A Cache-Based Hardware Accelerator for Memory Data Movements

A Cache-Based Hardware Accelerator for Memory Data Movements

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft, Nederlands,
op gezag van de Rector Magnificus Prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 13 oktober 2008 om 12:30 uur

door

Filipa DUARTE

Mestre em Engenheria Electrónica e Telecomunicações
verkrijging aan de Universidade de Aveiro, Portugal
geboren te Coimbra, Portugal

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. K.G.W. Goossens

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr. K.G.W. Goossens	Technische Universiteit Delft, promotor
Prof.dr. G. Brown	Indiana University
Prof.dr. W. Najjar	University of California Riverside
Prof.dr. L. Sousa	Universidade Tecnica de Lisboa
Dr.ir. J.S.S.M. Wong	Technische Universiteit Delft
Dr. L. Carro	Universidade Federal do Rio Grande do Sul
Prof.dr.ir. A.J.C. van Gemund	Technische Universiteit Delft
Prof.dr. C. Witteveen, reservelid	Technische Universiteit Delft

Prof.dr. Stamatis Vassiliadis provided substantial guidance in this thesis.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Author: Filipa Duarte
Title: A Cache-based Hardware Accelerator for Memory Data Movements
Cover: processor die photo from Intel® Core™2 Extreme mobile
Subject headings: cache, hardware accelerator, memory data movements.

Thesis Technische Universiteit Delft - Faculteit Elektrotechniek, Wiskunde en Informatica
Met samenvatting in het Nederlands.
ISBN 978-90-72298-01-0

Printed in The Netherlands

Give the software guys the hardware,
and they will use it!

Yale Patt in “The Future of Computing -
Essays in memory of Stamatis Vassiliadis”

A Cache-Based Hardware Accelerator for Memory Data Movements

Filipa Duarte

Abstract

This dissertation presents a hardware accelerator that is able to accelerate large (including non-parallel) memory data movements, in particular memory copies, performed traditionally by the processors. As today's processors are tied with or have integrated caches with varying sizes (from several kilobytes in hand-held devices to many megabytes in desktop devices or large servers), it is only logical to assume that data to-be-copied by a memory copy is already present within the cache. This is especially true when considering that such data often must be processed first. This means that the presence of the caches can be utilized to significantly reduce the latencies associated with memory copies, when a "smarter" way to perform the memory copy operation is used.

Therefore, the proposed accelerator for memory copies takes advantage of the presence of these caches and introduces a redirection mechanism that links the original data (in the cache) to the copied addresses (in a newly added indexing table). The proposed solutions avoid cache pollution and duplication of data, and efficiently schedule the access to the main memory, thus effectively reducing the latency associated with memory copies. Moreover, the proposed accelerator supports copies of cache line and word granularity, can be connected to a direct-mapped or a set-associative cache, and can efficiently reduce the memory copy bottleneck in single core processors and in multi-core processors that execute a message passing communication model.

The proposed solutions have been implemented in a FPGA as a proof of concept and incorporated in a simulator running several benchmarks to determine the performance gains of the proposal. In particular, for the receiver side of the TCP/IP stack, the proposed solutions can reach speedups from 2.96 to 4.61 times and reduce the number of instructions executed by 26% to 44%.

Contents

List of Acronyms

ASIC	application specific integrated circuit
BRAM	block RAM
BSD	Berkeley Software Distribution
CAM	content addressable memory
CB	control bus
CLB	configurable logic blocks
CO	cast-out
CPI	cycles per instruction
CPU	central processing unit
DB	data bus
DCU	data cache unit
DMA	direct memory access
DRAM	dynamic random access memory
DSOCM	data-side OCM

EDK Embedded Development Kit

FCP finite cache penalty
FE factor of eviction
FPGA Field-Programmable Gate Array
FSM finite-state machine
T fetching time

HDL hardware description language
HRO1 hit rate in OtherL1

ICU instruction cache unit
ILP instruction level parallelism
IOB input/output blocks
IP Internet Protocol
ISC Internet Systems Consortium
ISE Integrated Software Environment
ISOCM instruction-side OCM

JTAG Joint Test Action Group

L1 level 1 cache
L2 level 2 cache
LRU least recently used
LUT look-up-table

MPI	message passing interface
mr	miss rate
N.Ticks	number of ticks
OCM	on-chip memory
OS	operating system
OtherL1	miss rate of other L1 caches
OwnL1	miss rate of each L1 cache
PLB	processor local bus
PPC	PowerPC
RAM	random access memory
RDMA	remote direct memory access
ROM	read only memory
R	request rate
SRAM	static random access memory
SRL16	16-bit shift register
S	service time
TCP	Transmission Control Protocol
TE	trailing-edge delay

TLB translation lookaside buffer

TLP thread level parallelism

TOE TCP/IP offload engine

U utilization

V visitation probability

VHDL very high speed integrated circuit hardware description language

XST Xilinx Synthesis Technology

List of Figures

List of Tables

Chapter 1

Introduction

Data exchange operations exist in several types of processing: in inter-process communication (multitasking systems); in between processes running on the same memory space (multi-threading systems) and in multiprocessing systems. In particular, data exchange between different address spaces (inter-process communication and multiprocessing) requires physically moving data in the main memory, and therefore, several accesses to the memory hierarchy have to be performed. As memory bandwidth is scarce relative to processor bandwidth, data movements are expensive. One traditional way to address the imbalance between memory bandwidth and processor speed is the use of caches. As caches store the most recently used data, it is only logical to assume that data to be moved (e.g., by the inter-process communication in a multitasking system or by a multiprocessor system) is already present within the cache. This is especially true when considering that such data often must be processed first. Would it be possible, then, to take advantage of the presence of the caches to significantly reduce the latencies associated with memory data movements?

This chapter introduces, in Section 1.1, the motivation to address the memory data movements in a uniprocessor system and extents the motivation to a multiprocessor in Section 1.2. In Section 1.3, the related works are presented and in Section 1.4 the research questions addressed in this dissertation are presented. Finally, Section 1.5 describes the outline of the dissertation.

1.1 Motivation

The imbalance between memory bandwidth and processor speed, also refereed as "processor-memory performance gap", have been demonstrated over the year using several means. For commercial applications, node idle times were reported to reach 65% of execution time, and high-performance scientific computations reported to reach 95% node idle times: much of this is due to memory bottlenecks [53]. Hennessy and Patterson [29] demonstrated that the processor performance grew from 1980-1998 at 60% per year, while the access time to the memory improved at 10% per year, giving rise to an increasing "processor-memory performance gap". Moreover, this trend that existed in the past, is expected to continue into the future, as depicted in the STREAM Benchmark Reference Information web-site [85]. Furthermore, the processing performance from 1985 to 2001 increased at 50% per year on average, and sustainable memory bandwidth from 1975 to 2001 increased at 35% per year, on average, over all the systems measured in [85].

To understand better the impact of the memory data movements in the imbalance between memory bandwidth and processor speed, consider a extremely used application: the Transmission Control Protocol (TCP)/ Internet Protocol (IP) processing stack. The TCP was outlined in a 1974 paper by Kahn and Cerf [11] and was introduced in 1977 on the ARPAnet - and every network attached to the ARPAnet - forming the first network of computers. In 1980, IP [75] is added to TCP to provide the routing mechanisms of the networks. From then on, all networks that use IP are collectively known as the Internet.

Since August 1981, the Internet Systems Consortium (ISC) [33] keeps a record on the number of computers connected to the Internet. Besides the increasing number of users connected to the Internet, one also witnessed an increase in the variety and complexity of the services available. This imposes a demand for a faster network and faster computations. Figure 1.1 depicts the impressive growth in number of computers connected to the Internet and the evolution on network throughput. As the throughput of the network increases, the strain put on devices running the TCP/IP stack also increases, due to the increasing demand of performing more complex tasks in short time. The TCP/IP stack processing overhead is high whenever network bandwidth (packets arriving) is large in comparison to processor and memory bandwidths (packets processing).

