.7. Hazards 67
5.8. Validation of the design 67
5.9. Sum up 68

6. Results of the simulations *69*
6.1. Obtaining the hit ratio 69
6.2. Simulations 70

7. Conclusion *74*

Chapter 7 Conclusion **75**

1. Results *75*

2. Future work *75*

References **77**

Appendix A Number of status bits **81**

Appendix B The cache simulator **83**

Appendix C Selection of benchmarks **89**

Appendix D Validation of the implementation in Verilog **91**

Index **94**

List of figures

Figure 1: Repartition of memories....3
Figure 2: Performance gap between CPUs and memories....4
Figure 3: Usual memory hierarchy....4
Figure 4: Fully associative cache....6
Figure 5: Address in a fully-associative cache....7
Figure 6: Direct-mapped cache....7
Figure 7: Address in a direct mapped cache....8
Figure 8: N-way set associative cache....8
Figure 9: Address in a 2^n-way set associative cache....9
Figure 10: Miss ratio on different cache configurations with the PLRUm replacement policy....9
Figure 11: Improved LRU stack bits....12
Figure 12: 1-bit sequence....13
Figure 13: PLRUm sequence....14
Figure 14: Decision tree for the PLRUt algorithm....14
Figure 15: PLRUt sequence....15
Figure 16: Decision tree for the MPLRU algorithm....15
Figure 17: Side sequence....16
Figure 18: Level 1 data side memory implementation of the ARM11 MPCore processor....25
Figure 19: Hit ratio of the different usual replacement policies averaged over benchmarks....33
Figure 20: Average hit ratios on representative benchmarks with GRR and non-MRU....33
Figure 21: Hit ratio of the interesting replacement policies on the software applications....34
Figure 22: office_rotate hit ratio in a 4-way set associative cache....35
Figure 23: Markov chain for the Random replacement policy....37
Figure 24: Evolution of the probabilities p(k,n) in function of the allocations in the cache set....37
Figure 25: Average number of busy lines in function of the allocations in the cache set....37
Figure 26: Hit ratio for automotive_aifftr in a 4-way set associative cache....40
Figure 27: Hit ratio of the dirty bits implementation averaged over the selection of benchmarks....40
Figure 28: Memory requests of a core on a test bench....52
Figure 29: Architecture of the Dirty RAM solution....53
Figure 30: Architecture with the new RAM solution....56
Figure 31: Addressing the status bit RAM....57
Figure 32: Architecture of the RAM and cache solution....58
Figure 33: Status Bit Cache hit ratios for a 16-KB 4-way set associative cache on the selection of benchmarks....60
Figure 34: Status Bit Cache hit ratios for a 16-KB and 4-way set associative cache on software....60
Figure 35: Status Bit Cache hit ratios for a 4-way set associative data cache with different Status Bit Cache configurations and different data cache sizes on the selection of benchmarks....61
Figure 36: Hit ratio for a 4-way set associative cache on the SPEC92 benchmark suite [GEE93]....61
Figure 37: Deriving the address of the Status Bit Table for the two SBC configurations....62
Figure 38: Wave diagram of a SBRB....64
Figure 39: Wave diagram of a sequence for the overall SBC side....68
Figure 40: Improved LRU stack bits....82
Figure 41: PLRUt tree nodes numbering....86
Figure 42: MPLRU tree nodes numbering....86

List of tables

Table 1: Comparison of the different simple replacement policies 17
Table 2: Summarize of the enhanced LRU policies 18
Table 3: Cache characteristics of commercialized processors 24
Table 4: Hit ratio for the different policies with benchmarks with a 4-way set associative cache 32
Table 5: Average miss ratios on representative benchmarks of GRR and non-MRU 34
Table 6: Miss ratios of LRU, Random and PLRUm on specific benchmarks 36
Table 7: Miss ratios of PLRUm, PLRUt, Random and Round Robin on specific benchmarks 38
Table 8: Pseudo-LRU ways for PLRUm and PLRUt 42
Table 9: Non-optimality of PLRUt and quasi-optimality of PLRUm on a loop 43
Table 10: Sequence of 3 loops of N_{ways}+1 steps for LRU, PLRUm and PLRUt 43
Table 11: Number of hits for the N_{ways}+1-step loop with different filling orders 44
Table 12: The same N_{ways}+1-step sequence for LRU with a different filling order. 44
Table 13: Miss ratio in 4-way and 2-way set associative caches for software averaged over the different cache sizes 45
Table 14: Impact of high associativity on miss ratio for software and the selection of benchmarks 45
Table 15: Equivalent of Belady's anomaly for PLRUm 46
Table 16: Impact of the number of read buffers on the overall performance of maze and explorer 65
Table 17: Hit ratio of the Verilog version with benchmark office_rotate 71
Table 18: Miss ratio of the candidates on the different ARM11 cache configurations with maze and explorer 72
Table 19: Impact of the LFSR width for 1-bit with a 4-way set associative and 16KB data cache 73
Table 20: Test sequence of the optimal algorithm 84
Table 21: Test sequence for SIDE 85
Table 22: Test sequence of PLRUm algorithm 85
Table 23: Test sequence of MPLRU algorithm 86
Table 24: Test sequence for LRU 87
Table 25: Test sequence for ModBits 83
Table 26: Test sequence of the Status Bit Cache Logic with PLRUm replacement policy 92

List of codes

Code 1: 1-bit allocation and update of the status bits with lockdown feature 49
Code 2: PLRUm allocation and update of the status bits with lockdown feature 49
Code 3: PLRUt allocation and update of the status bits with lockdown feature 50

List of abbreviations

CP15	Coprocessor 15
CPU	Central Processing Unit
GRR	Global Round Robin
LFB	Line Fill Buffer
LFSR	Linear Feedback Shift Register
LFU	Least Frequently Used
LRU	Least Recently Used
MMU	Memory Management Unit
MPLRU	Modified Pseudo Least Recently Used
MRU	Most Recently Used
PLRUm	MRU based Pseudo Least Recently Used
PLRUt	Tree-based Pseudo Least Recently Used
RAM	Random Access Memory
RR	Round Robin
SBC	Status Bit Cache
SBG	Status Bit Group
SBRB	Status Bit Read Buffer
SBT	Status Bit Table
SBWB	Status Bit Write Buffer
SBT	Store Buffer
TLB	Translation Lookaside Buffer

Chapter 1

Introduction

В любом начинании только одна тайна: будет ли завершение?

Г. Александров[a]

1. Motivation

As processors become faster and faster, the performance gap between memories and processors gets wider and wider. Consequently, different solutions are proposed to cope with this issue, among them increasing cache performance. In this perspective, designing a cheap, low-power and efficient replacement policy appears as critical, as Eq. (2) p.6 shows. Besides, studies [MEG03, ZOU04] demonstrated that there is a significant gap between the optimal replacement algorithm and the one used in most embedded systems' processors, i.e. Random or FIFO. This observation motivates a deep study of these policies. However, because of the high locality in instruction caches, using a new replacement policy is not so critical for instruction caches [MIL03], particularly in comparison with the designing cost. This thesis is thus focused on the improvement of data cache performance by implementing a new cache line replacement algorithm.

The word implementation is crucial in this definition because all the studies published in the literature up to now deal with a cache simulator model, and thus are restricted to the theoretical efficiency of these algorithms. There is a lack of knowledge and data in this field, particularly in the embedded world. Moreover, all the works cited above focus on the x86 architecture and aim to find the most efficient algorithm. The only work found in the literature about replacement algorithms on ARM processors is [MIL03] but they use a simulator of the core and not the core itself, which could lead to some significant differences. Moreover, it was the previous generation of ARM cores, whose architecture has changed a lot. In this work, we will deal with a typical embedded processor, the ARM11 MPCore one. Like other embedded cores, there is no published work about the efficiency of different replacement cache line algorithms. This work will try to compensate this lack of knowledge. The originality of this study will be to take into account not only the efficiency of the algorithms but also their designing cost, their power consumption and their area; thereby leading to a good compromise between these factors. This is explained by the fact that whereas these factors are non-essential for desktop applications, they are critical for embedded systems.

Finally, all the studies found in the literature are focused on mono-processors whereas our implementation will take care of the process of invalidating lines. The impact of this feature is not studied in details in this thesis because of time restrictions. However, it is dealt with in the implementation since the ARM11 MPCore processor is a multi CPU system.

2. Methods and plan of the thesis

As explained in the previous paragraph, the aim of this internship is to implement a cheap, low-power, efficient and easy to design cache line replacement policy. In order to reach this goal, the thesis has been split into different parts:

[a] *At each beginning, there is only one mystery: will it be completed?*

G. Aleksandrov

- study of the literature,
- simulation of the algorithms and deep study of their characteristics,
- implementation of one or some algorithms in a hardware description language.

These steps allow dividing the thesis into different parts, with their own objectives. This method usually leads to a well organized work and simplifies meeting the requirements because there are various small frequent steps. This splitting is natural in the sense that we first study the existing material and try to improve the existing replacement algorithms. Then the chosen policies are simulated on our particular architecture in order to evaluate their performance and finally from these results and the constraints specific to embedded systems, some are chosen as candidates for implementation, which is the final step. The plan reflects this partitioning.

Chapter 2 presents the general features of caches memories. People already familiar with caches can begin the reading of this thesis at Chapter 3.

Chapter 3 deals with the common replacement strategies as well as their expected performance. The advantages and drawbacks of each policy are examined to select the algorithms that may meet embedded systems' constraints.

Chapter 4 briefly addresses the characteristics of the ARM11 implementation useful for this work.

Chapter 5 is devoted to the simulations of the replacement algorithms on a cache model. These results are then confronted to the literature and to the cost evaluations of their integration in the ARM11 MPCore data side architecture, thereby selecting some candidates for an implementation.

Chapter 6 focuses on the Verilog implementation of some replacement strategies. Different proposals are examined and the chosen one is addressed in details. Simulations are then performed on the enhanced version of the processor to corroborate the performance improvement.

Chapter 7 sums up the results of this work and suggests some ideas for future works on neighbouring themes.

Chapter 2

Theoretical Background

We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. Burks, H. Goldstine, J. von Neumann: *Preliminary Discussion of the Logical Design of Electronic Computing Instrument, Part I, Vol. I,* Report prepared for the U.S. Army Ord. Dept 1946

Before dealing in details with cache line replacement algorithms, the main characteristics of the cache memories that will be used in this thesis are briefly presented. The mapping, which will be intensely discussed in the final implementation, is dealt with in details.

1. The need of caches

1.1. Memories

Memories are one of the basic hardware elements with processors and communication. They enable data and instructions to be saved and they can be accessed thanks to the address of their elements. There are two types of memory accesses:

- the *reads* which give the value of an element stored at a defined address,
- the *writes* which store a value at a given address.

The access could concern only one word or some words, depending on the executed instruction.

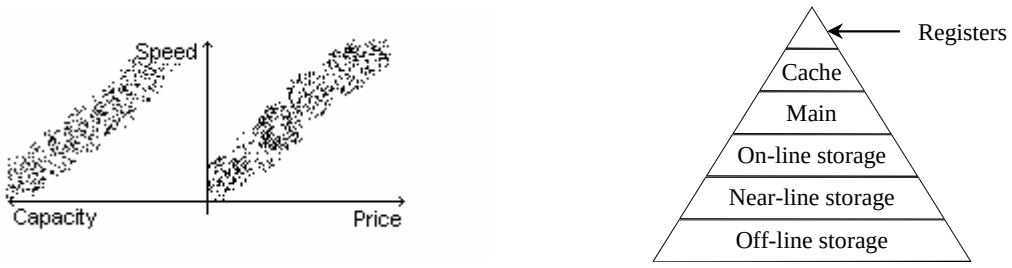

Figure 1: Repartition of memories[a]

Ideally, we would like to have an indefinitely large memory (with high capacities) which would be able to immediately access one element as well as instantly replace one of them. Unfortunately, it is only a dream. Various types of memories are thus used at different levels of the memory hierarchy, each of them resulting from specific compromises between price, capacity, speed, reliability, availability[b]. These criterions yield the classification of memories depicted above.

After the first glance on the figures above, one notices that the fastest memories are the most expensive. Apart from cost, fast memories have also low capacity, thus preventing engineers from

[a] These two pictures come from [SCU03]
[b] For more details about distinction between reliability and availability, see [DOU06]

designing fast and big memories. The different memories are usually regrouped as a pyramid in accordance with the previous observations. Registers lie at its top, as they are the fastest and the most expensive memory. Cost per byte, capacity and speed decrease downwards this pyramid to off-line storage.

1.2. Caches

During the last years, the gap between processors' speed and memories' speed increased in spite of the numerous attempts to cope with it (see Figure 2). As a result, memory access time appears now as a bottleneck for the design of fast systems. Indeed, the average access time is 60 ns for DRAM (usually used for the main memory) whereas a typical ARM11 MPCore processor clock period is 2 ns. Therefore, many endeavours have been performed to reduce the average access time. Fast memories located near the CPU (Central Processing Unit) and containing the most frequently used data and instructions should be a good solution to this issue: embedded SRAMs (commonly employed in cache memories) can exhibit an average access time equal to 1 ns. Unfortunately, these memories are also very expensive and their capacities are limited in virtue of the principles presented in Section 1.1. Processors would ideally always access the caches (except for cold misses), thereby requesting greater and greater capacities of the caches. It is obviously in opposition with the previous remark.

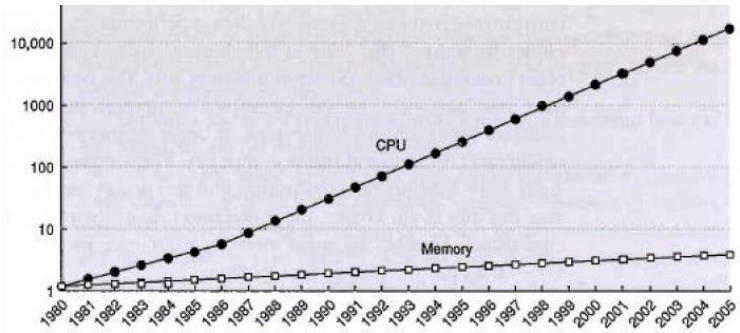

Figure 2: Performance gap between CPUs and memories[a]

Nowadays, processors are thus frequently provided with a second level cache (denoted as L2) in order to solve the cache size issue. This cache acts as an intermediary memory between the first level cache and the main memory. It has greater capacity but lower speed than first level caches (whose symbol is L1). The corresponding memory hierarchy is drawn on Figure 3 for a back-end L2. It can also be directly connected to the CPU and it is known as front-end.

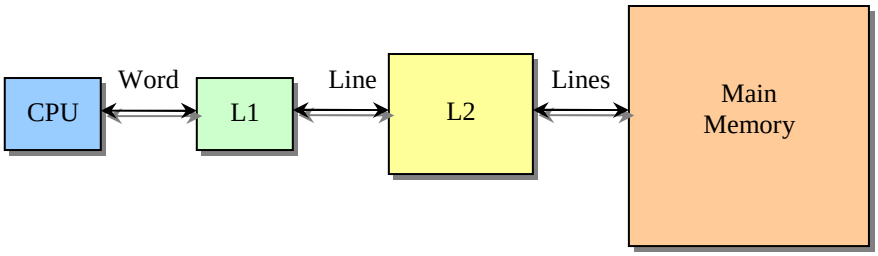

Figure 3: Usual memory hierarchy

In the next chapters, the term *upper level memory* will refer to the second-level cache if there is a back-end L2 or to the main memory for the other systems.

2. Generalities about caches

Various points about the characteristics of caches are addressed in this section: unified or separated caches, mapping, replacement policies…

[a] Picture from [HEN03]

2.1. Definitions

A *cache* is a small fast memory located near the processor which contains the data recently accessed. With caches, the memory access is not deterministic anymore but probabilistic.

A *cache line* is the smallest amount of data that can be transferred between the upper level memory and the cache. It takes advantage of the locality principle[a] to avoid numerous costing upper level memory accesses. A cache line is composed of *words.*

A *word* is the smallest amount of data transferred between the processor and the cache and usually corresponds to the size of the processor's registers.

If the data/instruction required by the processor is located in the cache, it is a *hit*. Otherwise, it is a *miss*.

The *mapping* is the technique used to assign one upper level memory block to one cache line.

2.2. Unified vs. separated caches for data and instruction

The information, that the processor needs accessing to, is the data and the instructions. They can be saved in the same memory or in two different memories because of their particularities[b].

Separating the data and the instruction caches doubles the bandwidth because the processor can issue simultaneously an instruction request and a data request. Indeed, a load or a store simultaneously calls for a data as well as an instruction. A unified cache can so be a bottleneck of the system in some specific situations. A separated cache has thus the advantage of a multi-port memory without its cost. Moreover, the logic that arbitrates priority between data and instructions in a unified cache is suppressed in a separated cache. Another advantage of separated caches is that the two caches can be located near the place where they will be used, thus saving some nanoseconds in the critical path.

On the other hand, separated caches raise important issues concerning the enforcement of the coherency between the two caches, especially for self-modifying codes. A thorny problem must be solved if data and instructions lie in the same memory line: each cache can own an erroneous copy of the memory line, thereby raising the coherency problem. Finally, the separation of cache and data leads also to an inefficient use of the memory because some memory space is wasted.

2.3. Write-through vs. write-back

When a data is located in the cache, the system has two copies[c]: one in the main memory, one in the cache. Two different policies govern the write:

- *write through:* the data/instruction is written both in the cache and in the main memory,
- *write back:* the information is written in the main memory only when the line is removed from the cache. To avoid useless back writings in the main memory, the cache lines are provided with a dirty bit. If it is set to 1, it means that the data/instruction has been modified in the cache. Otherwise, the data is clean and does not need to be written back in the main memory on eviction.

The main problem of the write through policy is its generation of a lot of unnecessary traffic. It leads to a situation where the processor may have to wait for the write buffer to perform the writes. With this policy, the main memory has always an up-to-date copy of the data/instruction, which simplifies coherency.

[a] For further details about the locality principle, see next chapter.
[b] On the figure 5.8 p. 406 of [HEN03], we see for instance that instruction caches have less miss rates than data caches.
[c] For systems with a L2 cache, the situation can be more complex: there can be up to three copies, one in L1, one in L2, one in the main memory. For non-inclusive L2, there may be a copy in L1 with no copy in L2.

The write back policy does not face the programmer with these problems but it has also drawbacks. It generates fewer stores and is so faster but by definition the discrepancies between cache and main memory last longer and the worst case sequence is longer too. Moreover, the implementation of this policy is more complex.

2.4. The question of the replacement policy

Since caches cannot contain the whole memory, designers face an important problem: which lines should be stored in the cache? The answer is obviously dynamic because the needs of the processor dynamically change over time. Consequently, the problem can be reformulated: in which emplacement of the cache should lines be stored, which lines should be kept in the cache and which ones should be discarded? Numerous cache lines replacement algorithms try to answer the replacement question in an efficient, low power consuming, fast, easy to implement and cheap way. The criticalness of this question is illustrated by the following equation:

$$\begin{aligned} \textit{Average access time} = \textit{Hit ratio} \times \textit{Average cache access time} + \\ (1 - \textit{Hit ratio}) \times (\textit{Average cache access time} + \textit{Average upper level access time}) \end{aligned} \quad (1)$$

The term *Average cache access time* comprises the time required for sending the Load/Store instruction to the cache, performing the cache lookup and forwarding the data in case of a hit. This step is performed by any instruction sent to the cache arbiter. For missed accesses, the line must then be fetched from the upper level memory. Replacing the times by their typical values in clock cycles, *Average cache access time* = 2 and *Average upper level access time* = 100, one gets:

$$\textit{Average access time} = 102 - 100 \times \textit{Hit ratio} \approx 100 \times \textit{Miss ratio} \quad (2)$$

Decreasing the miss ratio of only one percent improves the overall performance of one clock cycle! This observation explains why the theme of this thesis is considered as crucial.

2.5. The mapping

The replacement policy answered only partially to the questions formulated in the previous section: the first one remains. The answer is given by the mapping that assigns an upper level memory block to a cache line. Three mappings are usually used: direct mapped, fully associative and *N*-way-set associative.

2.5.1. Fully-associative cache

The mapping of a fully-associative cache is drawn on Figure 4. Each memory line can be mapped anywhere in the cache. This technique thus requires performing the lookups of all the lines in parallel, thereby generating a huge amount of hardware in the cache controller. It implies large access times and big capacitances. These constraints make this solution viable only for small caches.

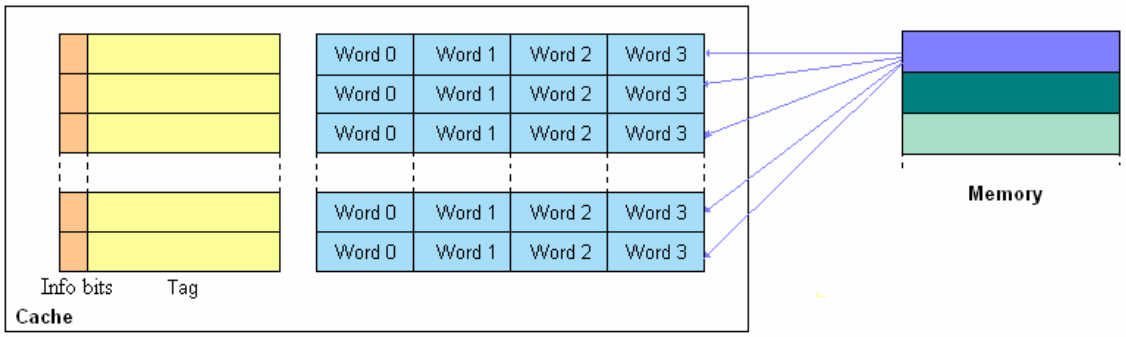

Figure 4: Fully associative cache

As the main memory block can be mapped anywhere, a means is needed to know which upper level memory address currently lies in the cache: this is given by the tag, a unique number for each upper

level memory block. Because of the huge amount of bits, this piece of information is usually stored in a dedicated RAM.

The memories and the cache are assumed to be byte-addressable therefore any byte of the cache lines can be accessed individually. The part of the address handling these byte accesses is called the offset.

Let us take the example of a 32-bit address and 2^s KB cache where s is an integer. The memory is assumed to be byte-addressable and the width of the cache line equal to 256 bits, i.e. 32 bytes. The main memory thus contains 2^{27} lines. This distinction between the lines can be performed according to the address cut drawn in Figure 5.

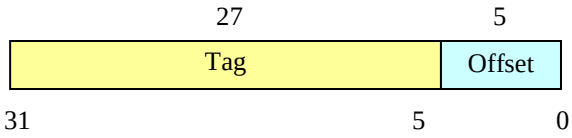

Figure 5: Address in a fully-associative cache

In Figure 4, the tag is bound with information bits. They allow the cache controller to know the state of the given cache line. In general, it contains the following:

- *validity bit*: the stored line is valid or not,
- *dirty bit*: the line has been modified since its allocation,
- *coherency protocol bits*: in multiprocessors environment, some additional bits are required to ensure coherency between the caches of the processors (MESI for instance needs four additional bits in one hot encoding).

These bits do not depend on the mapping and will be found through the three different mappings. Depending on the implementation, some other bits can be found too.

2.5.2. Direct-mapped cache

Here, each memory line is mapped to a particular line of the cache. The line on which the memory block is mapped is denoted by a part of the address called the index. On Figure 6, the cache lines are composed of 4 words. A common way to perform the cache line selection is:

$$Line = (Memory\ line\ \ address) \bmod (number\ of\ \ lines)$$

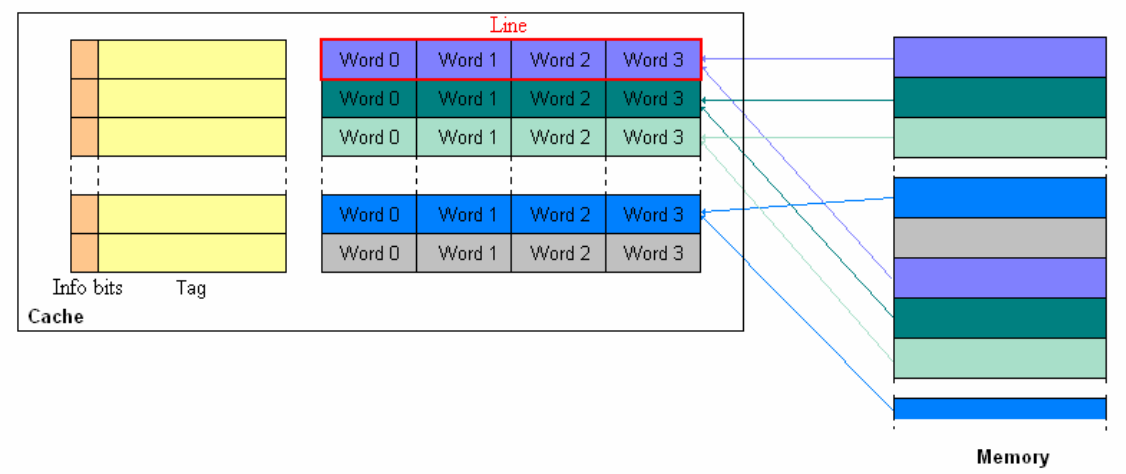

Figure 6: Direct-mapped cache

Let us retake the example of the previous section: a 32-bit address and 2^s KB cache. Thus, the cache contains 2^{s+13} bits. Knowing that a line is 256-bit wide, one deduces that there are 2^{s+5} lines per set. Therefore, the index is (s+5)-bit wide.

The memories are byte-addressable. Consequently the 32-bit address enables access to a 2^{35}-bit memory. As the width of the cache lines is 256 bits, the main memory contains then 2^{27} lines. As a result, 2^{22-s} different lines need to be distinguished per cache set. The tag is so (22-s)-bit wide. Moreover, a line is 32-byte wide, giving an offset of 5 bits. The address cutting is shown on Figure 7.

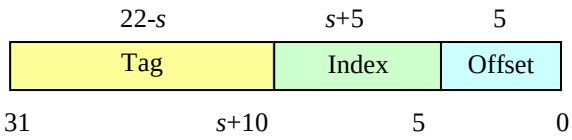

Figure 7: Address in a direct mapped cache

Direct mapping is a simple strategy but poorly effective. Indeed, as the cache size is smaller than the upper level memory size, different memory blocks are mapped to the same cache line, thereby leading to numerous conflicts misses. A means to prevent it is to allow a memory line to be mapped onto different cache lines. This solution is presented in the next section.

2.5.3. *N*-way set associative cache

The cache is split into *sets* where each set is composed of *N* cache lines. The cache scheme is drawn on Figure 8, where a set is represented as the union of the red rectangles.

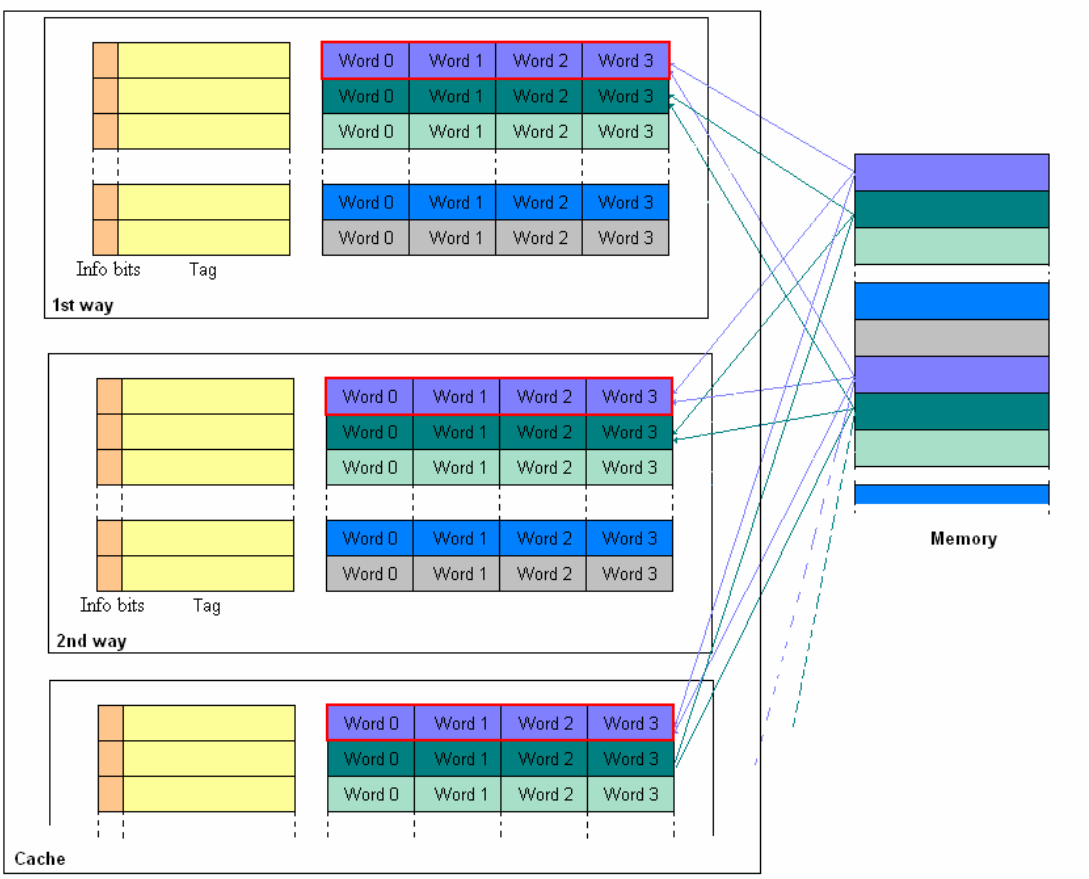

Figure 8: N-way set associative cache

The upper level memory line is affected to a set and can then be mapped on any of the *N* ways, thereby giving the possibility to decrease the number of conflict misses. Therefore, the cache is associative in a set; which explains the name of the mapping. The set selection can be performed by:

$$Set = (Memory\ line\ address) \bmod (number\ of\ sets)$$

As an example, we will consider here the case of a 2^s KB 2^n-way set associative cache. Thus, the cache contains 2^{s+13-n} bits per way. Knowing that a line is 256-bit wide, one deduces that there are 2^{s+5-n} lines per set. Therefore, the index is $(s-n+5)$-bit wide.

The memories are byte-addressable. Consequently the 32-bit address allows access to a 2^{35}-bit memory. As the width of the cache lines is 256 bits, the main memory contains then 2^{27} lines. As a result, there are 2^{22-s+n} different lines to distinguish per cache set. The tag is so $(22-s+n)$-bit wide.

A line is composed of 32 bytes, yielding an offset of 5 bits. An address scheme is drawn below.

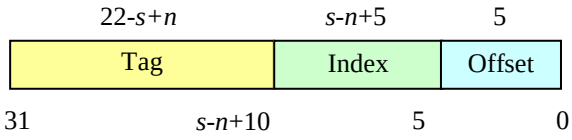

Figure 9: Address in a 2^n-way set associative cache

2.6. Different types of misses

A miss occurs when the data (or the instruction) required by the processor does not lie in the cache: it must be then fetched from the upper level memory. There are four types of misses:

- *compulsory misses*: these are the misses on a cold start. Data must be fetched at least once from the upper level memory to be present in the cache,
- *capacity misses*: these misses occur when the working set (data/instructions required for the program) exceeds the cache size,
- *conflict misses*: such misses result from the mapping of two different items to the same cache line,
- *coherency misses*: in a multiprocessing environment, the coherency protocol may invalidate a line. The next lookup for this line will be a miss.

This work is mainly devoted to single processors' data caches. Consequently, we will not deal further with the coherency misses. The different types of misses can be seen on Figure 10. It should be noticed that the *x*-axis of the graph is logarithmic.

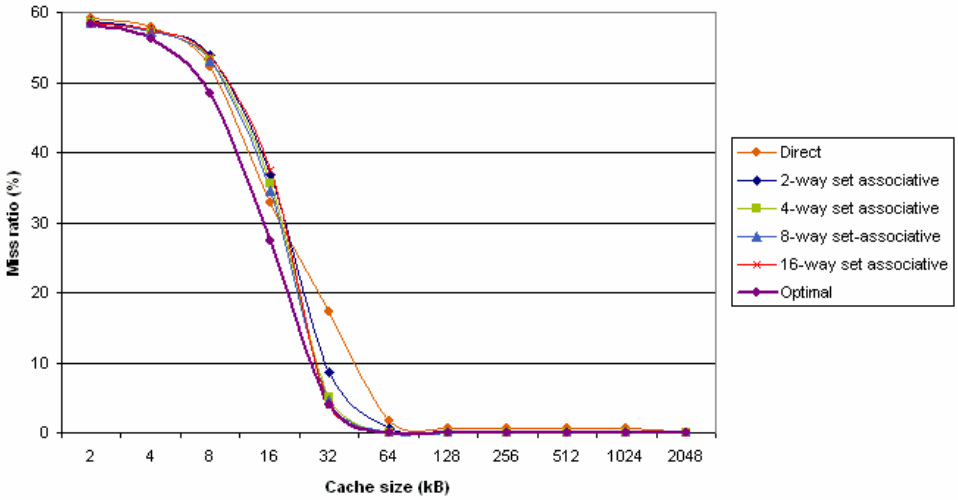

Figure 10: Miss ratio on different cache configurations with the PLRUm replacement policy [a]

Compulsory misses. First, one sees that for the highest cache sizes (128 KB and greater), the miss ratio is constant and tends to be zero. However, it is a bit greater than zero (around 0.06 %). Increasing further the cache size does not influence the miss ratio anymore. These remaining misses are the compulsory misses and correspond to the first use of the data. Thus, they are impossible to avoid, unless the cache becomes omniscient and predicts the data that will be used on start in order to preload them in the cache.

Capacity misses. When the working set cannot be contained in the cache, useful values evict one another from the cache. These misses are capacity misses and they can be seen on the graph in the part where the hit ratio continuously decreases. For instance, one deduces from Figure 10 that the working set of this program is around 32 KB. They can be avoided by increasing the number of cache lines or by enlarging the size of the lines in the cache. However, increasing the size of the lines without modifying the cache size leads to more conflict misses. So these two solutions extend the size of the

[a] This graph results from simulations run with a cache simulator created in this thesis. It will be presented in details in Chapter 5. The memory requests pattern was obtained from `maze`, a program looking for a path between two nodes of a tree.

cache, which will be more expensive, consume more power and more area. These solutions have thus strong negative impact on key points of embedded systems. Moreover, the size of the cache cannot be continuously increased because it makes the cache access time longer. Yet, the memory access time constrains the processor clock cycle and is considered as a limit in today's processors. As a result, dealing with conflict misses appears as the only efficient and cheap way to increase the efficiency of the caches.

Conflict misses. One sees that there is a difference between direct mapped cache and *N*-way set associative cache for data cache sizes between 8 KB and 64 KB. This difference between 16-way set associative cache and direct mapped cache is due to a diverse influence of conflict misses. Indeed, 16-way set associative cache avoids conflict misses by giving more possibilities for the allocation of a way, thereby leading to fewer conflicts. On the graph, the hit ratios of 8-way and 16-way set associative cache are almost equivalent. Thus, 16-way set associative cache can be considered as efficient as a fully-associative cache for this memory pattern. The difference between the optimal algorithm and the *N*-way set associative caches yields the ratio of conflict misses in the given configuration. This simulation confirms that the origin of conflict misses is the mapping of the upper memory level blocks to a cache line. Therefore, relaxing the constraints on the mapping should decrease the number of conflict misses. In this optics, *N*-way set associative caches (and ideally fully associative cache) appear as the best solution to this issue according to Figure 10. Nevertheless, increasing the associativity of the cache is also problematic because it requires a lot of lookups in parallel, which is power consuming, the key point for embedded processors. For this reason, an alternative for dealing with conflict misses is to propose another replacement policy, which would be as optimal as possible. When a conflict occurs, the choice of the line to evict at the profit of the incoming one is crucial since it determines the data present in the cache. This observation demonstrates the criticalness of the theme of this thesis.

Through this chapter, different characteristics of the cache have been presented. Among them, one appeared critical for the design of an efficient and low-power embedded cache: the cache replacement policy. This is the subject of this thesis and is investigated in details in the next chapter.

Chapter 3

Overview of the replacement algorithms

Savoir pour prévoir, afin de pouvoir[a].

A. Comte

We have seen in the previous chapter that different upper level memory lines are mapped to the same cache line. Thus, when the set of cache lines, which the upper level memory block can be mapped to, is full, the line that will be evicted at the profit of the new data must be chosen. The role of the replacement algorithms is to select the discarded line as optimally as possible. These algorithms should be implemented in hardware and execute rapidly in order not to speed down the processor. In this chapter, the principle of locality – on which almost all the replacement policies are based – is presented before introducing and comparing the usual replacement algorithms.

1. The optimal algorithm

This algorithm [BEL66] uses the cache in the optimal way. It replaces the lines that will not be used for the longest period of time. Consequently, it must know all the future accesses and thus is impossible to implement. Only simulations on predefined memory patterns can be carried out. As a result, it only appears as a means to measure the performance of the replacement algorithms. Moreover, Brehob *et al.* [BRE04] showed that computing the optimal policy is NP-hard for modern cache structures. Because of this huge computation time, it will be little used in this thesis.