As an example, consider a Pentium M processor (from 2003) with a DDR

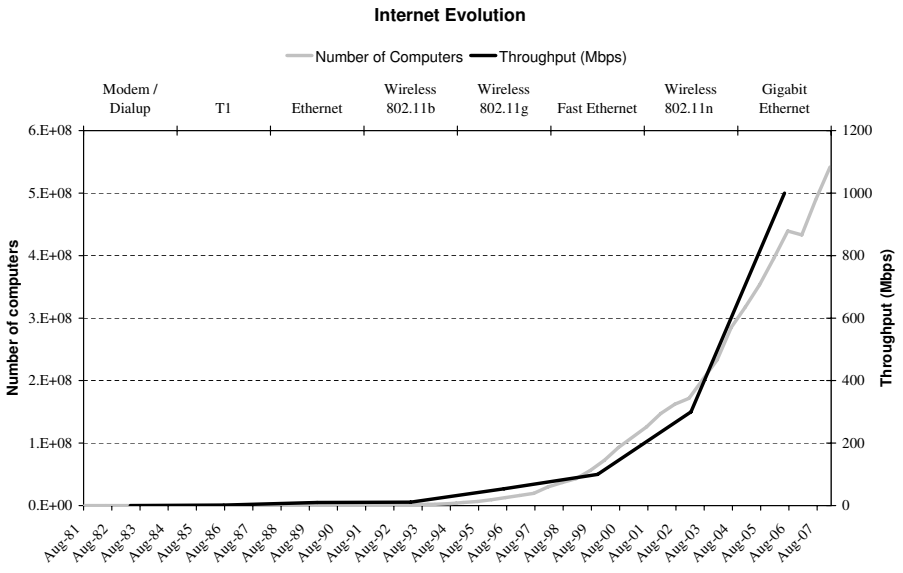


Figure 1.1: Internet evolution (source: ISC [33])

200 main memory running the TCP/IP stack. This system takes roughly 150 μsec to process 64 kB of data [109]. On a 100 Mbps network, this 64 kB of data will arrive roughly every 5 msec, however, on 10 Gbps network that time is roughly 50 μsec. Therefore, the time of arrival of packets has become of the same order of magnitude as the time it takes to process a single packet in software.

Network processing has been an object of study since its creation. In particular, in [16], [36], [47], [73] and [109], the authors present profiling information and different analysis of the TCP/IP stack in software. The main time-consuming parts were identified to be:

- Operating system integration: The operating system (OS) overhead is mainly due to interrupt processing, layered drivers and buffers management.
- Checksums: Checksum calculations are quite compute intensive, due to the heavy mathematical calculations needed.
- Memory copies: Memory copies are time-consuming mainly due to the difference in speed of the processor and the main memory, i.e., the "processor-memory performance gap".

Because the TCP/IP stack is deeply integrated with the OS, avoiding the OS integration overhead is a difficult task, though some work has been performed ([77] and [59]). The checksum bottleneck, however is easier to solve. As the checksum calculations are quite computing intensive, offloading them from the processor through dedicated accelerator included in the network cards, turned this quite time-consuming code into a negligible execution time. On the other hand, memory copies are time-consuming mainly due to the difference in speed of the processor and the main memory, so a typical offload of such functionality is not possible.

Analyzing in particular memory copies, Clark et al. [16] in 1989 demonstrated that 64% of the measured time of the authors experiment was attributable to check-summing and memory data movement, from it 48% was accounted for by data copying. Subsequent work has consistently demonstrated the same phenomenon, as the earlier Clark et al. study. Kay and Pasquale [36] reported results that separate the processing times for check-summing and for memory data movement operations. For the 1500 bytes Ethernet size, 20% of the total processing overhead time is attributable to data copying and the checksum accounted for 30% of the same processing time. This values corresponded to around 70% of all processing time of the TCP/IP protocol dedicated to memory data movements.

As the memory copies cross the processor/memory bus twice per copy (one transferring the data from the main memory to the processor and another transferring the data from the processor back to the main memory) it suffers twice from the "processor-memory performance gap". The main scheme to overcome this bottleneck is to utilize direct memory access (DMA) or a combination between DMA and software techniques. However, DMA-based approaches provide a limited solution mainly due to 3 reasons:

- DMAs are peripheral devices and therefore there is a significant overhead on the communication between the processor and the DMA device, as the initialization of the device has to be done explicitly;
- The notification of a DMA transfer completion is performed either through polling or interrupt, both being expensive;
- DMAs deal mainly with physical addresses and therefore user-level application cannot take advantage of them.

The software techniques used in combination with DMA typically restructure the OS to minimize or completely avoid memory data movements. In

particular, the OS virtual page remapping and the zero-copy techniques were presented by Druschel et al. [67] and Thadani et al. [37], respectively. However, OS virtual page remapping is only efficient if the size of the packet is bigger that the OS virtual page size and the zero-copy technique is only applicable to the traditional UNIX OS interfaces[1]. Therefore, the memory copies bottleneck is still an open issue in today's networking processing systems.

In order to evaluate the impact of such time-consuming parts of the network processing on a newer network standard, the Bluetooth standard was profiled, as it also uses the TCP/IP stack[2]. The main conclusion of the study was that the `memcpy` function is the most time-consuming function (except the interrupt-related functions)[3]. Four actions, that include the `memcpy` function, are performed by the OS when handling a Bluetooth 'file transfer': 'frame acknowledging', 'interrupt handling', 'receiving packet' and 'reassembling frame'. The copy size used in either these actions is also regular, being 339 bytes for 4 packets plus 151 bytes for the last packet on the 'receiving packet' action and 1507 bytes when 'reassembling frame'. A graphical view of these actions is depicted in Figure 1.2. Therefore, the conclusion goes in the same direction

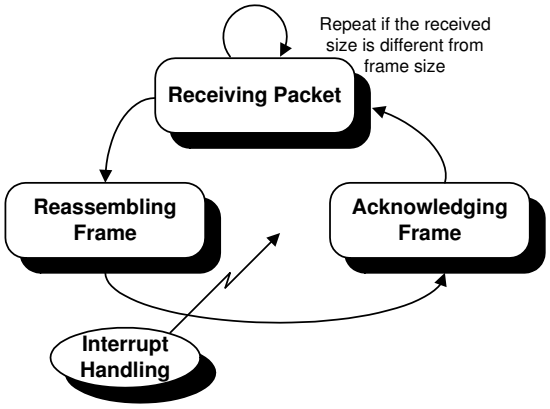


Figure 1.2: Bluetooth actions

of the related work on network processing: memory copies are a bottleneck in nowadays systems, even for new protocols.

[1] Further details on DMA and software techniques are introduced in detail in Section 1.3
[2] The details of this work can be found in Appendix A.
[3] The `memcpy` function and its variations are introduced in detail in Section 2.3.

1.2 Multiprocessor Platforms

Even though certain physical limits have been reached, it is continuously possible to put more transistors on a single integrated circuit die. However, the effects of these physical limitations can cause significant heat dissipation and data synchronization problems. The demand for more capable processors causes designers to use various methods to increase performance, such as instruction level parallelism (ILP) and thread level parallelism (TLP). ILP methods like superscalar pipelining are suitable for many applications, but are inefficient for others that tend to contain difficult-to-predict code. Many applications are better suited to TLP methods, and using multiple independent processors is one common method used to increase a system's overall TLP. A combination of increased available space on the integrated circuit die due to refined manufacturing processes and the demand for increased TLP is the reasoning behind the creation of multiprocessors, in particular, multi-core processors.

Most commonly the terms multi-core and multiprocessor have been both used to refer to several processing units[4] that are manufactured on the same integrated circuit die. In this dissertation[5], the term multi-core refers to several processing units manufactured on the same integrated circuit die. In contrast to multi-core processors, the term multiprocessor refers to multiple physically separate processing units (not in the same integrated circuit die), which often contain special circuitry to facilitate communication between each other. Summarizing:

- Uniprocessor: One processing unit with its caches in one integrated circuit die with the necessary interfaces to communicate with an main memory and peripherals;
- Multiprocessor: Several processing units with their caches, connected through an interconnect network, that allows for distributed execution of tasks; can be in several or one integrated circuit die.
  - Multi-core processor: Several processing units with their caches in one integrated circuit die and its bus-based interconnection, with the necessary interfaces to communicate with an main memory and peripherals;

[4] Also referred as central processing unit (CPU).

[5] The definition of the terms multi-core and multiprocessor have evolved to a gray area, where different authors actually mean different systems.

Multiprocessors systems are built on top of architecturally different platforms that support a wide variety of parallel programming models and communication models. However, the choice for a particular programming model greatly depends on the communication model utilized, as programs are typically tailored towards it. Moreover, from the hardware point of view, the communication model is tightly-coupled with the platform used, as the choice of the communication model for a particular platform can significantly impact the performance and ease of use of such systems. The communication models can be classified as:

- Shared memory: where the communication is implicitly performed via loads and stores to a global shared address space; and
- Message passing: where the communication is performed explicitly by utilizing messages containing the data to be communicated, e.g., message passing interface (MPI) [55] or SHMEM [79] implementations.

Looking into the hardware support for multiprocessors systems, there is a convergence to mainly two types of platforms:

- Tightly-coupled: multiple processing units connected through a high-performance interconnect, accessing a shared memory hierarchy and implementing a shared memory communication model.
- Loosely-coupled: nodes of processing units, in which the message passing communication model is implemented in software across the nodes. Examples of such systems include Linux Beowulf cluster [4].

An extreme case of tightly-coupled systems is the multi-core processor, where the number of processing units is small (normally called cores, and reaching a maximum of 8 per chip) and the interconnect is bus-based. As the cores in a multi-core processor share the memory hierarchy, the obvious communication model is the shared memory.

With the increasing demand on processing power, the processing units of a traditional node of a loosely-coupled system have evolved from a uniprocessor per node to one or more multi-core processors per node. Therefore, the applications that were developed for nodes of uniprocessors (that utilize a message passing communication model) are now being executed in nodes of multi-core processors. Consequently, there is a need to carefully evaluate the impact of the

message passing communication model on the multi-core processors. Moreover, research has proven there are benefits in using message passing communication models with tightly-coupled systems (not specifically multi-core processors) for particular applications ([40], [44] and [78]).

As the message passing communication is performed with send and receive messages, the manner this data is transferred can impose penalties that will greatly impact the performance of the overall system. Again, one of the main bottlenecks identified of using a message passing communication model concerns the memory data movements. As the trend is to increase the number of processing units (being cores or dedicated application specific processing units) in a single integrated circuit die, the impact of memory data movements is also expected to increase.

1.3 Related Work

A wealth of data from research and industry presented that memory data movements are responsible for substantial amounts of processing overhead, in particular, memory copies that can be the source of a considerable part of this processing overhead (as already introduced in the previous sections). It further demonstrated that, even in carefully implemented systems, eliminating these memory copies significantly reduces the overhead, as referenced below. Firstly, several studies that analyze the TCP/IP stack are presented, followed by studies that evaluate the impact on caches due to the execution of the TCP/IP stack. Secondly, several software approaches to the memory copy bottleneck are presented, followed by the approaches that combine changes to the network cards and software. Thirdly, hardware approaches to the problem are presented, divided by DMA-based approaches and non DMA-based. Finally, studies that analyzed memory copy bottleneck in multiprocessor environment are presented.

Clark et al. [16] in 1989, concluded that the TCP overhead processing is attributable to both per-packet operations (costs due to the OS integration such as interrupts, context switches, process management, buffer management, timer management) and the costs associated with processing individual bytes (specifically, computing the checksum and moving data in main memory). The authors found that moving data in main memory is the one of the most important of the costs, and their experiments concluded that memory bandwidth is the greatest source of limitation. The data presented demonstrated that 64% of the measured time was attributable to per-byte operations, check-summing

and memory data movement, from it 48% was accounted for by data copying. Subsequent work has consistently demonstrated the same phenomenon, as the earlier Clark et al. study. Kay and Pasquale [36] reported results that separate the processing times for check-summing and for memory data movement operations. For the 1500 bytes Ethernet size, 20% of the total processing overhead time is attributable to data copying and the checksum accounted for 30% of the processing time.

A number of studies reported results that per-byte operations dominate the processing costs for messages longer than 128 bytes ([13], [16], [21], [36], [49], [69] and [73]). For smaller messages, the main costs are demonstrated to be per-packet operations ([12] and [36]). However, the percentage of overhead due to per-byte operations increases with packet size, since the time spent on per-byte operations scales linearly with the message size. As networks get faster, data copying and check-summing become the dominating overhead, both because the other overheads are amortized over large packets and because per-byte operations stress a critical resource, the memory bus.

There has been some work evaluating the impact on caches while executing the TCP/IP stack ([30], [62] and [110]). Nahum et al. in [62] presented work that reached the following main conclusions, when executing the TCP/IP stack: i) instruction cache behavior is significant; ii) cold cache performance falls dramatically; and iii) larger caches and increased associativity improve performance. Zhao et al. in [110] studied the cache behavior for several TCP/IP data and implemented a specific and dedicated to networking actions cache. The authors have demonstrated that the header and the payload do not present temporal locality (as they have just arrived to the system), however the payload alone does provide spatial locality. Moreover, Huggahalli et. al. in [30] demonstrated that almost 100% of all incoming data from the network card, is subsequently read by the processor. This is the main reason for the authors proposing an approach to locate the data arriving immediately on the processor's cache.

There are many examples of copy elimination by using a variety of different software approaches. Such approaches that typically restructure the OS software to minimize or completely avoid data movements, have demonstrated significant improvement in the system performance ([13], [14], [37], [67] and [68]) The work presented by Thadani et al. [37] extended the traditional UNIX OS interfaces to avoid transfers of data between user-defined buffers and the kernel. Therefore, these interfaces lend to an efficient zero-copy data transfer. In this work, the network throughput was improved by more than 40%

and the processor utilization reduced by more than 20%. The work presented by Druschel et al. [67], developed a new facility in the OS called *fast buffers* (or *fbufs*). It combined the virtual page remapping with shared virtual memory and exploited the locality of input/output (I/O) traffic. The authors claim that the usage of *fbufs* can provide the same performance as the fastest page remapping in literature and it offers better performance than shared memory. The same page remapping concept was also used by Chu [14], with a the copy-on-write technique. The authors present performance improvement on TCP/IP stack executing a Solaris OS however, the performance is dependent on the performance of the cache of the system. Pai et al. [68] presented a unified I/O buffering system for a general-purpose OS. It provided a layer of abstraction that eliminates the redundant copies and multiple buffers of data. The authors provided performance improvements between 40% and 80% on a prototype implementation in FreeBSD [9]. More recent work by Chase et al. [13], measuring the processor utilization, concluded that avoiding copies reduces the processing time spent on data access from 24% to 15% at 370 Mbps for 32 kB data. This is an absolute improvement of 9% due to copy avoidance. The total processor utilization was 35%, with data access accounting for 24%. Thus, the relative importance of reducing copies is 26%. At 370 Mbps, the system is not very heavily loaded. The relative improvement in achievable bandwidth is 34%. This is the improvement seen if copy avoidance were added when the machine was saturated by network I/O.

A number of studies performed improvements on the network cards to reduce the number of copies ([38], [72], [73], [81], [87] and [92]). Steenkiste et al. in [38] and [81] presented a Communication Accelerator Board, where at its core is a memory used for outboard buffering of network packets. Moreover, the memory feeds three DMA engines and provide checksum calculation. In order for this accelerator to be supported by the OS, the necessary extensions were implemented. The authors claim that their solution can be 3 times more efficient than the original implementation. Walsh [92] presented a high-performance network adapter for a bus. This accelerator consists on a data and control memory interface (to communicate directly with the main memory), a master and slave interface to the bus and the necessary interfaces to the send and receive parts of the network card. In order to support such accelerator, the authors also developed the software, such as a device-driver and small changes to the OS. More recently, Regnier et al. [72] presented the Embedded Transport Acceleration, where one of the available cores in a multi-core processor is used to perform a packet processing engine tasks. This implies a partition of tasks between the general host core and the packet processing engine

core. Results presented by the authors depict an approximately 50% increase in transmitting performance and a throughput that can reach 4 Gbps. In [73], the same authors extend the previous work by introducing a memory-aware reference stack, that takes advantage of three latency reduction techniques: i) light-weight threading; ii) direct cache access; and iii) asynchronous memory copies. The authors show that the combination of these techniques can double the network throughput and reduce the number of clock cycles spend per packet by one third. Finally, remote direct memory access (RDMA) [74] is a technology that allows computers in a network to exchange data in main memory without involving the processor, the cache, or the OS of either computers. Like locally-based DMA, RDMA improves throughput and performance because it frees up resources. RDMA also facilitates a faster data transfer rate. RDMA implements a transport protocol in the network card hardware and supports a feature called zero-copy networking. Zero-copy networking makes it possible to read data directly from the main memory of one computer and write that data directly to the main memory of the other computer. The communication is performed through messages, that are "one-sided" in the sense that they will be processed by the adapter that receive them without involving the processor on the system that receives the messages.