Since processors are not omniscient, a method predicting the next accessed data is needed to approach the performance of the optimal algorithm. To that end, replacement policies resort to the principle of locality.

2. Principle of locality

Programs tend to reuse the data and the instructions that they have used recently: this is the temporal locality. Moreover, a program tends to use the instructions and the data that are located in the vicinity of the used one. This is the spatial locality. For instance, a program spends in average 90% of its execution time in 10% of its code [HEN03]. As a result, we can reasonably predict the next used data and instructions from the previously accessed ones. Replacement algorithms try to take advantage of the locality to be as near as possible to the optimal replacement choice.

3. Usual algorithms

As stated in the previous chapter, the eviction occurs among the cache lines of a given set, which is the set of an *N*-way set associative cache or all the cache lines for a fully associative cache. Replacement algorithms are responsible for assigning this victim line. Of course, there is no need of such an

[a] From knowledge comes prediction and from prediction action

algorithm for direct mapped caches where there is not any selection to operate. Through this chapter, a single cache set will be considered; it is assumed that the situation is duplicated for the other cache sets.

3.1. Random

The replacement policy chosen here is the simplest: the discarded line is randomly selected. This algorithm is easy to implement as a pseudo-random counter for the whole cache, consumes few resources but performs badly because it is not usage-based. Its performance relies on the real randomness of the sequence. It can be implemented with Linear Feedback Shift Registers but this solution is poorly efficient. Random is reported to perform 22% worse than LRU on average [ZOU04]. This inefficiency is balanced by another advantage of the Random policy: in opposition to Round Robin, its variations depend less on parameters such as working set size, number of cache lines.

3.2. Least Recently Used (LRU)

The LRU algorithm evicts the least recently used line. The idea behind is to keep the data recently used which should be used soon, thanks to the principle of locality. It has been demonstrated that LRU never results in more than *p* times more misses than the optimal algorithm where *p* is proportional to the number of ways [SLE85]. All the accesses to the blocks are recorded and the replaced block is the one which has been the least recently used. It is thus very computationally expensive to implement for large caches with a great number of ways. Moreover, it does not take into account the frequency of accesses.

Replacement in a set of *N* elements requires *N(N-1)/2* bits with an upper triangular matrix *R* (without diagonal) if the traditional encoding is used. When a line *i* is referenced, row *i* is set to 1 and column *i* is set to 0. The LRU line is the one whose row is entirely equal to 0 (for those bits in the row because the row can be empty) and whose column is entirely 1. This algorithm should execute rapidly and quite easy to implement. As the complexity grows with the size, it is preferable to have small set sizes but small set size generates more conflicts.

Nevertheless, another solution was developed in this thesis, which yields fewer bits, but at the price of a bit more complicated encoding which would require more hardware. However, it should be acceptable considering the fact that the ARM11 data cache has only four ways. Furthermore, the point is that the critical factor of the design will be the size of the additional storage (see Chapter 6) and not the hardware itself. Amongst other things, it is explained by the sharing of the hardware by all the data cache sets. The proposal relies on the squandering of the storage in the traditional encoding. Let us look at the example of a sequence successively accessing ways 3-2-0-1 in a 4-way set associative cache. This situation is drawn in Figure 11. The MRU way lies on the top of the stack and the LRU on the bottom. The two first ways are encoded with two bits. Once it is known that the two first ways in the LRU stack are ways 1 and way 0, only two possibilities remain: the access order is either 2-3 or 3-2. It is justified by the fact that the stack can always be considered as full. Therefore, it can be encoded in one bit instead of the three previously used. This solution should save 2 bits per set, which is crucial when keeping in mind that usual cache configurations are composed of 512 lines [ARM01]. This solution is the most compact encoding (for a theoretical demonstration, see Appendix A). Unfortunately, there is no easy expression of the number of bits in the general case (for more details see Appendix A).

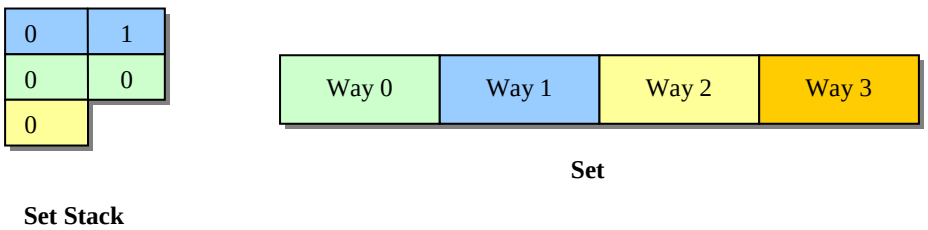

Figure 11: Improved LRU stack bits

LRU is widespread in the industry although there are well-known situations where it performs far from optimal. The classical example is a loop of N_{ways}+1 steps, each step *j* accessing data *j*. At each step, the data that will be used at the next access is discarded, thereby leading to a miss ratio of 100% percent whereas the optimal policy generates only 25% (one out of four) of miss. For this reason, other

algorithms, which require even much more hardware, are developed. Therefore, they appear as quite inefficient candidates for implementation in embedded cores.

3.3. First In First Out (FIFO)/ Round Robin

Since the LRU replacement policy is complex to implement and computational consuming, a popular approximation, FIFO, has been developed. It removes block in the order they were brought in the cache, thereby taking advantage of the locality principle in a simpler way. FIFO yields a miss ratio 12%-20% higher than LRU in average [SMI83] but is far less expensive in hardware and in computation time. It can be designed with a $\log_2(N_{ways})$-bit counter per set, which points to the next evicted way of the set. This counter is incremented on every cache miss. Provided that the counter is initially set to 0 when the cache is cleaned, the lines are discarded in the order they entered the cache.

Apart from its relative poor performance, this algorithm presents a major drawback: contrary to the first impression, increasing the size of the cache can worsen the efficiency of the algorithm. This phenomenon is known as *Belady's anomaly* [BEL69].

3.4. Least Frequently Used (LFU)

This algorithm keeps track of the frequency of accesses of the lines and replaces the LFU one. Therefore, the lines which have been accessed very frequently and that will not be needed again tend to remain in the cache, preventing useful data to be cached. Usually, an aging policy is used to avoid this *cache pollution.* It requires logarithmic implementation complexity in cache size and will then not be studied further in this thesis focused on embedded and hardware cheap solutions.

4. Approximations of the LRU policy

The LRU policy gives good performance but it requires a lot of hardware to keep track of the last accessed data in the cache. This hardware complexity has a strong negative impact on the average access time, thus justifying the use of approximations to this policy. On the bottom of the LRU stack, the probability that the processor hits a line is almost constant, so a complete order is not required, only a partial one is needed. As a result, approximations of the original LRU algorithm should perform well.

For all the figures drawn in this section, the following convention is respected: the green colour corresponds to blocks considered as Most Recently Used (MRU) by the replacement algorithm and the yellow one stands for the pseudo-LRU line(s)[a].

4.1. The 1-bit replacement policy

It is one of the simplest approximations of LRU and requires only one bit per set [SO88]. The bit partitions the set into two groups: one which contains the MRU line, the other that does not contain. The aim is to protect the MRU line and its neighbours according to the principle of spatial locality. On a cache request, this bit is updated to point to the part where the MRU line lies. The discarded line is then randomly selected in the non-MRU half. A sequence of the algorithm is shown on Figure 12. After an access, the pointed half (in green in the sequence below) is always the one where the access occurred.

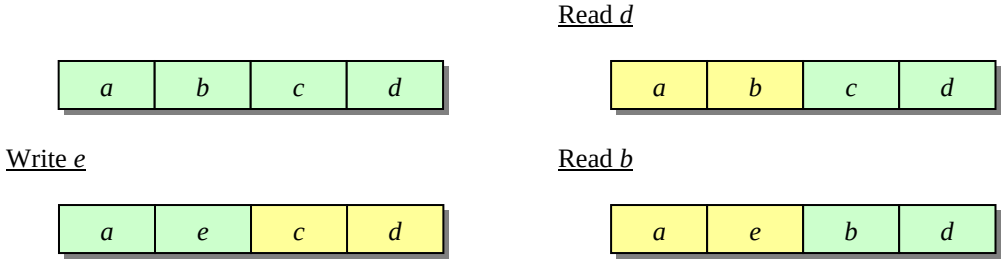

Figure 12: 1-bit sequence

[a] There is only one for the pseudo-LRU algorithms but $N_{ways}/2$ for 1-bit policy and $1..N_{ways}$ for Side

The simplicity of this policy is obtained at the price of the performance. For high associativity (greater than 8), this policy performs as bad as Random. Nevertheless, for low associativity, its results are good (5-10% worse than LRU), rising it up as a credible candidate.

4.2. MRU based Pseudo LRU (PLRUm)

In this approximation, each block is assigned a MRU bit. When the cache controller replaces a line, it switches the MRU bit 0 to 1. If all the MRU bits are equal to 1 after the modification, all the bits except the last accessed are reinitialized to 0. According to [ZOU04], PLRUm and PLRUt are very good approximations of LRU. PLRUm outperforms PLRUt and is even better than LRU for some patterns. It explains that PLRUm has been used in IBM computers (for instance IBM3033). Consequently, it appears as one leading candidate (cheap, efficient, easier to implement) for embedded applications. This algorithm can be conceptualized by a finite state machine, as Fatemi *et al*. proved [FAT94]; thereby leading to different possibilities for the design of such a system. A sequence is drawn on the figure below.

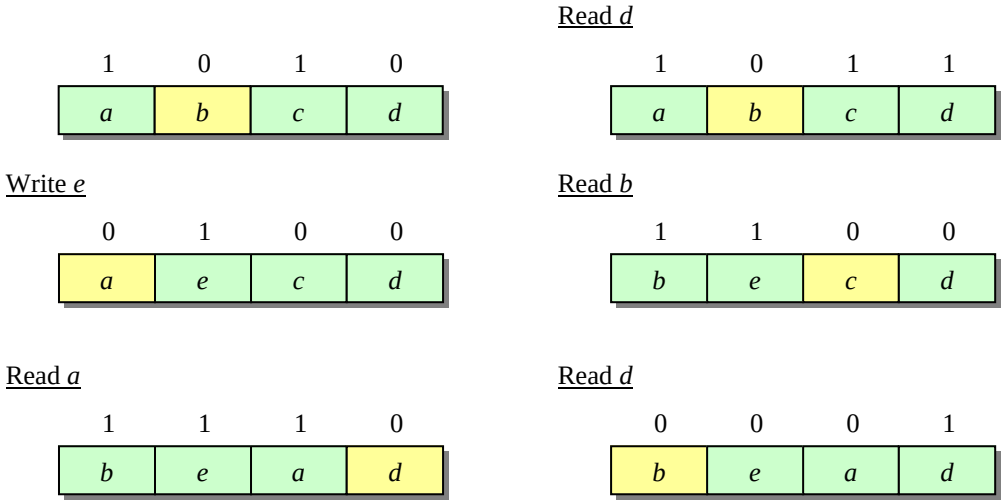

Figure 13: PLRUm sequence

4.3. Tree-based Pseudo LRU (PLRUt)

This binary tree approximation of the LRU algorithm [KAR94] requires *N-1* bits in an *N*-way associative cache. Such a tree is drawn on Figure 14.

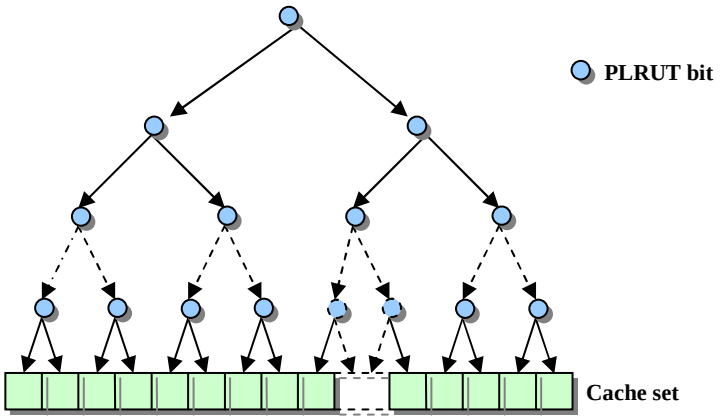

Figure 14: Decision tree for the PLRUt algorithm

The tree bits encode the paths towards the leaves corresponding to the different ways of the set. Reading them points to the pseudo LRU line. On hit, the bits on the path towards the hit line are inversed to indicate the opposite part of the tree as pseudo LRU. The idea behind is to protect the last accessed data from eviction by inversing the nodes towards it. For instance, let us take a look at the example of a 4-way set associative cache, where the initial data in the set are *a*, *b*, *c* and *d*, which points

to *d* as the pseudo LRU way (see Figure 15). The tree bits and the contents of the set are drawn *after* the memory access occurred. Let us take the example of the third instruction `Read` *b*. Data *b* lies in way 1 of the cache thus the path goes through the top node and the left leaf one. These bits are modified to point to the other halves, i.e. the top bit points to the right half and the leaf bit to its left half. The principle is the same for all the other instructions of the sequence.

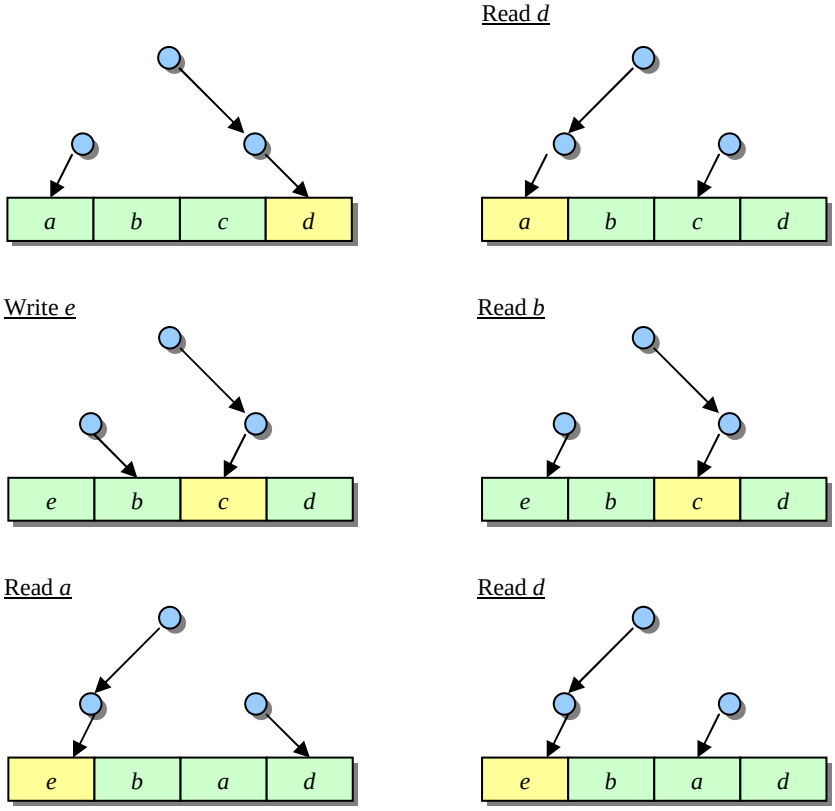

Figure 15: PLRUt sequence

The disadvantages of this algorithm come from its binary nature, which makes it simple. The node at the top of the tree contains only one piece of information (one bit) and cannot reflect sufficiently exactly the history of the leaves of the tree: using a bit implies a loss of history. Another problem is that decoding the bits is a sequential process. Indeed, the nodes must be stridden from the upper node downwards to a leave. Designers are thus prevented from performing parallel computations. However, it is widely used in data caches thanks to its high induced hit ratio (Al-Zoubi *et al.* reported a miss ratio 1-5% worse than LRU [ZOU04]).

4.4. Modified Pseudo LRU (MPLRU)

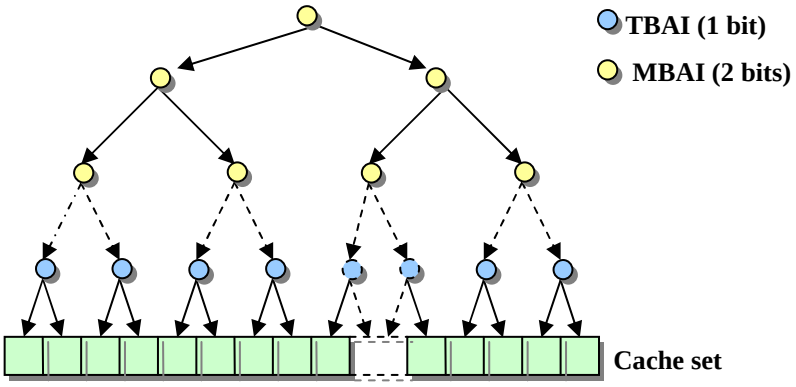

Figure 16: Decision tree for the MPLRU algorithm

Ghasemzadeh *et al.* [GHA06] introduced an algorithm which is supposed to partially solve the issue of history loss presented above about PLRUt. They split the nodes in two groups: the ones that keep

information about two leaves of the tree (the Two Block Access Indicators) and the ones which give information about more than two blocks (the Multiple Block Access Indicators). Two bits model the MBAIs: one for the previous status and one for the current status. TBAIs are equivalent to PLRUt tree bits. On a miss, previous bit is used to determine which line to discard. The resulting tree is drawn on Figure 16.

For a 32KB 4-way set associative cache, the authors find an average miss rate of 2% for MPLRU whereas around 2.1% for PLRUt. This figure seems to increase with cache size and associativity. MPLRU performs around 8.5% better than PLRUt while keeping the low overhead of the latter. Compared to PLRUt, it has few more operations like copying bits but their amount is quite insignificant. The major negative impact is thus the storage of one more additional bit per MBAI, which corresponds to a single bit per set for a 4-way set associative cache. At first glance, it is an acceptable amount but it should be kept in mind that this will impact on all the cache sets. Therefore, this could significantly affect the power consumption. The final decision about the relevancy of this algorithm will then be made with respect to the results of simulations.

4.5. SIDE algorithm

Deville [DEV90] introduced a new policy which allies the simplicity of FIFO with an almost LRU performance. It has one counter per set like FIFO but the update of these counters is usage based, thereby improving the performance of FIFO. The counter points to an element among the least recently used. Let N be the set associativity. If there is a hit on element i, the counter c is updated to $(i+1)$ mod N if $i=c$ and remains unchanged otherwise. On a miss, c is updated to $(c+1)$ mod N. Consequently, this counter partitions the set in two groups: the possibly MRU on the left, the surely LRU elements on the right. The discarded line is the one pointed by the counter, i.e. the possibly LRU line located at the frontier with the surely LRU group. All the information is lost at each beginning of a phase, when the counter is reset to 0 (all the ways are identical for the replacement algorithm). The hardware implementation shows that it can be designed to work also in FIFO mode. The additional logic compared with a FIFO is an N-bit-comparator and an N-bit-encoder, which are shared by the sets.

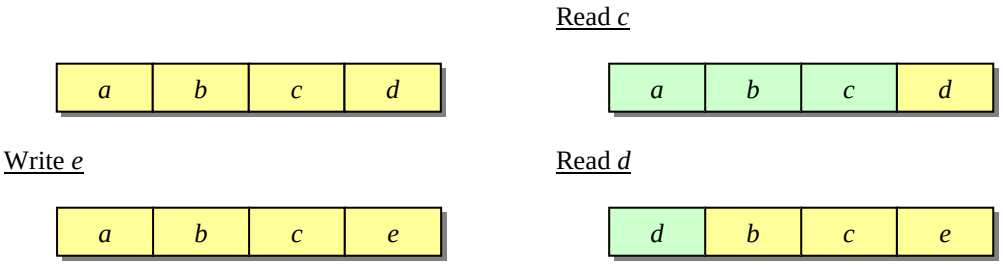

Figure 17: Side sequence

On a 4KB cache, the improvement is around 10-15% for a 4-way-set-associative. This figure is strongly dependent on the associativity and the algorithm performs almost as well as LRU (only 0-5% worse) on low associativity while being much simpler to implement.

4.6. Comparison of these policies

We have seen a lot of different policies, understood their principle of working but let us now sum up everything to compare them. This is done in Table 1.

5. Enhanced LRU policies

In Section 3.2, it has been shown that LRU performs far from optimized on some patterns, although its results are good on average. Besides, LRU is badly efficient on burst access sequences: data that are accessed only once are cached and take the place of lines that may be useful. To solve these issues, improvements have been developed; among them some could be implemented on pseudo-LRUs but, up to now, I have found no data about such an attempt. These algorithms are thus very useful for disk caches or file copying when a lot of burst accesses are necessary. All the policies presented below are based on the same idea: partition the cache in two parts. One part contains the data referenced once; the other the data accessed at least twice. The relative sizes of the partitions are dynamic or fixed, depending on the algorithms. The flows between the two pools are specific of each algorithm too.

Policy	*Bits required*	*4-way*	*What to do when hit on i?*	*What to do when miss?*	*Logic required (only estimations)*	*Performance to LRU*
Random	$\log_2(N_{ways})$	2	-	Update LFSR	LFSRs and XORs to generate the random	22% worse [ZOU04]
FIFO	$N_{sets}.\log_2(N_{ways})$	2. N_{sets}	-	Increment FIFO counter	Incrementing logic and N_{sets} $\log_2(N_{ways})$-bit-counter registers. Low overhead in principle.	12-20% worse [ZOU04, SMI83]
LRU	N_{sets}. $N_{ways}.\log_2(N_{ways})$	8. N_{sets}	Update the LRU stack	Update the LRU stack	Can be implemented in a matrix form. Seems too intricate and computation time consuming for embedded systems	-
PLRUm	N_{sets}. N_{ways}	4. N_{sets}	Update the MRU bits	Update the MRU bits	Around N_{ways}. N_{sets} registers (for the MRU bits), N_{ways}. N_{sets} muxes, 3.N_{sets}. $\log_2(N_{ways})$ AND/OR	1% worse to 3% better [ZOU04]
PLRUt	$N_{sets}.(N_{ways}-1)$	3. N_{sets}	Update the tree bits	Update the tree bits	Around $N_{sets}.(N_{ways}-1)$ registers, $N_{sets}.(N_{ways}-1)$ Muxes, $N_{sets}.(N_{ways}-1)$ inverters, N_{sets}. $\log_2(N_{ways})$ OR	1-5% worse [ZOU04]
MPLRU	$N_{sets}.(3N_{ways}/2-2)$ [Appendix A]	4. N_{sets}	Update the tree bits	Update the tree bits	Around $N_{sets}.(3N_{ways}/2-2)$ registers, $N_{sets}.(3N_{ways}/2-2)$ Muxes, 2* $N_{sets}.(3N_{ways}/2-2)$ AND/OR	1-4% worse [GHA06]
SIDE	N_{sets}. $\log_2(N_{ways})$	2. N_{sets}	Update the counter c to $(i+1)$ mod S if $i \geq c$ Nothing if $i < c$	Increment counter and random selection	N_{sets} $\log_2(N_{ways})$-bit-registers for the counters. N_{ways} -bit comparator and incrementation. LFSRs and XORs to generate the random during the discarding selection.	0-20% worse. Strongly dependent on associativity. On low associativity (2-8), 0-5% [DEV90]
1-bit	N_{sets}	1. N_{sets}	Update the bit	Update the bit	LFSRs and XORs to generate the random in the LRU partition, N_{sets}. N_{ways} OR/AND	10-20% worse for high associativity; 5-10% worse for low [SO88]

Table 1: Comparison of the different simple replacement policies

Policy	*Lists*	*User defined parameters*	*What to do when hit on i?*	*What to do when miss?*	*Complexity per request*	*Performance to LRU*
2Q	1 LRU list and 1 FIFO list split into two sub lists	K_{in} and K_{out}	Move it the MRU side of the LRU list Am. If it was in A1, remove it from A1.	Move the line to the back-end of the FIFO A1in. Move the discarded line to the FIFO A1out. Discard line from this FIFO.	Constant	5-10% better [JOH94]
LRU-K[a]	*K* history lists	CIP[b]	Update the CRP[c] of the data and its *K* history lists.	Select a victim by exploring the list of 1st and *K* access. Initialize the CRP and the *K* history lists of the data.	Logarithmic time (because of its priority queue)	Around 50% better for very large database buffers [ONE93]
SLRU	2 LRU lists of fixed size $c_1+c_2=c$	PbS size	Move *i* to the MRU position of the PtS[d]. Move the LRU line of the PtS to the MRU position of the PbS[e] if necessary. Discard the LRU line of the PbS if necessary.	Move the line to the MRU part of the PbS. Discard the LRU line of the PbS if necessary.	Constant	Around 5% better [KAR94]
ARC	2 LRU lists of size *c*	-	Move *i* to the MRU position of T2	Miss on the global list: Delete the LRU page in B1 if $\|\|T1\|\| < c$, in T1 otherwise. Move *i* to the MRU in T1. Hit on the global list: Update *p* (addition and min/max operations) and move *i* to the MRU position of T2.	Constant	50-200% better [MEG03]

Table 2: Summarize of the enhanced LRU policies

[a] The performance figures are here given for $K=2$
[b] Correlation Information Period: approximately the time a data accessed only once should stay in the cache.
[c] Correlated Related Period: the time since a given data has not been accessed
[d] Protected Segment: segment of fixed size which contains the element accessed at least twice.
[e] Probationary Segment: segment which contains data that have been accessed only once recently.

5.1. 2Q

The two queues presented above have fixed size in this implementation. The first queue, which contains the data accessed once, is treated as a FIFO queue and the second one is managed as a LRU stack. The first queue is also partitioned in two queues (A1in and A1out) because experimental results showed that the optimal size of the first queue strongly depends on the trace. This can be considered as a trick to dynamically adjust the fixed size of the queue. K_{in} and K_{out} are respectively the size of $A1_{in}$ and $A1_{out}$. The reported improvement over the LRU reaches 5-10% [JOH94]. However, the problem is that two queues as well as migrations from one to another must be managed, which is hardware expensive and cycle consuming.

5.2. LRU-*K*

In this algorithm, O'Neil *et al*. [ONE93, ONE99] split the cache in *K* different pools which correspond to the lines that were accessed between 1 and *K* times quite recently (in reference with a backward *K* distance). The idea is basically the same as 2Q: keeping track of the history to predict the next accesses. A history list of the *K* last accesses of each data accessed recently[a] should be maintained and would be logic-consuming. On a miss, the victim is chosen by exploring these lists and finding the data which has not been accessed quite recently and whose last K^{th} access was the LRU. Nevertheless, this algorithm seems to be really efficient only for big sizes (for instance for disk buffering). A simplified version may however be efficient also for embedded caches too. The issue, which is common to all the algorithm presented in this section, is to estimate the importance of these patterns on a L1 data cache and hence the need of such improvements for embedded caches.

5.3. Segmented LRU (SLRU)

As we have seen, LRU caches can be filled by lines which are accessed only once, thus discarding from the cache lines which should be useful. The principle here is similar to LRU-*K* but seems easier to implement. The cache is divided into two segments: the protected segment and the probationary segment. On a miss, data is then pended on the MRU part of the probationary segment. Hits are added to the MRU part of the protected segment. As the protected segment has a definite size, adding a line into the protected segment pushes the LRU line of the protected segment to the MRU part of the probationary segment. This method avoids flooding the cache with data that will not be reused, because the protected segment contains lines which have been accessed at least twice. The supplementary storage is thus one bit per each cache line (a flag which indicates whether the line belongs to the protected or the probationary part of the cache). However, the hardware needed to handle the two queues should not be forgotten. Globally, there is one pointer that marks the border between the two zones. There is extra maintenance due to moving the lines in the list on a hit. The best results were obtained when the size of the protected segment is around 80% of the cache. It performs around 3-4% better than LRU for a cache size of 0.5 Mb [SO88].

5.4. Adaptive Replacement Cache (ARC)

The algorithm designed by Megiddo and Modha [MEG03] maintains the history of 2*c* lines where *c* is the cache size. This 2*c*-history is divided in two subgroups: L1 and L2 of length *c*. L1 contains the lines recently referenced once and L2 the lines recently referenced at least twice. These two subgroups are then dynamically split in two subsets T(op) – which contains the MRU part – and B(ottom) – which contains the LRU part – so as $||T1 \cup T2|| = c$. A parameter *p* represents this partition: *p* can be seen as the target size of the list T1. At a given time, the ARC algorithm performs as a fixed replacement policy which keeps *p* lines in T1 and *c-p* in T2. This parameter *p* is dynamic and tuned in function of the demand in order to "invest" in the most active list. The increment and the decrement steps of *p* depend also on the respective size of the sets B1 and B2.

This algorithm performs as well as the fixed replacement policy with the optimal *p* but it is dynamic and no parameter needs to be tuned before and hence should perform the same way through all workloads and cache parameters, contrary to the other policies presented above. It is scan-resistant, has

[a] This recency is defined by the Correlated Information Period, which is an *a priori* defined parameter.

a constant overhead (with contrast to the logarithmic overhead of the LRFU and LRU-*K*). For instance, on a SPC1 benchmark, the hit ratio of ARC was 23.82% and only 4.24% for the LRU but with a cache of 4GB. The space overhead seems to be small, 0.75% of the cache according to the authors. Nevertheless, the huge cost in hardware counterbalances the first optimism and assigns this solution only for desktop systems and possibly for some L2 embedded caches.

5.5. Summary of the enhanced LRU policies

We have seen a lot of different enhanced policies, understood their principle of working but they are quite complex. As it was done for the pseudo-LRU policies, we summarize their principle and performance in Table 2. The common point of these policies is their hardware cost. While LRU needed to handle one queue, they introduce two stacks with floods between them. This should require too much hardware for embedded L1 caches. As a result, they will not be studied further but were addressed here to give some knowledge about enhancements of LRU.

6. Ideas of improvement

Some other algorithms, which are not based on LRU, were proposed. Some of them present interesting features which could be used in further developments of cache policies.

6.1. Cacheable/Non Allocatable

Tyson *et al*. [TYS95, TYS97] present a technique to improve the replacement algorithms using Selective Cache Line Replacement. The authors point out that a large percentage of data misses are caused by a small number of instructions (less than 5% of the load instructions cause 99% of cache misses). The idea is thus to mark these instructions as C/NA (the load instructions and not the data!). The decision of allocating the data is then taken in respect with this flag. They develop static and dynamic method.

- *Static method:* instructions that cause a miss higher than 75% are marked CNA. It is useless for us because it requires pre-runs of the program,
- *Dynamic method:* each load instruction has a 2-bit counter, which is incremented at each miss and decremented at each hit. On a hit, the instruction that brought the cache line in cache is also decremented. When it reaches “11”, the load instruction is marked as C/NA. It gives an average improvement of 20% on bandwidth requirements (for the SPEC2000 benchmarks) but is insignificant for the hit rate (less than 1% on average).

6.2. Selective cache way

Inoue *et al*. [INO99] proposed an implementation where the hardware tries to predict which way will be used and gives power only to this predicted way. It saves power but if the prediction is erroneous, a cycle is lost. MRU (a 2-bit flag for a 4-way associative cache) is frequently used to predict the way. Reading the MRU bits before accessing the cache makes the cache access longer but it can be performed in an earlier pipeline stage. On the other hand, it decreases access time because there is no way selection delay.

Various replacement policies were presented in this chapter. Balancing the estimated hardware cost and the power consumption with their expected performance led us to select a class of replacement policies in respect with the constraints of embedded systems: the pseudo-LRU algorithms. The latter will be simulated on the ARM11 architecture to compare their theoretical performance with the simulated one. For that end, a study of the ARM11 data cache side is beforehand required.

Chapter 4

The current cache implementation

Any sufficiently advanced technology is indistinguishable from magic.

A. Clarke, *Profiles of the Future: An Inquiry into the Limits of the Possible*

The different cache lines replacement policies studied in the previous chapter raised the issue of the hardware implementation. Before dealing with it in details, the main features of the ARM11 architecture, which will be useful in this thesis, are addressed. In this work, the target processor is an ARM11 core, so the presented microarchitecture is the ARM11 one, an implementation of the ARMv6 instruction set architecture [ARM11, COR02].

1. The ARM11 microarchitecture

1.1. Architecture vs. microarchitecture

The *architecture* is the general description of the behaviour of the microprocessor, its interface with the outside world, but without specifying the internal design. So it can be seen as a description of the instruction set and of the programmer's model.

The *microarchitecture* is the detailed definitions of the internal design and hardware, which support a given architecture. These specifications usually concern points that are invisible to the programmer.

For the ARM11 MPCore processor, the architecture is ARMv6.

1.2. Memory

The ARM processors must work with a *byte*-addressed memory and suppose the alignment. Double word, word, half-word and byte accesses are supported. During the three first accesses quoted above, bits [0:2], [0:1] and 0 are respectively ignored. Usually, the main memory is cacheable and bufferable. The memory system endianness and the ARM processor should match one another or be configured for that aim in CP15 register 1.

1.3. System Control Coprocessor

All the memories and the system features are controlled by coprocessor 15 (CP15), also known as the System Control Coprocessor. It contains up to sixteen 32-bit primary registers, whose permission (read-only, write-only or read-write) depends on their functionality.

The characteristics of the caches are accessible through register 0 of the system control coprocessor. This register indicates the type (unified or separated), the size of the data and instructions caches, the write policy (write-through or write-back) and the cache associativity.

Caches, MMU, write-buffers, branch prediction and replacement strategy can be enabled or disabled on register 1 of the System Control coprocessor. For more information about other registers of CP15, one can read Chapter B2 of *ARM Architecture Reference Manual* [ARM01].

1.4. Memory Management Unit (MMU)

The MMU converts the virtual address into a physical address. It is also responsible for controlling whether a program is allowed to access a memory area. This permission depends on the running mode (User, Privileged) and ensures improved security of the system.

1.5. Generalities about caches

The caches are separated for data and instructions and no coherency is implemented. A detailed examination of the ARM11 MPCore level1 data side memory system will follow this brief overview of the main characteristics of the data cache in the ARM11 MPCore processor.

1.5.1. Data cache characteristics

The data cache is available in three configurations:

- 64 KB 4-way set associative cache: 512 sets[a], 9-bit index, 18-bit tag, 5-bit offset,
- 32 KB 4-way set associative cache: 256 sets, 8-bit index, 19-bit tag, 5-bit offset,
- 16 KB 4-way set associative cache: 128 sets, 7-bit index, 20-bit tag, 5-bit offset.

The data cache is write-back write-allocate.

1.5.2. Replacement policy

One of the most important characteristics of the data cache for the present study is the replacement policy. In ARM11 MPCore, an altered version of the Round Robin algorithm is implemented. It will be denoted Global Round Robin (GRR) in this thesis to distinguish it from the usual Round Robin. Two major design decisions distinguish it from the literature algorithm. First, it cares about the presence of free ways in the set before allocating a way. In a unique processor environment, this does not enhance the overall performance. Yet, it should give a significant improvement in multi-processor systems due to the invalidation of shared lines after writes, which creates free lines disassociated from the Round Robin counter. The second and the most significant difference is the global nature of the counter which is not owned by a set anymore. This ensures using only a 4-bit register to save the counter but at the expense of a loss of performance. Nevertheless, this is acceptable (see Chapter 5) and justifies the current implementation.

1.5.3. Non-blocking misses

One of the most interesting features in the data cache management of the ARM11 microarchitecture is the non-blocking and hit-under-miss operation. It allows the processor not to stall, if a memory request results in a cache miss. The cache immediately issues a line fill request and waits for the data to be fetched from the upper level memory. The pipeline goes on and performs the next instruction, provided that there is no dependency between the instructions: the processor is not blocked. If the instruction is a Load, the microarchitecture avoids processor's stalling if the look-up yields a hit (*hit-under-miss*): the data cache handles the lookup while waiting for the line fill. Up to two successive cache read misses are supported [MPC05] before stalling the processor.

1.5.4. Lockdown blocks

The cache improves in general the performance of memories but also worsens the worst case: as it has been stated in Eq.2, an instruction resulting in a cache miss is executed in *Average cache access time + Average upper level access time* and not only *Average upper level access time*. This overhead for misses can be problematic for some programs whose frequently accessed data are evicted from the cache whereas useless or simply unimportant data still pollute it. This issue can be solved by locking the ways where these critical data are stored: the replacement algorithm will never discard the lines located in these ways. As a result, these crucial data are protected from eviction, thereby improving the

[a] The number of lines, index width, tag width and offset width are derived from equations of section 2.5.3

overall performance. In ARM processors, only ways and not parts of cache sets can be locked. This is performed in two phases:

- modifying CP15 register 9 (data cache lockdown register) to lock ways 0..*W*,
- fetching the data in the cache. If there are not any free ways, the data is allocated to (one of) the locked way(s).

Once this is completed, the replacement algorithm will not discard ways 0..*W* until CP15 register 9 has been modified.

1.6. ARM11 vs. competitor configurations

An overview of the solutions chosen up-to-now by different leading companies is drawn in Table 3. One can see the predominance of *N*-way set associative caches among the present configurations, with $N = 2$ or $N = 4$ for the embedded processors. These figures agree with the different configurations available on ARM processors, which are 4-way set associative. This quite low figure is explained by the overhead of the cache lookup for higher associativity, which significantly increases the access time, the area and the power. Four appears as a quite good compromise, as it will be demonstrated in Chapter 5 Section 4.