Only recently hardware solutions started to appear to solve the data movement costs. The traditional DMA solution has been used extensively to transfer data between network cards and the main memory without much processor intervention or control. However, it needs to be explicitly initiated by the OS (since it is treated as a peripheral device). Therefore, a large overhead is incurred and user applications cannot directly utilize this solution making it limited in use. Intel's I/O Acceleration Technology [87] presents a set of hardware features that also include DMA. It attempts to alleviate the receiver packet processing overheads by using split headers (TCP/IP processing), an asynchronous DMA copy engine (memory copies between network cards and main memory) and multiple receive queues (memory bandwidth). The asynchronous DMA copy engine is in the OS kernel space and has direct access to the memory to improve performance. However, as it is based on a DMA device, the accesses to the device have to be explicitly managed by the application and, therefore, there are overheads that cannot be avoided.

Non-DMA based solutions were presented in [30], [73], [99] and [109]. The TCP/IP offload engine (TOE) [99] has emerged as an attractive solution which can reduce the host processor overhead and improve network performance at the same time. This is accomplished by offloading the TCP/IP stack from the processor into a dedicated accelerator that will perform the process-

ing involved in the TCP/IP stack ([30] and [73]). Zhao et al. in [109] present a hardware support for memory copies. This work presents a copy engine that is able to duplicate the data in the main memory by adding new features to the traditional memory controller. This provides reduction of cache pollution, however it will result in an unnecessary overhead if the copied data is used (touched) by processor, as it was demonstrated by [30].

In multiprocessor environment, memory copies have also been identified as a bottleneck for several cache coherent systems. Shan et al. in [78] compare the performance of the implementation of the three major programming models (shared address space, MPI [55] and SHMEM [79]) on a cache coherent multiprocessor. The authors concluded that removing the extra copy and using lock-free management queues in the message passing models can improve performance, however that implied changing the MPI and SHMEM implementation. With these improvements the three implementations performed quite similarly up to 16 processors and for small problem sizes. For more processors and bigger problem sizes, the following situations impact the performance of the system: i) remote accesses of cache line granularity and poor spacial locality on the remote data; ii) explicit transfers that either put data in the cache or in the main memory of the destination; iii) difference in cache conflict behavior; iv) situations when the cache coherence protocol degrades performance; and v) the implementation of barriers. A more recent study by Leverich et al. in [44] compares the message passing and shared memory communication model in a chip multiprocessor. The authors demonstrate that both communication models scale well, however the message passing model benefits from having a cache coherent multiprocessor as it enhances locality and can actually be easier to use.

One of the first machines supporting shared memory and message passing communication models were the Cray T3D [19] and the Stanford FLASH [40]. The designers of both machines identified the need to alleviate expensive operations in the path of `send` and `receive` messages, in order to provide the expected performance. For that, the solution relied on avoiding message copying through direct transfer of data between processes, and overlap computation with communication. The solution implemented in the Cray T3D was the use of a system level block transfer engine, which used DMA to transfer large blocks of contiguous or strided data to or from remote memories. Based on the Stanford FLASH, Heinlein et al. [28] implemented a custom programmable node controller containing an embedded processor that can be programmed to implement both cache coherence and message passing protocols.

However, the need for a data transfer engine is still a matter of debate. Woo et al. in [98] analyzed the performance of integrating a data transfer engine in a system closely resembling the Stanford FLASH architecture. According to the authors, the benefits of block transfer are not substantial for cache coherent multiprocessors. The reasons given are: i) the relative modest fraction of time applications spend in communication; ii) the difficulty of finding enough independent computation to overlap with communication latency; and iii) the cache lines often capture many of the benefits of block transfer. However, in a more recent work presented by Buntinas et al. in [10], the authors analyze the performance of transferring large data in symmetric multiprocessors. The authors analyze five different mechanisms (shared memory buffers, message queues, Ptrace system calls, kernel module copies and network cards) in terms of latency, bandwidth, cache usage and suitability to support message passing communication protocol. The main conclusion is that, as soon as the proper mechanism is chosen, these mechanisms do provide performance benefits and are suitable for message passing, contradicting the conclusions reached by Woo et al.. Another software solution for optimizing memory copies in multiprocessor systems has also been presented by Prylli et al. in [71]. The authors designed and implemented new protocols of transmission targeted to parallel computing of the high speed Myrinet network. Nieplocha et al. in [63] introduced a new portable communication library that provides one-sided communication capabilities for distributed array libraries, and supports remote memory copy, accumulate, and synchronization operations optimized for non-contiguous data transfers.

Summarizing, DMA-based approaches provide only limited solutions due to the high overhead introduced to explicitly initialize the devices. Software techniques are either OS dependent (e.g., [37]), or not valid for all cases (like when a packet is small then the OS virtual page in the case of [67]). Furthermore, today's network cards commonly offload the checksums, which removes the one part of per-byte overhead (the other part is the memory copy operation). They also coalesce interrupts to reduce per-packet costs. Thus, today copying costs account for a relatively larger part of processor utilization than previously, and therefore relatively more benefit is to be gained in reducing them. Moreover, from the cache studies previously presented, there is a cache behavior that can be taken advantage of: the fact that the payload presents spacial locality and that almost all the incoming (received) data is subsequently read by the processor, as presented by [62].

The solutions presented in this dissertation do not incur in the penalties of the DMA-based approaches and they are not platform or OS dependent.

Moreover, they take advantage of the presence of the cache (as it was demonstrated that the performance of the cache can have impact on the performance of the memory copy). Furthermore, the solutions presented in this dissertation can also be applicable to multiprocessors platforms and efficiently reduce the impact of memory copies in message passing communication protocol.

1.4 Research Questions

The previous sections have demonstrated that memory copies are a bottleneck in several systems (uniprocessor and multiprocessor) and for different standards (networking standards based on TCP/IP and the message passing communication model). Moreover, the previous proposals to solve this bottleneck still cannot provide the necessary performance in some circumstances or can only be applied to limited number of cases. It has been also presented in the previous sections, that the processor's evolution has increasingly taken benefit of the presence of caches, intended to reduce the "processor-memory performance gap". The possible benefits of utilizing such caches and a trend to continue in such direction has been identified. Therefore, the research questions this dissertation addresses are:

- Can the presence of caches in today's processors be exploited to solve the memory copy bottleneck?
- How do the proposed solutions in this dissertation perform compared with existing approaches?
- How can the proposed solutions in this dissertation be adapted to support multiprocessor platforms?

In order to address these questions, prototyping platforms are chosen to implement the proposed solutions. Their hardware implementation will demonstrate how feasible the solutions are and estimate their real performance. Moreover, in order to study the performance benefits a simulator is utilized. Utilizing a simulator (with performance numbers from the hardware implementation) will allow to further perform performance studies for multiple benchmarks and real applications. The benefits of utilizing these two analysis is two-fold. A study on raw performance and of the quantity of hardware resources necessary to implement the proposed solutions can be derived from the prototyping platforms and an event-driven simulator provides the cycle accurate timing evaluation of the proposed solutions. Therefore, these two meth-

ods provide the possibility of performing an accurate and complete analysis of the proposed solutions. Moreover, in order to evaluate the multiprocessor solution, a analytical study is utilized that provides the theoretical benefits of the proposed solutions.

1.5 Outline

This section discusses the organization of the reminder of the dissertation which consists of the following chapters:

- Chapter 2 introduces some basic concepts and the necessary background to better understand the remainder of this dissertation. It introduces the definition, organization, policies and design of caches and the general behavior and utilization of a memory copy operation. Moreover, the platforms utilized in prototyping and simulating the proposed solutions are also introduced in this chapter.
- Chapter 3 introduces the concept of the cache-based memory copy hardware accelerator. It also describes the design of the proposed solutions for different cases and presents the expected benefits compared with the traditional approach.
- Chapter 4 presents the methods utilized to demonstrate the proposed solutions. It introduces the details of the platforms chosen and the details of implementation of the proposed solutions on these platforms.
- Chapter 5 introduces the results of the synthetic benchmarks executed in the previous presented platforms. Moreover, it presents the performance evaluation of the proposed solutions and discusses the results.
- Chapter 6 describes the applicability of the proposed solutions to a multiprocessor platform. It describes the system targeted, the analytical analysis used to demonstrate the benefits of the proposed solutions and the proof of concept.
- Chapter 7 presents the conclusion of this dissertation and describes the main contributions of the research. Finally, several future work directions to continue the described research are presented.

Chapter 2

Background

In many current-day systems, processors perform many of the mentioned memory copies. Moreover, such processors often are tied with or have integrated caches with varying sizes (from several kB in hand-held devices to many MB in desktop devices or large servers), to improve performance.

Section 2.1 motivates the topics presented in this chapter and Section 2.2 presents the necessary concepts on cache design tradeoffs and implementation. Section 2.3 introduces the memory copy operation in more detail. Section 2.4 presents the prototyping platforms based on the Xilinx Virtex family and Section 2.5 introduces the details of the simulator used to demonstrate the performance benefits of the proposed solutions. Finally, Section 2.6 summarizes this chapter.

2.1 Introduction

As presented in the Chapter 1, one traditional way to address the imbalance between memory bandwidth and processor speed, i.e., the "processor-memory performance gap", is the use of caches. Moreover, it was also presented that the payload of a packet received through TCP/IP does provide spatial locality and that almost all incoming data from the network card is subsequently read by the processor (i.e., it has to go through the cache at some point). Therefore, a "smarter" way to perform the memory copy operation could take advantage of the presence of the caches. Consequently, in Section 2.2 the concepts on cache design, tradeoffs and implementation details are introduced.

In Chapter 1 was also motivated that memory copies are a bottleneck in

today's processing system. In order to understand the extend of this bottleneck, examples where the memory copy operation is utilized and its details are introduced in Section 2.3.

The solutions presented in this dissertation utilize two different platforms to demonstrate their benefits: the Xilinx Virtex Field-Programmable Gate Array (FPGA) and the Simics simulator. The advantage of using two different platforms is two-fold. The prototyping in real hardware provides an estimate of hardware resources utilized and the raw performance of the proposed solutions. The numbers gather when prototyping are afterwards utilized to correctly model the proposed solutions under the simulator. This approach provides more accurate measurements when evaluating the proposed solutions with a simulator. Therefore, Section 2.4 presents the details of the prototyping platforms, in particular the Xilinx Virtex family, and Section 2.5 introduces the simulator platform, Simics.

2.2 Caches

The concept of cache became popular in the 1970's (the papers that introduced the cache concept and design in 1968 were [18] and [45]) as a way of speeding up main memory access time. The basic idea of a cache is to predict what data is required from main memory to be processed. Therefore, a cache is used by the processor to reduce the average time to access the main memory. The cache is a smaller and faster memory which stores copies of the data from the most frequently used memory locations. When the processor wishes to read from or write to a location in the main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to the main memory.

The utilization of caches in processors is expected to be more prominent as technological advances continue to allow more transistors on a single chip with increasingly less transistors being dedicated to logic. The evolution of on-chip caches is depicted in Table 2.1.

To understand the impact of the use of a cache, consider a system with a processor and a main memory that takes 1 μsec to read an instruction. A program is made up of a series instructions each one being stored in a location in main memory (typically, located in sequential memory addresses), say from address 100 upwards. The instruction at location 100 is read out from the main memory and executed by the processor, then the next instruction is read from

Date	CPU	Cache size on-chip
April 89	80486DX	8 kB L1
September 91	80486SX	8 kB L1
March 92	80486DX2	8 kB L1
March 93	Pentium	(8 kB Inst. + 8 kB Data) L1
March 94	80486DX4	8 kB L1
November 95	Pentium Pro	(8 kB Inst. + 8 kB Data) L1 + 256 kB L2
January 97	Pentium MMX	(16 kB Inst. + 16 kB Data) L1
May 97	Pentium II	(16 kB Inst. + 16 kB Data) L1 + 512 kB L2
August 98	Celeron	(12 kB Inst. + 8 kB Data) L1 + 128 kB L2
February 99	Pentium III	(16 kB Inst. + 16 kB Data) L1 + 256 kB L2
November 00	Pentium IV	(12 kB Inst. + 16 kB Data) L1 + 256 kB L2
May 01	Xeon	8 kB + 256 kB + 512 kB
June 01	Itanium	32 kB + 96 kB + 2 MB
March 03	Pentium M	(32 kB Inst. + 32 kB Data) L1 + 1 MB L2
April 05	Pentium D	(12 kB Inst. + 16 kB Data) x 2 + 2 MB x 2
July 06	Core 2 Duo	(32 kB Inst. + 32 kB Data) x 2 + 2 MB
January 07	Core 2 Quad	(32 kB Inst. + 32 kB Data) x 4 + 4 MB x 2

Table 2.1: Cache presence and size evolution

location 101 and executed, then 102, 103, etc. If the processor takes 100 nsec to execute the instruction, it then has to wait 900 nsec for the next instruction. Now, lets introduce in the system a cache with an access time of 250 nsec between the processor and the main memory. When there is a request for the first instruction at location 100, the cache requests addresses 100, 101, 102 and 103 from the main memory all at the same time, and stores them in the cache. Instruction at location 100 is passed to the processor for processing, and the next request, for 101, is provided by the cache. Similarly, 102 and 103 are provided at the much increased speed of 250 nsec. When the processor requests the instruction at location 104, the process is repeated to reload the cache with the next instructions being requested. Therefore, a cache provide fast access to the data, by keeping a copy of a range of sequential memory addresses.

The reason why requesting sequential addresses (in the previous example, the cache sequentially requests locations 100, 101, 102 and 103) pays off is because program code and data have temporal and spatial locality. This means that, over short periods of time, there is a good chance that the same code or data gets reused. In temporal locality, a memory location that is referenced by a program code or data at one point in time is likely be referenced again in the near future. In spatial locality, a program code or data is more likely to reference a particular memory location if the program has referenced a nearby memory location in the recent past. Realizing that locality exists is key to the concept of caches as used today.

Therefore, the utilization of the cache reduces the access time of the processor to data requested. Other intermediary storage devices between the processing unit and the hard disk drive (HDD) intent to achieve the same objective. The closest storage device (therefore, on-die and the smallest one) from the processing unit is the register file (RF). Next to it cames the cache, which can be on-die or off-die. The next intermediary storage device is the main memory and finally the HDD. Therefore, as the size of the RF is smaller than the cache, it is accessed faster. The same happens with the cache, as it is smaller then the main memory, is accessed faster. And finally, the size of the main memory is smaller than the HDD, thus being accessed faster. Therefore, there is a clear continuum on distance from the processing unit and size of the intermediary storage device. Figure 2.1 depicts a schematic analogy of such distance/size tradeoff.

2.2.1 Cache Organization

Most modern processors have at least three independent caches: an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store, and a translation lookaside buffer used to speed up virtual-to-physical address translation for both executable instructions and data (this last type of caches are introduced in Section 2.2.6).

A typical instruction or data cache is divided in two main parts: a cache directory and cache data-array. The cache directory can be seen as a list of memory addresses of the data stored in the corresponding location of the cache data-array (which is the one that contains the data). In a typical cache design, the cache directory is constituted by two different arrays: a tag-array and a valid-array. Figure 2.2 depicts the referred cache organization. The address provided by the processor is divided into 3 parts (Figure 2.3 depicts this organization): the index, the tag, and the offset. The index is used to access the

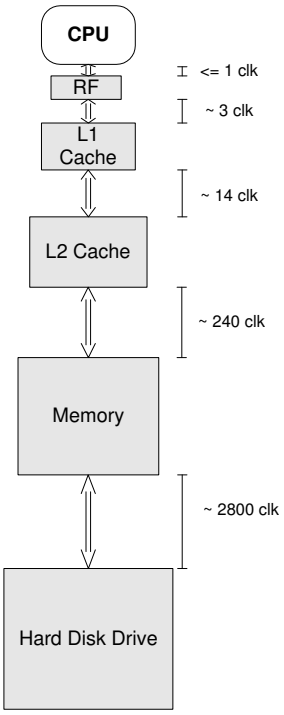


Figure 2.1: Typical memory hierarchy.

cache directory and the cache data-array. The tag is used to compare with a tag already in the tag-array (on a read). If the tag supplied by the tag-array is the same as the tag of the address requested by the processor and the valid bit supplied by the valid-array is set, a cache read hit is registered. On a cache read hit, the data supplied by the cache data-array (the cache line) is accessed and, based on the offset, the correct word is provided. If a write occurs, the tag is written to the tag-array. Based on the offset, the correct word is accessed and its content modified. If the architecture supports byte accesses, besides the tag, the index and the offset, also a byte write is used to identify which byte, within the selected word, is to be written.