The ARM11 block size lies in the range [32:64] of typical values for embedded processors. This figure can be explained by the will to take advantage of the locality principle and not to pollute the cache with neighbour data which will not be accessed soon.

LRU and pseudo-LRUs are widespread in the desktop systems thanks to the high induced hit rate. However, they have a negative impact on area and power consumption. Nowadays, this class of replacement policies enters the embedded world too. The introduction of a system-on-chip L2 platform based on a pseudo-LRU replacement policy by MIPS is one example [MIP06]. Therefore, integrating a modified low-power version in ARM processors must be investigated and considered as one of the permanent ARM efforts aiming better performance in the embedded world without sacrificing power and area.

The major technical features of the ARM11 microarchitecture have been addressed. A detailed examination of the ARM11 data cache management is now needed. The following section deals with it.

2. The ARM11 MPCore level 1 data side memory system

The data side memory block organization is drawn on Figure 18. The different modules and their role are described below.

2.1. Slots Unit

The Slots Unit is responsible for handling the memory requests from the core. It is composed of three slots, each of them being in charge of a single memory access[a]. When a request arrives, the micro-TLB converts the virtual address to a physical address and gets the protection and cache attributes. In respect with these attributes, the slot then generates an abort if needed. In case of a write, the request is sent to the Store Buffer and the slot is freed. If the Store Buffer is full, the request waits in the slot until the Store Buffer accepts it. If the access is a read, the slot asks the arbiter for an access to the RAMs. If the lookup results in a miss, a line fill request is sent to the Line Fill Buffer. Otherwise, the data is forwarded to the processor through the Droute module.

The Slots Unit also computes the sequentiality of the access. The information is then used to power off the four Tag RAMs and some Data RAMs. Indeed, if the first lookup of the burst access succeeds, the slot is aware of it on the second sequential access. Knowing that the data lies in the cache, only the Data RAM containing the requested data is enabled (the other RAMs are disabled). This feature ensures saving much power.

[a] It is a single access and not a single memory request. Indeed, processor's memory requests can be merged (for instance two successive reads at the same line) or can induce more than one memory access (for instance instructions like LDMIA, STMIA…)

	AMD Athlon [AMD02]	Hitachi SH3-DSP [SH3]	Intel Pentium III [HEN03]	Intel Pentium 4 [INT06]	IBM PowerPC 405 CR [HEN03]	Sun UltraSparc III [SPA03] [HEN03]	DEC Alpha 21064 [WIK06] [HEN03]	DEC Alpha 21364 [WIK06] [HEN03]	Freescale PowerQuicc III [GEN04]
Instruction architecture	80x86	SuperH	80x86	80x86	PowerPC	SPARC v9	Alpha	Alpha	PowerQuicc
Intended application	desktop	Communication, embedded	desktop	desktop	embedded	server	Workstations, servers	Workstations, servers	Wireless, embedded
Instructions/ clock	3	?	3	3	1	4	1/2.51	1/0.6	?
Clock rate	1400 MHz		900-1200 MHz	3 GHz	266 MHz	600-900 MHz	100-200 MHz	800-1300 MHz	1.3-1.5 GHz
Instruction cache / Data cache	64 KB 2-way/64 KB 2-way divided into 8 banks	16 KB 4-way unified Way 2 and 3 lockable	16 KB 2-way/ 16 KB 2-way	~ 96 KB / 8 KB 4-way	16 KB 2-way/8 KB 2-way	32 KB 4-way/64 KB 4-way	Direct mapped 8 KB / 8 KB	64 KB 2-way/64 KB 2-way	32 KB 8-way/ 32 KB 8 way
On-chip L2 cache	256 KB 16-way (exclusive)	-	256-2048 KB 8-way	256 KB 8-way (not inclusive)	-	-	-	1536 KB 6-way	256 KB 8-way
Off-chip L2 cache	-	-	-	-	-	8 MB 1-way	2 MB 1 way	-	-
Block size (bytes) L1/L2	64		32	64/128	32	32	32	64	32/32
Replacement Policy	LRU	LRU	Pseudo LRU	Pseudo LRU	?	?	-	?	Pseudo LRU
Write update Policy L1/L2	Write-back	Write-back and write-through	Write-through/ Write-back	Write-through/ Write-back	?	Write-through no-write allocate	Write-through	Write-through	Write-back/ Write-through

Table 3: Cache characteristics of commercialized processors

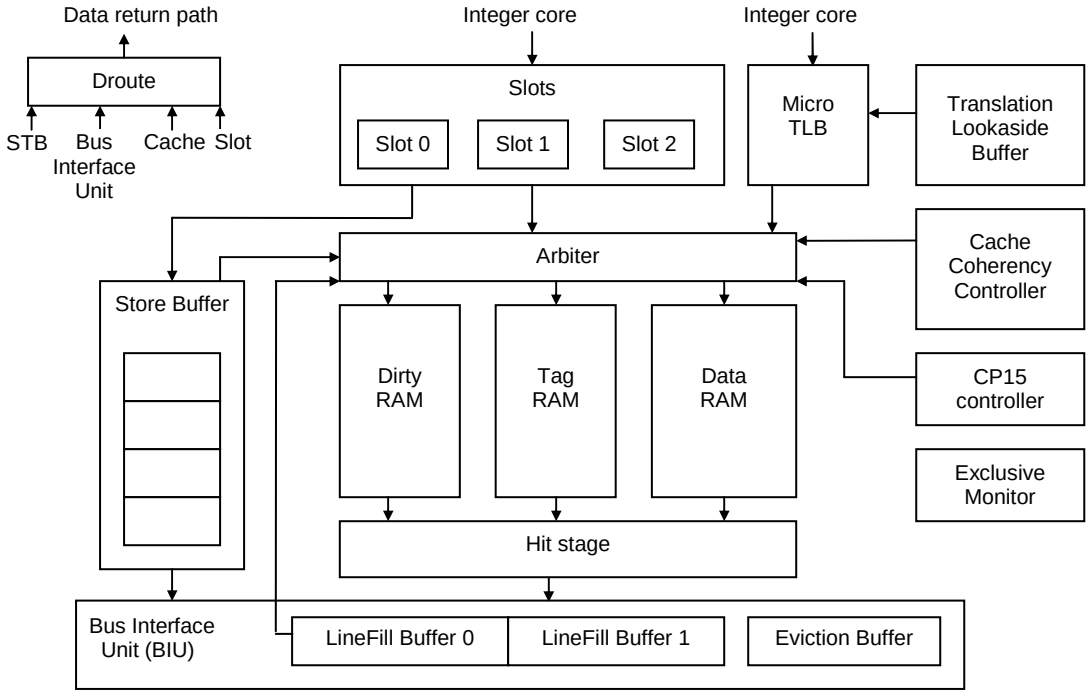

Figure 18: Level 1 data side memory implementation of the ARM11 MPCore processor[a]

2.2. Micro Translation Lookaside Buffer

The micro-TLB (micro Translation Lookaside Buffer) is an 8-entry fully associative cache of the main TLB. The lookup is performed very quickly and the physical address and page attributes are available in the same clock cycle as the arrival of the virtual address. In case of a miss, the address is sent to the main TLB which will handle it.

2.3. Arbiter

The arbiter deals with the accesses to cache lines. Among the requesting modules, a single one is granted access, in respect with fixed priorities. It also transmits the information (particularly the address) to the Dirty RAM, the Tag RAMs and the Data RAMs. The different requesting modules are the two Line Fill Buffers, the Eviction Buffer, the Store Buffer, the Slots Unit, the CP15 controller and the Cache Coherency Controller Block.

2.4. RAMs

In the current implementation, the RAMs can be regrouped in three types: the Tag RAMs, the Data RAMs and the Dirty RAM. All the RAMs are physically addressed: the cache is said to be PIPT (Physically Indexed Physically Tagged).

2.4.1. Tag RAMs

The Tag RAMs store the tag of the different cache lines as well as their validity bits. Since the cache size is up to 64 KB, the Tag RAMs comprise up to 512 lines of 22 bits (20 tag bits, 1 validity bit and 1 MESI exclusive bit).

2.4.2. Data RAMs

It stores the data and is organized in the same way as the Tag RAMs. Its capacities are up to 512 lines of 32 bytes (8 words).

[a] The original picture from [MPC05] has been modified so that it only presents the essential modules.

2.4.3. Dirty RAM

The Dirty RAM contains the information of a set: dirtiness and MESI protocol bits. One line is 24-bit wide and stands for the information of the four ways of a given index.

2.5. Hit stage

This is the module responsible for computing the hit/miss information. This information is returned to the module which was granted access by the arbiter. In case of a miss, the evicted way is also passed on to the requesting module.

2.6. Store Buffer (STB)

The Store Buffer is composed of four slots. Each slot has space for the 64-bit data and for the 32-bit physical address. It receives the write requests from the Slots Unit and merges them with the existing one when it is possible, thereby avoiding numerous cache accesses. If the request is non-cacheable, the write is transmitted to the Bus Interface Unit. If it is cacheable, the Store Buffer requests an access to the RAMs for the lookup. If a miss occurs, the slot requests for a line fill. In case of hit, the slot asks for a new access to write the data: in overall, at least two cycles are necessary to handle the request. The overhead of writes over read does not really matter because the reads can hit in the Store Buffer. Indeed, from the point of the view of the core, everything happens as if the data is already written in the cache.

2.7. Line Fill Buffer (LFB)

Two 256-bit Line Fill Buffers are responsible for fetching a line from the upper level memory. They are endowed with the merging ability, which allows saving power and bus traffic. Another important enhancement is the LFB hit ability: a lookup resulting in a miss can hit in one LFB and the data will be forwarded to the processor as soon as possible. When the words are received from the upper level memory, they are streamed to the CPU; preventing it from waiting that the whole line has been fetched to get the requested data. Once the line fill has been completed, the LFB asks the arbiter for an access to the RAM. When granted, it writes the whole line in one clock cycle: tag, data and attributes.

2.8. Eviction Write Buffer (EWB)

This buffer receives the line that has been discarded from the cache. Then it requests access to the bus and writes back the data in the upper level memory.

2.9. Droute

It only selects the valid data from the different sources and returns it to the core. It can be considered as a multiplexer.

In this chapter, the different modules which are involved in handling a data cache request were presented. With this technical background, the issue of getting results about efficiency of the different replacement policies can thus be discussed.

Chapter 5

Replacement policies simulation

In der Praxis muß der Mensch die Wahrheit, d. h. die Wirklichkeit und Macht, seines Denkens beweisen.[a]

K. Marx, *Thesen über Feuerbach*

In order to compare the replacement policies and their efficiency for ARM processors, a cache simulator has been developed. A detailed interpretation of the simulation results will follow the presentation of the simulator assumptions and of its implementation. It will allow us to select the replacement strategies whose hardware realization will be carried out in the next chapter.

1. Principles

1.1. Source files

The cache simulator and its replacement policies were written in C. For further details of the code, one can see Appendix B.

1.2. Choice of the policies

The cache simulator affords different replacement policies: 1-bit, Global Round Robin, Modbits, MPLRU, non-MRU, Optimal, PLRUm, PLRUt, Random, Round Robin, Side and true LRU. They can be divided into two subgroups: the academic policies already presented in Chapter 3 and some additional algorithms simulated to confirm or infirm some hypotheses.

1.2.1. Academic policies

These algorithms were chosen because they are widely used (particularly in desktop applications), should enhance the cache performance significantly and should not cost so much in terms of gates and power, which is a major point of concern for the ARM embedded processors. This selection is based on the study presented in the previous chapters, particularly on Table 1.

The algorithms may not correspond exactly to the literature because they were implemented in the way they will be used in the ARM processor. Indeed, the simple modification of calling the replacement policy only when there is no free ways yields improved performance for the Random algorithm. The aim of the thesis being to implement the most efficient algorithms, this solution is simulated in this chapter. Section 3.1 demonstrates that this enhancement is at the origin of the slight difference observed between the results published in the literature and the ones obtained in this work. Besides, the ARM architecture differs a bit from the x86 one, which is the tested architecture in most articles. It partially contributes to the reported deviance too.

[a] *Man must prove the truth - i.e. the reality and the power - of his thinking in practice.*

1.2.2. Additional policies

Modbits and non-MRU are quoted for the first time in this work. They do not come from the literature but are ideas for costless replacement policies. Like the other replacement strategies, both policies rely on the locality principle. They also helped verifying some hypotheses and better understanding the behaviour of the replacement algorithms.

Non-MRU. Like 1-bit, non-MRU's aim is to protect the most recently used line from eviction. Contrary to 1-bit, it stores the MRU way and therefore requires $\lceil \log_2(N_{ways}) \rceil$ storage bits per cache set. The expected performance is slightly better than 1-bit since it does not induce any collateral protection. Nevertheless, it can be balanced by positive effects of the spatial locality provided that the MRU region is wider than a way. This assumption is in accordance with the work of So *et al.* [SO88].

Modbits. Modbits deduces the recently accessed ways from the dirty bits of a set. Indeed, it considers the dirty bits as an approximation of the LRU information and examines them in the same way as the MRU bits for PLRUm. Using the dirty bits creates a discrepancy between the writes and the reads; which can lead to some strange patterns. However, the cost of this policy is negligible because it requires no additional bits. Yet, it is at the expense of continuously accessing the dirty bits. Depending on the cache implementation, this access can be highly expensive in terms of power, turning on a RAM for instance. The behaviour of the simulations will help making decision about this algorithm.

1.3. Cache simulator

A cache simulator was designed in order to get clues about the efficiency of the different replacement algorithms. The hypotheses of the written model are depicted below.

1.3.1. Physical parameters

First, the cache characteristics (cache size, number of ways) can be modified in the configuration file and are given as a range: the program will simulate all the legal cache configurations corresponding to these ranges. The block size and the line width are fixed settings that filly with the ARM11 implementation. The other cache attributes (number of lines, amount of sets…) are inferred from these basic ones. In order to be as close as possible to real cache's behaviour, the latency is also implemented: the time needed to get a line from the upper level memory (L2 cache or main memory) is simulated. Only when the data is available, the previous cache block is discarded; i.e. the evicted cache block is available while fetching the data from the upper memory. This latency is characterized by three parameters:

- the transfer time t_{L2} from L2 cache,
- the transfer time t_m from the main memory,
- the average hit rate h_{L2} of L2 cache.

The set L2 cache – main memory is modelled as a single module which forwards the requested data in a time t_{L2} with a probability h_{L2} and in a time t_m with a probability (1- h_{L2}).

As it will be seen in the next chapter, a considered implementation of the replacement algorithms requires a fully-associative cache of the RAM storing the status bits[a]. The physical parameters describing the cache are its number of lines and its number of elements per cache line. The behaviour of this cache – particularly its hit ratio – was also investigated to estimate the efficiency of this solution. Like the latency time to fetch a data from the upper level memory, this specialized cache can be enabled or disabled in the configuration file. However, its results will be presented in the next chapter.

1.3.2. Inputs of the simulator

The inputs feeding the cache simulator are memory requests from an ARM11 core, on which different benchmarks run. Because of the huge size of the log files, the original TARMAC [SHA01] disassembler was modified to lighten log files, which supply the very single information that we need:

- time when the core requests a data from the memory,
- type of the access (read or write),
- address of the wanted data.

[a] Status bits is the name given in this thesis to the bits required by the replacement policies. Their name comes from the fact that they encode the status of a cache set for eviction.

From these information, the memory request patterns which will feed the cache simulator are deduced, thus preventing from running time-consuming SystemC simulations of the core for each policy but only once for each benchmark. Of course, the way the cache reacts can impact on the memory request pattern from the core but it should be negligible for the scope of this study considering that the difference can only be seen for the requests whose lookup result has been changed and for which the core is stalled. Yet, the tested core is provided with the hit-under-miss[a] feature, so this situation seldom occurs. It must be noticed that the data itself does not matter because data values are needed by the core itself, which is simulated as a fixed parameter in this model in the sense that the modifications of the data cache implementation do not influence the behaviour of the processor in a first approximation. Therefore, the model addressed in this section is minimal: only the very necessary information and parameters are taken into account.

The MMU was turned off in order to speed up the SystemC simulations. Indeed, it does not alter the behaviour of the cache since the address conversion from virtual address to physical address is a flat one by default. Furthermore, the software has already been verified so there is no access to an unauthorized domain from the benchmarks.

1.3.3. Multiple loads and stores

The ARM assembly provides instructions to load and store multiple registers from the *same memory line*. This particular feature must be specifically handled in the cache simulator. Indeed, considering them as usual memory accesses would overestimate the cache hit ratio: the first instruction is a miss or a hit but all the other lookups would result in a hit, provided that the latency simulation is disabled. Now, in case of a miss, the cache does not have time to fetch the data from the main memory. Consequently the following memory accesses of this core instruction must be marked as a miss too. In order to detect them, the timestamps of the instructions are monitored. If the timestamps are equal, then the memory access result (hit or miss) is considered as the same as the previous one. If the latency is enabled, this test appears redundant. Thus, it is performed only when the latency is disabled.

1.3.4. Optimal algorithm

The optimal algorithm implemented in this work is not the real optimal version of the algorithm as the name could suggest. It is only a very good approximation of optimality in the sense it looks for the future accesses of a set and evicts the way that is accessed the last. This is a sufficient condition for optimality under usual circumstances [BEL69]. The issue raised in modern optimized data caches originates from instructions such as LDMIA. As it has already been stated previously, this class of instructions generates many memory accesses that must be globally considered either as a hit or as a miss and not as independent memory requests. The nature of the ARM assembly yields some equivalent situations. A solution could be to affect them a weight equal to the number of data cache line accesses that it induces, but determining this on execution would be almost impossible since it assumes the exact knowledge of the cache contents and of its precise internal organization at every rising clock edge. However, this could be roughly approximated by assessing the number of bits it requires and dividing it by the width of a data cache line. It would be a quite optimistic approximation but the deviance from optimality should be tiny. Besides, state-of-the-art data caches are optimized in numerous manners, which prevent researchers from getting a realistic overview of the next lookups. It explains the NP-hardness of the optimality issue [BRE04]. Since this thesis aimed to find and to implement a cache replacement policy and keeping in mind that the optimal algorithm is only a view of spirit and not a realistic replacement strategy, it was decided to keep this good approximation of the optimal algorithm. Indeed, it is sufficient to get some absolute clues about the efficiency of the considered replacement algorithms. Moreover, in order to avoid numerous reads of the TARMAC file, this replacement policy has been enhanced with a buffer to allow faster simulations.

1.4. Benchmarks

The benchmarks running on the SystemC model of an ARM11 processor come from different sources:

- *3D graphics*: it is a benchmark suite that stands for the graphic kernel of Quake2. The benchmarks are available in 32-bit floating point and integer version,
- *500* benchmarks: these benchmarks perform usual operations like quick sorting, computation of Huffman bits,

[a] See Chapter 4 Section 1.5.2 p.22 for more details.

- *EEMBC benchmarks (versions 1 and 2)*: these benchmarks are supposed to account for usual application work in the embedded world, particularly for automotive, telecommunication and networking (for more information about each benchmark and the datasheets, see [EEM06]).

Combined together, these benchmarks should represent a good overview of the work and of the memory access patterns required by usual embedded applications, which is the market of the ARM processors. For a deeper description of these benchmarks, one can have a look at Appendix C.

In this work, all the simulated benchmarks exhibit quite large data sets since this property induces more realistic patterns and ensures that the hit ratios do not reflect the filling of the cache or the booting phase. Indeed, the impact of the replacement policies must not be overwhelmed by these unavoidable effects, which would lead us to wrong conclusions.

Selection. From all the tested benchmarks, a selection of representative benchmarks was performed. By representative, it is meant that the cache is stressed and that some differences in the hit ratio between the different cache sizes are observable. This last element insures that we do not lie in the part of the compulsory misses, on which the replacement policy does not impact at all (see Figure 10 in Chapter 1). Moreover, these representative benchmarks exhibit realistic hit ratio (i.e. their hit ratios are not all equal to 99.91-99.94%).

Representativeness. All the benchmarks needed to be adapted to the ARM validation environment. Nevertheless, they remained quite useless even after modification since they lead to hit ratio over 99% in their original form, thereby raising the issue of their representativeness. Besides, if benchmarks were completely trustworthy, there would be no need for a L2 cache, which is obviously not the case in real embedded applications. For this reason, these benchmarks were tuned in order to obtain larger cache misses ratio. Unfortunately, although it gave us interesting characteristics about the different replacement algorithms, it was not sufficient to stress the largest data caches (64 KB and even 32 KB). Benchmarks `mpeg4_decode` and `mpeg4_encode` show the worthlessness of increasing the size of the inputs data: the proportion of conflict misses remains the same because the algorithms work only on a local window of the working set at a given moment and not on the whole set. As a result, increasing the size of data does not influence the hit ratio.

A scale factor can be evoked to extend the conclusions of the smallest data cache sizes to the greatest ones but this is not a true scaling. Indeed, not all the cache parameters which potentially bias the cache behaviour can be scaled down: block size, cache line size, word size remain unchanged. Consequently, a new means to measure the relative performance of the different algorithms was required. Software directly run on the processors without any operative system seemed a good solution. Indeed, the aim of the benchmarks is to be synthetic and thus they only model the typical behaviours of the programs. So, they do not exhibit useless memory requests which help stressing the data cache. Furthermore, the introduction of these useless accesses complicates the management of the data cache sets and thereby should lead to greater differences between the cache replacement strategies.

1.5. Software

The lack of representativeness reported above has been identified for a long time, thus some pieces of software have already been developed in the company to solve this issue. They are reused and adapted a bit to the needs of this work. These software applications perform different operations:

- `Maze` finds a path between two nodes through a connection set,
- `Explorer` computes a factorial, evaluates if different numbers are primes or not, performs permutation on different strings, calculates the remainder of integer divisions (modulo operations).

The software applications' characteristics, among them data sets, can be easily modified in order to stress the caches. The simulations on the cores took much longer time, preventing us from carrying out the study only on these more realistic patterns. Indeed, in opposition to the benchmarks, design issues lead us to simulate the software applications on the Verilog core and not on the SystemC version. Besides, basing this work only on the two software applications would have exposed us to specific characteristics of the programs, thereby negating the statistical average performed over the wide range of benchmarks.

1.6. Remarks about the simulations

1.6.1. Data cache parameters

One of the first questions raised before launching simulations is the range of the data cache size. At a first glance, only the data cache sizes 16 KB, 32 KB and 64 KB should be studied since they correspond to the cache configurations available with the target architecture of this work, the ARM11 one. However, after the first simulations, it clearly appeared that this will not be sufficient for the scope of this work: stressing the 32-KB and even more 64-KB data cache was extremely complicated and the observed differences for the 16-KB cache were small. Therefore, it has been decided to add the value 8-KB to the simulation set. It is firstly needed by the stressing issue presented above. Secondly, it is justified by the fact that benchmarks stand for real applications but are a sum up of some of their patterns and therefore the cache is less stressed than it will be in the real life. Finally, it should be reminded that the data cache will be faced with more complicated patterns in real products: many threads will preempt one another and a light version of an operative system will be responsible for handling these threads, their priority and interruptions from the outside world. All these factors contribute to stress the data cache much more and to give it a smaller efficient cache size. For these reasons, the simulated values of the data cache size are 8, 16, 32 and 64 KB.

The simulator is able to deal with up to 32 ways, provided that true LRU is not used and 8 ways if true LRU is the replacement strategy. However, the number of ways of almost all the simulations introduced in this chapter will be four because it corresponds to the implementation of the data cache in the ARM11 MPCore processor. Simulations were also performed with 2-way, 8-way and 16-way set associative caches to evaluate the impact of the associativity but their results will be separately presented in Section 4.

1.6.2. Smoothing

Before dealing with the interpretation of the results, it should be reminded that the results presented here were performed on numerous tests. Since each test has its own characteristics, figures can be smoothed because of the mathematical average. This is particularly the case for tests, which have different significant zones: for instance, one benchmark should exhibit interesting features for 8 KB and 16 KB data cache but not for higher cache sizes because its working set would entirely fit in the data cache. If the others benchmarks do not exhibit any difference between the replacement algorithms for these cache configurations, this specific feature will not be visible in the final data, averaged over all the benchmarks. For these reasons, each benchmark result has been studied independently from the others and only then the overall results were analyzed. When a benchmark exhibits an interesting feature, it is specifically mentioned in the text and explained why it should be important. In other cases, only the averaged results are addressed. Yet, smoothing is essential because it prevents us from founding the conclusions on a particular and rare phenomenon specific to a program.

1.6.3. Validity of the results

In this work, we are faced with an almost unsolvable issue: in order to stress the data cache at a significant level, we were forced to use specific software which should reflect the usual observed values partially; it implied relying a part of our study on a restricted number of software, thereby exposing us to some particular behaviours specific to a software or a special data alignment because they are not smoothed anymore among a statistical average. A solution would have been to run a lot of different real software and to compare their results. Unfortunately, this would have been very much time-consuming and would have prevented us from diving into the design part, which was a major point of the original theme of the thesis. Furthermore, it would have required writing the different programs in order not to stress the same way the data cache and to face it with various memory request patterns. However, it can be admitted that program `explorer` already performs it in this way but at a lower scale: it runs different small programs like factorial, computing the power of a number, permuting strings, computing the modulo of a number, popping off and pending elements from/to stacks. Therefore, the best solution is after all to keep this issue in mind and to examine the obtained results carefully.

2. Simulation results

The relative efficiency of the different replacement algorithms must be examined in details for some candidates of implementation to emerge from the others. To that end, simulations were performed. A comprehensive analysis of their results will follow the presentation of the obtained figures. In a first time, only the results of the replacement policies appearing as potential enhancements are introduced and then a comparison with the current implementation of the ARM11 MPCore processor (Global Round Robin) will be performed. Finally, the replacement strategies will be submitted to the memory requests of real programs. This complete process will allow us to get an almost thorough overview of the possibilities of each policy.

2.1. Benchmarks with usual replacement policies

The results of the benchmarks' simulations are given in Table 4. The corresponding graphs are drawn on Figure 19. From these results, one can distinguish different groups of efficiency among the replacement policies: Random and Round Robin, 1-bit, Side and LRU and the pseudo-LRUs (PLRUm, PLRUt, MPLRU). According to the obtained figures, the latter appear as the best candidates for implementation, even though 1-bit must be further considered too.

Policy	*Cache size (KB)*	*Ways*	*All benchmarks*			*Representative benchmarks*		
			Hit ratio (%)	*Miss ratio / LRU*	*Average Miss ratio / LRU*	*Hit ratio (%)*	*Miss ratio / LRU*	*Average Miss ratio/ LRU*
1-bit	8	4	96.32	0.987	0.985	97.04	0.984	0.996
	16	4	97.15	0.983		97.90	1.010	
	32	4	98.28	0.999		98.74	0.997	
	64	4	98.56	0.972		99.10	0.991	
LRU	8	4	96.27	1.000	1.000	97.00	1.000	1.000
	16	4	97.10	1.000		97.92	1.000	
	32	4	98.27	1.000		98.74	1.000	
	64	4	98.51	1.000		99.09	1.000	
MPLRU	8	4	96.26	1.002	0.997	97.01	0.994	0.997
	16	4	97.13	0.988		97.94	0.989	
	32	4	98.27	1.001		98.73	1.007	
	64	4	98.52	0.998		99.09	1.000	
PLRUm	8	4	96.34	0.981	0.981	97.07	0.975	0.976
	16	4	97.17	0.977		97.94	0.988	
	32	4	98.30	0.982		98.78	0.968	
	64	4	98.54	0.985		99.11	0.974	
PLRUt	8	4	96.26	1.003	0.999	97.01	0.995	0.998
	16	4	97.13	0.989		97.94	0.988	
	32	4	98.27	1.003		98.73	1.010	
	64	4	98.51	1.000		99.09	1.000	
Random	8	4	96.25	1.004	0.999	96.92	1.024	1.021
	16	4	97.15	0.984		97.84	1.037	
	32	4	98.26	1.007		98.71	1.026	
	64	4	98.52	0.999		99.09	0.998	
Round Robin	8	4	96.18	1.023	1.018	96.86	1.045	1.031
	16	4	96.98	1.042		97.79	1.063	
	32	4	98.27	1.000		98.75	0.990	
	64	4	98.50	1.008		99.06	1.026	
SIDE	8	4	96.23	1.010	1.009	96.99	1.001	1.009
	16	4	97.10	0.999		97.90	1.008	
	32	4	98.25	1.015		98.74	0.994	
	64	4	98.50	1.001		99.06	1.033	

Table 4: Hit ratio for the different policies with benchmarks on a 4-way set associative cache

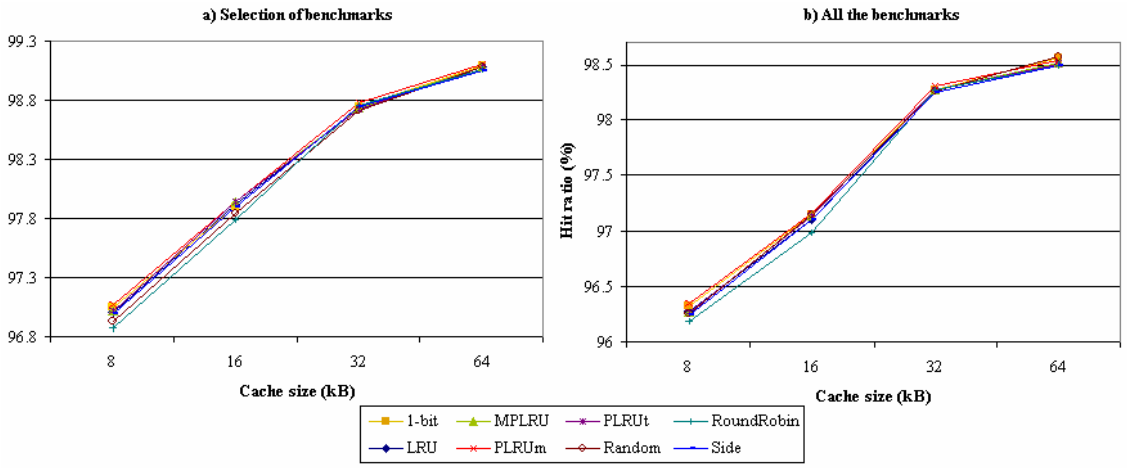

Figure 19: Hit ratio of the different usual replacement policies averaged over benchmarks

At first glance, it could be surprising that the hit ratios are higher for the representative group than for the usual one, although it has been claimed that they present more realistic hit ratios. It is simply due to the presence of some specific benchmarks – among others `500_mandeld`, `consumer_rgbcmyk`, `consumer_rgbcmy`… – in the complete set. These benchmarks exhibit a tiny difference between the different replacement algorithms across the different cache sizes but yield a quite important amount of misses (miss ratio around 2-3%). Capacity misses govern the behaviour of these benchmarks. Therefore, the study can be pursued with the single group of representative benchmarks.

2.2. Non-MRU and Global Round Robin study

In the continuity of the previous section, the algorithms were compared to the current replacement policy of the ARM11 MPCore processor, Global Round Robin. Non-MRU is investigated in this section too since it is a comparison for 1-bit at the same degree as GRR for the other strategies. This study is carried out in this section and the results are given in Table 5. The benchmarks ran correspond to the selection of representative benchmarks. For the sake of clarity, only the policies, which were short-listed in pursuance of the first benchmarks' results, along with some reference algorithms (Round Robin and Random), are presented below. This choice of algorithms also provides us with a continuous range of complexity of implementation, from no bit for Random to 4 bits for PLRUm, where the complexity is still acceptable.

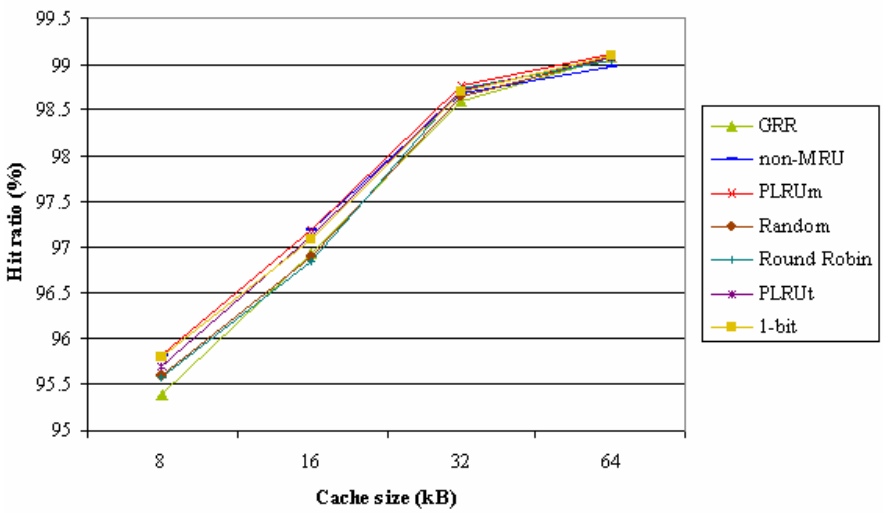

Figure 20: Average hit ratios on representative benchmarks with GRR and non-MRU

From these figures, it is striking that the data caches are not stressed enough for 64 KB and even for 32 KB where the observed hit ratios are abnormally high to hope distinguishing usage-based policies from one another. This comment justifies the appeal to software applications in the next section.

Policy	*Cache size (KB)*	*Ways*	*Hit ratio*	*Miss ratio / PLRUm*	*Average miss ratio / PLRUm*
Global Round Robin	8	4	95.38	1.108	1.095
	16	4	96.92	1.097	
	32	4	98.59	1.147	
	64	4	99.08	1.026	
Round Robin	8	4	95.58	1.059	1.070
	16	4	96.84	1.126	
	32	4	98.74	1.026	
	64	4	99.04	1.070	
1-bit	8	4	95.80	1.006	1.031
	16	4	97.09	1.037	
	32	4	98.69	1.061	
	64	4	99.07	1.019	
Non-MRU	8	4	95.83	0.999	1.056
	16	4	97.18	1.005	
	32	4	98.68	1.072	
	64	4	98.98	1.148	
Random	8	4	95.60	1.055	1.075
	16	4	96.90	1.106	
	32	4	98.64	1.101	
	64	4	99.07	1.038	
PLRUt	8	4	95.71	1.028	1.034
	16	4	97.11	1.029	
	32	4	98.72	1.041	
	64	4	99.07	1.036	
PLRUm	8	4	95.83	1.000	1.000
	16	4	97.19	1.000	
	32	4	98.77	1.000	
	64	4	99.11	1.000	

Table 5: Average miss ratios on representative benchmarks of GRR and non-MRU

2.3. Software

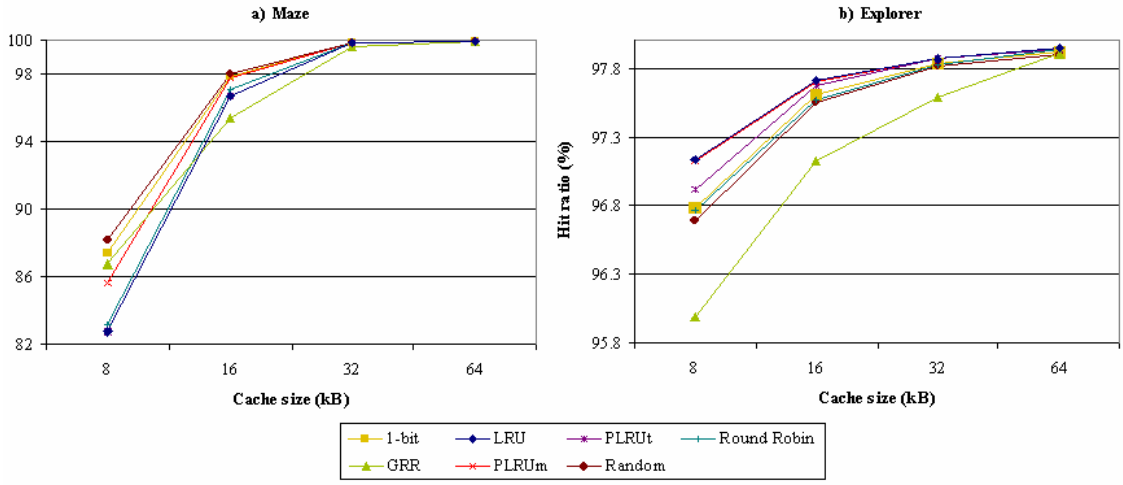

Figure 21: Hit ratio of the interesting replacement policies on the software applications

The results are given in the figures above. For sake of clarity, only the policies that emerged before are represented here as well as Random and Round Robin as references. The applications used are `maze` and `explorer`. Their data sets are modified so that caches can be more or less stressed. However, changing the work set does not change the global pattern access scheme: the program is the same and performs the same operations, only on a different input. Nevertheless, these modifications can be very

important and impact on the global memory pattern scheme, for instance on trees. Moreover, one of their advantages is that the result can be checked, giving a means to be certain that the memory patterns reflect reality. Of course, the correctness of the results does not prove that the program operates as wanted but the probability is very high. For the scope of this work, it will be considered as a sufficient verification. The graphs show that even with these software applications, it was hard to stress the cache for 64 KB; it is almost constant between cache sizes 32KB and 64 KB. However, we should keep in mind that the usual configurations sold by ARM are up to 32KB. This is all the more true since all cores are now provided with an on-chip L2 cache, which explains that the size of L1 caches is reduced.

3. Interpretation

The results presented in the previous section mostly agree with the figures reported in the literature [DEV90, MIL03, SMI83, ZOU04]. In opposition to the articles cited above, the major point of this chapter will be to explain the behaviour of the strategies, analyzed one after another.

3.1. Random

3.1.1. General considerations

As expected, the Random policy is less efficient than the other algorithms; it is almost always outperformed by pseudo-LRUs. A difference of 5-7% to the LRU figures is observed, which disagrees with the results of Al-Zoubi *et al.* [ZOU04]: they reported a performance loss roughly equal to 20–25 % on the SPEC CPU2000 benchmarks suite [SPE00]. In some of the benchmarks simulated in this work (`3d_persptris_f32`, `networking_ospf`, `office_bezier`), Random is even slightly better than the LRU policy. It is the expression of specific patterns where the width of the history retained by LRU is inadequate (loops of N_{ways}+1 steps for instance). This explanation is confirmed by the observation of the same behaviour for the pseudo-LRUs. This phenomenon will be more precisely studied in Section 3.8. Though it is not the common situation, it corroborates the absence of an absolute replacement policy; this choice must be performed in accordance with the average characteristics of the running programs. This explains the improved performance observed on the `maze` pattern too since the exploration of the connections of the tree does not yield strong spatial locality. Thus, evicting the lines in a random manner gives a better probability that the nodes, which will be accessed in the future when the program searches back from a leaf, are still located in the cache.

The comparison of Random with Round Robin shows the roughly equivalence of the two algorithms in terms of performance. The efficiency then depends on the specificities of each benchmark.

3.1.2. Lack of stress

If we restrict our study to the small cache size part of the benchmarks (where the conflict misses are more frequent), greater differences up to 25% are noticed. The figures are given in Table 6 for some of these benchmarks.

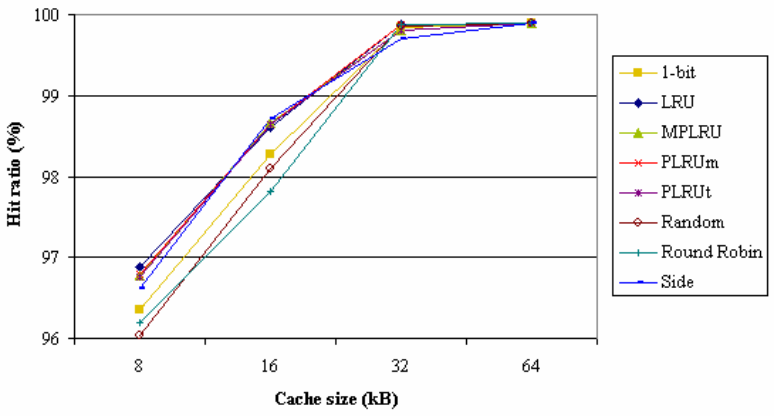

Figure 22: office_rotate hit ratio in a 4-way set associative cache

Their average miss ratio compared to the LRU ones are:

- 1.27 for 8 KB data cache,
- 1.10 for 16 KB data cache,
- 1.04 for 32 KB data cache,
- 0.96 for 64 KB data cache.

Benchmarks	*Cache size (KB)*	*LRU Hit ratio (%)*	*Miss ratio / LRU miss ratio*		
			Random	*PLRUm*	*PLRUt*
automotive aifftr	8	98.67	1.373	1.062	1.003
	16	99.01	1.076	0.863	0.906
	32	99.48	1.052	0.884	0.924
	64	99.95	1.017	1.001	0.998
automotive tblook	8	99.23	1.048	0.993	0.993
	16	99.28	1.012	1.016	1.002
	32	99.29	1.000	1.002	1.000
	64	99.29	1.000	1.000	1.000
mpeg4 decode	8	97.87	1.063	1.002	1.007
	16	98.01	1.032	1.003	1.003
	32	98.21	1.049	1.004	1.002
	64	98.25	1.019	1.001	1.002
mpeg4 encode	8	98.24	1.047	1.000	1.004
	16	98.35	1.024	1.002	1.003
	32	98.56	1.055	1.006	1.002
	64	98.57	1.013	1.000	1.000
networking route lookup	8	99.60	1.889	1.000	1.067
	16	99.97	1.071	1.002	1.376
	32	99.97	1.002	1.002	1.000
	64	99.97	1.000	1.000	1.000
office dither	8	99.39	1.235	0.997	0.983
	16	99.43	1.143	0.993	0.962
	32	99.47	1.012	1.008	0.988
	64	99.73	0.673	0.695	0.766
office rotate	8	96.88	1.271	1.030	1.041
	16	98.61	1.366	0.973	0.972
	32	99.88	1.135	0.987	1.546
	64	99.90	0.999	1.001	0.999

Table 6: Miss ratios of LRU, Random and PLRUm on specific benchmarks

One sees that the figures match the study of Al-Zoubi for small cache sizes. For bigger cache sizes, the cache is probably not stressed enough for us to observe these differences at the same intensity. A typical graph of these benchmarks is shown on Figure 22. The issue of lack of stress for high cache sizes is striking for all the replacement strategies. For sake of concision, it will not be further mentioned in the chapter but it should be reminded that it applies to each policy. However, this only explains the difference for these benchmarks. Another parameter may significantly affect the overall performance.

3.1.3. Importance of looking for free ways

It should not be forgotten that the implemented algorithm differs from the academic one. Indeed, the latter does not look for a free way before allocating a way whereas the simulated one does. This may partly explain the difference between the simulated results and the literature ones, which are far from optimal. The impact of this hypothesis is studied here with a simple probabilistic model. Let us consider only a set of the data cache (the other sets being symmetrical). Denoting *p(k,n)* the probability that there are *exactly k* free ways in the set after *n* allocations to the set, the initial conditions are:

$$P(0) = [\, p(4,0), p(3,0), p(2,0), p(1,0), p(0,0)] = [1, 0, 0, 0, 0]$$

where *P(n)* is the probability vector. With the same notations as above, *S(k)* is the state corresponding to the assertion *exactly k* ways store a valid data. The transitions between the different states can be represented by the Markov chain drawn on Figure 23.

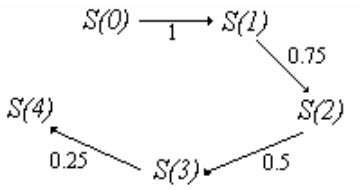

Figure 23: Markov chain for the Random replacement policy

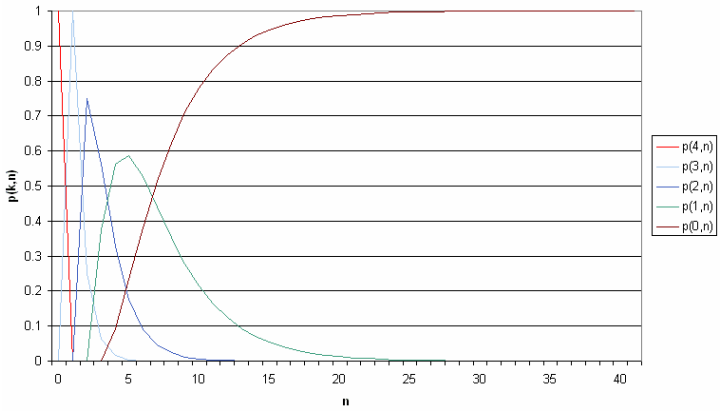

Figure 24: Evolution of the probabilities p(k,n) in function of the allocations in the cache set

Since the replacement policy is Random and since the four ways of the set are equivalent, the transition rate between states *S(k)* and *S(k+1)* equals (4-*k*)/4. The invalidations of the lines which enable a transition between the states in the inverse order are neglected here because of their tiny probability in multi-processing platforms and their absence in mono-processing environments. Evictions would have the same effect but at the difference that an allocation at the same line immediately follows them. The set thus remains in the same state. As a result, it can be assumed in a first approximation that there is no transition between states *S(k)* and *S(k-1)*. Combining this to the fact that only one line can be allocated at a time, it implies that all the possible transitions have been dealt with.

The transition matrix *M* is then:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0.25 & 0.75 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0.75 & 0.25 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

And the state evolution is given by:

$$P(n) = P(0) \times M^{n}$$

The evolution of states *S(k)* in function of the number of allocations is drawn on Figure 24. From these probabilities, the average number of busy ways *N(n)* can be easily deduced by the simple equation:

$$N(n) = 4.p(0,n)+3.p(1,n)+2.p(2,n)+p(3,n)$$

The resulting curve is drawn on Figure 25 as well as the result for the ARM implementation.

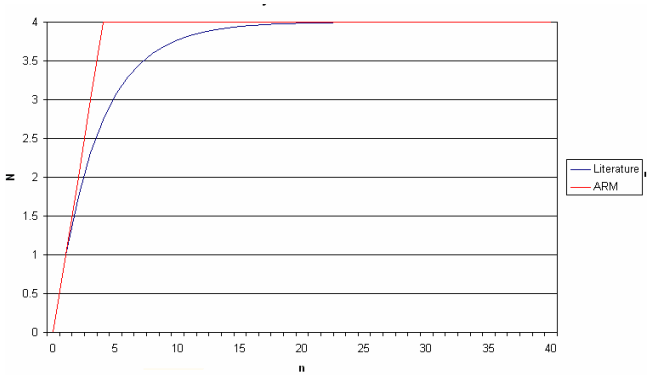

Figure 25: Average number of busy lines in function of the allocations in the cache set

There exists a significant difference between the two implementations for the values of *n* lying in the range [2:10]. Supposing that the accesses are uniformly spread among the 512 data cache sets, it

implies that the effect can be detected on the overall cache behaviour only for the first 5,000 cache misses. For these requests, the implemented solution roughly performs 1.25 times better than the literature one: its number of busy lines, which is a measure of the effective capacity of the cache in some manner, is 1.25 greater than the value obtained with the literature method. On the benchmarks simulated above, the average hit ratio is 95-99%. Therefore, a good approximation would be to evaluate the 5,000 cache misses roughly equivalent to 500,000 cache requests. As a result, a difference around 1.25 is observed between the two implementations up till 500,000 requests.

Let us apply this reasoning to the benchmark `networking_ospf` which issued 673,677 data cache requests. It then corresponds well to the bounds inferred above. Its hit ratios are 94.46% for Random and 94.30% for LRU. At a first approximation, we can apply the 1.25 factor to the whole benchmark (the bound 500,000 is a very rough approximation), which yields a corrected miss ratio for Random of 6.925%. Expressing it in function of LRU miss ratio, one obtains a ratio of 1.21. The corrected difference is then in the same range as the one obtained by Al-Zoubi *et al*. Moreover, it should be noticed that the benchmarks presented in Table 6 issued a huge amount of cache requests (over 2,000,000). Because of this huge amount of requests, the difference has been pad out in the overall pattern. Consequently, after examination with a quite coarse model, the simulated results match the literature ones.

3.2. Round Robin

After examination of the simulation results, it appears that Round Robin performs as bad as Random on benchmarks and on software applications. There is no obvious better policy between Random and Round Robin: on some benchmarks, one is better, on other benchmarks, it is the opposite. The comparison with the LRU policy yields even worst results (around 1-5%). This result disagrees with the work of Al-Zoubi *et al.* which found a difference of 15-20% but agrees with the figures of Deville [DEV90] for low associativity. This discrepancy is probably due to the lack of stress on the caches for most benchmarks. It is corroborated by the software study. On `explorer` patterns, the difference is 9% on average with peaks to 15% on low size. The increase of the gap for low sizes reinforces the hypothesis of lack of stress. On `maze`, Round Robin performs on average 2% better than LRU. Once more, it underlines the specificity of `maze` which helps putting the conclusions in perspective with the particular characteristics of each program.

Benchmark	*Cache size (KB)*	*LRU hit ratio (%)*	*Miss ratio / LRU miss ratio*		
			PLRUm	*PLRUt*	*Round Robin*
automotive matrix	8	93.49	0.924	0.950	1.091
	16	97.82	0.894	0.978	1.098
	32	99.02	0.975	1.028	1.078
	64	99.77	1.006	1.128	1.150
networking route lookup	8	99.60	1.000	1.067	1.888
	16	99.97	1.005	1.372	1.073
	32	99.97	1.002	1.000	1.002
	64	99.97	1.000	1.000	1.000
networking tcp mixed	8	79.85	0.955	1.001	1.000
	16	86.25	0.807	0.914	0.999
	32	98.76	0.980	1.009	1.005
	64	98.97	0.845	0.911	1.005
office dither	8	99.39	0.997	0.983	1.222
	16	99.43	0.993	0.962	1.146
	32	99.47	1.008	0.988	1.071
	64	99.73	0.695	0.766	1.207

Table 7: Miss ratios of PLRUm, PLRUt, Random and Round Robin on specific benchmarks

Although Round Robin is not a so bad approximation of the LRU policy in overall (see Table 4), a significant difference is observed with the following benchmarks: `automotive_matrix`, `networking_route_lookup`, `networking_tcp_mixed`, and `office_dither`. One notices that these are almost the same benchmarks as Random, thereby contradicting the potential hypothesis according to which only the benchmarks demonstrating what we expect are selected. The figures of

these specific benchmarks are given in Table 7. The Round Robin performance appears here as 10-15% worst than the LRU one, which is in the range of the results given by Al-Zoubi *et al.* The figures are:

- 1.300 for 8 KB data cache,
- 1.079 for 16 KB data cache,
- 1.039 for 32 KB data cache,
- 1.091 for 64 KB data cache.

The results are the worst for the smallest data cache sizes, where the conflict misses are the most important. It confirms the hypothesis of a lack of stress to explain the difference between Al-Zoubi's results and ours. The difference in handling the free ways which explained the discrepancy for Random does not apply here. Indeed, as long as the line invalidations are negligible, the Round Robin policy assigns the four free ways in the increasing order. Then the cache set is full and the ways are evicted in the order they came in.

The strong dependence of the Round Robin replacement policy on the cache configuration emerges from the simulations. Even for patterns apparently equivalent, the results are quite different. It can be explained by data alignments inside the memory or simply a high sequentiality of the optimal replacement in some cases. The overall poor performance of the Round Robin strategy can be explained by the lack of history in its computing. Indeed, it is well adapted for patterns where the accesses are performed sequentially but its results are poorly efficient as soon as the ways of a set are not accessed in the same order as the counter, which can lead the policy to evict the MRU line. As a result, the relatively poor observed efficiency is not surprising.

3.3. Global Round Robin

These simulations demonstrate that Round Robin with a global counter performs worse than true Round Robin and worse than the pseudo-LRU algorithms. The figures correspond well to the intuition since GRR is a pseudo-random strategy. As the set locality is strong due to the spatial locality, the GRR counter can be seen as an efficient counter on two or three sets, thereby presenting the same drawbacks as Round Robin without its advantage of well-fit to the highly sequential accesses.

On the other hand, Global Round Robin performs 1% better on `networking_tcp_mixed` and 3% better on `maze` (up till 5% on small cache sizes) than Round Robin whereas the latter outperforms it of 1-2% on `explorer`. I expected that Round Robin would perform much better than Global Round Robin because there are quite common situations where Global Round Robin evicts the last written line. For instance, let us consider four lines λ_1, λ_2, λ_3, λ_4 mapped to distinct sets and which are not lying in the cache. Let us now assume that a line λ_5 is mapped to the same set as λ_1. Then the execution of the sequence λ_1, λ_2, λ_3, λ_4, λ_5 will evict the line λ_1 which was the MRU of its set! Such situations are not so rare and should have a negative impact on the overall hit ratio. True Round Robin avoids these situations and then should be more efficient. However, simulation results differ with this hypothesis: the difference is not as significant as expected so this type of situations is not as common as thought at first glance. This is confirmed by the results of the non-MRU which slightly performs better than Random. Yet, non-MRU aim is to avoid evicting the last written line. The results are thus consistent.

The difference of efficiency between Global Round Robin and its competitors is sufficient to justify implementing a pseudo-LRU algorithm. Indeed, in hit ratio – which is the interesting figure which reflects the most accurately the efficiency of the replacement algorithms – the difference is a bit less than 1% but the hits here are important and the cache must be stressed further to study in depth this feature. Besides, the difference reaches two percents of the memory requests on benchmarks `networking_route_lookup`, `office_rotate` and `networking_pktflow`.

3.4. 1-bit

The 1-bit policy aims to protect the MRU region and particularly the last written line from eviction. Its results are good and better than Random. The policy thus seems quite efficient at a low cost but there are some applications patterns where it performs badly: its performance is not steady across all the benchmarks. While it performs a bit and even significantly better than LRU for some applications, the gap separating it from the best simulated replacement policy is important on some benchmarks. A typical graph of these situations is drawn on Figure 26. The random assignment inside the non-MRU half benefits from the same features as Random, which could explain the instability of the policy.

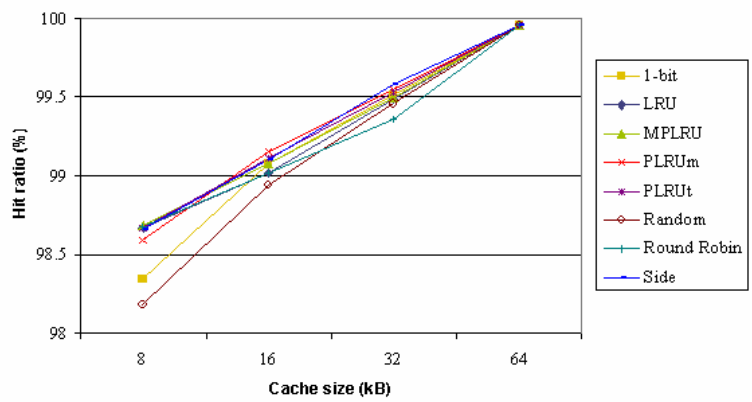

Figure 26: Hit ratio for automotive_aifftr in a 4-way set associative cache

This solution yields an improvement compared to the Global Round Robin, up to one percent in hit ratio. Furthermore, using only one bit instead of three or four for the pseudo-LRUs would save not only area but also power, because there would be fewer accesses to the RAMs storing these bits. Consequently, this policy suits the aims of this thesis and is one final candidate for implementation in spite of its unsteadiness.

The choice of encoding this policy with one bit protects also the neighbours[a] of the last written line. In order to investigate whether this protection affects the overall hit ratio significantly, an algorithm was implemented: the non-MRU policy.

3.5. Non-MRU

The principle is the same as for 1-bit but the LRU way – and not its half – is stored so that there is no collateral protection. Thus, 2 bits per set are required instead of one. After examination of the results, non-MRU performs roughly the same as 1-bit and then appears as a quite non-optimal solution: 1-bit requires less additional storage and its updating and decoding processes cost less hardware. Thus, the hypothesis raised in the previous section is not verified. Splitting the cache into two halves is well-adapted to 4-way set associative caches with reasonable data line width. It gives clues about the size of the MRU region of typical embedded programs too: it should be included in the range 1-2 lines. Non-MRU will not be further considered in this work because of its ratio hardware cost-replacement efficiency.

3.6. Modbits

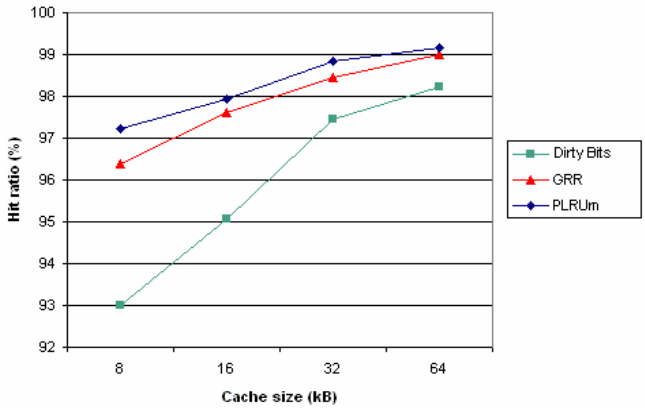

Figure 27: Hit ratio of the dirty bits implementation averaged over the selection of benchmarks

While looking for the cheapest algorithm, using the dirty bits was an idea whose ratio efficiency-cost might be interesting. The results of the simulations were quite disappointing: around 2 % worst in hit ratio as it can be seen on Figure 27. The difference even reaches 3% in hit ratio on low size where

[a] By neighbours of way w is meant: $\{y \in \mathbf{N} / E(w/2) = E(y/2)\} \setminus \{w\}$, i.e. simply $2.E(w/2) + (1-w \bmod 2) = 1-w + 4.E(w/2)$ in the case of a 4-way set associative cache.

conflict misses are more important. It may be explained by the fact that this policy does not take into account the read and can even evict them when they are MRU. Moreover, once all the ways of a set are written, there is no means to differentiate them and it is equivalent to the Random policy until the next allocation. As a result, the read requests evict one another whereas writes remain in the cache and pollute it. One solution to this issue would be to add an aging policy to this algorithm but then the design cost would be too expensive: the Dirty RAM should be powered on at each access, thereby leading to high power consumption, which is specifically what must be avoided in embedded systems. As the age should be stored somewhere (flip-flops or RAM), it will increase the power consumption of this policy. Considering this complexity with its poor efficiency, it will not be mentioned further in this work.

3.7. Side

As expected and stated in [DEV90], Side performs a bit better than Round Robin and its hit ratios are roughly equivalent to the 1-bit policy. The updating process of the counter is based on usage, which provides the algorithm with a better adaptability to the patterns. This counter splits the cache into two groups and thus establishes a parallel with 1-bit. It is the origin of the close performances of the two algorithms. However, it still partially exhibits the same drawback as Round Robin: in the absence of a hit, the ways are discarded in the growing order. Moreover, it is strongly dependent on data alignments, non-cyclic accesses and cache parameters such as cache size, block size… Thus, it does not appear as a good candidate. Compared with 1-bit, Side thus appears not so efficient: it uses a 2-bit counter per set instead of one additional bit.

3.8. LRU and pseudo-LRUs

3.8.1. Results

The LRU algorithm and the pseudo-LRU ones present quite the same performance and much better than Global Round Robin since they are usage-based. Depending on the benchmarks, their relative efficiencies change but the differences are slight: pseudo-LRU algorithms are an efficient approximation of the LRU replacement policy. Consequently, implementing a true LRU algorithm is not worthy because of its high complexity – which is the origin of bugs in general –, its overhead, its consumption and the amount of required hardware. Pseudo-LRUs yield the same performance with simpler, cheaper and faster implementation. The issue is now to choose a pseudo-LRU algorithm.

The algorithms are different in nature: PLRUt and MPLRU implement a tree implementation whereas PLRUm is MRU based. Inside the first group, the differences are small: MPLRU performs a bit better than PLRUt but at the price of one additional bit per set and additional complexity in interpreting the status bits. This result is consistent with the simulation results obtained in [GHA06] where a significant improvement was only observed for high associativity (eight and above). Moreover, for the same amount of bits in a 4-way set associative cache, PLRUm performs much better and the decoding is simpler. A difference between PLRUt and PLRUm is observed: while PLRUt is a good approximation of LRU, PLRUm outperforms it on almost all the input patterns. This superiority of PLRUm over PLRUt is confirmed when examining in details the simulation results. Depending on the data sets, PLRUm is 1-5% better in hit ratio than PLRUt and LRU, particularly for small cache sizes, where misses occur more often. Thus, PLRUm appears as a better implementation candidate but the tree-base version will still be considered further. The candidates among pseudo-LRUs are then:

- PLRUt : only three bits per set and as efficient as LRU in spite of its acceptable unsteadiness observed across the different benchmarks,
- PLRUm: four bits per set but simpler to encode, decode and interpret. Moreover, its hit ratios are the highest in average and on most patterns.

Compared to Global Round Robin, PLRUm performs around 2% better in hit ratio as it is shown in Figure 21. However, there is a case when Global Round Robin performs better than the pseudo-LRUs: on `maze` with a very small data cache. This is probably a specific resonance between the data and the global Round Robin designation. Other algorithms, including true Round Robin, do not exhibit such a feature, confirming that it should be a special alignment which produces this performance. Furthermore, this is corroborated by the hit ratio measured for the next data cache sizes: Global Round Robin performs much worst. Finally, this specific feature has not been observed on the benchmark suite; thereby reinforcing our idea that it is only a specific alignment created by the application.

3.8.2. The origin of the discrepancies

The improved performance of PLRUm has already been reported in the literature [MIL03, ZOU04] but no explanation about this enhancement has been advanced yet. This paragraph will try to fill this lack of knowledge. To that end, some typical sequences are studied; which will help us understanding the underlying phenomena governing replacement strategies' behaviour.

3.8.2.a. PLRUm vs. PLRUt

First, let us be interested in the two pseudo-LRU algorithms and have a look at Table 8. In this table, the errors in designing the LRU way are highlighted in grey: the darkest one for an error of 2 in the LRU stack and the lightest one for an error of one. Of course, error is a quite inappropriate term since an "error" in the LRU stack can help being nearer to the optimal choice. While PLRUt misled us three times, PLRUm made only two mistakes but one was quite important in the sense that it designed a way located at the second place of the MRU stack. This deviance originates from the reset of the status bits whereas element 0 was accessed during the previous step. The erroneous interpretation is then transmitted to the next step. Thus, the reset phase seems to be the major inconvenient of PLRUm. However, it constitutes its superiority over PLRUt too as it will be seen in the next paragraphs.

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
Sequence	0	1	2	3	0	3	1	2	1	0	3	0	1
PLRUm													
PLRU way	1	2	3	0	1	1	2	0	0	3	0	1	2
status bits	0001	0011	0111	1000	1001	1001	1011	0100	0110	0111	1000	1001	1011
PLRUt													
PLRU way	1	2	3	0	1	2	2	3	0	3	2	1	2
status bits	011	110	101	000	011	010	110	001	100	111	010	011	110
LRU way	1	2	3	0	1	1	2	0	0	3	2	2	2
Optimal way	0	1	2	2	0	3	3	2	2	2	2-3	0-2-3	0-1-2-3

Table 8: Pseudo-LRU ways for PLRUm and PLRUt

3.8.2.b. Amount of information held by the status bits

As it has already been stated in Chapter 3, the main difference between the pseudo-LRU algorithms consists in the binary tree of PLRUt, which assigns various efficient weights to the different status bits. Over the first ten accesses of Table 8, PLRUm yields the exact LRU way whereas PLRUt is mistaken twice since node 0 does not held sufficient information about the previous accesses. Indeed, if the true LRU way is a neighbour of the last accessed way, it will not be discarded. This could lead to far from optimal evictions. As all the MRU bits are almost equivalent, PLRUm does not exhibit this property. Nevertheless, a slight discrepancy is introduced by the order of examination. While computing the way to be discarded, the algorithm looks for a MRU bit equal to 0 from bit 0 to bit N_{ways}-1; bit k is thus more probably subjected to eviction than any bit j for $j>k$. This discrepancy is however far less significant than the one observed in PLRUt. Therefore, it can justify the superiority of PLRUm over PLRUt.

Another example of the lack of information in PLRUt is given in Table 9, where this issue is striking. The sequence corresponds to the execution of a loop 0-4-0-5-0-6 after the data cache has been filled with elements 0-1-2-3. This type of loop is quite common and the principle of temporal locality is strongly verified for this pattern. Since the loop contains only four elements, a protection of the loop elements is expected and hence a quasi-optimal performance should be reached. LRU performs optimally whereas PLRUm misses once more before keeping the four data as requested. Regarding PLRUt, its performance deceives the expectations since way 1 is never evicted and pollutes the cache although it is not required in the loop. Once more, this discrepancy originates from the tiny amount of information held in node 0.

If this phenomenon was the only one explaining the performance of PLRUt, MPLRU would lead improved efficiency over PLRUt and a performance roughly equal to PLRUm. This is obviously not the case on Table 4 and can be explained in looking at Table 8 more precisely. The sequence presented there is a target sequence of the MPLRU algorithm, where the MBAIs should help reaching a better approximation of LRU. In MPLRU, the replacement decision is based on the previous state of the MBAIs but the issue of the amount of information held by a bit partially remains since the neighbours are still protected due to the tree structure. The retained history is a bit longer, which explains the slightly better observed results on some benchmarks but an insignificant improvement in overall. The

advantage of PLRUm over PLRUt is then the greater history that it can retain in spite of its reset phase, even over MPLRU which considers the previous access.

Sequence	0	1	2	3	0	4	0	5	0	6	0	4	0	5	0	6
PLRUm																
way 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
way 1		1	1	1	1	4	4	4	4	6	6	6	6	6	6	6
way 2			2	2	2	2	2	5	5	5	5	5	5	5	5	5
way 3				3	3	3	3	3	3	3	3	4	4	4	4	4
status	0001	0011	0100	1000	1001	1011	1011	0100	0101	0111	0111	1000	1001	1101	1101	0010
PLRUt																
Way 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Way 1			2	2	2	2	2	2	2	2	2	2	2	2	2	2
Way 2		1	1	1	1	4	4	4	4	6	6	6	6	5	5	5
Way 3				3	3	3	3	5	5	5	5	4	4	4	4	6
status	011	110	101	000	011	110	111	010	011	110	111	010	011	110	111	010
LRU																
way 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
way 1		1	1	1	1	4	4	4	4	4	4	4	4	4	4	4
way 2			2	2	2	2	2	5	5	5	5	5	5	5	5	5
way 3				3	3	3	3	3	3	6	6	6	6	6	6	6
Optimal																
way 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
way 1		1	1	1	1	4	4	4	4	4	4	4	4	4	4	4
way 2			2	2	2	2	2	5	5	5	5	5	5	5	5	5
way 3				3	3	3	3	3	3	6	6	6	6	6	6	6

Table 9: Non-optimality of PLRUt and quasi-optimality of PLRUm on a loop

On the other hand, PLRUm performs also far from optimal in certain situations where the reset leads to discard the way lying in the second position of the MRU stack (see step 11 of Table 8), which explains that it is outperformed by PLRUt on some benchmarks. However, this situation is much rarer than the loss of information for PLRUt and then the impact is tinier.

3.8.2.c. PLRUm outperforming LRU

Sequence	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
PLRUm															
way 0	0	0	0	0	4	4	4	2	2	2	0	0	0	3	3
way 1		1	1	1	1	0	0	0	0	4	4	4	4	4	4
way 2			2	2	2	2	1	1	1	1	1	1	1	1	1
way 3				3	3	3	3	3	3	3	3	3	2	2	2
status bits	0001	0011	0111	1000	1001	1011	0100	0101	1101	0010	0011	0111	1000	1001	1011
PLRUt															
way 0	0	0	0	0	4	4	4	4	3	3	3	3	2	2	2
way 1			2	2	2	2	1	1	1	1	0	0	0	0	4
way 2		1	1	1	1	0	0	0	0	4	4	4	4	3	3
way 3				3	3	3	3	2	2	2	2	1	1	1	1
status bits	011	110	101	000	011	110	101	000	011	110	101	000	011	110	101
LRU															
way 0	0	0	0	0	4	4	4	4	3	3	3	3	2	2	2
way 1		1	1	1	1	0	0	0	0	4	4	4	4	3	3
way 2			2	2	2	2	1	1	1	1	0	0	0	0	4
way 3				3	3	3	3	2	2	2	2	1	1	1	1
Optimal															
way 0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2
way 1		1	1	1	1	1	1	1	1	1	1	1	1	1	1
way 2			2	2	2	2	2	2	3	3	3	3	3	3	3
way 3				3	4	4	4	4	4	4	4	4	4	4	4

Table 10: Sequence of 3 loops of N_{ways}+1 steps for LRU, PLRUm and PLRUt

The remaining mystery is now the unexpected good performance of PLRUm on some benchmarks where LRU can be clearly outperformed. The non-optimality of the LRU algorithm has already been

reported in many articles but the fact that it is outperformed by its own approximations is strange. The explanation advanced in this thesis relies on the reset feature, which allows PLRUm to be more flexible for some specific patterns and specifically for the loops composed of N_{ways}+1 allocations to the same set per loop. This contributes also to the enhanced performance of PLRUm over PLRUt. Indeed, the reset ensures a more adaptive history, thereby helping avoiding the cache pollution by less useful data. One example is given in Table 10.

Sequence	1	2	0	3	4	0	1	2	3	4	0	1	2	3	4
PLRUm															
way 0	1	1	1	1	4	4	4	2	2	2	0	0	0	3	3
way 1		2	2	2	2	2	1	1	1	1	1	1	1	1	1
way 2			0	0	0	0	0	0	0	4	4	4	4	4	4
way 3				3	3	3	3	3	3	3	3	3	2	2	2
status bits	0001	0011	0111	1000	1001	1101	0010	0011	1011	0100	0101	0111	1000	1001	1101
PLRUt															
way 0	1	1	1	1	4	4	4	2	2	2	2	1	1	1	1
way 1			0	0	0	0	0	0	0	4	4	4	4	3	3
way 2		2	2	2	2	2	1	1	1	1	0	0	0	0	4
way 3				3	3	3	3	3	3	3	3	3	2	2	2
status bits	011	110	101	000	011	001	100	111	010	001	100	111	010	001	100
LRU															
way 0	1	1	1	1	4	4	4	4	3	3	3	3	2	2	2
way 1		2	2	2	2	2	1	1	1	1	0	0	0	0	4
way 2			0	0	0	0	0	0	0	4	4	4	4	3	3
way 3				3	3	3	3	2	2	2	2	1	1	1	1

Table 11: The same N_{ways}+1-step sequence with a different filling order.

Whereas PLRUt and LRU miss at each access, PLRUm attunes better and yields three hits over the fifteen requests. Nevertheless, it is far from the optimal algorithm which misses only seven times. This discrepancy in the behaviour of such patterns could partially explain the improvement observed for PLRUm. But one noticed in Table 4 that PLRUt outperforms LRU too but at a slighter degree than PLRUm. At first glance, it is in contradiction with the results of Table 10 since PLRUt apparently exhibits the same drawback as LRU. Yet, the filling order of the set is a parameter that affects the overall performance significantly (see Table 11 and Table 12). Indeed, the replacement strategies deduce the future accesses from the previous ones and are then sensible to the first filling. One sees that the best improvement is observed for PLRUm but the difference between PLRUt and LRU is not negligible too since it is a 0.5 miss for a sequence of 15 elements. This type of loops can be quite common in embedded systems where numerous matrix multiplications are called for picture computing and telecommunications for instance. The phase called “filling” of the cache corresponds to the state of the set before the loop begins and is then strongly dependent on the initialization operations performed on the matrix. For this reason, there is no common pattern for the initial state and the 24 possibilities must be studied, which is done in Table 12. The importance of the initial state of the cache reflects the hysteresis of the considered algorithms.

Start sequence	0123	0132	0231	0213	0312	0321	1023	1032	1203	1230	1302	1320	2013
LRU	0	1	1	1	2	2	1	2	1	1	2	2	2
PLRUm	3	4	2	4	5	3	4	5	4	3	5	4	5
PLRUt	0	1	2	1	2	3	1	2	2	2	3	1	2
Start sequence	2031	2103	2130	2301	2310	3012	3021	3102	3120	3201	3210	*Average*	
LRU	2	2	2	2	2	3	3	3	3	3	3	1.92	
PLRUm	3	5	4	3	4	6	4	6	5	4	5	4.17	
PLRUt	3	3	3	3	2	3	4	4	4	4	3	2.42	

Table 12: Number of hits for the N_{ways}+1-step loop with different filling orders

4. Cache set associativity

The question seems evident but as it asked only a small additional effort to check it, simulations were performed to investigate the impact of associativity on the hit ratio. The results are split in two categories: on one hand the reasonable values for implementation and on the other hand the results

lightening the behaviour of the replacement strategies. This division is performed in accordance with the number of requested parallel lookups, which induce important power consumption.

4.1. 2-way vs. 4-way set associative caches

4.1.1. Equivalence of the policies

This simulation allowed us to check the equivalence of LRU, MPLRU, non-MRU, PLRUm, PLRUt, Side and 1-bit in a 2-way set associative cache, thereby validating the coded algorithms. In accordance with the theoretical equivalence of these strategies for 2-way set associative caches, *exactly* the same hit ratios were observed.

4.1.2. Impact on hit ratio

The results of Table 13 demonstrate the performance improvement due to the associativity increase, on average 0.5% in gross figures of the hit ratio. The figures are given here for the software because it seemed important to stress the data cache as near to reality as possible. The gain is 5-10%, depending on the policies and on the memory request patterns. Increasing the associativity releases constraints on the placement of a line and then implies less conflict misses. This explains the observed performance enhancement that justifies the use of four ways instead of two. This is all the more true as the power consumption and the hardware cost of performing four look-ups in parallel are still reasonable.