Reads dominate processor cache accesses. On a read request, if the data is in cache (read hit), the processor will have the data available on the next clock cycle, as the data can be read at the same time that the tag-array and the valid-array are read and compared. Therefore, the data read begins as soon as the address is available. If the data is not in cache (read miss) the processor has to stall until data is provided by the main memory (the time

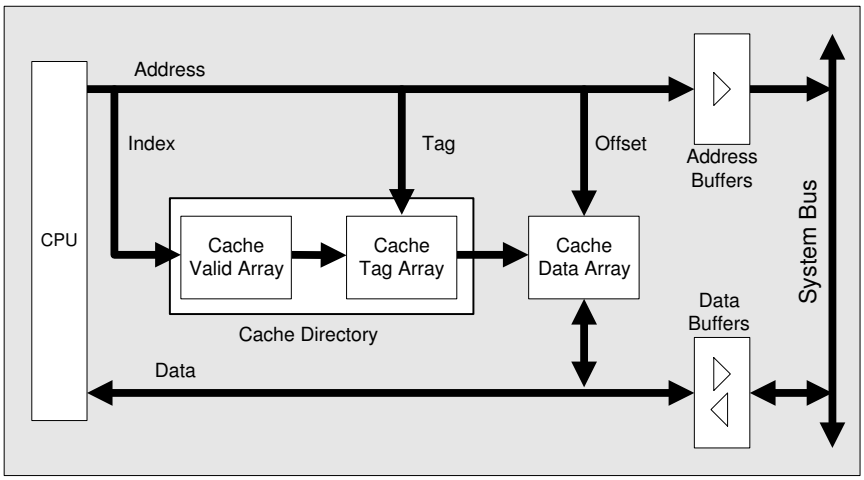


Figure 2.2: Typical cache organization.

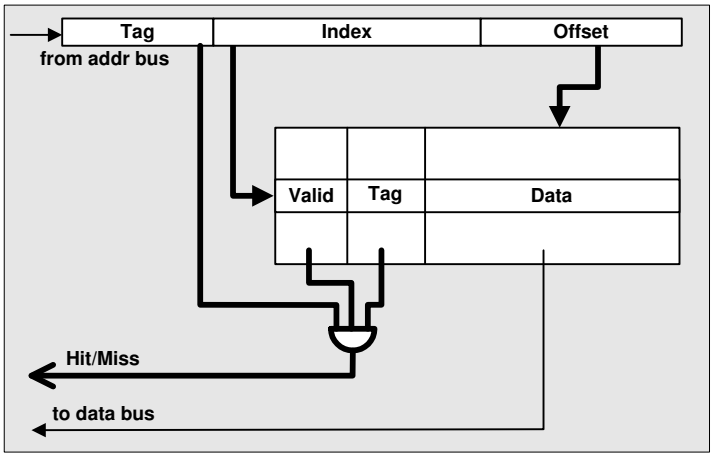


Figure 2.3: Typical cache implementation.

to provide the data by the main memory is depended on the main memory technology and implementation). If the read is a miss, the tag-array and the valid-array are read and compared at the same time as the data is read. There is no benefit on performing such operation but there is also no harm (except power consumption), just ignore the value read.

The previous description applies to both instruction and data caches. However, in an instruction cache there is no write as instructions are only read by

the processor. Therefore, the next paragraphs describing the behavior of a write request to a cache do not apply to instruction caches.

On a write request several options are available that constitute performance tradeoffs. When data is written to the cache, it must at some point be written to main memory as well. The timing of this write is controlled by what is known as the write policy. The write policies on write hit often distinguish cache designs:

- Write-Through: The information is written to both the cache line in the cache and to the main memory.
  - Advantage:
    - easy to implement;
    - the main memory always has the most current copy of the data (consistent).
  - Disadvantage:
    - write is slower;
    - every write needs a main memory access;
    - as a result, the system uses more memory bandwidth.
- Write-Back: The information is written only to the cache line in the cache. The modified cache line is written to main memory only when it is replaced. To reduce the frequency of writing back cache lines on replacement, a dirty bit is commonly used. This status bit indicates whether the cache line is dirty (modified while in the cache) or clean (not modified). If it is clean the cache line is not written on a miss.
  - Advantage:
    - writes occur at the speed of the cache;
    - multiple writes within a cache line require only one write to main memory;
    - as a result, the system uses less memory bandwidth.
  - Disadvantage:
    - harder to implement;
    - the main memory is not always consistent with cache.

There are two common options on a write miss:

- Write-Allocate: The cache line is loaded on a write miss, followed by the write-hit action.
- No Write-Allocate: The cache line is modified in the main memory and not loaded into the cache.

Which write policy to choose is dependent on the available hardware resources and the maximum latency allowed by the application. Therefore, a study of the application behavior should be performed before choosing the write policy, as there is no perfect option to use.

2.2.2 Cache Miss Types

A cache miss refers to a failed attempt to read or write data in the cache, which results in a main memory access with much longer latency. In order to lower cache miss rate[1], a great deal of analysis has been performed on cache behavior. Sequences of memory references performed by benchmark programs were saved as address traces. Subsequent analysis simulated many different possible cache designs on these long address traces. Making sense of how the many variables affect the cache hit rate[2] can be quite confusing, however it is possible to separate misses into three categories:

- Compulsory misses: are those misses caused by the first reference to the data. These always happen when a application starts executing or when there is a context switch (swapping between applications) as the data is not in the cache. This is called cold-start. Cache size and associativity (introduced in the Section 2.2.3) have no impact in the number of compulsory misses.
- Capacity misses: are those misses that occur due to the finite size of the cache. Caches almost always have nearly every line filled with a copy of some line in main memory, and nearly every allocation of a new line requires the eviction of an old line. The relation between capacity miss rate and cache size measures the temporal locality of a particular application.

[1] The miss rate is the ratio between the number of accesses (both read and write) that miss in the cache (i.e., whose data is not present in the cache), and the total number of cache accesses.

[2] The hit rate is the ratio between the number of accesses (both read and write) that hit in the cache (i.e., whose data is present in the cache), and the total number of cache accesses. Therefore, it can also be defined as $1-$miss rate.

- Conflict misses: are those misses that could have been avoided, had the cache not evicted an entry earlier. Conflict misses can be further broken down into mapping misses (due to mapping of different addresses to the same index of the cache), that are unavoidable given a particular amount of associativity (introduced in the Section 2.2.3), and replacement misses (due to the choice of which line to replace), which are due to the particular victim choice of the replacement policy (introduced in the Section 2.2.4).

Several factors influence the cache miss rate, which typically result from a combination of cache size, cache line size, cache associativity (introduced in Section 2.2.3) and cache policies (introduced in Section 2.2.4).

2.2.3 Associativity

Cache associativity was introduced to reduce the conflict misses. Taking into account that cache lines are evicted to give place to new cache lines being loaded, consider an application using data that maps to the same cache line. Every new load evicts the previous data (stored in the same cache line) that will be needed after. Being able to store both data (that previously mapped to the same cache line) reduces the conflict misses. This can be accomplished by allowing addresses with the same index to be in the cache at the same time, and use the tag to differentiate among them.

Associativity is a tradeoff. If there are ten places that a new cache line can be mapped to, then when the cache is checked for a hit, all ten places must be searched. Checking more places (even if done in parallel) requires more power and area. On the other hand, caches with higher associativity suffer fewer conflict misses, so there is less time spent in servicing those misses. To determine which of the available places is used to hold the just loaded cache line, a replacement policy is used (introduced in the Section 2.2.4). Therefore, it is based on the replacement policy that the decision is taken where in the cache a copy of a particular entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called fully-associative. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is direct-mapped. Many caches implement a compromise and are described as set-associative.

One of the advantages of a direct-mapped cache is that it allows simple and fast access, as only one index can have a copy of the data (the cache line). That cache line can be read in parallel with the tag matching calculation, and when

the matching calculation is finished (and if it is a match) the data is available to the processor immediately. If the tag does not match the requested address, there is a cache miss, the data provided by the cache is ignored and an access to main memory is initiated. On a set-associative or fully-associative cache, the tag matching cannot be performed in parallel with the access to the data, as the location of the data is dependent of the tag.