Replacement policy	1-bit	LRU	MPLRU	PLRUm	PLRUt	Random	Round Robin	Side
4 ways / 2 ways in miss ratio	0.885	0.951	0.923	0.900	0.954	0.921	0.945	0.959

Table 13: Miss ratio in 4-way and 2-way set associative caches for software averaged over the different cache sizes

Specificity of 1-bit. The specificity of the 1-bit improvement comes from its particular good results on the `maze` program. Indeed, its structure seems very well adapted to such a program. An explanation can be obtained by examining the C code in details. First, the structures node and connections are respectively 12 and 20-byte wide. Thus, a node lies with two neighbour nodes on its cache line; the relative positions inside the line depend on data alignments. The MRU node should be protected from eviction in order to be available if the algorithm has to come back from a dead path. All the policies perform it but another important point is not to discard the neighbour nodes too because they are very likely to be accessed too. Indeed, the nodes are usually numbered so that it is connected to its neighbours. This characteristic feature explains the particular good performance of 1-bit. Apart from 1-bit, the results are the same for all the replacement policies. In the further study of associativity, only the candidates for implementation will then be simulated.

4.2. High associativity

Although the increases in power and area of the execution of eight or even sixteen lookups in parallel appear as too important, associativities of eight and sixteen were simulated in order to investigate the behaviour of the replacement policies. The results are given in Table 14. The selection of benchmarks was used to smooth the 1-bit specificity on `maze` a bit.

Replacement policy	1-bit	GRR	PLRUm	PLRUt
8 ways / 4 ways in miss ratio	0.997	1.004	1.000	0.998
16 ways / 8 ways in miss ratio	1.000	1.006	1.003	1.000

Table 14: Impact of high associativity on miss ratio for software and the selection of benchmarks

Increasing the associativity to eight still yields some improvements. The benchmarks and the software applications can be regrouped in two sets: the first one where the improvement is almost zero and the other where the enhancement is significant. These comprise `maze`, `telecom_viterbi`, `networking_pktflow`, `networking_ospf`, `automotive_matrix` and `office_bezier`; they correspond to an important working set, which explains that the impact is visible. For the others, only side-effects can be detected. Finally, these numbers must be confronted with the improvements

obtained by the replacement algorithms. An improvement of one percent from 4 to 8 thus appears as a quite small increase. Adding it to the cost of lookups, it justifies keeping the current organization.

One notices that increasing to 16 ways is useless. This figure seems strange in the sense that the performance degrades with higher associativity but this phenomenon has been already reported in the literature [MIL03, ZOU04]. For 1-bit, there is a loss of performance but almost insignificant. This can be explained by the fact that the original aim of the algorithm (protect the MRU region) is perverted by the fact that the region it protects for high associativity is greater than the MRU region, thereby contributing to a pollution of the cache. This is balanced by the impact of the evolution on higher associativity caches for programs such as maze. The phenomenon of the Belady's anomaly is a well-known phenomenon for the FIFO algorithm [BEL69]. GRR obviously presents the same anomaly, which can partially explain the above evolution.

Sequence	3	2	1	0	3	4	1	2	3	4	0	2	3	1	0
3-way															
way 0	3	3	3	0	0	4	4	2	2	4	4	2	2	1	1
way 1	-	2	2	2	3	3	3	3	3	3	3	3	3	3	3
way 2	-	-	1	1	1	1	1	1	1	1	0	0	0	0	0
status bits	001	011	100	101	010	011	100	101	010	011	100	101	010	011	100
4-way															
way 0	3	3	3	3	3	3	3	2	2	2	0	0	0	1	1
way 1	-	2	2	2	2	4	4	4	3	3	3	2	2	2	0
way 2	-	-	1	1	1	1	1	1	1	1	1	1	3	3	3
way 3	-	-	-	0	0	0	0	0	0	4	4	4	4	4	4
status bits	0001	0011	0111	1000	1001	1011	0100	0101	0111	1000	1001	1011	0100	0101	0111

Table 15: Equivalent of Belady's anomaly for PLRUm

Belady's anomaly for PLRUm. Like Round Robin, PLRUm exhibits Belady's anomaly as it is shown in Table 15 where the accesses resulting in a miss are highlighted in grey. The number of misses is 11 for the 3-way implementation and 13 for the 4-way implementation. Belady's anomaly comes from the reset phase which induced an erroneous assignment of the pseudo-LRU block on step 6. Although Belady's anomaly is a well-known phenomenon and though pseudo-LRUs are widespread in the industrial world, it is surprisingly the first time such a phenomenon is reported, at least at the knowledge of the author of this thesis and of its researches through the internet. Besides, this anomaly could significantly affect the overall behaviour since it seems a not so rare situation and can be easily encountered in implementations allowing the designer to lock some ways.

5. Conclusion: which replacement algorithms will be selected?

In pursuance of the detailed investigation performed in this chapter, three candidates are eligible for implementation:

- PLRUm: its performance is the best one observed, it can easily be implemented in hardware. Unfortunately, it requires four bits per set, which is one more than PLRUt. It can be a crucial design criterion when there is a large number of cache lines,
- PLRUt: it is a very good approximation of the LRU algorithm, although it is always outperformed by PLRUm. Whereas its encoding/decoding is a bit more complex than the PLRUm one, it requires only three bits per cache set, thereby saving some storage space in comparison with PLRUm. This could be a clincher for the implementation where a status bits cache is designed,
- 1-bit: it performs quite well, almost in the same range as PLRUt but is also unsteady. Furthermore, it raises the problem of generating truly random sequences for the selection of the evicted line among a set half. Nevertheless, it should be acceptable, considering its low designing cost and the final decision concerning this strategy will be taken in the next chapter.

The other replacement algorithms do not suit the aims of this thesis: either their performance is not sufficient or they design cost is too important. The replacement policies studied further are then the two pseudo-LRU algorithms PLRUt and PLRU as well as the 1-bit strategy.

Chapter 6

The cache implementation

Theory is when everything is known but nothing works. Practice is when everything works but nobody knows why. If practice and theory are spliced, nothing works and nobody knows why.

A. Einstein

In the previous chapter, different replacement policies were studied and their impact was compared with their estimated designing cost. The choice of different candidates naturally leads us to deal with the different solutions for implementation. They will be presented in this chapter but before dealing with the details of each proposal, general characteristics common to the three implementations are handled, among them replacement policies' hardware description.

1. Replacement policies implementation

The specific characteristics of the ARM11 architecture, which will have to be handled in this thesis and which have already been presented in Chapter 4, are briefly addressed here as well as their interactions with the replacement strategies.

1.1. Integration of the lockdown feature

As it has already been stated in Chapter 4, the ways of the data cache can be locked: they must not be replaced and not considered as candidates for eviction by the replacement policy. This feature can be handled thanks to the following methods suggested for each algorithm:

***1-bit*:** if the bit points to a completely locked half, the discarded way is randomly chosen among the MRU group. Otherwise, the random selection is performed in the pointed half.

***PLRUm*:** while examining the ways to see if a reset is required, the locked ways must be counted as 1 even if their status bit is 0. On update, they must also be considered as high. During the reinitialization of the status bits, the bits corresponding to the locked ways must be set to 1 too.

***PLRUt*:** the situation depends on bit 0. If it points to a fully locked half, it must be inversed provided that the other half is not locked too. Though a complete locked set is a situation rejected by the ARM11 specification, this algorithm is able to handle it. Otherwise, the bit only stands for the negation of the hit half. The process is identical for bits 1 and 2 except that these bits represent a fully locked group. In this case, it is decided that they point to their first half (i.e. they are set to 0).

Therefore, the locked ways are always considered as locked, on update as well as on allocation; thereby avoiding any conflict between the lockdown feature and the integration of a new replacement policy in the ARM11 architecture. In respect with the specification, the lockdown is reversible for the replacement policy too. At any moment, the ways can be locked/unlocked and the replacement strategy will immediately adapt its computations.

The only point is that the algorithm remembers these ways as recently accessed. Yet, removing the locking on these ways is a programmer action, thereby surely meaning that these data are useless now and can be evicted not to pollute the cache. The implemented solution does not take care of this but it should be an interesting improvement for the system to test: while removing the locking on some ways, these ways are marked as LRU on their next accesses. Of course, this would imply many technical issues since a cleaning of these status bits could not occur without degrading significantly the current performance. A solution would be to perform this modification only on the next access on this set but it assumes that the information whether the modification has been done and which ways have been unlocked should be stored somewhere. Because of these technical problems, the lack of time and the quite rare occurrence of these patterns, this potential enhancement was not studied in this thesis. However, it should be an interesting idea for further works on neighbouring themes.

1.2. Hazards

Like any digital systems, the data side architecture is faced with hazards that have already been solved by the ARM engineers. These issues are briefly addressed in this subsection to evaluate their impact on the considered implementations. The other hazards that are specific to the proposals of implementation will be handled in the following sections.

RAW (Read After Write) and WAR (Write After Read). These patterns do not modify the updated status bits because they access the same data cache line sequentially. Indeed, accessing twice the same line is transparent for the replacement policies because the status bits point to the same evicted line or evicted group.

Cache line hazards. If the data has been modified between the lookup phase and the write of a data, a new cache lookup is performed. Thus, a new computation of the status bits will be done; which makes the status bit compliant with the state of the line.

1.3. Updating the status bits

Before addressing the different implementations in details, the updating process of the different algorithms are dealt with in this section. They are independent of the implementation and it will allow us examining the impact of the replacement policy in terms of gates and power. The signal `way`, present in the codes below, is either the data cache hit way or the way used by allocation, write… The other signals' names are transparent. In the codes below, the following convention is applied: all signals whose name is `*_reg` are flip-flops, those which finish in `_i` are inputs of the module.

1.3.1. Sharing the updating hardware

Since only one data cache set is accessed at a given time, there is a need for a single status bit updater for the data cache. The hardware may be shared among all the data cache sets. Yet, the presence of modules dedicated in the RAM and of buffers on some of the implementations presented in this chapter will increase the need of updating modules to three or four. This result depends on the implementation and will be addressed in details in the next sections.

1.3.2. 1-bit

The Verilog code of the updating process for 1-bit policy is shown below. It is a bit more complicate than the original algorithm to take care of the locked ways.

```
wire       first_half;
wire       second_half;
wire       first_half_locked;
wire       second_half_locked;
wire       updated_sb;
reg [3:0]  lfsr_reg;
wire       lfsr_feedback;

// Before updating, a check on locking is performed
assign first_half_locked  = cp15_locked_ways_i[0] && cp15_locked_ways_i[1];
assign second_half_locked = cp15_locked_ways_i[2] && cp15_locked_ways_i[3];

// Update with 1-bit
```

```
assign first_half  = way[0] || way[1];
assign second_half = way[2] || way[3];
assign updated_sb  = (first_half && ~second_half_locked) || first_half_locked;

// Assigning the way to evict
always @(posedge clk_i or negedge nreset_i)
  if (~nreset_i) lfsr_reg <= 4'b0100;
  else lfsr_reg <= { lfsr_reg[2:0], lfsr_feedback};

assign lfsr_feedback       = ~(lfsr_reg[2] ^ lfsr_reg[3]);
assign alloc_way_busy_1bit = { status_bits_reg &&  lfsr_reg[3],
                               status_bits_reg && ~lfsr_reg[3],
                              ~status_bits_reg &&  lfsr_reg[3],
                              ~status_bits_reg && ~lfsr_reg[3]};
```

Code 1: 1-bit allocation and update of the status bits with lockdown feature

The overall required hardware for the 1-bit policy is thus around 10 gates and one 4-bit register with negative reset for the generation of the random sequence. The random sequence is performed thanks to a Linear Feedback Shift Register whose tap configuration is optimal according to [XIL96]. The LFSR width is a good compromise between the wanted performance and the hardware cost. Nevertheless, it should be easily modified to be nearer to a truly random sequence by incrementing the width of the LFSR sequence. This study will be performed in Section 6.2.5. The initial value set on a reset is not important, provided that it is non zero. Eventually, the cost of the implementation of the 1-bit policy is negligible in amount, area and power.

1.3.3. PLRUm

Let us write Verilog code which implements the update of the PLRUm status bits and compute the allocated way in Code 2. The status bits are first modified to be compliant with the current locked ways. Then the update is performed in accordance with the hit information. If the lookup resulted in a miss, the computation of the way to discard is based on the status bits altered to take the lockdown in account.

```
wire       reset_status_bits;
wire [3:0] new_status_bits;
wire [3:0] status_bits_pregated;
wire [3:0] sb_initial_value;
wire [3:0] sb_initial_value_int;

//Locked ways must be considered locked
assign status_bits_pregated    = status_bits_reg | cp15_locked_ways_i;
assign status_bits_gated_plrum = (&status_bits_pregated) ? cp15_locked_ways_i :
status_bits_pregated;

//Update of the status bits with PLRUm
assign sb_initial_value_int       = cp15_locked_ways_i | plru_way;
assign sb_initial_value           = (&sb_initial_value_int) ? cp15_locked_ways_i :
sb_initial_value_int;
assign new_status_bits            = plru_way | status_bits_gated_plrum;
assign reset_status_bits          = &new_status_bits;
assign updated_status_bits_plrum  = ( {4{reset_status_bits}} & sb_initial_value) |
( {4{~reset_status_bits}} & new_status_bits);

//Allocation of the evicted way with PLRUm
assign status_bits_alloc_init = status_bits_reg | lfb_cp15_locked_ways;
assign status_bits_alloc      = (&status_bits_alloc_init) ? lfb_cp15_locked_ways :
status_bits_alloc_init;

always @(status_bits_alloc)
  if (~status_bits_alloc[0]) alloc_way_busy_plrum = 4'b0001;
  else if (~status_bits_alloc[1]) alloc_way_busy_plrum = 4'b0010;
  else if (~status_bits_alloc[2]) alloc_way_busy_plrum = 4'b0100;
  else if (~status_bits_alloc[3]) alloc_way_busy_plrum = 4'b1000;
  else alloc_way_busy_plrum = 4'bXXXX;
```

Code 2: PLRUm allocation and update of the status bits with lockdown feature

On examination of the code, one notices that around 60 gates and a 4-bit register with asynchronous negative reset are required. Once again, this can be considered as negligible in amount of gates and power in comparison with the figures of the core.

1.3.4. PLRUt

Let us write Verilog code which implements a PLRUt status bits update, in order to evaluate the area of these update features. This is done in Code 3. The process is similar to the one presented for PLRUm.

```
wire [3:0] updated_plrut;
wire       first_half;
wire       second_half;
wire       first_half_locked;
wire       second_half_locked;

// Before updating, check on locking is performed
assign first_half_locked  = cp15_locked_ways_i[0] && cp15_locked_ways_i[1];
assign second_half_locked = cp15_locked_ways_i[2] && cp15_locked_ways_i[3];
assign first_half         = way[0] || way[1];
assign second_half        = way[2] || way[3];

// Update with PLRUt
assign updated_plrut[0] = (first_half && ~second_half_locked)|| first_half_locked;
assign updated_plrut[1] = cp15_locked_ways_i[0] || (~cp15_locked_ways_i[1] &&
    (first_half && way[0] || second_half && status_bits_reg[1]));
assign updated_plrut[2] = cp15_locked_ways_i[2] || (~cp15_locked_ways_i[3] &&
    (second_half && plru_way[2] || first_half && status_bits_reg[2]));
assign updated_status_bits_plrut[3] = 0;

// Allocation of the evicted way with PLRUt
assign lfb_first_half_locked  = lfb_cp15_locked_ways[0] & lfb_cp15_locked_ways[1];
assign lfb_second_half_locked = lfb_cp15_locked_ways[2] & lfb_cp15_locked_ways[3];
assign sb_gated_plrut[0] = lfb_first_half_locked || (~lfb_second_half_locked &&
status_bits_reg[0]);
assign sb_gated_plrut[1] = lfb_cp15_locked_ways[0] || (~lfb_cp15_locked_ways[1] &&
status_bits_reg[1]);
assign sb_gated_plrut[2] = lfb_cp15_locked_ways[2] || (~lfb_cp15_locked_ways[3] &&
status_bits_reg[2]);
assign alloc_way_busy_plrut ={sb_gated_plrut[0] &&  sb_gated_plrut[2],
                               sb_gated_plrut[0] && ~sb_gated_plrut[2],
                              ~sb_gated_plrut[0] &&  sb_gated_plrut[1],
                              ~sb_gated_plrut[0] &&  sb_gated_plrut[1]};
```

Code 3: PLRUt allocation and update of the status bits with lockdown feature

The PLRUt updating and allocating module is estimated to around 30 gates and a 4-bit register. The difference between the figures of the two pseudo-LRUs may question. However, this is explained by the nature of the algorithms. Indeed, PLRUm must not only modify the original status bits in regards with the locked ways like PLRUt but also perform a reset check on these status bits. This additional step is the origin of the additional hardware and of its improved performance. Nevertheless, the two implementations require a negligible amount of hardware and their power consumption will be insignificant too.

The required hardware for updating the status bits is negligible for the three different policies. Registers storing the status bits must be added to this estimation, but it is only four flip-flops… The power consumption will be insignificant too. The update is one of the two aspects of the replacement policy, the other being assigning the evicted way. This is the theme of the next subsection.

1.4. Allocating a way

The question of allocating a way can appear simple at first glimpse. However, it must be noticed that the allocation is not immediate: the Line Fill Buffers require time to fetch the data and only then request an access to the Data RAMs. As a result, there is here a possibility of hazards: a way of a set may be allocated twice successively, thereby leading to the eviction of the MRU line. Fortunately, this can be quite easily solved by storing the last allocated lines in registers. According to the specification, only two successive misses dealt by the Line Fill Buffers will not stall the processor. Consequently, keeping information about the two last allocated lines and their index will enable avoiding these

hazards. Of course, this piece of information is erased once the corresponding allocation has been performed. These lines will then be considered as temporarily locked by the replacement policies if the index matches the index of the current access. This complicates the allocation of the way by the algorithm but is necessary. After all, it adds only some gates but the cost is more important for registers. The 9-bit wide index as well as the way must be stored, thereby increasing the overall gates by an amount of 26 registers. Considering the importance of such a hazard which may lead to inefficient replacements of the MRU line of a set, this cost is acceptable and the solution approved. The codes presented above already integrate this feature: the signal `lfb_cp15_locked_ways` integrates the locked ways and the way where the Line Fill Buffers will write if the current index matches the allocated indexes.

According to the codes given above, the hardware required for updating the status bits and allocating the way to discard in accordance with these policies is negligible in comparison with the size of a processor. The real impact on the system will be the access to the status bits and the way they will be stored and accessed. This issue is addressed in the next section.

2. Status bits implementation

2.1. How to update the status bits

When the hit information about a memory request is available, the address presented at the RAMs has already changed and may differ from the data address of the hit/miss computed in hit stage. It is thus impossible to update directly the status bits in the RAM as soon as we receive the old status bits because they would be written to a wrong address or an erroneous update based on wrong hit information would be computed. A specific system ensuring the status bits to be written must be designed. In the implementations presented here, the updated status bits are computed by module hit stage. This calculation can be done only when the hit/miss and hit way computation are performed. Status bits thus need to be registered before updating, which explains the 4-bit registers already counted in the previous section. According to the estimations of the previous subsections, only four stages of basic gates are required, which should not constrain further the critical path in hit stage since equivalent operations are performed in the current implementation.

2.2. Storage of the status bits

The replacement policies evoked in the previous chapters need to store pieces of information for each data cache index. This observation raises the issue of storing these additional bits. The different considered solutions are presented in the subsections above. Hereafter, the term status bit group will be used to designate the status bits of a data cache set. For PLRUm, it would be four-bit wide and for PLRUt only three.

2.2.1. Flip-flops

The first solution is to store the status bits in flip-flops. As the status bits can be read for memory access *n+1* and updated for memory access *n* during the same clock period, there would be two different addresses: one for writing and one for reading. It would suppose to latch the data address at the output of the arbiter to ensure that this address is available when the hit stage module computes the updated value. There should be some hazard if read and write addresses are equal but this can be further studied and solved if this solution will finally be chosen.

The advantage of this solution is its simplicity of implementation but it is at the price of gates. Indeed, it would require for a 64 KB cache 512 *N* flip flops (*N* is the number of bits required by the replacement policy from 1 for 1-bit to 5 for LRU in our encoding) and 2 9-bit decoders handling accesses to the different flip-flops.

2.2.2. Reuse the available RAMs

Because of the large size and the important consumption of the flip-flops, another solution has been naturally investigated: storing the status bits in one of the available RAMs. Indeed, the data side is already provided with RAMs which store tags, dirty bits, modified bits, MESI bits… The two possibilities are examined: Dirty RAM and Tag RAM.

Dirty RAM. The Dirty RAM presents the advantage of being more natural because it is a stock per set. We will need up to four bits so there is enough space and keeps the advantage of modularity for the choice of the replacement policy.

Tag RAM. This possibility seems not very functional because the information is specific to a cache set and not to a way but one bit per way can be stored in each Tag RAM. This solution would be practical for PLRUm but wastes space and gives strange control for the other replacement policies. Moreover, it would require activating the four Tag RAMs to get the status bits whereas the Dirty RAM will require powering on only one RAM. Definitely, it does not appear to be the smartest choice.

Reusing the Dirty RAM and not the Tag RAM seems a more efficient and less power consuming possibility. The drive of the enable signals on sequential accesses would have to be modified a bit with this solution. The main advantage is to be integrated in the actual architecture and then does not require any additional control hardware for the access itself. A lot of area is also saved by avoiding so many flip flops. Finally, only a few additional bits are required so there remains "virtual free bits" for future evolutions.

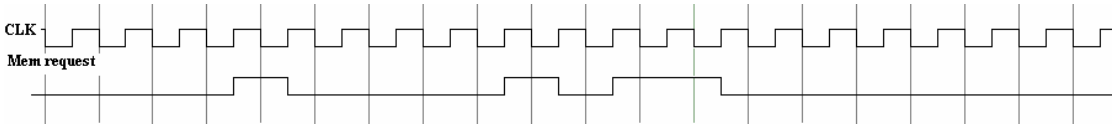

Figure 28: Memory requests of a core on a test bench

The inconvenient is the growing complexity to deal with the RAM but it allows us to include ourselves in the actual architecture and to mime the manner the hazards are solved. Moreover, there will be a loss of cycles for non-sequential reads to allow storing the updated status bits. Fortunately, this will not be seen by the core provided that the read data will be available at the same moment as before. As the ModelSim simulation shows (see Figure 28), there are usually a quite important number of cycles between two memory requests from the core, thereby giving us the possibility to use some of these "lazy" cycles to store our updated bits.

2.2.3. A new RAM

The solutions above present some important drawbacks in terms of power: the Dirty RAM has to be activated on almost each cache access… Therefore, the creation of a dedicated RAM has been studied. Indeed, the accessed data are neighbours in time or in space in virtue of the principle of locality. This principle will also apply to the requested cache sets. Consequently, many accesses to the RAM could be merged, thereby saving power. This gain could be even more significant if this RAM is endowed with a small fully-associative cache, which would enable updating the status bits in the same cycle as the hit information is provided.

The main problems considering the storage of the status bits have been addressed in the two sections above. Three possibilities of implementation emerged from the previous study:

- reusing the Dirty RAM for its simplicity of implementation and its probably interesting results,
- storing the status bits in a dedicated RAM in order to merge the different accesses and saves some power,
- storing the status bits in a dedicated RAM endowed with a small fully-associative cache in order to decrease the number of power consuming RAM accesses.

These propositions are examined in details in the following sections. Their advantages and drawbacks will be evoked, thereby leading us to the final choice of implementation.

3. In the Dirty RAM

The implementation inside the Dirty RAM is developed below and the resulting architecture is given on Figure 29. On this figure as well as on Figure 30 and Figure 32, the modules that should be significantly modified are represented in orange whereas the blue colour stands for created modules and connections.

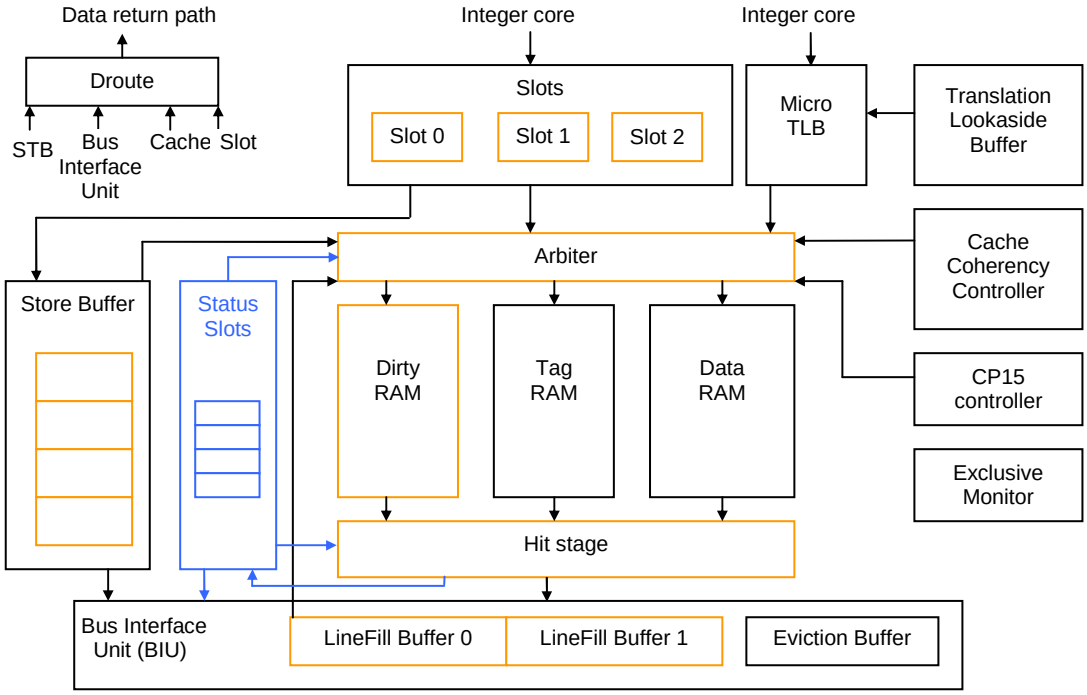

Figure 29: Architecture of the Dirty RAM solution

3.1. Creation of a status slots module

In the previous section, it has been shown that the updated bits cannot be stored as soon as they are available and thus some clock cycles could be wasted. In order to avoid stalling the RAM while storing the updated status bits, a small module is created in which the updated status bits to be stored are stocked, waiting for the arbiter to give them access to the RAMs. This module will be called the status slots module. The advantage of this solution is to access the RAM only when no other module requests it. Provided that the module is well-dimensioned, it will not influence the replacement policy's performance but will induce a better usage of the lazy cycles presented on Figure 28.

3.1.1. Priority of the module

The system performance is dictated by the core thus its requests must have priority for accessing to the RAMs. This would be a pity if the increase in performance due to the new replacement policy would be compensated by delaying core memory accesses (particularly the sequential ones). To avoid this situation, the module should be granted a low priority even lower than the Store Buffer: the other features are essential for the functioning, not ours. As there are unused clock cycles, the update is able to be performed later.

3.1.2. Dimensioning the status slots module

The module is divided into slots, each of them corresponding to the status bits of a single cache set. Therefore it stores thirteen bits, four of them being the binary representation of the status bits and the nine remaining encoding the cache set information. A good comparison for the dimensioning of this module is the size of the Store Buffers and of the Slots Unit, which are approximately faced with the same workload. Considering that the new module will receive fewer requests than these slots[a], three or four slots should be sufficient. However, it must be noticed that the module is given the least priority, thus it will be difficult for it to be granted access, which balances the observation of a less important traffic. As a result, four or five slots should be a good compromise. Of course, the implementation is worthy as long as there are lazy cycles to avoid losing too many status bits. Indeed, if the module is full, the LRU information for the corresponding set will be lost since it is non-essential for the functioning of the data side. This comparison with the Store Buffer and the Status Slots Unit is all the

[a] As it is explained on the behaviour of the module in the subsections below, the Status Slots Unit will receive only the lookups that resulted in a hit.

more valid since the Status Slots Unit will be able to merge data which are already present with new data. The data being regrouped in the Status Slots Unit, the locality should be even stronger.

3.2. Actions on memory requests

We have seen that the status slots module will receive the updated status bits and store them before writing them back when it receives grant from the arbiter. Unfortunately, this situation wastes some clock cycles only to update the status bits and is consequently not optimal. As a result, the storing of the updated status bits can be improved by performing it meanwhile other cache actions occurring at the same data cache index: eviction, invalidation, data write of a write request, second sequential access of a burst… This optimization is enabled by the correlation of these actions: either the last one induced the modification of the status bits, or they result from a cache action which implied the update of the status bits too. This optimization has also drawbacks: the updated status bits ready to be stored back in the Dirty RAM can be stored on different emplacements, not only the status slots module but other modules too. However, each of these modules has an exclusive copy: only this module and the RAM own the status bits of the set. When a module owns a copy, the RAM copy is dirty and the module one is clean. When no module owns the cache set status bits, the copy in the RAM is up to date. This issue is dealt with in much more details in Section 3.3.

In the subsections below, it is thus assumed that the status bits entering the hit stage module are up to date and it is not cared where they come from. The different types of cache actions available in the data side of the ARM11 MPCore processor are addressed below and are the following:

- read request: sequential or not,
- write request: sequential or not,
- eviction of a line,
- invalidation of a line.

3.2.1. Read requests

If the lookup results in a miss, the allocated way is deduced from the status bits and then sent to the Slots Unit. In case of hit, the hit stage also computes the updated status bits. The action then depends on the type of the transaction:

- if it is a sequential access (i.e. burst), the second sequential access[a] (if it exists) can be used to update the status bits. Indeed, the sequential requests access to the same line and thus do not alter the status bits. So, the Slots Unit can write the update bits on a sequential access. The Dirty RAM must be specifically enabled for this purpose during the second sequential access.
- if it is not sequential or if the sequentiality is less than 3, the updated status bits are sent to the status slots module. This sequentiality is known by the Slots Unit and it should forward this signal to the status slots module for it to be aware of taking the updated status bits on a hit or not.

This feature may save a bit clock cycles but may not be implemented in a first version.

3.2.2. Line Fill

During a line fill, all the information relevant to the line is written; among them some are stored in the Dirty RAM. As a result, it seems possible to write the updated status bits in the same time as the dirtiness information. Three solutions were considered:

- The Line Fill Buffer writes itself the updated status bits and fetches them from hit stage on a miss. The role of hit stage here is only to transmit the status bits. The drawback of the implementation is that the two Line Fill Buffers must be significantly modified and that the storage for the status bits should be added to each buffer (up to 5 bits). It creates a copy of the status bits in a different module and may create additional conflicts. Nevertheless, it appears efficient in the sense that it optimizes the RAM access usage,
- The status bits are stored in the status slots on miss too and the Line Fill Buffer gets them when granted access. This assumes that the Line Fill Buffer communicates with

[a] The second sequential access is the third access of the burst. Indeed, the first one has never been considered as sequential since the second access is required to compute it. The information is thus really available on the third one.

the status slots before being granted; which is quite complex and somehow far from optimal because it introduces unnecessary communication in the system,
- The Line Fill Buffer does not write the updated status bits and does not store it. It is left to the Status Slots Unit but the utilization of the RAM accesses is then non optimal.

The first solution is preached despite its greater complexity since it should have strong positive effects on the power consumption.

3.2.3. Write requests

The hit stage computes the updated status bits and forwards them to the Line Fill Buffer and to the Store Buffer. If it is a hit, the Store Buffer accepts it and the Line Fill Buffer does not. On a miss, it is the opposite. The case of a write miss is so managed by a Line Fill Buffer. These requests have already been evoked in the previous section so only the write lookups resulting in a hit are considered further here.

On a hit, the Store Buffer stores the updated status bits in a slot and then requests the arbiter an access to the RAMs as usual. When it gets grant, it will store the updated status bits in the Dirty RAM while it stores the data in the cache. The Store Buffer thus plays an equivalent role to the Line Fill Buffers: it stores the updated status bits which it will soon be able to write in the Dirty RAM. Some bits must also be added to the basic slot to make this solution feasible.

3.2.4. Eviction

The line is marked as invalid and is now recognized as free by the replacement policies. These algorithms taking care of this piece of information, there is no need to modify the status bits on eviction; it will be done when the set of the line will be used again.

3.3. Obtaining the up-to-date version of the status bits

The status bits are stored in the Dirty RAM and in different modules: STB, status slots, LFB0, LFB1… The exclusivity of this copy among the modules has been admitted. This hypothesis is checked after examination of the actions on different memory requests. Indeed, it has been shown by construction that only one module receives the status bits for a given action and these status bits are the most recent ones. Therefore, either a single module has the up-to-date version of the status bits and the Dirty RAM a dirty copy of it or no module is in possession of it and the up-to-date copy lies in the RAM.

As a result, while the address is presented to the RAMs, it can also be sent to the storing elements which will transmit whether they own the given status bits. This lookup will be fast because it is performed on flip-flops and it is a parallel execution across all the modules that forward the ownership information to the hit stage module (this signal must be gated to arrive at the same time as the data from the RAMs). If none of the signals is high, the status bits from the RAM are up to date. If one of them is high, the status bits from the given module are taken. This allows us to get the up to date version of the status bits. This feature relies on the property demonstrated in the previous paragraph.

3.4. Sum up

Compared with the ARM11 MPCore processor version, the additions are:
- the Status Slots Unit (five 13-bit registers, five 9-bit comparators and the control logic),
- Line Fill Buffer: 2x4 bits and 2 comparators and a few gates to handle these registers,
- Store Buffer: 4x4 bits, 4 comparators and a little logic,
- logic for updating the status bits (roughly 10 registers and 50-100 gates per algorithm)
- Slots Unit: 3x4 bits and the logic needed to handle the specificity of sequentiality.

These elements seem quite acceptable additions for the implementation of a new system. However, attempts to reach optimality as near as possible (particularly in consumption) lead us to implement a solution where the status bits are widespread in the system. Exclusivity should be ensured but it seems a quite intricate system to manage and source of numerous hazards (especially if the killing signals and other ARM11 optimizations are taken into account). Combining this with the numerous writings in the Dirty RAM, it makes us thinking of a new implementation. The first idea is to store the status bits in an

external RAM, where a line will store different status bit groups. This solution relies on the principle of locality and should yield interesting enhancements. The detailed study is presented below.

4. In a new RAM

Using the Dirty RAM presented the drawback of accessing a whole RAM line to store the updated status bits whereas the other components of the line still remain unchanged in most cases. This proposal is then far from optimal. Moreover, it is possible to take more advantage of the principle of locality: a separated RAM solution would allow us to order the RAM in a more efficient way by regrouping neighbouring sets, thereby decreasing the amount of accesses to the RAM and thus saving power. This is beneficial provided that a significant amount of accesses is merged. Apart from the organization of the RAM, which is dealt with in the following subsection, the designed system is almost the same as the previous solution. For this reason, the other parts are briefly addressed at the end of this section. The resulting architecture is drawn on Figure 30.

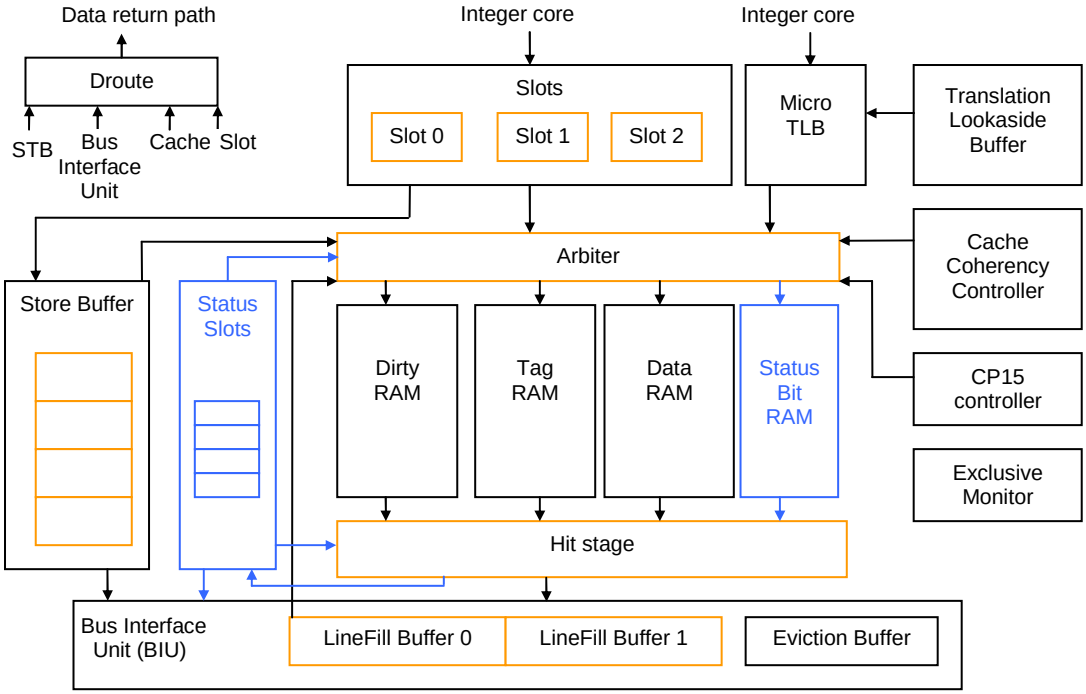

Figure 30: Architecture with the new RAM solution

4.1. The RAM

We suppose that a new RAM is created where four status bits groups are located on the same line. This figure can be explained by the capacity of the RAMs. The single information that must be stored on a status bit RAM line is the status bits groups. For the sake of convenience and of decoding of the addresses, it is a common solution that the number of ways is a multiple of two. Indeed, the address cutting is very simple and fast – one of the major points of concern – for these configurations. The replacement policies that have been evaluated as candidates for implementation require up to four bits per status bit group. If four sets are stored, the required storage is sixteen bits for the replacement strategy. For the next step (eight status bit groups), thirty two bits must be stored, which would be an acceptable figure for the RAM. However, this choice also impacts on the slots module. Indeed, the implementation of a slot will require keeping track not only of the status bits but also of the status bit tag. For a data cache size of 64 KB, there are 512 sets. If we store 4 status bit groups, the status bit tag will be 7-bit wide and 6-bit wide for 8 status bit group. A slot of the slots module must then contain:

- 8x4 + 6 = 38 bits if 8 status bit group are stored per status bit RAM line,
- 4x4 + 7 = 23 bits for 4 status bit groups.

Of course, this is multiplied by the number of slots of the slot module. Therefore, saving four status bit group seems a good compromise to avoid adding numerous registers which are power consuming.

Using the previous data, it is deduced that the created RAM has up to 128 lines. Therefore, addressing it requires seven bits which are derived from the RAM 32-bit address. Applying the same reasoning as in Chapter 2 Section 2.5.3, one deduces that these seven bits are the MSBs (Most Significant Bit) of the index part of the address. The cutting is therefore:

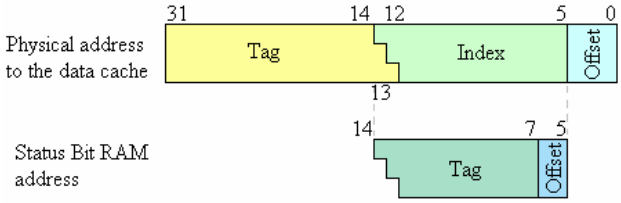

Figure 31: Addressing the status bit RAM

At the output of the RAM, the full 16-bit wide line is forwarded. A multiplexer (whose selector will be the two LSBs (Least Significant Bit) of the index part of the address) multiplexes them and yields the requested output.

4.2. Slots module

The slots module is adapted in the same manner to this new storage. The external constraints applied to the module in terms of power and capacity differs a bit from the first implementation. Indeed, the only significant difference is the width of a slot but this issue has already partially been taken into account while ordering the status bit RAM. However, the greater width of the slot makes us select preferentially a solution with a bit fewer slots: two should be sufficient when it is noticed that a slot corresponds to 16 data cache lines and that there are “only” two Line Fill Buffers. Thus, a slot stores 16 bits for the replacement policy and 7 bits for the Status Bit Cache tag. This is consistent with the higher locality of the module.

In opposition to the first implementation, the merging feature will be critical here. On free cycles, the slots module will be granted access to the RAM and will update the corresponding lines. It should be noticed that there is no need to enable the other RAMs when this module is granted access. If the buffer is full, the LRU information will simply be lost because it is not strictly required for the core to operate correctly. The merging feature helps limiting the number of slots to only two.

4.3. Actions on memory requests

The situation is identical to the first proposal except that the status bit RAM will be checked and invoked instead of the Dirty RAM when stated before. Therefore, this version is faced with the widespread of the status bits through the system too. It must be highlighted that all the memory requests are read for the status bit module. The single difference between writes and reads is that the cache hit way is used for the read requests whereas the specified way is invoked for the write and allocation requests. The writes on the Status Bit Cache is then performed by the updater module.

4.4. Sum up

In comparison with the original ARM11 MPCore processor implementation, the additional hardware is:

- the Status Slots Unit (two 23-bit registers, two 7-bit comparators and a few gates to handle the registers),
- Line Fill Buffer: 2x4 bits, 2 comparators and a little logic,
- Store Buffer: 4x4 bits and 4 comparators,
- logic for updating the status bits (around 10 registers and 50-100 gates per algorithm),
- Slots Unit: 3x4 bits as well as the logic required to handle the specificity of sequentiality.

One of the important improvements is to take further advantage of locality: different accesses to neighbouring sets are merged, thereby preventing from sending write requests to the RAM. However, the other drawbacks of the first implementation remain. Among them the necessity of going through the arbiter and a still important amount of requests to the RAM can be cited. It is observed that the module can act nearly as a cache if the workload from the core is sufficient. This remark sows the seeds

of an idea of improvement: avoiding even further useless accesses to the RAMs which are power consuming by endowing the status bit RAM with its own dedicated cache. This solution that is implemented in this work is dealt with in the next section.

5. In a new RAM with a small cache

Taking further advantage of the principle of locality, this implementation should avoid useless accesses to the RAM and thus will save power. The status bit RAM is named hereafter SBT (Status Bit Table) and the cache of the SBT is SBC (Status Bit Cache). The architecture is presented graphically below. The newly created modules are drawn in blue.

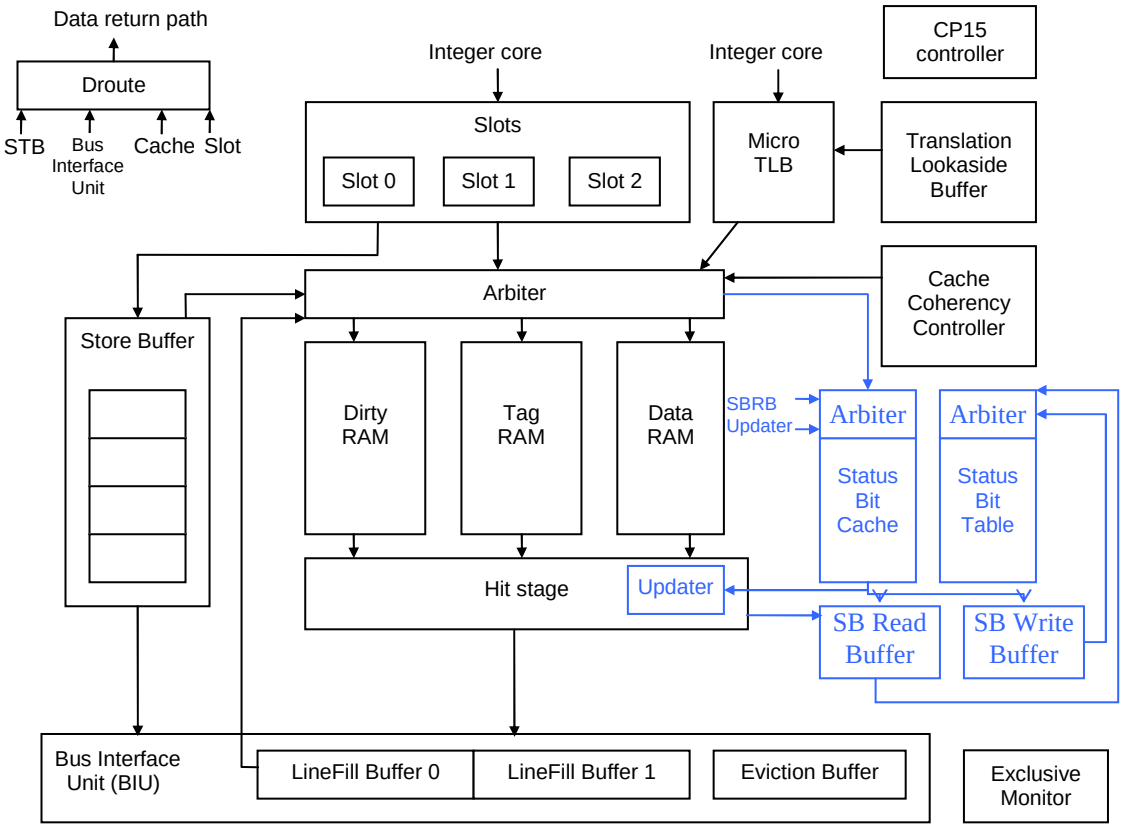

Figure 32: Architecture of the RAM and cache solution

This solution will be advantageous as long as the creation of the cache allows us to save clock cycles and thus to update directly the status bits, as soon as the hit information is known. Therefore the SBC must be implemented in logic (flip-flops). It implies a small cache size but ensures full associativity and requires less power. An order of 16 should be well adapted, which is confronted to simulations in the next subsection.

The updating feature is now performed by the update module which is also responsible for computing the evicting line. Since this module is integrated in hit stage, there is no modification with the previous proposal. This split is the result of modularity: the module is now instantiated in hit stage as well as in the two Status Bit Read Buffers which will require updating the status bits that they fetched from the Status Bit Table.

If there is a hit in the SBC, the situation is the same as before. If it misses, the evicted line (decided by Round Robin for instance) must be written back in the SBT and the requested line must be fetched from the SBT before being stored in the SBC. Since the SBT is a RAM and cannot be immediately and simultaneously accessed to two different addresses, these pieces of information must be stored. Buffers are therefore created:

- Status Bit Write Buffer (SBWB) which contains the line discarded by the SBC and writes it back in the SBT,

- Status Bit Read Buffer (SBRB) which stores the line read from the SBT. It stores the hit information to update the read line. This line is then written back in the SBC.

5.1. Status bit RAM

The internal organization of the RAM is kept identical to the version described in the previous section because the constraints remain roughly unchanged: the amount of bits to be stored is not altered, the impact of the number of status bit groups on the status bit Slots Unit is replaced by the influence on the width of the Status Bit Cache line and on its flip-flops, the external workload does not change… In this configuration, the SBT stores up to 128 lines, each of them contains four status bit groups. Seven tag bits are required to distinguish the lines. Consequently, an SBT line is 23-bit wide, assuming that the replacement policy's bits number is four (PLRUm for instance).

5.2. Status Bit Cache

From the previous paragraph, it is deduced that a SBC line is 23-bit wide. We suppose that the SBC controller is included in the term SBC. In order to obtain some clues about the real efficiency of the dedicated cache, some simulations have been performed with an enhanced version of the cache simulator, which implements a status bits cache. The parameters describing this cache are:

- the number of lines,
- the number of status bit groups it contains per cache line (i.e. the number of data cache sets a Status Bit Cache line stands for).

As it is only a simulation, it is supposed that in case of miss, the data is immediately available. Of course, this is only an approximation but it is quite realistic considering that the access to the RAM, takes in average 1 clock cycle provided that the RAM is available. In comparison with the time required to fetch data from a L2 cache (10 clock cycles) or even from the main memory (100 clock cycles), this time can be really considered as zero. The only equivalent time is the time of a cache access and thus the only issue would be in a sequential access if the cache hits and the Status Bit Cache misses. However, considering the probability of the sequential accesses and the average hit ratio in the SBC and in the data cache, this is a quite uncommon situation. For the scope of this work, this will be considered as a sufficient enough description of the Status Bits Cache.

5.2.1. Dimensioning

One of the problems raised by the dedicated cache implementation is the dimensioning of this cache. The number of lines is constrained by the fully associativity of the Status Bit Cache: there must be a small number of lines in order to limit the control logic (which grows quickly for a fully associative cache) and not to waste power in the flip-flops. Considering it, an interval [4, 16] seems good bounds for this study even if the figure 16 appears quite important. Simulations will show whether it can be further constrained.

The amount of status bit groups per cache line must be decided too. For sake of convenience and dealing with the different addresses, it should be a power of two but other integer solutions can be simulated for knowledge and trends. Since a Status Bit Group is up to 4-bit wide, the range of simulations will be [2, 5] Status Bit Groups per line. It will allow us to confront the dimensioning of the RAM already performed with the simulation and thus confirm or infirm the hypothesis.

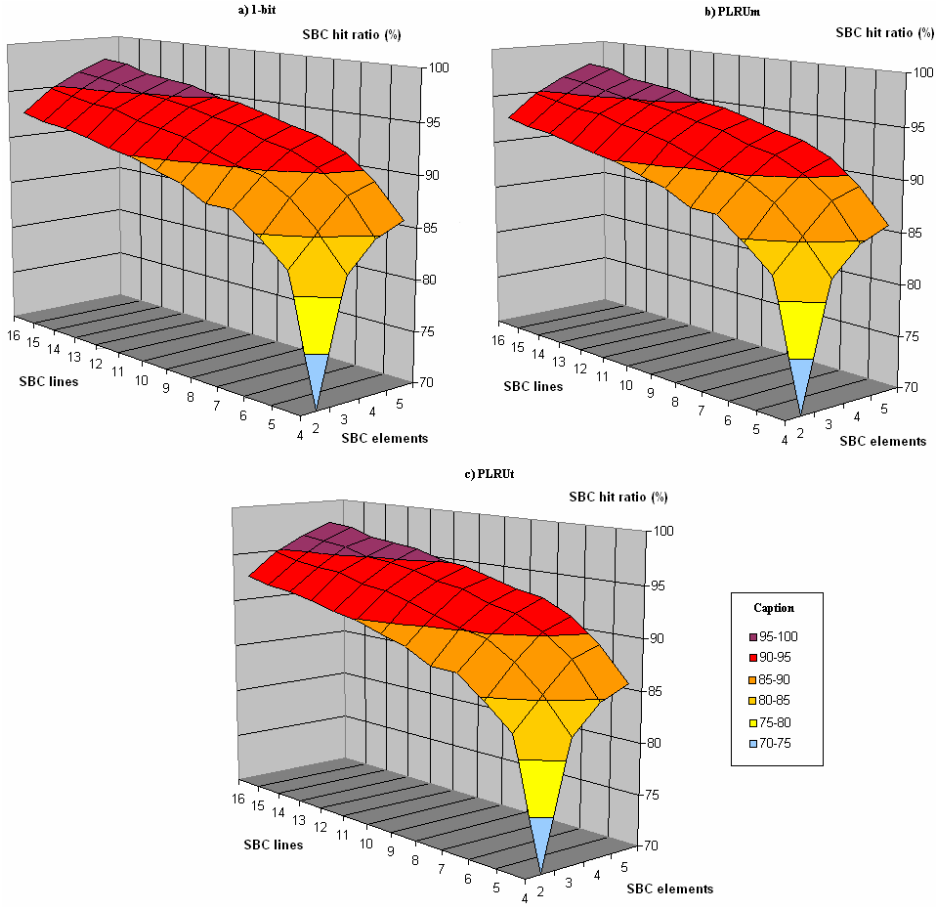

Figure 33: Status Bit Cache hit ratios for a 16-KB 4-way set associative cache on the selection of benchmarks

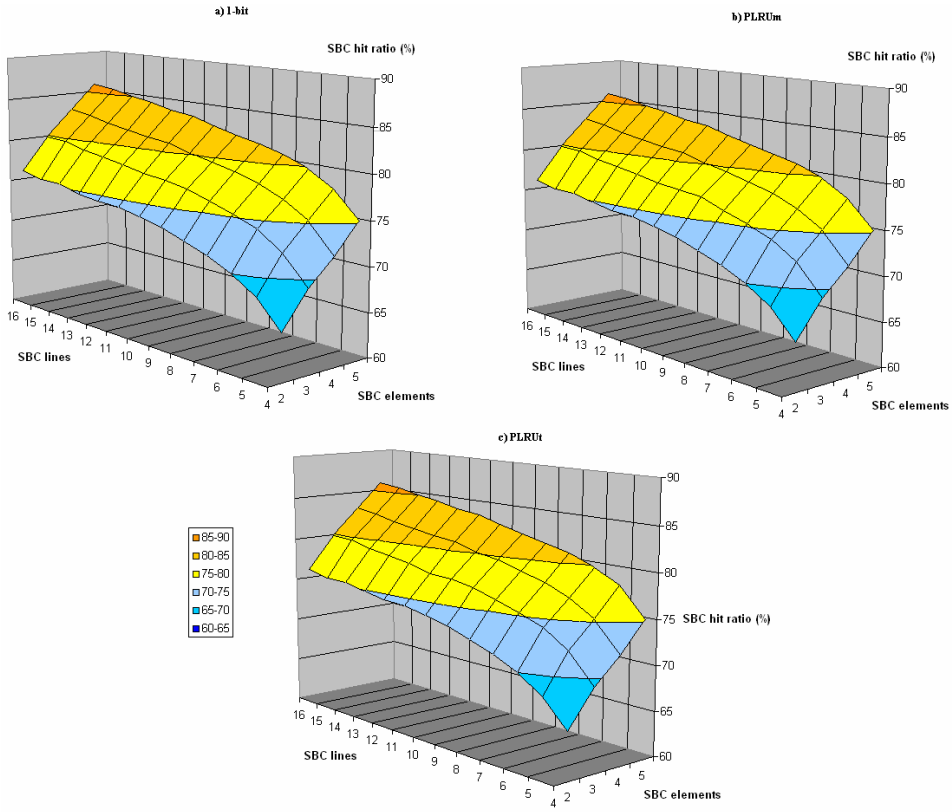

Figure 34: Status Bit Cache hit ratios for a 16-KB and 4-way set associative cache on software

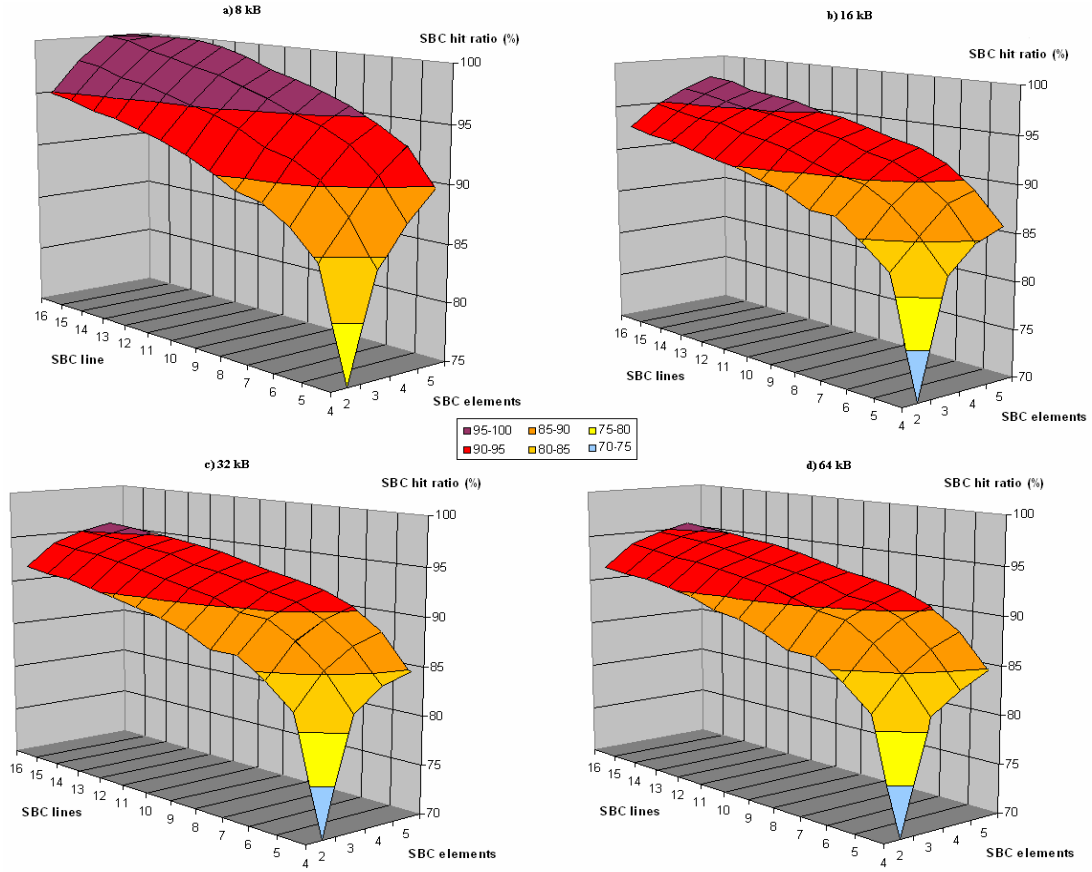

Figure 35: Status Bit Cache hit ratios for a 4-way set associative data cache with different Status Bit Cache configurations and different data cache sizes on the selection of benchmarks

After examination of these figures, it appears that the improvement on the SBC hit ratio is not so important above 8 or 9 lines: the surface flattens. This justifies the choice of 8 for this implementation. Note that the fact that it is a power of 2 does not matter here because the cache is fully associative. Regarding the SBC elements, increasing their amount significantly impacts on the hit ratio only for a tiny number of lines. Consequently, the couple 8 lines and 4 elements per SBC line appears as a good compromise between performance in hit ratio and amount of logic and power involved. The performance simulations corroborate the SBT dimensioning where four appeared as the best compromise in respect with capacity and consumption. Eight lookups in parallel will not consume so much power since it will be done on registers and hence will not require powering on a RAM.

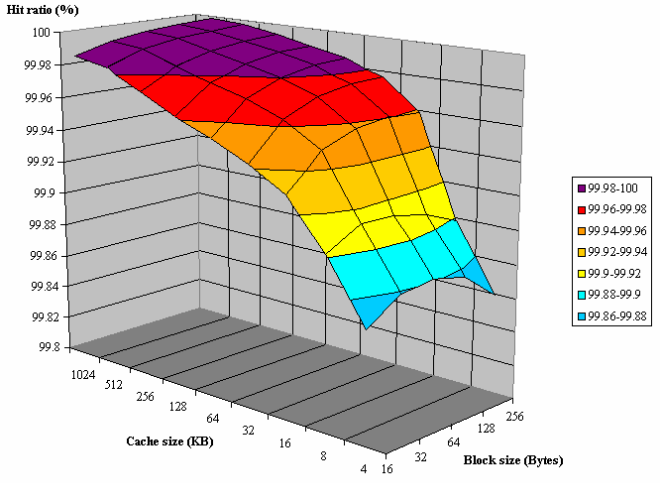

Figure 36: Hit ratio for a 4-way set associative cache on the SPEC92 benchmark suite [GEE93]

The results obtained by these simulations are in compliance with the trends reported in [GEE93] for a data cache submitted to the SPEC92 benchmark suite. The figure above shows this information graphically. The block size of this figure is equivalent to the number of SBC elements and the cache size almost corresponds to the amount of SBC lines. The flattening for high cache size is observed too whereas the description for high block size is more complete than ours in the sense that it goes further and then it exhibits the decrease of the hit ratio after the most efficient value. However, our simulations showed this flattening and hence confirm that the range simulation was well-defined. This comparison confirms that the characteristics of the SBC can be derived from the general data cache ones, in agreement with the first impressions. Indeed, the SBC accesses reflect the data cache requests and thus are faced with almost the same constraints with a higher locality; which justifies that the optimal number of elements is much smaller than the optimal block size of Gee *et al.*'s work.

The average obtained hit ratio is around 90% on benchmarks and 80% on software, which strengthens the hypothesis of power saving. The real efficiency will depend on the chosen replacement policy in the sense that 1-bit would store much more status bits per line and hence its hit ratio would be greater. Indeed, the SBT could almost be generated in logic for 1-bit. However, 1-bit is not the favourite candidate for implementation because of its dependence on an almost true random sequence, which costs a lot of hardware and is quite intricate. The tiny width of a status bit group of the 1-bit algorithm is at the origin of the cache structure difference. The address cuttings are then a bit dissimilar too and are dealt with in the next subsections.

5.2.2. Addressing

Pseudo-LRUs. The cache aims to store the status bits corresponding to an index of the data cache. As it has been seen in the previous section, a good compromise between all the constraints is to design a Status Bit Cache which has eight lines of four status bit groups each. Therefore, the offset of the Status Bit Table address is 2-bit wide. Moreover, the system supports three data cache sizes: 16 KB, 32 KB and 64 KB, which respectively tally with 512, 256 and 128 data cache sets. Consequently, the SBT contains 128, 64 or 32 lines. The Status Bit Cache being fully associative, the status bit tag is 5, 6 or 7-bit wide. The cutting is drawn on Figure 37.a.

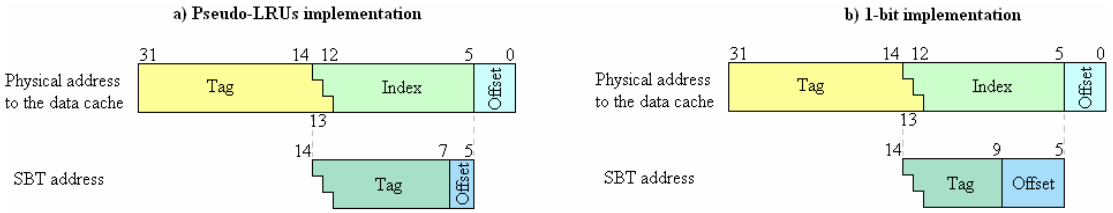

Figure 37: Deriving the address of the Status Bit Table for the two SBC configurations

1-bit. The only difference with the pseudo-LRU implementation is the number of status bit groups per line: sixteen instead of four. Thus, the offset is 4-bit wide and the tag up to 5-bit wide. The address cutting is drawn on Figure 37.b.

5.2.3. Replacement policy inside the Status Bit Cache

Like any cache, the SBC needs its own replacement policy. Since the cache is fully associative, the replacement policy must not only be efficient but also must cost little power and require few gates. According to Chapter 5 and its conclusions, the strategies that meet these constraints are 1-bit, PLRUm and PLRUt. Because of its poor efficiency for high associative caches, 1-bit is discarded. The two remaining candidates are the two pseudo-LRUs. The selected replacement policy will be PLRUm because it is a bit easier to implement and to check that PLRUt, especially when taking into account the last evicted ways for the allocation of the victim way.

5.2.4. Behaviour of the Status Bit Cache

Generalities

The Status Bit Cache aims to afford the status bits as soon as they are required and to immediately store their updated value. By immediately, it is meant on the next rising edge of the clock. The hit information is computed in less than one clock cycle thanks to the implementation in flip-flops. The

Status Bit Cache is then able to handle simultaneously a write and a read request, even if they correspond to the same status bit line. In this case, the value returned by the read request is the one which will be written and not the one which currently lies in the registers. This simultaneity of the read and write requests is the main advantage of this cache over the RAM implementation. Moreover, the power consumption is quite small thanks to the restricted size of the Status Bit Cache.

The single requester for a read is the RAM arbiter which is always granted access. Indeed, the Status Bit Write Buffer gets the evicted line as soon as the new one is written; thereby avoiding a specific access for the SBWB. As a result, the core requests will never be delayed.

Two modules compete for an access to the write part of the cache: the updater and the Status Bit Read Buffers. Since the updater deals with the current access and cannot store the whole SBC line but only a status bit group, it is assigned the highest priority. It must be underlined that it does not constrain further the Status Bit Read Buffer. The presence of an updater request implies that the previous SBC lookup resulted in a hit, thus the Status Bit Read Buffers are not faced with a request (see Figure 39). Besides, the hit-in-SBRB feature ensures that the delaying of the SBRB request does not affect the overall behaviour. For a more detailed explanation of this feature, one can refer to Sections 5.3 and 5.6. The actions one each type of data cache request can now be addressed.

Actions on memory requests

The cache accesses can be split into two groups: the lookups and the writes. The first category comprises the reads and the write lookups whereas the second one is composed of the second phase of the write and of the line fill. The evictions are somehow particular and their handling is addressed before.

Eviction. The line is marked as invalid and is now recognized as “free”. Consequently, there is no need to modify the status bits; it will be done when the way of this line will be used again.

Lookups. The status bits are looked up in the SBC while the cache request is forwarded to the RAMs. The result of the SBC lookups dictates the following action:

- on a hit, the updated status bits are written back at the next rising edge of the clock,
- on a miss, a SBRB fetches the status bits from the SBT, update them and compute the way to discard if needed. Finally the updated line is stored back in the SBC.

It must be noticed that the status bits are updated for any lookup, which may seem queer for the write lookup but which prevents the SBC from allocating the way a second time in the near future. Indeed, if a miss at the same cache set occurred between the two phases of the write, it would assign the same way to the two requests. Thus the MRU line would be evicted, which is clearly what must be avoided.

Writes. The process is identical to the one described above except that the updated value of the status bits is not computed from the hit way value but from the way given by the write requests. This way is obtained by the preliminary lookup that yielded either the hit way value to the Store Buffer or the way to allocate for the Line Fill Buffer. Therefore, the latency time required to fetch a data from the SBT does not impact on the overall performance, provided that this time is smaller than the latency time from L2. It will be demonstrated in the next section that this is equal to two clock cycles, which is negligible in comparison with the L2 latency times roughly equal to 10 clock cycles.

5.3. Read Buffers

The need for buffers has already been stated. This section focuses on the read buffers. A detailed examination of its assumptions and of its architecture is followed by the investigation of the minimal amount of read buffers.

5.3.1. Architecture of a read buffer

Each SBRB is provided with an updater, which allows it to update the status bits if there was a hit or a miss in the data cache. This module is a slightly optimized version of the one located in the hit stage module.

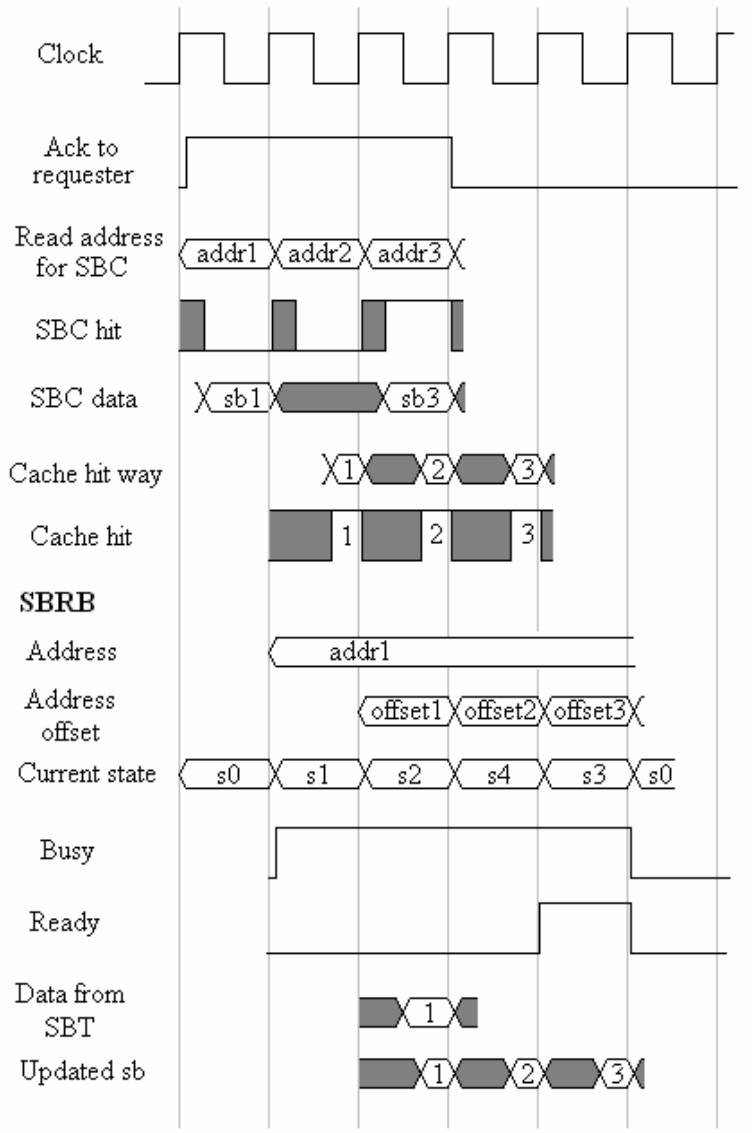

Figure 38: Wave diagram of a SBRB[a]

The two Status Bit Read Buffers' behaviour was implemented as a Finite State Machine. Indeed, it seemed very well adapted to this kind of logical functions. Six states were defined:

- *s0 (000):* the buffer is free and is waiting for a miss in the Status Bit Cache,
- *s1 (100):* the buffer becomes busy and asks for an access to the Status Bit Table. If there is a new miss in the SBC that matches the SBT line address of the buffer, it goes to state *s2*. Otherwise, the next state is *s5*. The higher priority of the read buffers over the SBRB for an access to the SBT prevents them from being blocked. Moreover, they cannot simultaneously request an access to the SBT since it would imply that the two buffers reacted to the same SBC miss. It is avoided by construction. Therefore, it is ensured that the read buffer has been granted access to the SBT when it leaves this state,
- *s2 (001):* the buffer fetches the status bits from the RAM and updates them in respect with the hit information stored at the previous positive edge of the clock. In case of a SBC miss, this state is responsible of sending the allocated way to the Line Fill Buffers. If a SBC miss matches the SBT line stored in the buffer, it goes to state *s4*. Otherwise, it goes to state *s3* ,
- *s3 (011):* the buffer asks for a write to the SBC arbiter and sends the stored updated value to it. If a SBC miss matches the data stored in the buffer, it goes to state *s4*. It stays in this state as long as the SBC has not granted it the access,

[a] Here addresses 1, 2 and 3 correspond to the same SBT line, i.e. their bits [8:2] are the same.

- *s4 (101):* a SBC miss matched the SBT line stored in the read buffer. The stored status bits are updated using the data cache hit information received during the previous clock cycle. As long as a SBC miss matches the SBT line stored in this buffer, this state is not leaved. Otherwise, it goes to state *s3*,
- *s5 (111):* this state is similar to state *s3*: it asks for a write to the SBC arbiter but the sent value is directly updated from the SBT one.

The numbers written in parentheses are the binary representation of the states. These encoding are the results of optimizations in order to reduce the number of gates describing the control signals. An example of its behaviour is given on Figure 38.

5.3.2. Amount of read buffers

Two read buffers are sufficient to ensure theoretically that no Status Bit Cache line is lost, as it can be seen on the wave diagram of Figure 38. This limit can be understood if it is kept in mind that a lookup results in a single request to the read buffers and that the latter need only two clock cycles to fetch a data from the Status Bit Table. The priority of the Status Bit Read Buffers being higher than the Status Bit Write Buffers' one, it ensures that up to two Status Bit Cache misses can be active at a time. Indeed, the SBRB is not able to take the request only while in state *s1*, *s2* or *s4* and thus only these states are further considered for the study of this property. As long as the SBC miss matches the SBC index stored by this SBRB, the buffer remains in state *s4* and there is no need for another buffer (situation on Figure 38). If a SBC miss index differs, the considered buffer goes to its final state and will be able to take the next SBC miss. Hence, the theoretical upper bound of the amount of read buffers is two.

Policy	*Software*	*Cache size (KB)*	*Number of buffers*	*Hit ratio (%)*	*Execution time (ns)*
PLRUm	Explorer	16	1	97.468	310789775
			2	97.471	310726675
		32	1	97.691	308503725
			2	97.693	308502375
	Maze	16	1	80.755	2791796225
			2	80.752	2791176425
		32	1	97.177	1658053275
			2	97.234	1654942675
PLRUt	Explorer	16	1	97.468	310789775
			2	97.471	310726675
		32	1	97.691	308503725
			2	97.693	308502375
	Maze	16	1	80.112	2835813825
			2	80.093	2838222375
		32	1	96.999	1670535925
			2	96.997	1670541025

Table 16: Impact of the number of read buffers on the overall performance of maze and explorer

Yet, the situations where the two status bit buffers are active at the same time are not common. Thus, simulations were performed to investigate the loss of performance if only one buffer is implemented. This is evaluated in terms of hit ratio and of run time, which is one of the most important factors for the user. This figure can be biased by the long booting sequence which does not depend on the implementation. However, the boot is equivalent for the two proposed systems and the execution time can still be considered as an interesting measure. The results of the different simulations on the Verilog code are given in Table 16. For 1-bit, such a problem is not faced since the Status Bit Cache holds eight lines of sixteen status bit groups and thus stands for 128 sets. As a result, the SBT is not needed for the 16-KB data cache. For 32 KB and 64 KB, the principle of locality should allow us to use only one Status Bit Read Buffer. Consequently, Table 16 only presents the two pseudo-LRU replacement strategies.

The introduction of a second buffer, which enables the application of the strict replacement strategy, does not yield a significant performance improvement. Comparing this small enhancement with the number of gates of a SBRB, the best solution for embedded systems is thus to design only one such buffer. Two would be all the more unjustified because the single read buffer implementation even exhibits better hit ratio on some patterns. Instantiating a second buffer can be useful for desktop

systems in order to enforce a strict replacement algorithm but it is not the case for embedded systems, where area and consumption are the major points of concern. Nevertheless, including `define TWO_SBRBS in the Verilog code instantiates the second read buffer.

5.4. Write buffer

For the write buffer, it could be thought that the constraints are the same but the modified bit ensures writing back only the lines altered. Combining it to the strong locality of the accesses which consequently do not change the status bits for most requests, it relaxes the constraints on the Status Bit Write Buffer. By the way, it justifies the higher priority of the read buffers for accesses to the Status Bit Table. It should be noticed that this prioritization forces the write buffer to wait longer for a grant but the principle of locality guarantees that read buffer will not access to the SBT RAM continuously, thereby letting the write buffer performing its write back in the SBT RAM. Furthermore, the original ARM11 design has only one victim buffer for its cache and thereby confirms the need of a single write buffer. Consequently, a single buffer will be then implemented in this design.

5.5. Updater

The updater is equivalent to the one presented in the other proposals. The output of the SBC is registered to ensure that the module receives the data cache hit information and the status bits of a given set during the same clock cycle. The origin of this delaying is striking on Figure 39. Apart from it, the module tallies with the code written in Section 1.3.