The rule of thumb is that doubling the associativity, from direct-mapped to 2-way, or from 2-way to 4-way, has about the same effect on hit rate as doubling the cache size. Associativity increases beyond 4-way have much smaller effect on the hit rate.

2.2.4 Cache Policies

In order to make room for the new entry on a cache miss (both when writing and reading), the cache generally has to evict one of the existing entries. The heuristic that it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache line is least likely to be used in the future. Predicting the future is difficult, especially for hardware caches which use simple rules amenable to implementation in circuitry, so there are a variety of replacement policies to choose from and no perfect way to decide among them. One popular replacement policy, least recently used (LRU), replaces the least recently used entry (other replacement policies can be found in [29]). This algorithm requires keeping track of what was used when, which is expensive if one wants to ensure the algorithm always discards the least recently used item. In the implementation of this technique, every time a cache line is used, the age of all other cache lines changes. Therefore, the implementation requires the usage of “age-bits” to keep information about cache lines accesses and track the least recently used cache line based on the “age-bits”.

Another issue is the fundamental tradeoff between cache access latency and hit rate. Larger caches have better hit rates but longer latency. To address this tradeoff, many processors use multiple levels of caches, with small fast caches backed up by larger slower caches. Multi-level caches generally operate by checking the smallest level 1 cache (L1) cache first; if it hits, the processor proceeds at high speed. If the smaller cache misses, the second larger level 2 cache (L2) cache is checked, and so on, before main memory is checked. As the latency difference between main memory and the fastest cache has become larger, some processors have begun to utilize as many as three levels of on-chip cache. The tradeoff between size and access times was already presented

in Figure 2.1.

The design tradeoff between cache access latency and hit rate provided by the utilization of multi-level caches, introduce new design decisions. For instance, in some processors, all data in the L1 cache must be contained in the L2 cache. These caches are called strictly inclusive. Other processors have exclusive caches - data is guaranteed to be in at most one of the L1 and L2 caches, never in both. Still other processors do not require that the data in the L1 cache also reside in the L2 cache, although it may often do so. There is no universally accepted name for this intermediate policy, although the term mainly-inclusive has been used.

The advantage of exclusive caches is that they store more data. This advantage is larger when the L1 cache size is comparable to the L2 cache size, and diminishes if the L2 cache is many times larger than the L1 cache. When the L1 misses and the L2 hits on an access, the hitting cache line in the L2 is exchanged with a line in the L1. Exclusive caches require both caches to have the same cache lines sizes, so that cache lines can be swapped on a L1 miss, L2 hit. However, this exchange involves more work (specifically, more transactions on the bus) than just copying a line from L2 to L1, which is what an inclusive cache does.

One advantage of strictly inclusive caches is that when peripheral devices (or other processors in a multiprocessor system) wish to remove a cache line from the processor's cache, they need only to check the L2 cache (remove the line in the L2 cache implies removing it also from L1 due to inclusion). In cache hierarchies which do not enforce inclusion, the L1 cache must be checked as well. As a drawback, there is a correlation between the associativity of L1 and L2 caches: if the L2 cache does not have at least as many ways as all L1 caches in the system together, the effective associativity of the L1 caches is restricted.

2.2.5 Multiprocessor Cache Coherence

Multiprocessor systems can be viewed as a set of several uniprocessors (as introduced in Section 1.2). As processes (typically one process is assigned to one processor as explained in Section 1.1) need to work on data, caches were also introduced to speed up accesses and thus increase performance. Therefore, typically each processor in a multiprocessor system has, at least one, own cache.

Data needed by one process/processor might be present in a cache of an-

other processor (specially true if the data was processed by a process assigned to another processor). Therefore, caches in a multiprocessor system, may contain several copies of the same data stored in the main memory. Moreover, when one process/processor updates the data in its cache, copies of data in other caches in the system will become out of date. Therefore, there is a need to utilize communication protocols between the cache controllers to keep the data coherent and consistent. More precisely, coherency defines what value is returned on a read, and consistency defines when it is available.

Cache coherence defines the behavior of reads and writes to the same memory location. The coherence of caches is obtained if the following conditions are met:

- A read made by a processor P to a location X that follows a write by the same processor P to X, with no writes of X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P. This condition is related with the program order preservation, and this must be achieved even in uniprocessor architectures.
- A read made by a processor P1 to location X that follows a write by another processor P2 to the same X location must return the written value made by P2 if no other writes to X made by any processor occur between the two accesses. This condition defines the concept of coherent view of memory. If processors can read the same old value after the write made by P2, it is said that the memory is incoherent.
- Writes to the same location must be serialized. In other words, if location X received two different values A and B, in this order, by any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

These conditions are defined supposing that the read and write operations are made instantaneously. Unfortunately, this does not happen given the main memory latency and the architecture design. This defines the need for consistency models, i.e, determine when the data is available. If, for example, a write to location X on processor P1 precedes a read to location X on processor P2 by a very small time, it may be difficult to ensure that the read returns the value of the data written, since the written data may have not left the processor P1 at that point. The most straightforward model for consistency is called sequential consistency. It requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among

different processors were interleaved. More details on consistency models can be found in [29].

The two most common types of cache coherence protocols that are typically utilized are bus-based and directory-based, each having their own benefits and drawbacks. Bus-based protocols tend to be faster, provided enough bandwidth, since all transactions are a request/response seen by all processors connected to a logical or physical bus. The drawback is that bus-based protocols are not scalable. Every request must be broadcasted to all nodes in a system, meaning that as the system gets larger, the size of the bus and the bandwidth it provides must grow. Directory-based cache coherence protocols were invented as means of dealing with cache coherence in systems containing more processors than can be accommodated on a single bus. In directory-based systems, the directory can be held centrally with the main memory or can be distributed among the caches as singly or doubly linked lists. Therefore, directory-based protocols tend to have longer latencies but use much less bandwidth since messages are point-to-point and not broadcast. For this reason, many of the larger systems ($>$64 processors) use this type of cache coherency protocol. More details of both cache coherence protocols can be found in [29].

The most used bus-based cache coherency protocol is the MESI protocol, where every cache line is marked with one of the four following states:

- M - Modified: The cache line is present only in the current cache, and is dirty; it has a different value that the one stored in the main memory. The cache is required to write the data back to the main memory at some time in the future, before permitting any other read of the data.
- E - Exclusive: The cache line is present only in the current cache and it is clean; it matches the main memory.
- S - Shared: Indicates that this cache line may be stored in other caches of the system.
- I - Invalid: Indicates that this cache line is invalid.

All caches on the bus monitor (or snoop) the bus to determine if they have a copy of the block of data that is requested on the bus. A cache may satisfy a read from any state except Invalid. The Modified and Exclusive states are always precise, i.e., they match the true cache line ownership situation in the system. The Shared state may be imprecise: if another processor discards a Shared line, and this processor becomes the sole owner of that cache line, the

line will not be promoted to Exclusive state (because broadcasting all cache line replacements from all processors is not practical over a broadcast bus). In that sense the Exclusive state is an opportunistic optimization: if the processor wants to modify a cache line that is in the Shared state, a bus transaction is necessary to invalidate all other cached copies. The Exclusive state enables modifying a cache line with no bus transaction. Table 2.2 depicts the change of the state of a cache line as the result of either internal or external activity related to that line.

	M **Modified**	**E** **Exclusive**	**S** **Shared**	**I** **Invalid**
Is this cache line valid?	Yes	Yes	Yes	No
The copy of this cache line in memory is	out of date	valid	valid	-
Are there copies of this cache line in the caches of the other processors?	No	No	Maybe	Maybe
A write to this cache line	does not go to the bus	does not go to the bus	goes to the bus and updates the cache	goes directly to the bus

Table 2.2: Cache line state changes due to MESI

An Invalid line must be fetched (changing its state to Shared or Exclusive states) to satisfy a read. A write may only be performed if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. A cache may discard a non-Modified line at any time, changing to the Invalid state. A Modified line must be written back first before discarding it. A cache that holds a line in the Modified state must snoop (intercept) all attempted reads (from all other processors in the system) to the corresponding main memory location and provide the data it holds. This is typically done by forcing the read to back off (i.e., to abort the memory bus transaction), then writing the data to main memory and changing the cache

line to the Shared state. A cache that holds a line in the Shared state must also snoop (intercept) all invalidate broadcasts from other processors, and discard the line (by moving it into Invalid state) on a match. A cache that holds a line in the Exclusive state must also snoop (intercept) all read transactions from all other processors, and move the line to Shared state on a match. The MESI state diagram is depicted in Figure 2.4.