The behaviour of the SBC system has been explained and it has been demonstrated that the *considered* status bits always describe the current state of the data cache: either they are located in the Status Bit Cache and immediately updated on a memory access, or they are fetched from the Status Bit Table and the induced value of the way to discard is transmitted to the Line Fill Buffers. The word *considered* is of utmost importance in the assertion since the status bits may temporarily not reflect the status of a data cache set – they have not been fetched from the SBT or have not been updated yet… – but they will be up-to-date once these status bits are required. The assertion must also be understood in respect with the loss of accuracy introduced by the restriction of the amount of the status bit buffers. Nevertheless, it has been shown that this loss is negligible and that the assertion is valid. In addition to this basic behaviour, the design was optimized to decrease the power consumption and to increase the performance of the design. These features are handled in the following section.

5.6. Optimizations

5.6.1. Updater

The Status Bit Cache is designed to take advantage of the principle of locality. Therefore, successive accesses on the same set often occur. This would lead to unnecessary writes in the Status Bit Cache and could prevent the Status Bit Read Buffers from writing a line in SBC since the updater has priority for a write access to the SBC. A trick has been introduced to solve this issue: the updater checks that the updated value differs from the incoming one before sending it to the cache. The required hardware is only a four-bit comparator, so the cost is quite very negligible in comparison with the potential gain.

5.6.2. Status Bit Cache

This optimization is based on the modified bit commonly found in modern caches. This bit states whether the Status Bit Cache line has been altered since it was fetched from the Status Bit Table. As the status bit line could have been modified by one of the two Status Bit Read Buffers, the latter forward this piece of information on a SBC line fill. This avoids useless writings to the Status Bit Table and also helps maintaining the number of Status Bit Write Buffers to one.

5.6.3. Status Bit Read Buffers

The SBRBs are probably the most optimized part of the design. First, the design supports hit in the SBRBs: if a request misses in the SBC and hits in a buffer, this one waits for the data cache hit information and updates the line in consequence. Furthermore, the same tip as in the updater of hit stage module is used here in order not to mark lines as modified, when the update has not altered the

line. Finally, SBRBs are able to fetch a data from SBWB if the requested line lies in this buffer. This consists in the means used to avoid some hazards too (see next subsection). In this case, the SBWB is prevented from writing its line in the cache and considers in the next cycle that it is empty.

5.7. Hazards

The implementation was source of different hazards. They all come from the fact that the status bits travel through the system, thereby leading to unwanted situations. In this section, the terms read and write refer to reads and writes to the Status Bit Cache.

The first hazard corresponds to a situation when simultaneous read and write occur at the same address. The read value must be equal to the value written at the rising edge of the clock. Indeed, if the returned value was the value lying at this moment in the Status Bit Cache, the updated value currently presented to SBC would be lost or even could lead to unpredictable… Thus, on a write and a read at the same address, the value returned by the cache is the data to write in conformity with the specifications asserted in Section 5.2.4. This assignment bypasses the registers of the Status Bit Cache and hence leads to numerous bugs and infinite loops in the delta computation. Fortunately, all these problems have been solved by carefully caring the signals in the finite state machine of the SBRBs.

The priority of SBRB over SBWB for an access to the SBT induces the possibility of a hazard. Let us assume that the evicted line still lies in the SBWB and that a request to the same line is presented to the cache arbiter. The SBRB deals with the request and fetches data from the SBT. Since the updated data have not been written back yet, it receives the old value of the status bits which will then be written in the cache. However, it should be noticed that this hazard occurs only with two SBRBs. Indeed, the SBWB must be prevented from accessing to SBT. Since the only other requester is the SBRB module, it implies that one of the buffers requests for accessing to SBT. As the buffer which will deal with the last request was at the previous clock cycle either in state *s0* or one of the two final states *s3* or *s5*, it cannot have sent this request. Thus, it was transmitted from the second buffer. Nevertheless, this hazard is avoided by the hit in SBWB feature that has been presented in the section about optimizations. When a SBC request misses, SBWB compares the tag of the line it stores and the tag of the requested line. In case of equality, the line is forwarded to the Status Bit Read Buffer which receive these data and cancel its request to SBT arbiter.

It is seen on Figure 39 that the status bits are always updated on the next clock cycle. Thus, the only possible hazard has been already solved: it is the presentation of the same read and write addresses. Otherwise, the read data is the valid one. There can remain some peripheral hazards but none has been detected.

5.8. Validation of the design

In order to test the design and to verify that it operates as it should, different tests were performed. First, the design was written in a modular manner, which ensures checking each level after its sub modules have been validated. To that end, testbenches and some small C programs were developed to help performing automatically these verifications. Besides, some signals controlling that the behaviour is not crazy were also introduced. For instance, a signal asserts that there are not two lines of the cache holding the same data. These signals efficiently contributed to the detection of bugs.

Some basic assembly programs were written and integrated into the validation environment in order to test more accurately the newly created modules. The instructions were then followed through the design and their interpretation studied. Of course, it was not as simple as it seems because of optimizations inside the data cache side. The major point of concern was the out-of-order execution of instructions which reorders the instructions. This is a problem for validation and for the replacement strategies. Indeed, it is equivalent to modifying the order of the memory accesses and can yield a data cache status different from the expected one. The importance of this issue has already partially been addressed in the previous chapter and may explain the slight difference observed between the results of Chapter 5 and the ones given in Section 6.

Thus, the issue was approached in another manner, consisting in launching some basic tests, such as Dhrystone, which calls for various data. The instructions and their impact on the status bits through all the design were then investigated and compared with the results manually obtained. This method is

obviously quite far from optimal and very much-time consuming. Nevertheless, it helped correcting some errors and better understanding all the characteristics of the system.

In order to enable an automatic validation of the design, the Verilog code was modified to store the memory requests reaching the data cache. It distinguishes itself from the TARMAC logs by the point of view: whereas TARMAC keeps track of the memory requests of the processor, this logger stores the memory requests which enter the data cache. Memory requests are thus seen after reordering in the case of out-of-order execution and after potential merges. These log files fed the cache simulator of Chapter 5, whose results were compared with the Verilog figures. It raises the issue of the diversity of points of view on this thesis. Since a single designer, the author of this work, implemented the C and the Verilog versions, some interpretation errors can possibly be found in the two versions. Nevertheless, it must be reminded that the cache simulator's results matched the values reported in the literature. Hence the bug probability can be considered as tiny though non null.

Finally, the system was submitted to real software running on the processor. No error on the output of the system or on any of the inner controlling signals has been observed, thereby validating the module. For more information on each step of the validating process, one can refer to Appendix D.

5.9. Sum up

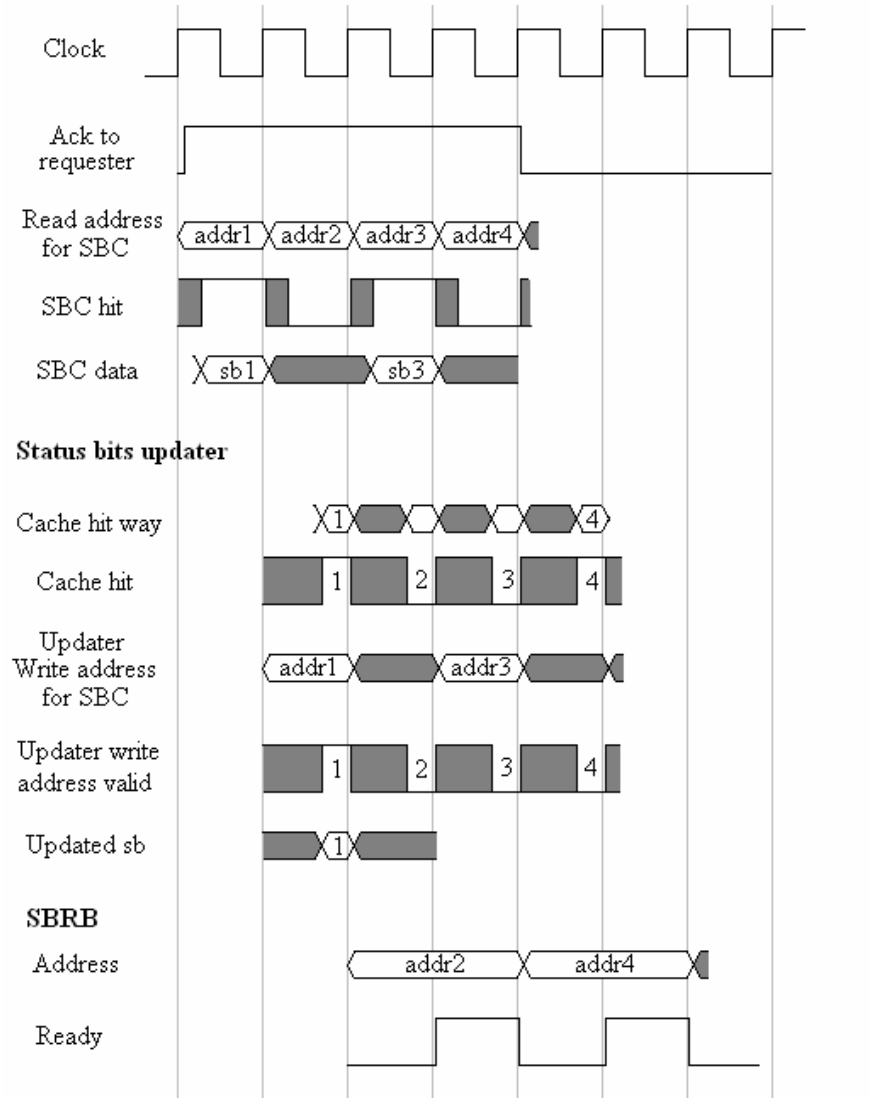

Figure 39: Wave diagram of a sequence for the overall SBC side

The system finally designed corresponds to the wave diagram drawn in Figure 39. This solution saves some power though it adds a small cache along with its power consumption. This addition is overbalanced by the reduction the number of accesses to the RAM, which is one of the most power

consuming elements of a design. The high SBC hit ratio observed with only 8 lines of sixteen bits (i.e. 128 bits) which avoids numerous accesses to the 2-KB RAM (up to 128 lines of sixteen bits) ensures an important gain in energy and in performance. It should be stated too that this implementation avoids numerous costly accesses to the L2 as shown by Eq.2. This power consumption of the module must then be compared to the value of the power consumption of an access to the L2 cache. The power consumptions are denoted *W* and the hit ratios *h*. It is easily obtained that:

$$\Delta W = W_{new} - W_{current} = (1 - h_{SBC})\, W_{SBT} + W_{SBC} + \Delta h_{L1}.W_{L2} \quad , \quad \text{with} \quad \Delta h_{L1}=h_{L1,new} - h_{L1,current}$$

ΔW stands for the power consumption increase between the two implementations. From the simulations performed in this chapter and in the previous one, the following typical values are deduced h_{SBC} = 0.9and Δh_{L1} = -0.005. Thus:

$$\Delta W / W_{L2} = 0.1\, W_{SBT} / W_{L2} + W_{SBC} / W_{L2} - 0.005$$

Let us evaluate the activity of a register of the Status Bit Cache. The probability that one of the lines is modified on an access is $(1-h_{SBC})$. The eight lines being equivalent to one another, the probability that a given line is altered is then $0.125(1-h_{SBC})$. Finally, the probability that a given bit of the line is modified knowing that the line is modified is 0.5 since it is assumed that 1s and 0s are evenly distributed. As a result, the probability that a given register is altered is $(1- h_{SBC})/16=6.25\text{x}10^{-3}$. Adding it to the tiny power consumption of a register in comparison to the L2 access, the SBC power consumption can thus be neglected in spite of its 128 registers. Finally, the power difference is:

$$\Delta W / W_{L2} = 0.1(2/250)-0.005 = -0.0042$$

One thus sees that the impact on the power consumption is null, even a bit beneficial. In spite of the quite coarse evaluation, it demonstrates that the proposed implementation is very well adapted to embedded systems' constraints.

One means to estimate the area and is to count the number of registers created in overall. For the pseudo-LRUs, it is roughly equal to 330 flip-flops including the SBRB[a], the SBC, the updater and the SBWB. The SBT RAM must be added to this amount. The 1-bit implementation requires little less hardware with only 320 flip-flops in average. In these two cases, the major part of the required flip-flops is due to the Status Bit Cache and its controlling logic. This is consistent with the negligible amount of logic required for the update of the bits itself. The slight difference between 1-bit and the pseudo-LRUs originates from the width of the tags of the SBC lines which impacts on the amount of registers required by the SBRB and the SBWB, as well as on some intermediate register. The figure given for 1-bit is obtained and depends significantly on the amount of flip-flops.

Finally, the implementation of a RAM endowed with its dedicated cache appears as the most promising solution because of the power it will save and the tiny amount of hardware required. Moreover, it prevents us from continuously accessing the Dirty RAM and avoids significant modifications in the current module, which ensures that the verification can be performed in an easier and more modular manner.

6. Results of the simulations

In the continuity of Chapter 5, the selection of benchmarks and the software applications were simulated on the enhanced version of the processor. The results and their interpretation are presented here.

6.1. Obtaining the hit ratio

Computing the hit ratio, the means used to measure the efficiency of the policies, required getting the number of requests and counting the cache hits. At first sight, this task seems trivial but reality is far from that. Indeed, numerous problems have to be solved to get this figure. Assuming that we got the means to count the cache request, getting hit information is obvious: when there is a cache request, the counter is incremented in concordance with the hit information, which is available on the next clock cycle. Therefore, the problem can be summed up to enumerating the cache requests. This issue is handled in the sections below.

[a] The implementation of a single Status Bit Read Buffer is assumed.

6.1.1. Read requests

The situation for the read requests is quite simple. Indeed, the Slots Unit asks for a cache lookup that induces a line fill if it misses. Consequently, monitoring the cache lookup requests from the Slots Unit would be a simple means to count the read requests from the core. Unfortunately, the situation is not as simple as it seems, because the data cache side is optimized and lookups are merged when it is possible. Moreover, the order of instructions issuing is not preserved in the data cache side for data which are independent from one another. It leads to a situation a bit different from the one expected by the enforcement of the replacement algorithm on the memory requests of the processor. These optimizations common to state-of-the-art processors cannot be ignored and must be studied. Relying once more on the TARMAC disassembly could help us solving this issue. Unfortunately, correlating the memory requests of the TARMAC file with the hit information of the data cache is a very tough task since there is no fixed time interval or fixed order between the different accesses and all these parameters impact on the merge of the requests directly. Consequently the read lookups are considered as an appropriate indicator of the read requests.

6.1.2. Write requests

The issue raised by the write requests is a bit more complicate. The writes are constituted by two phases: the lookup and the write or line fill. If this split up was permanent, monitoring the lookups issued by the Store Buffer would be sufficient to deduce the number of the write requests. However, the writes as well as their lookups can be merged to optimize the data cache accesses. This is quite similar to the read requests but the problem is even harder when the sequential accesses are taken into account. The lookup is performed by the first access and provided that there is no intervention of any other requester on this data cache line, the other writes are performed without any lookup, either by the Store Buffer in case of a hit or by the Line Fill Buffer in case of a miss where the writes are merged. This pathological situation is not faced by the read requests since the read is still visible. Monitoring the write accesses thus appears as the best reply but the issue would have been even more intricate since the writes reflect only the write requests whose lookup resulted in a hit. Therefore, the Line Fill Buffer accesses should have been counted too, thereby leading to a tortuous design where the Line Fill misses from the Store Buffer and from the Slots Unit must be distinguished whereas these misses can have been merged if they correspond to the same line. Adding the fact that the Store Buffers can hit into the Line Fill module, one understands easily that this solution would not have been efficient.

For these reasons and in spite of its drawback about the sequential accesses, the choice was made to count the write lookups, which is the means that reflects the best the situation seen from the core. It is all the more justified since sequential accesses seldom occur.

6.1.3. Other requesters

Among the modules requesting a data cache access, only the following ones remain: the Coprocessor 15 Controller, the Cache Coherency Controller, the Eviction Buffer and the two Line Fill Buffers. All these accesses can be considered as cache maintenance in the sense they are either consequences of data cache accesses which have already been taken into account or pure cache maintenance:

- the requests from CP15 controller are pure cache maintenance and should not be taken into account (invalidating, flushing…),
- the Cache Coherency Controller's requests are pure cache maintenance due to multi processing,
- the eviction and the line fill requests are due to misses. Yet these misses are the results of the Slots Unit lookup or of the Store Buffer lookups and have thus already been counted in the number of requests.

As a result, these cache accesses are ignored when counting the requests.

6.2. Simulations

6.2.1. Simulated benchmarks and software

Since the Verilog simulations are very time consuming and since they must be performed for the four replacement policies and for two cache sizes (64 KB could be simulated too but it is very hard to stress, specifically for the benchmarks as it has been seen in Chapter 5), only a subset of the benchmark selection was considered. The benchmarks which yielded the most significant differences between the replacement strategies for the low data cache sizes of the ARM11 MPCore processor form it, i.e.

`automotive_matrix`, `mpeg4_decode`, `office_rotate` and `networking_tcp`. Indeed, the difference in gross hit ratio on all the benchmarks is small for the commercial sizes of the data cache, even for 16 KB. Whereas the main reported differences in Chapter 5 concerned 8KB, differences were however still observable in gross hit ratio figures for some patterns. Among them, `automotive_matrix`, `office_rotate` and `mpeg4_decode` were the most significant contributors. It should be noticed that they belong to different application fields and afford a broad range of embedded workload. As a result, it should not restrict the extent of this work.

In the same optics of optimizing the simulations, the software applications `maze` and `explorer` are investigated too. For `explorer`, the elementary functions as well as the data sets were altered to stress the cache differently whereas only the input trees were modified for `maze`.

The striking issue of restricting the study to the two pieces of software and some benchmarks is the exposure to the particularities of the latter. As it has already been stated in the previous chapter, `maze` exhibits very good results for the Global Round Robin policy, particularly for low-size caches, in contradiction with the other benchmarks and software. It has however been kept as a reference test since it constrains well the cache and reminds us the exceptions of some patterns.

6.2.2. Results and general interpretations

Benchmarks

Among the simulated benchmarks, `automotive_matrix` exhibited a specific behaviour. For this reason, its detailed examination follows the general presentation.

	Cache size		
Policy	16 KB	32 KB	64 KB
1-bit	98.21	98.80	99.82
PLRUm	98.70	99.81	99.82
PLRUt	98.69	99.81	99.82
Global Round Robin	98.39	99.81	99.82

Table 17: Hit ratio of the Verilog version with benchmark office_rotate

Conformity of the simulations. The results of a typical benchmark are given in Table 17. As the figures are in compliance with the cache simulator ones (see Figure 22 p.35), the Verilog simulation seems to match the previous study performed in Chapter 5. However, a slight difference is observed between the two simulations but it is very tiny (less than 0.05% in hit ratio). This deviance can be all the more neglected since the relative efficiencies of the replacement strategies remained the same and the exhibited trends are identical. The small disagreement in absolute hit ratio originates from the implementation of the cache simulator. Indeed, its inputs are memory requests from the core but they are then reordered and merged if there are successive reads or writes to the same memory lines. Because of these merges, the requests are a bit miscounted and the impact of a hit can be overestimated or underestimated. It should be however kept in mind that the cache simulator took the multiple load-and-store instructions' particularity into account; which explains that the observed difference is tiny in overall.

Another interesting feature is the dissonance of counted instructions between the cache simulator and the Verilog implementation. This issue has already been addressed in Section 6.1 and is evoked briefly here. Whereas the cache simulator considers all the memory requests, the latter only deals with the accesses once they have entered the data cache. Furthermore, if successive memory requests correspond to the same cache line, they are considered as distinct requests by the cache simulator whereas they may be merged in one in the real version. Another factor interferes but at a slightest degree: the cache simulator was not aware of the turning on of the data cache and then dealt with all the requests, even the booting ones and the CP15 starting cleaning of the cache. Obviously, this is significant only for the short running programs and could be ignored for the long-running ones. Since the benchmarks were chosen in part for their high amount of instructions, this factor is not the major contributor to the observed discrepancy.

According to the concise presentation of the general behaviour, the Verilog simulation seems to corroborate the cache simulator's results perfectly. `Automotive_matrix` demonstrates the opposite below.

The particular case of `automotive_matrix`. In opposition to the other benchmarks, `automotive_matrix` did not exhibit any significant difference between the replacement strategies, which appears discordant with the cache simulator. It suggests the presence of a bug in the design but the output of the cache simulator, running the stored memory requests which have reached the data cache, corroborates the obtained result and confirms the validity of the simulator. This outstanding phenomenon originates from the nature of the benchmark and the causes are addressed below.

Firstly, it should be mentioned that the performance enhancement was observable on the cache simulator only for 8 and 16-KB. Thus, the lack of differentiation for the two highest commercial cache sizes is in accordance with the previous simulation. Secondly, `automotive_matrix` has the particularity to frequently load and store multiple registers and to access the same data routinely. It exposes the benchmark to numerous merges and hits in the Store Buffer and in the Line Fill Buffers. These features thus modify the overall behaviour of the cache and strongly smoothes the differences among the replacement strategies. Therefore, the small difference reported in Chapter 5 shrinks and explains the performance leveling.

Software

Because of the particularities of `maze` already evoked in Chapter 5, the results are presented separately for the two applications in Table 18.

		Explorer		*Maze*	
Cache size (KB)	*Policy*	*Miss ratio (%)*	*Miss ratio compared to PLRUm*	*Miss ratio (%)*	*Miss ratio compared to PLRUm*
16	1-bit	3.55	1.516	22.45	1.869
	PLRUm	2.34	1.000	12.01	1.000
	PLRUt	2.37	1.010	12.30	1.024
	GRR	2.47	1.055	11.80	0.983
32	1-bit	2.32	1.096	2.90	1.802
	PLRUm	2.12	1.000	1.61	1.000
	PLRUt	2.13	1.005	1.77	1.098
	GRR	2.32	1.094	1.48	0.917
64	1-bit	2.08	1.029	0.51	2.816
	PLRUm	2.03	1.000	0.18	1.000
	PLRUt	2.03	1.002	0.21	1.162
	GRR	2.06	1.016	0.17	0.952

Table 18: Miss ratio of the candidates on the different ARM11 cache configurations with maze and explorer

Comparing these results with the ones obtained by the cache simulator, one sees that they are almost equivalent. The numbers of requests differ a bit too but as it has been explained for the benchmarks, it can be neglected. Two differences strike the observer: the poor performance of the 1-bit strategy and the improved performance of the Global Round Robin strategy. Their causes along with the overall interpretation of the policies are addressed in the next subsections.

6.2.3. Pseudo-LRUs

The performance hierarchy of the two pseudo-LRU algorithms remains unchanged. The performance of PLRUm over PLRUt is particularly noticeable for program `maze`, as in the previous chapter. Besides, the trends are equivalent too: the smaller the cache, the more significant the efficiency difference. It must be mentioned that the difference is a bit smaller in gross hit values. All the phenomena evoked in the previous sections – reordering and merges of the instructions, execution of the lookups while the data has not been fetched from the upper level memory – apply here too and

explain the overestimation of the previous chapter. Indeed, the status bits are updated during the lookup for the cache simulator whereas in the Verilog one, they are modified on the lookup on a hit and during the allocation on a miss. These small differences along with cache optimizations such as hits in the Line Fill Buffers and in the Store Buffer, which are not modelled in the simple cache model used in the previous chapter, induced this slight deviance. Nevertheless, the increased performance of PLRUm over PLRUt is visible on all the simulations and is sufficient to justify the selection of this strategy despite it requires one additional bit per set in comparison with PLRUt. As a result, PLRUm is the best alternative for implementation and is the solution preached in this work.

However, it should be kept in mind that the cache simulator aimed to give clues about the efficiencies of the replacement strategies in order to lead us to a final implementation. Moreover, verification and timing constraints encouraged the author to write a slight simpler cache but with the assurance that it operates in respect with the specifications. The simulations thus confirm these choices: even though some particularities were modelled in a too optimistic way or not taken into account – among them the reordering of the instructions – the results are globally in accordance with one another.

6.2.4. Global Round Robin

Global Round Robin performs much better than it had in the cache simulator. Since all the results are a bit shifted due to the distinct means used to count the requests and the hits, the difference between PLRUm and GRR is considered as reference. Indeed, the pseudo-LRUs operated almost exactly as in Chapter 5. The observed difference is in average 0.73% in the hit ratio over the commercial cache sizes for the cache simulator and 0.12% for the Verilog version. Because of its nature, GRR is the algorithm which the most took advantage of the instruction merge. Indeed, two successive misses to the same line may have led it to be in lag to the most efficient Round Robin algorithm, which induces numerous non-optimal replacements. This assertion may question the reader: if the accesses are merged then they correspond to the same data cache line and thus the second access should not have been considered as a miss in the cache simulator. This is true for most requests but it should be reminded that the requests generated by LDMIA instructions for instance are globally handled by the cache model and the number of misses can thus be incremented twice. In the Verilog one, this does not occur and GRR remains in phase with a quite efficient replacement, thereby reducing its distance from optimality. This is particularly true for the multiple load-and-store instructions. Furthermore, it is less sensitive to the reordering of the instruction since the sequence 1-2 or 2-1 makes it point to the same way. Thus, the differences between the cache simulator and the Verilog version are all profitable to GRR; thereby increasing its performance.

It must be noticed that expressing these results in function of the miss ratio of the PLRUm policy still yields important differences for `explorer`, around 6%. These differences are in the same order as the ones obtained for the benchmarks. On `maze`, the Global Round Robin policy performs really well as it has been already shown in the previous chapter but it is here at a higher degree. The reasons set out above apply here too. However, it has been shown during the general study of benchmarks and in this chapter that patterns such as `maze` are rare, as it is confirmed by the simulation results across the other benchmarks. But it helps keeping in mind that there is no absolute better algorithm and relativizing the results.

6.2.5. 1-bit

After examination of Table 17 and Table 18, 1-bit obviously operates poorly. Comparing these results with the cache simulator ones, it seems that the algorithm has changed. Yet, the very difference between the Verilog implementation and the C one lies in the random generation. In C, the `rand()` routine yields an almost truly random sequence whereas it is produced by four LFSRs in Verilog. It could be argued that 4 is a quite low figure and that longer sequences would have been more random. Such a study is performed in the table below. All the taps were optimal and obtained from [XIL96].

Amount of LFSRs	4	8	16	32
Hit ratio on explorer (%)	96.181	96.144	96.156	96.112

Table 19: Impact of the LFSR width for 1-bit with a 4-way set associative and 16KB data cache

The impact of the LFSR width is almost insignificant and even decreases the hit ratio. The origin of the inefficiency must then be searched elsewhere and probably comes from the reorganization of the instructions and of the merge of the requests. Since the 1-bit policy protects the MRU half from

eviction, the merges and the reorganization of the data significantly influence its behaviour. While the other algorithms protect the MRU way and consider the four ways of the cache, everything happens for 1-bit as if the associativity is equal to two. It is then easily understood that it is more sensitive to the instructions merge. Let us suppose that the processor memory access sequence is *a*-*b*-*a* where *a* and *b* are mapped to the same set and initially do not lie in the cache. If the requested data are independent and if the requests are sufficiently near from one another in time, it could be seen in the data cache as *a*-*b*. On the next miss, the policy will evict a way of the first half, so the MRU data of the processor, i.e. *a*, can be discarded… This high sensitivity of 1-bit explains the loss of performance.

6.2.6. Need of a new implementation

The final issue is to compare PLRUm with the current ARM11 implementation and to put these results in perspective with the cache simulator conclusions.

The specificity of `maze` brings into question the need for such an enhancement. Although the working sets were altered, the algorithms principles were not modified and the manner the software deals with the data is kept constant. Consequently, we are exposed to the particularities of some programs and their relative frequency should be evaluated. To that end, benchmarks are very useful since they enable that the simulation range is wide enough and stands for the real embedded systems, at least for its main characteristics. Looking at the results of the cache simulator, it is deduced that the characteristics of the program `explorer` are nearer to the typical values than `maze`. This assertion is based on the comparison of the behaviours of benchmarks on the one hand and on `maze` and `explorer` on the other hand. For this reason, more credit is given to the results of `explorer` than to `maze`'s ones, although this last should not be forgotten.

The difference between the PLRUm strategy and the GRR algorithm is 0.25% in gross miss ratio on average across the benchmarks and `explorer`. As Eq. (2) demonstrated in Chapter 2, this difference can be considered as sufficient to justify the implementation of the PLRUm policy. Moreover, the same reasons as the ones presented in Chapter 5 advocate for the introduction of such a policy in the current data cache architecture. The development of on-chip L2 caches reduces the size of L1 caches and thus reinforces the call for an efficient replacement policy. Besides, real systems are faced with an operative system and numerous applications running in the meantime; which further reduces the efficient cache size. This justifies the focus of the conclusions on the low-size end of the experiments. In this part, the performance enhancement is noticeable yet not outstanding. For all these reasons, PLRUm is recommended for implementation in embedded systems. For the chosen implementation of Section 5, the implementation cost in terms of power consumption and hardware is negligible. As a result, it pleads for the integration of the PLRUm replacement policy in the ARM processors.

7. Conclusion

After having examined the different replacement strategies in details along with their impact on the data cache architecture, PLRUm appears as the most well-adapted replacement algorithm to embedded systems. In spite of the acceptable performance of the Global Round Robin policy, PLRUm outperforms it on almost all the memory patterns and can be easily integrated in the current implementation with low power and area costs. Though the target processor of this work was ARM11 MPCore, the conclusions and the results should be extended to the other processors since the memory workload should not be altered so significantly.

Chapter 7

Conclusion

Корни всякого открытия лежат далеко в глубине, и, как волны, бьющиеся с разбега о берег, много раз плещется человеческая мысль около подготовляемого открытия, пока придет девятый вал.

В. Вернадский[a]

1. Results

After having addressed the different replacement policies characteristics along with their expected performance, the algorithms were simulated on a cache model designed for this purpose. The relative fast running time of this simulator allowed us to deal with numerous replacement proposals across a broad range of embedded applications and benchmark suite. Belady's anomaly for the PLRUm replacement strategy has been thus disclosed and reported for the first time in a scientific document. Three candidates for a final design emerged from performance and coarse power and area considerations: 1-bit, PLRUm and PLRUt. The proposed integration of these algorithms meets all the constraints of the embedded systems: negligible increase in power consumption and in area as well as performance enhancement. Finally, the simulations performed on the enhanced ARM11 MPCore processor confirmed the improved hit ratio, even if the final results are a bit smaller than expected. Nevertheless, it justifies the integration of such a strategy in the next generation of processors. Whereas it is not crucial for the highest data cache sizes, it significantly improves the efficiency of the replacement for low data cache sizes.

2. Future work

This work was devoted to the first level data caches. For the second level data caches, modifying the replacement policy can even improve the efficiency of the system in a more significant way, because the average access time is even greater (see Eq. 2). The results of this thesis can partially be applied to L2 caches but these ones have their own characteristics. For instance, the associativity and the size of the blocks are parameters among others which differ and imply different constraints for the two solutions. Some studies have already been performed [WON00] but they were dedicated to desktop systems and then not optimized for embedded processors' points of concern such as power, area… Carrying out such a study would then be a potential source of improvement.

The influence of the multiprocessing environment has not been studied in this thesis too. It introduces new data cache actions (invalidation by the coherency protocol) whose impact has not been the object of a deep study yet. In this work, the actions were taken into account in the implementation part but their interaction with the replacement policy has not been investigated. At first glance, it should not influence significantly the replacement algorithms but improvements such as taking into account the MESI state of the line for eviction should be examined in further details. Indeed, at the end of the stack, the probability of hit is almost equivalent and adding the shared state information to this eviction could be useful. In fact, for the two last elements of the stack or of the pseudo-stack, data that are modified or exclusive may be preferentially kept in the cache to shared data that could soon be modified by another

[a] *The roots of each discovery go deep and like a wave lapping again and again the coast, human ideas come back and forth many times preparing the discovery until the huge wave comes.*

V. Vernadsky

core. Of course, this is a potential error if the data will be modified by this processor. It is only some of the possible ideas, which could improve the data cache efficiency in multiprocessing systems.

References

[AMD02] AMD Athlon™ Processor x86 Code Optimization Guide, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf, February 2002

[ARM01] *ARM Architecture Reference Manual*, Addison-Wesley, 2nd edition, 2001

[ARM11] ARM11 Family, http://www.arm.com/products/CPUs/families/ARM11Family.html

[BEL66] L.A. BELADY, A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM Systems Journal*, Vol. 5, N.2, 1966

[BEL69] L.A. BELADY, R.A. NELSON AND G.S. SHEDLER, An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine, *Communications of the ACM*, Vol.12, N.6, 1969

[BRE04] M. BREHOB, S. WAGNER, E. TORNG AND R. ENBODY, Optimal Replacement is NP-hard for Nonstandard Caches, *IEEE Transactions on computers*, Vol. 53, N. 1, pp. 73-76, 2004

[COR02] D. CORMIE, The ARM11 Microarchitecture, ARM Ltd, http://www.arm.com/pdfs/ARM11%20Microarchitecture%20White%20Paper.pdf, April 2002

[DEV90] Y. DEVILLE, A Low-Cost Usage-Based Replacement Algorithm for Cache Memories, *SIGARCH Comput. Archit. News*, Vol.18, N.4, ACM Press, pp. 52-58, 1990

[DOU06] E. DOUBROVA, Fault-Tolerant Design: an Introduction, Kluwer Academic Publishers, Draft, 2006

[EEM06] EEMBC: Embedded Microprocessor Benchmark Consortium, http://www.eembc.org

[FAT94] O. FATEMI, F. IDRIS, AND S. PANCHANATHAN, FPGA Implementation of the LRU Algorithm for Video Compression, *IEEE Transactions on Consumer Electronics*, Vol. 40, N. 3, pp. 337-344, 1994

[GEE93] J.D. GEE, M.D. HILL, D.N PNEVMATIKATOS AND A.J. SMITH, Cache performance of the SPEC92 Benchmark Suite, *IEEE Micro*, vol. 13, no. 4, pp. 17-27, Jul/Aug, 1993

[GEN04] P. GENUA, A Cache Primer, Freescale Semiconductor, Inc., AN2663, 2004

[GHA06] H. GHASEMZADEH, S. MAZROUEE AND M.R. KAKOEE, Modified Pseudo LRU Replacement Algorithm, *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, IEEE Computer Society, pp. 368-376, 2006

[HEN03] J.L. HENNESSY AND D.A. PATTERSON, Computer Architecture: A Quantitative Approach, 3rd edition, Morgan Kaufmann Publishers, 2003

[INO99] K. INOUE, T. ISHIHARA, K. MURAKAMI, Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption, *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pp. 273-275, ACM Press, 1999

[INT06] IA-32 Intel Architecture Optimization Reference Manual, 248966-013US, http://download.intel.com/design/Pentium4/manuals/24896613.pdf, 2006

[JOH94] T. JOHNSON AND D. SHASHA, 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases,* Morgan Kaufmann Publishers Inc., pp. 439-450, 1994

[KAR94] R. KAREDLA, J. S. LOVE AND B.G. WHERRY, Caching Strategies to Improve Disk System Performance, Computer, Vol. 27, N. 3, *IEEE Computer Society Press*, pp. 38-46, 1994

[MEG03] N. MEGIDDO AND D.S. MODHA, ARC: A Self-Tuning, Low Overhead Replacement Cache, *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies,* USENIX Association, pp. 115-130, 2003

[MIL03] A. MILENKOVIC, M. MILENKOVIC AND N. BARNES, A Performance Evaluation of Memory Hierarchy in Embedded Systems, *Proceedings of the 35th South eastern Symposium on System Theory*, pp. 427- 431, 2003

[MIP06] MIPS Technologies, SOC-it L2 Cache Controller, http://www.mips.com/content/Products/Platforms

[MPC05] ARM11 MPCore Processor r0p3 Technical Reference Manual, Ref DDI0360C, http://www.arm.com/pdfs/DDI0360D_arm11mpcore_r1p0_trm.pdf, Dec 2005

[ONE93] E.J. O'NEIL, P.E. O'NEIL AND G. WEIKUM, The LRU-K Page Replacement Algorithm for Database Disk Buffering, *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data,* ACM Press, pp. 297-306, 1993

[ONE99] E.J. O'NEIL, P.E. O'NEIL AND G. WEIKUM, An Optimality Proof of the LRU-K page Replacement Algorithm, *J. ACM,* Vol. 46, N. 1, pp. 92-112, 1999

[SCU03] T. SCHWARZ, Santa Clara University, Computer Engineering, COEN 180, http://www.cse.scu.edu/~tschwarz/coen180/LN/MemoryHierarchy.html, Memory Hierarchy, 2003

[SH3] Hitachi SH3-DSP: Processor Overview, Berkeley Design Technology, Inc. (BDTI), http://www.bdti.com/procsum/sh3-dsp.htm

[SHA01] D. SHARP, TARMAC, A Dynamically Recompiling ARM Emulator, University of Warwick, http://www.davidsharp.com/tarmac/, 2001

[SLE85] D.D. SLEATOR AND R.E. TARJAN, Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM*, Vol. 28, N. 2, pp.202-208, 1985

[SMI82] A.J. SMITH, Cache Memories, *ACM Computers Survey*, Vol. 14, N.3, ACM Press, pp. 473-530, 1982

[SMI83] J.E. SMITH AND J.R. GOODMAN, A Study of Instruction Cache Organizations and Replacement Policies, *SIGARCH Computer Architecture News,* Vol. 11, N. 3, ACM Press, pp. 132-137, 1983

[SO88] K. SO AND R.N. RECHTSCHAFFEN, Cache Operations by MRU Change, *IEEE Transaction Computers,* Vol. 37, N. 6, IEEE Computer Society, pp. 700-709, 1988

[SPA03] Ultra-Sparc Family, http://www.sun.com/processors/Ultra-Sparc-III/index.xml

[SPE00] CPU-Intensive Benchmark Suite SPEC CPU2000, http://www.spec.org/cpu/

[TYS95] G. TYSON, M. FARRENS, J. MATTHEWS AND A.R. PLESZKUN, A Modified Approach to Data Cache Management, *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture,* IEEE Computer Society Press, pp. 93-103, 1995

[TYS97] G. TYSON, M. FARRENS, J. MATTHEWS AND A.R. PLESZKUN, Managing Data Caches Using Selective Cache Line Replacement, *Int. J. Parallel Program.*, Vol. 25, N. 3, Kluwer Academic Publishers, pp. 213-242, 1997

[WIK06] WIKIPEDIA, DEC Alpha, http://en.wikipedia.org/wiki/DEC_Alpha, 2006

[WON00] W. WONG, J.-L. BAER, Modified LRU Policies for Improving Second-Level Cache Behavior, *Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[XIL96] P. ALFKE, Efficient Shift Registers, LFSR Counters and Long Pseudo-Random Sequence Generators, XAPP052, http://www.xilinx.com/bvdocs/appnotes/xapp052.pdf, 1996

[ZOU04] H. AL-ZOUBI, A. MILENKOVIC AND M. MILENKOVIC, Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite, *ACM-SE 42: Proceedings of the 42nd annual southeast regional conference*, ACM Press, pp. 267-272, 2004

Appendix A

Number of status bits

A.1. PLRUt algorithm

A decision tree for the PLRUt algorithm is drawn on Figure 14 p.14. The tree comprises $N_{steps} = \lceil \log_2(N_{ways}/2) \rceil$ steps. This is equivalent to the following formulation:

$$\exists ! \left(\vartheta, N_{steps} \right) \in [0,1[\times \mathbf{N}, \quad \log_2 \left(\frac{N_{ways}}{2} \right) = N_{steps} - \vartheta$$

Each step i is composed of 2^i nodes. Therefore, the number of required additional bits per set is:

$$\sum_{i=0}^{N_{steps}} 2^i = \frac{1-2^{1+N_{steps}}}{1-2} = 2^{\vartheta} N_{ways} - 1$$

For a 2^n-way set associative cache, $N_{sets}.(N_{ways}-1)$ additional bits are thus required for the whole cache.

A.2. MPLRU algorithm

A decision tree for the MPLRU algorithm is drawn on Figure 16 p.15. One sees that the tree is composed of $N_{steps} = \lceil \log_2(N_{ways}/2) \rceil$ steps. Contrary to PLRUt, there are two types of nodes: the MBAIs, which require 2 bits, and the TBAIs, which require 1 bit. The TBAIs are the last step of the tree, and the MBAI the other ones. From this observation, one deduces the required number of additional bits:

$$\left\lceil \frac{N_{ways}}{2} \right\rceil + 2 \sum_{i=0}^{N_{steps}-1} 2^i = \left\lceil \frac{N_{ways}}{2} \right\rceil + 2 \frac{1-2^{N_{steps}}}{1-2} = \left\lceil \frac{N_{ways}}{2} \right\rceil + 2^{\vartheta} N_{ways} - 2$$

For a 2^n-way set associative cache, the MPLRU algorithm requires $N_{sets}.(3N_{ways}/2-2)$ additional bits.

A.3. LRU algorithm

Number of ways. There are two means to encode the status bits for the LRU algorithm. The first version is the simplest one but quite expensive in terms of bits. This method is derived from the expression of the number of status bits found in [ZOU04]. A stack of N_{ways} elements must be maintained in this method where each element of the stack contains the number of the way it points to, hence requiring $\lceil \log_2(N_{ways}) \rceil$ bits for each stack element. For the whole cache, the number of required bits is:

$$N_{sets}. N_{ways}. \lceil \log(N_{ways}) \rceil$$

In opposition, the coding developed in this thesis (see Chapter 3 Section 3.2) requires N bits where:

$$\frac{N}{N_{sets}} = \sum_{p=0}^{k} \frac{N_{ways}}{2^{p+1}} \log_2 \left(\frac{N_{ways}}{2^p} \right) \qquad \text{with} \quad k = \left\lceil \log_2 \left(N_{ways} \right) \right\rceil - 1$$

One thus obtains:

$$\frac{N}{N_{sets}} = N_{ways} \left(\sum_{p=0}^{k} \frac{\log_2 (N_{ways})}{2^{p+1}} - \sum_{p=0}^{k} \frac{p}{2^{p+1}} \right)$$

Restricting our study to the 2^n-way set associative cache, one gets:

$$N_{ways}\left(\log_2(N_{ways})\frac{N_{ways}-1}{N_{ways}}-\sum_{p=0}^{k}\frac{p}{2^{p+1}}\right)$$

It is impossible to get a beautiful expression of this sum, so we only give an equivalent:

$$\sum_{p=0}^{k}\frac{p}{2^{p+1}}=\frac{1}{4}\sum_{p=0}^{k}(p+1)\left(\frac{1}{2}\right)^{p}\underset{k\to\infty}{\approx}\frac{1}{4}\frac{1}{\left(1-\frac{1}{2}\right)^{2}}=1$$

For a high number of ways, the number of required bits is then:

$$N\underset{N_{ways}\to\infty}{\approx}N_{sets}N_{ways}\left(\log_2\left(N_{ways}\right)-1\right)$$

The first values obtained with this encoding are $N(2) = 1$, $N(4) = 5$ and $N(8) = 17$.

These figures are to be compared with the previous ones and the one presented in the general section *N(N-1)/2*. This encoding is much more compact but at the price of complexity of decoding, encoding. In the case of our design, we save 1536 bits compared to the first situation, thereby leaving free space in the Dirty RAM for future improvements. One could argue that this encoding is only valid for the situations where the set stack is full. However, it is not the case. Indeed, as long as some lines are invalid, giving an order of preference between them does not matter because there is no loss of information. Once the line is filled, the stack is updated to be consistent with the access order. Thus, free or invalid ways are equivalent and can be encoded in this way. It introduces a slight dissymmetry but which would be seen on the whole life of the product (maybe faster deterioration of some hardware modules but it can be considered as harmless). The scheme of this implementation is done below for four ways.

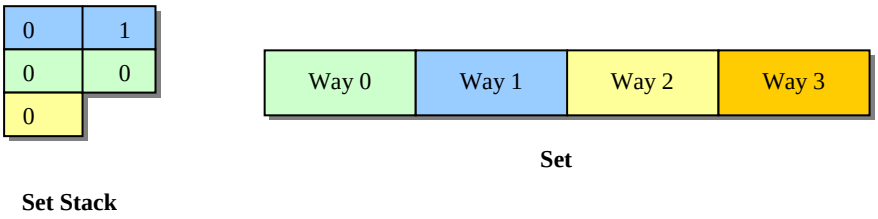

Figure 40: Improved LRU stack bits

Compacity. In Chapter 3 Section 3.2, it was stated that this algorithm is the most compact one. The proof of this assertion is given here as well as the assumptions made.

Proof. The information that we want to store is the sequence of access. According to the previous remark, the stack can always be considered as full. Thus, there are 4! different sequences to encode. Apart from the filling of the cache, the ways can be considered as equivalent. Indeed, during the filling of the cache, the ways are filled from 0 to 3. Apart from this slight dissymmetry, they are equivalent and all the sequences have the same probability p=1/24. Therefore, the Shannon entropy is:

$$H=-\sum_{i=0}^{23}p_i\log_2 p_i=-24p\log_2 p=\log_2 24\approx 4.585$$

According to Shannon's theorems, it is thus impossible to find a code whose average length is less than *H*. The word *average* is crucial in the previous assertion. Indeed, finding a coding as near as optimal as we want usually implies to have words of various lengths. This would imply to completely reorganize the way the status bits are addressed: decoding the address width would not allow one to directly access the elements because the widths of its predecessor are not known. In order to avoid this intricate issue, it is requested that all the code words have the same length. Indeed, if we do not assume this and if we want in the same time to keep a simple addressing for the elements, it would lead us to select the upper bound of the code widths as the basic width and would be a far from optimal solution. Therefore, the code width is the same for all the codes. Combining this with the entropy equation, it yields that the optimal width is 5 (a physical width must be an integer whereas an average one is a real). Consequently, the encoding proposed in this thesis is one of the optimal ones.

Appendix B

The cache simulator

As we have seen before, an implementation in C of the different cache algorithms was performed. Before giving the TARMAC outputs as input patterns of our implementation, some tests were developed to check the different features of the replacement policies. They are presented here to help understanding the behaviours of the cache lines replacement strategies.

B.1. How to use the cache simulator

The program is implemented in such a way that the number of ways can be modified but it must be a power of two. However, when the true LRU policy is used, the number of ways must be less or equal than 8. For the other algorithms, it must between 1and 32, since the status bits are represented as an integer (more than 32 ways is an uncommon situation in most caches as shown in Table 3 p.24).

The cache size corresponds to the three possibilities of the ARM caches available up to now on the ARM processors (16, 32 and 64 KB respectively encoded here as 0, 1 and 2). All the other parameters are derived from the data found in `replacement.cfg`. This file has to be present in the same directory as the executable file and must be structured like:

```
NUMBER_OF_WAYS_MIN=2
NUMBER_OF_WAYS_MAX=4
CACHE_SIZE_MIN=-1
CACHE_SIZE_MAX=2
CLK_PERIOD=10
TIME_TO_L2=10
TIME_TO_MAIN=100
HIT_L2=90
INSTRUCTION_SIZE=8
DEBUG_ALGO=0
DEBUG_LOOKUP=0
LFB_ENABLE=1
SBCACHE_ENABLE=1
SBCACHE_SIZE_MIN=6
SBCACHE_SIZE_MAX=16
NB_SBC_LINE_ELTS_MIN=4
NB_SBC_LINE_ELTS_MAX=10
LOG_EVICTION=1
```

The minimal and maximal values correspond to the bounds of the simulations done by the cache simulator. The output file can be then read with any spreadsheet or data software. The parameters are:

- `NUMBER_OF_WAYS`: number of ways of the data cache; this should be a power of two to match the usual implementations relying on cutting the memory address,
- `CACHE_SIZE`: cache size encoded as $2^{4+CACHE\_SIZE}$ KB,
- `CLK_PERIOD`: clock period defined in ns,
- `TIME_TO_L2`: average time required to fetch a data from L2 cache in clock cycles,
- `TIME_TO_MAIN`: average time to fetch a data from the main memory in clock cycles,
- `HIT_L2:` average hit ratio in the L2 cache given in percentage,
- `LFB_ENABLE:` boolean which enables the latency simulation,
- `SBCACHE_ENABLE:` enables the Status Bit Cache defined in Chapter 6,

- `SBCACHE_SIZE:` size of the status bits cache defined in number of lines,
- `NB_SBC_LINE_ELTS:` number of status bits groups per line of the status bits cache.

The parameters `DEBUG_LOOKUP` and `DEBUG_ALGO` simply activate printing of the results of elementary operations of the cache lookup and of the replacement algorithm. As its name suggests, `LOG_EVICTION` activates logging the different cache lookups and the result of these lookups (hit/miss, evicted way, true eviction[a] or not…).

The status bits are defined after the access has been performed, as the contents of the ways. All the bits are written in big endianness, i.e. `bits = "001"` is equivalent to:

`bits[0] = 1`, `bits[1] = 0` and `bits[2] = 0`.

Finally, the command to launch a simulation is:

`replacement` *`input_file policy1`* [*`policy2 policy3...`*]

where *`policy`* is one of the following: GRR, LRU, ModBits, MPLRU, non-MRU, OPTIMAL, PLRUm, PLRUt, RANDOM, 1-bit, ROUNDROBIN or SIDE.

B.2. Basic cache behaviour

The first thing to test is the mapping: each main memory line should be mapped on the appropriate cache line. It is done with a "*modulo*" computation, as defined in Chapter 2, section 2.5 p. 6. Writes and reads on random addresses are checked and match the expected behaviour. The replacement algorithms must be called only where the set is full. Thus, a test independent of the policy was written to check that the cache simulator uses free ways when available. It only fills a data cache set and after some accesses in other sets, it requests once more all the data of this cache set. The case where the core accesses another word of the line while the LFB fetches a data is tested too.

B.3. Test sequences of the replacement policies

In this chapter are addressed some of the tests developed for the verification of the cache simulator. One is presented for each algorithm and its results are dealt with if there is a complication. These sequences were written such that they test the particularities of each algorithm and their result on each step is checked.

a. Remark about the filling of the cache

In this appendix, all the sequences begin with the filling of the cache set considered by the replacement policies. The addresses are computed such that they match to the same cache set for sizes of 8, 16, 32 and 64 KB. This part is identical for all the algorithms, since it is managed by the free ways part of the program. The filling sequence 0x300, 0x4300, 0x8300, 0xc300 is not represented on the tables.

b. Optimal algorithm

Address accessed	*Cache set*						*Hit/ Miss*
	Way 0	*Way 1*	*Way 2*	*Way 3*	*Next accessed data*	*Evicted way*	
0x1000	-	-	-	-	-	-	-
0x10300	0x300	0x4300	0x10300	0xc300	1,3,0,2	2	Miss
0x4300	0x300	0x4300	0x10300	0xc300	-	-	Hit
0xa378bc10	-	-	-	-	-	-	-
0xc300	0x300	0x4300	0x10300	0xc300	-	-	Hit
0x300	0x300	0x4300	0x10300	0xc300	-	-	Hit
0x8300	0x300	0x4300	0x10300	0x8300	0,1,2	3	Miss
0xaaaaaaaa	-	-	-	-	-	-	-
0x300	0x300	0x4300	0x10300	0x8300	-	-	Hit
0x4300	0x300	0x4300	0x10300	0x8300	-	-	Hit
0x10300	0x300	0x4300	0x10300	0x8300	-	-	Hit
0x4300	0x300	0x4300	0x10300	0x8300	-	-	Hit

Table 20: Test sequence of the optimal algorithm

[a] A true eviction occurs when there was a valid line in the way

The optimal algorithm is the reference algorithm of our study. Its implementation was developed to get an absolute reference of the performance of a replacement algorithm. The TARMAC file provides us with the knowledge of the future accesses of the core and makes this algorithm feasible. Unfortunately, it has not been used intensively in this thesis since the computation time was too important and since the cache optimizations were not taken into account. The feature tested is the computation of the next accessed data, which gives the evicted line. The cases where the four ways are accessed in the future and where only a part of them is requested are studied. The simulated results match the theoretical result of Table 20. The implementation of the buffer has also been checked in a similar way as before. The test was made for sake of simplicity on the configuration with 2 ways and then checked on 4 ways.

c. SIDE

The update of the counter on misses and on hits in the surely LRU and possibly MRU part is checked here. The value of the evicted way is also tested. The representative test sequence is given in the table below.

Address accessed	*Cache set*						*Hit/ Miss*
	Way 0	*Way 1*	*Way 2*	*Way 3*	*Next accessed data*	*Evicted way*	
0x4300	0x300	0x4300	0x8300	0xc300	2	-	Hit
0x300	0x300	0x4300	0x8300	0xc300	2	-	Hit
0x8300	0x300	0x4300	0x8300	0xc300	3	-	Hit
0x1c300	0x300	0x4300	0x8300	0x1c300	0	3	Miss
0x10300	0x10300	0x4300	0x8300	0xc300	1	0	Miss
0x4300	0x10300	0x4300	0x8300	0xc300	2	-	Hit
0x14300	0x10300	0x4300	0x14300	0xc300	3	2	Miss
0x300	0x300	-	-	-	1	0	Miss
0x4300	0x300	0x4300	-	-	2	1	Miss
0x8300	0x300	0x4300	0x8300	-	3	2	Miss
0xc300	0x300	0x4300	0x8300	0xc300	0	3	Miss
0x4300	0x300	0x4300	0x8300	0xc300	2	-	Hit

Table 21: Test sequence for SIDE

d. Round Robin

A quite simple test is written for a simple algorithms with not so many features. The sequence 0x10300, 0x4300, 0x1c300, 0x8300, 0x14300, *0x16300, 0x12300, 0xe300, 0xa300, 0x2300*, 0x18300 gives the following succession of evicted ways: 0, -, 1, -, 2, -, -, -, -, - and 3. The italic numbers correspond to set which do not lie in the current set.

e. Global Round Robin

The test is almost the same as Round Robin but now the Round Robin counter is global and this could lead to situations where the MRU line is evicted. The same sequence as above was tested. The evicted ways are 0, -, 1, -, 0, 1, 2, 3, 0 and 1.

f. PLRUm

The sequence is explained in the table below and matches the result obtained with the simulation. The accesses are done in the set 1. Those that are not mapped onto this set are marked in this table as -. The values shown below correspond to the value after the instruction has been taken into account.

Address accessed	*Cache set*						*Hit/ Miss*
	Way 0	*Way 1*	*Way 2*	*Way 3*	*Status bits*	*Evicted way*	
0x8300	0x300	0x4300	0x8300	0xc300	1100	-	Hit
0x10300	0x10300	0x4300	0x8300	0xc300	1101	0	Miss
0x14300	0x10300	0x14300	0x8300	0xc300	0010	1	Miss
0xc300	0x10300	0x14300	0x8300	0xc300	1010	-	Hit
0x10300	0x10300	0x14300	0x8300	0xc300	1011	-	Hit
0x1c300	0x10300	0x14300	0x1c300	0xc300	0100	2	Miss
0x10300	0x10300	0x14300	0x1c300	0xc300	0101	-	Hit
0x20300	0x10300	0x20300	0x1c300	0xc300	0111	1	Miss
0xaaaaaaaa	-	-	-	-	-	-	-
0x1c300	0x10300	0x14300	0x1c300	0x1c300	1000	-	Hit

Table 22: Test sequence of PLRUm algorithm

In this sequence, there are 5 hits. The different features tested are:

- the global phase (the bits status are reset in order to avoid that all bits are high),
- the eviction with respect to the status bit (the discarded way is the first one encountered with a low status bit).

g. PLRUt

The features tested are the designation of the evicted way and the update of the status bits on miss/hit. The numbering is described in Figure 41. In the algorithm, one needs the relation between a node and its children. Let k be a node. The node k lies on step s = E(log2(k+1)) where E(x) gives the integer part of x. Let $n1$ be the number of nodes between our node k and the last node of the step s (including the boundary node) and $n2$ the number of nodes on the stage s+1 between the first node and the first child of k (including boundaries). The children of node k are then $k+n1+n2$ and $k+n1+n2+1$. By construction of the tree, we thus get that the upper boundary nodes of a step s are $2^{k+1}-2$. The numbering described above is then used for the modification of the tree bits and their reading is presented. Noting the bits in big endianness, the sequence 0x10300, 0x300, 0xc300, 0x18300 induces the status bits 011,110,010 and 001. The decoding of them assigns the ways to discard 0, 2, -(hit) and 1.

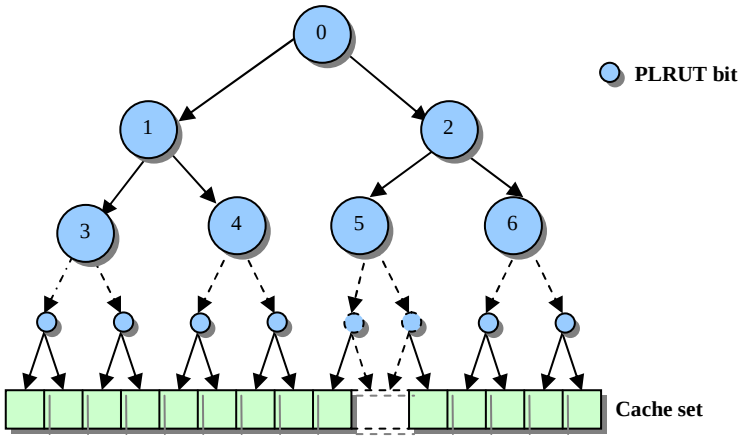

Figure 41: PLRUt tree nodes numbering

h. MPLRU

The numbering is given in Figure 42 where the notation n/n+1 stands for the couple previous-current bits of a MBAI. As before, the status bits are written in big endianness. The tested features are the computation of the way to discard and the update of the status bits.

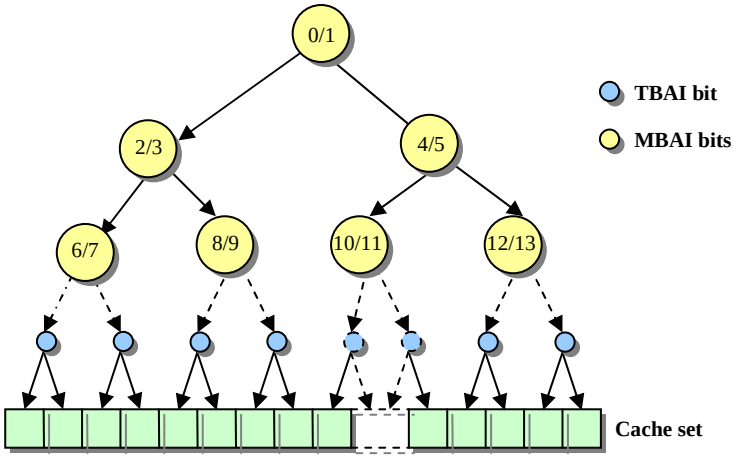

Figure 42: MPLRU tree nodes numbering

Address accessed	*Cache set*						*Hit/ Miss*
	Way 0	*Way 1*	*Way 2*	*Way 3*	*Status bits*	*Evicted way*	
0x300	0x300	0x4300	0x8300	0xc300	0110	-	Hit
0x8300	0x300	0x4300	0x8300	0xc300	1101	-	Hit
0x10300	0x300	0x4300	0x8300	0x10300	0100	3	Miss
0xc300	0x300	0xc300	0x8300	0x10300	0010	1	Miss
0x300	0x300	0xc300	0x8300	0x10300	0111	-	Hit
0x14300	0x300	0xc300	0x14300	0x10300	1101	2	Miss
0xc300	0x300	0xc300	0x14300	0x10300	1010	-	Hit
0x4300	0x4300	0xc300	0x14300	0x10300	1111	0	Miss

Table 23: Test sequence of MPLRU algorithm

i. LRU

Address accessed	*Cache set*						*Hit/ Miss*
	Way 0	*Way 1*	*Way 2*	*Way 3*	*MRU→LRU*	*Evicted way*	
0x10300	0x10300	0x4300	0x8300	0xc300	0,3,2,1	0	Miss
0x300	0x10300	0x300	0x8300	0xc300	1,0,3,2	1	Miss
0x10300	0x10300	0x300	0x8300	0xc300	0,1,3,2	-	Hit
0x14300	0x10300	0x300	0x14300	0xc300	2,0,1,3	2	Miss
0x18300	0x10300	0x300	0x14300	0x18300	3,2,0,1	3	Miss
0x300	0x10300	0x300	0x14300	0x18300	1,3,2,0	-	Hit
0xc300	0xc300	0x300	0x14300	0x18300	0,1,3,2	0	Miss
0x300	0xc300	0x300	0x14300	0x18300	1,0,3,2	-	Hit
0x4300	0xc300	0x300	0x4300	0x18300	2,1,0,3	2	Miss
0x18300	0xc300	0x300	0x4300	0x18300	3,2,1,0	-	Hit

Table 24: Test sequence for LRU

j. Dirty bits

The principle of the test is the same as before. However, to distinguish dirty cache locations and non-dirty cache locations, a distinction between reads and writes is done. The corresponding test sequence is given in the table below. The filling is done by three successive writes and a read request.

Address accessed	*Type of the request*	*Cache set*					*Hit/Miss*
		Way 0 - D0	*Way 1 - D1*	*Way 2 - D2*	*Way 3 - D3*	*Evicted way*	
0x10300	R	0x300 - 1	0x4300 - 1	0x8300 - 1	0x10300 - 0	3	Miss
0x300	R	0x300 - 1	0x4300 - 1	0x8300 - 1	0x10300 - 0	-	Hit
0xc300	R	0x300 - 1	0x4300 - 1	0x8300 - 1	0xc300 - 0	3	Miss
0x18300	W	0x300 - 1	0x4300 - 1	0x8300 – 1	0x18300 - 1	3	Miss
0x10300	R	? - ?	? - ?	? - ?	? - ?	? Rand.	Miss

Table 25: Test sequence for ModBits

B.4. Advanced behaviour of the cache

Finally, the advanced features of the modelized cache were investigated too. The latency to fetch the data from the upper level memory and the behaviour of the Status Bit Cache are tested here. Like the replacement algorithms, different tests were written to verify that it performs as expected. The latency tests are based on the time analyses and the management of the buffers. The ARM11 hit in LFB is not implemented here since it would have assumed to model the arrival of data in different groups whenever the requested address matches the one stored in LFB. This optimization is considered as negligible in a first approximation. The data is then not present until the time limit it has been asserted on miss (depending on the random number) and then is present on next access and written immediately in the cache. These two phases of assigning the time limit and waiting on the one hand and writing the data back in the cache are validated thanks to some different tests which include the extreme conditions too. For the SBC, the tests were a bit simpler since it is almost the same implementation as the simulated cache. The single difference relies on the fully associativity of this cache and on the fact that all the memory requests can be considered as reads for the SBC. Besides, it is made easier by the non-simulation latency for this cache since the aim is to have coarse clues about the optimal size. In this optics, tests equivalents to the data cache one were run.

The different operations of the cache simulator were tested and the major sequences were presented in this chapter. The validation of the Verilog implementation is performed in Appendix D.

Appendix C

Selection of benchmarks

C.1. Benchmarks

As it has been seen before, the benchmarks came from different sources. The simulated benchmarks are listed below:

- `3d_geometry_f32`: usual geometry transformations used in 3D graphics,
- `3d_persptris_f32`: computes 3D perspective,
- `500_huffmann`: implements the Huffman coding,
- `500_mandeld`: computes a Mandelbrot set (fractals),
- `500_qsort`: quick sort algorithm,
- `automotive_aifftr`: computes the inverse FFT (Fast Fourier Transform) on complex inputs (two arrays of real and imaginary parts),
- `automotive_idctrn`: computes an inverse discrete cosine transform used in digital videos and graphics applications such as image recognition,
- `automotive_iirflt`: computes an IIR (Infinite Impulse Response) on fixed-point values,
- `automotive_matrix`: computes the LU (Lower x Upper) decomposition of a matrix, its determinant and its product with another matrix,
- `automotive_tblook`: simulates a table lookup particularly used in ABS, ESP…,
- `consumer_rgbcmy`: performs the conversion from RGB to CMY especially used in printers,
- `consumer_rgbcmykv`: performs conversion from RGB to CMYK conversion extensively used in printers,
- `mp3`: decodes a mp3 input,
- `mpeg4_deblock`: heart routine of the MPEG4 algorithm,
- `mpeg4_decode`: decodes an MPEG4 file,
- `mpeg4_encode`: encodes in MPEG4 format,
- `networking_ospf`: implements the Dijkastra/Shortest Path First algorithm which is widely used in routers,
- `networking_pktflow`: simulates a network router work but focused on checksum and comparison operations,
- `networking_route_lookup`: implements an IP tree and simulates the work of a router,
- `networking_tcp`: simulates TCP traffic in networks (the number of packets and the size of the segments can be modified to simulate networks from FTP to Gigabit Ethernet),
- `office_bezier`: computes Bezier curves,
- `office_dither`: uses the Floyd-Steinberg Error Diffusion dithering algorithm to convert a greyscale picture for printing,
- `office_rotate`: it rotates clockwise a binary image of 90 degrees. This operation is common in printers,
- `telecom_viterbi`: it implements the Viterbi algorithm widely used in Error Correcting Codes (ECC).

C.2. Selection of benchmarks

The criteria which governed the selection of benchmarks were the following:

- the cache is stressed: there are evictions from the data cache and the replacement policy is called many times to select the victim line,
- there are differences in the ratio hit between the different cache sizes: this element insures that the impact of the replacement policy is visible and that the simulation environment does not correspond to the high data cache size of the curves. This term high ends refers to the ratio data cache size over working set size. For high values of this ratio, compulsory misses are the main verily the single contributors to miss ratio. As it has already been stated, replacement policies do not influence this type of misses and thus these simulations are meaningless for the scope of this work. This element is also checked by the number of true evictions,
- the representative benchmarks exhibit realistic hit ratio (i.e. they are not all equal to 99.91-99.94% even for small data cache), which should ensure that the memory patterns are realistic.

Finally, the group of representative benchmarks is composed of:

- 3d : `3d_persptris_f32`,
- 500: `500_huffman`,
- EEMBC: `automotive_aifftr`, `automotive_iirflt`, `automotive_matrix`, `consumer_mpeg4_decode`, `consumer_mpeg4_encode`, `networking_ospf`, `networking_pktflow8`, `networking_route_lookup`, `networking_tcp`, `office_bezier`, `office_dither`, `office_rotate`, `telecom_vitterbo`.

This selection thus affords a wide range of embedded applications that stress the data cache noticeably. Among this selection, some are more interesting for the scope of this study since they exhibit a greater differentiation of the replacement strategies. The benchmarks above mentioned are: `automotive_matrix`, `consumer_mpeg4_decode`, `networking_tcp` and `office_rotate`.

Appendix D

Validation of the implementation in Verilog

Different checks have been performed at each step of the development in Verilog and the main lines of these verifications are handled in this appendix. Knowledge about the solution implemented in this thesis – the Status Bit Table endowed with the Status Bit Cache – is assumed. If it is not the case, you are kindly advised to read the relevant chapter.

D.1. Status Bits Updater

For the updating process, the module was submitted to different inputs, generated by test benches written to this purpose. The outputs were monitored and compared to the results obtained by the replacement algorithms written in C. Obviously this automatic check is valid for the configurations without locked ways. Indeed, this feature was not taken in account by our cache simulator, since locking is seldom used and not on a long time slot. Otherwise, the results were checked manually.

Different conflict signals are asserted in the test bench in order to detect any bug like allocating a locked way, allocating a valid way when there is a free one, not including the locked ways in the updated status bits. All these verifications helped assuring that the updater operates correctly.

D.2. Status Bits Cache

As for the updater, different signals are asserted to check the different modules: particularly one makes certain that there is no double copy of an element inside the Status Bit Cache. Besides, a modular approach was adopted which enables to debug small pieces of hardware which are independent one from another, and then to dive into the issue of their relative timing and behaviours. A dedicated test bench was written for each module to test its function. Different types of inputs fed the Status Bit Cache and this work was eased by the development of a small C program which interprets a text file where the input sequences are written in a more human friendly manner. Then it writes a Verilog file as where the different signals were asserted and unasserted. This Verilog file was included directly in the test bench with an `` `include `` directive. This program ensured to save a lot of time and to test quickly different configurations. This “human friendly” language comprehends the following instructions:

- `wlin line_number address status_bits_to_be_written`: writes a complete line,
- `wstb line_number address status_bits_to_be_written`: writes a status bit group,
- `rlin addr`: reads a whole line,
- `rstb addr`: reads a status bit group,
- `wrll write_line_number write_address read_address status_bits_to_be_written`: writes and reads two (different or not) whole lines,
- `wrls write_line_number write_address read_address status_bits_to_be_written`: writes a whole line and reads a status bits group,
- `wrsl write_line_number write_address read_address status_bits_to_be_written`: reads a whole line and writes a status bit group ,
- `wrss write_line_number write_address read_address status_bits_to_be_written`: writes and reads status bit groups.

One sees that the line number is required for the writes. Indeed, it assumes that the write is split up in two phases by the arbiter which is located before the SBC. Thus, seen from the cache, the write lookup

is a read and the write phase is a write where the line has already been known. A sequence example is given in the table below, where the grey shaded lines correspond to the filling of the cache.

Access	*SBC Index*	*Hit or written way*	*Data from SBT*	*Allocated way to LSU*	*Allocated way to LFB*	*SBC line changed*	*SBC line data*	*Data to SBT*
`readh`	001001011	0010	3a51	-	-	1	3a51	-
`readh`	011011010	-	91bc	-	0010	0	91bc	-
`readh`	110001000	0001	ab47	-	-	2	ab47	-
`readh`	101010101	0100	871e	-	-	3	875e	-
`readh`	101010001	0001	1072	-	-	4	1072	-
`readh`	101000110	0100	4301	-	-	5	4701	-
`readh`	011001111	1000	7a13	-	-	6	8a13	-
`readh`	101110010	0010	1047	-	-	7	1247	-
`readm`	010000100	-	2034	-	0001	0[a]	2034	3a51[b]
`lfill`	010000100	0010	3146	-	-	0	2036	-
`lookh`	101010101	0010	ab67	-	-	(3)	-	-
`wdata`	101010101	0010	ab67	-	-	3	877e	-
`lookm`	010000001	-	481e	-	0010	1	481e	-[c]
`lfill`	010000001	1000	481e	-	-	1	489e	-
`lookm`	110001000	-	2479	1000	-	(2)	(ab47)	-

Table 26: Test sequence of the Status Bit Cache Logic with PLRUm replacement policy

D.3. Integration of Status Bit Cache in the ARM11 MPCore processor

Once the modules worked outside the ARM11 Data side of L1 level, it had to be included in the ARM11 MPCore processor. This integration had also to be checked and the sequences and methods of these verifications are presented in the tables below. The tests were run on T1-32 and Dhrystone. T1-32 was used for the first verifications. The tests are usually split into two main parts:

- the beginning of the sequence after having turned on MMU and Data Cache,
- the normal running mode which corresponds to accesses with status bits already different from zero and thus more complicated situations.

The issue of relying on existing programs is that the cache is not stressed as strong as it would be necessary. For this reason, some basic assembly programs were written to stress more efficiently the cache and to test specific situations of the data cache. These verifications are made tough by reordering of the instructions and potential merges. Nevertheless, they were very useful for the validation of the implemented modules.

All together, these programs tested all the possible situations:

- Fast path cache lookup with hit,
- Fast path cache lookup with miss,
- Hit in Status Bit Cache and miss in data cache,
- Miss in Status Bit Cache and miss in data cache,
- Hit in Status Bit Cache and hit in data cache,
- Miss in Status Bit Cache and hit in data cache,
- Line Fill…

The final tests targeted real applications which should go through numerous situations. Thus, benchmarks and software were run on the modified version of the processor. There were two types of verification. Firstly, the control signals ensured that the behaviour of the cache was not crazy and the checking of the results of the programs confirmed that it operated correctly. But, it was not sufficient to assure that the implementation was correct. Therefore a second and more comprehensive check was performed. All the memory requests entering the data cache were logged and the obtained log file fed

[a] Because the replacement policy is PLRUm and a filling of the cache is the beginning of a new phase where the information about last accesses is lost: only the MRU line is protected from eviction at the beginning of a new phase but it is a better solution than accessing free ways in order 7..0

[b] There is a write-back of line 0 in Status Bit Table because the line has been modified on its first write in Status Bit Cache

[c] The status bits have not been modified (it is a `readm`) and there is no need to write back the status bits

the cache simulator designed previously. The results of this last one were then compared to the result of the Verilog implementation. However, this was very time consuming since the files needed to be modified for the requests to match the input requirements of the cache simulator. It explains why this verification has been carried out on some benchmarks only, especially at the beginning of the overall verification phase, and not on all the simulated software and benchmarks.

The steps presented above allowed us to detect and correct bugs in a hierarchical manner. The final tests ensure that the design works properly for most cases. However, it could not be assured certainly that the design is bug free since the verification was not formal and since the testing vectors are not a generating family of the space of possibilities. This remark raises the issue of the validation of a design and of its fault-tolerance. This theme is beyond the scope of this thesis and the performed verifications are considered as sufficient to assert that the design operates properly.

Index

Architecture 21
Cache
  direct-mapped 7
  fully associative 6
  N-way-set-associative 8
Cache line 5
Coherency protocol bits 7
Dirty bit 7
Global Round Robin 22
GRR ..... *See* Global Round Robin
Hit 5
Hit-under-miss 22
Lockdown 22
LRU ..... *See* Least Recently Used
Memory Management Unit 22
Microarchitecture 21
Miss 5
  capacity 9
  coherency 9
  compulsory 9
  conflict 9
MMU ..... *See* Memory Management Unit
Principle of locality 11
Replacement algorithm
  1-bit 13
  Least Frequently Used 13
  Least Recently Used 12
  Modbits 28
  MPLRU 15
  non-MRU 28
  optimal 11
  PLRUm 14
  PLRUt 14
  Random 12
  Round Robin 13
  SIDE 16
SBC 58
SBRB 59
SBWB 58
Status Bit Group 51
Status Bit Read Buffer 59
Status Bit Table 58
Status Bit Write Buffer 58
Status bits 28
Validity bit 7
Word 5
Write through 5
Write update 5