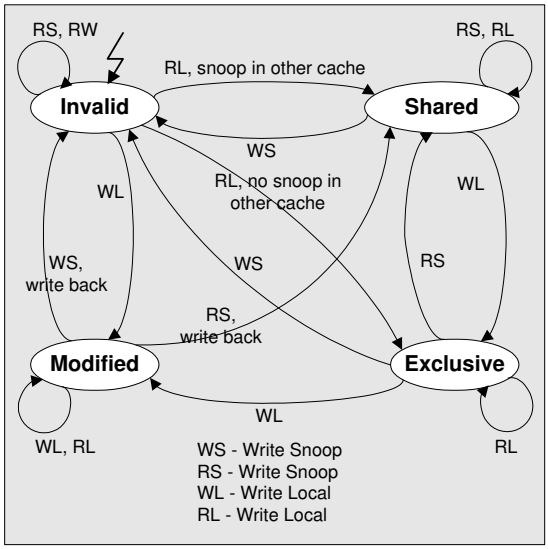


Figure 2.4: The MESI state diagram

2.2.6 Address Translation

A different kind of cache from the instruction and data caches presented in the previous sections, is the translation lookaside buffer (TLB). This cache performs the bridge between physical and virtual addresses. Each program running on the processor sees its own simplified address space, which contains code and data for that program only. Each program places data in its address space without regard for what other programs are doing in their address spaces. Virtual memory requires the processor to translate virtual addresses generated by the program into physical addresses in the main memory. The portion of the processor that does this translation is known as the main memory management unit and its fast path performs those translations through the TLB.

Caches have historically used both virtual and physical addresses for both

cache indexes and tags, although using virtual tags is now uncommon. Virtually indexed caches use a portion of the virtual address for their index, which is available earlier than the physical address. If the cache is physically indexed there is no need to consult the TLB to determine which data to feed back to the processor, and so the cache can be very fast. The speed of this recurrence (the load latency) is crucial to the processor's performance, and so most modern L1 caches are virtually indexed, which allows the TLB lookup to proceed in parallel with fetching the data from the cache. But virtual indexing is not the best choice for all cache levels. It introduces the problem of virtual aliases, as multiple virtual addresses can map to a single physical address, which in turn implies that the cache may have multiple locations with a single physical address. The cost of dealing with virtual aliases grows with cache size, and as a result most L2 and larger caches are physically indexed.

2.2.7 Cache and Memory Controllers

To implement the policies discussed in the previous sections (direct-mapped vs full-associative cache; write-through vs write-back cache; write-allocate vs no write-allocate cache; inclusive vs exclusive cache; coherence protocol; replacement policy), a cache controller is used. This controller is the logic that controls the determination of a cache hit or miss and reacts accordingly (i.e., loads a new cache line from main memory and stores it in the correct place on the cache, depending on the cache policies). Typically, caches are implemented using static random access memory (SRAM). Figure 2.5 depicts a simple write-allocate cache controller finite-state machine (FSM).

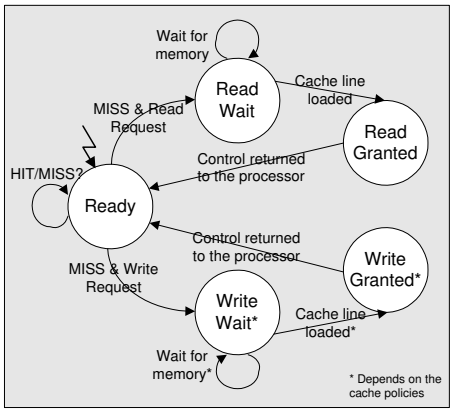


Figure 2.5: A simple write-allocate cache controller

The cache controller and the memory controller have to work in conjunction. A memory controller is then the logic that manages the flow of data going to and from the main memory. Typically, processors have a memory controller implemented within a chipset located on the motherboard. On the other hand, more modern processors have a memory controller on the processor die to reduce the memory latency. While this has the potential to increase the system's performance, it locks the processor to a specific type (or types) of memory, forcing a redesign in order to support newer memory technologies.

Modern computer systems typically implement main memory using different types of dynamic random access memory (DRAM). Therefore, memory controllers connected to this type of memory contain the logic necessary to read and write to the DRAM and to "refresh" the DRAM by sending electrical current through the entire device. Without constant refreshes, DRAM will lose the data written to it as the capacitors leak their charge within a number of milliseconds. Reading and writing to DRAM is facilitated by use of multiplexers and de-multiplexers, by selecting the correct row and column address as the inputs to the multiplexer circuit, where the de-multiplexer on the DRAM can select the correct memory location and return the data (once again passed through a multiplexer to reduce the number of wires necessary to assemble the system).

Several types of DRAM are used in modern computer systems. The synchronous DRAM or SDRAM differs from the standard DRAM in that it does not run asynchronously to the system clock the way DRAM does. SDRAM is tied to the system clock and is designed to be able to read or write from main memory in burst mode (after the initial read or write latency) at one clock cycle per access (zero wait states). SDRAM accomplishes its faster access using a number of internal performance improvements, including internal interleaving, which allows half the module to begin an access while the other half is finishing one. However, with the "processor-memory performance gap" increasing, there was a need for a new standard: Double Data Rate SDRAM or DDR SDRAM. DDR SDRAM is similar in function to regular SDRAM, but doubles the bandwidth of the memory by transferring data twice per cycle - on both the rising and falling edges of the clock signal.

2.3 Memory Copy Operation

Chapter 1 demonstrated that the memory copy operation is involved in several tasks (or processes) performed by the OS. As this operation moves data from a

memory location to another and as accessing the memory is time-consuming, current applications and the OS can spend a considerable amount of time performing this operation. Two common examples, involving two applications and the OS, where the memory copy operation is utilized, are depicted in Figures 2.6 and 2.7.

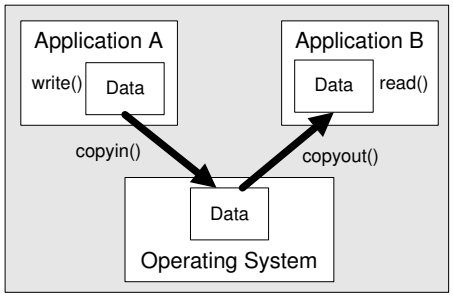


Figure 2.6: Memory copy example in the `pipe`

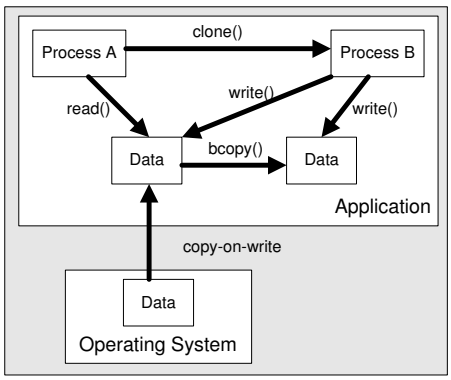


Figure 2.7: Memory copy example in the inter-process communication

In the first example, the copy operation between the OS and two applications utilizes a `pipe`. For security reasons (to avoid that the memory space of both applications being corrupted) the data transfer has to be performed through the OS. Application A starts by preparing some data that it wishes to send to application B. It calls the OS to transfer this information using the `write` system call. The OS performs a copy to transfer the data to the OS memory space using the `copyin` function. Then, application B requests the data using the `read` system call and the data is copied to application B's memory space using the `copyout` function. The two applications are able to

communicate using the OS `pipe` functionality.

A second example is the memory copy performed by the inter-process communication. At some point in the execution time, an application performs a copy of its processes using the `clone` system call. This requires the OS to perform a copy to recreate the process and provide it with all appropriate descriptors used by the parent. In order to minimize the memory utilization, shared structures are not immediately copied, but are marked copy-on-write by the OS. Therefore, when process A reads the data, the shared copy of the data is accessed and no additional memory is required. However, when process B attempts to write the data, the operation cannot proceed as the write may cause an incorrect value to be read by process A at a later time. The memory system, having marked the data as copy-on-write, throws an exception which is handled by the OS. The data is copied (utilizing the `bcopy` function) by the OS to a new location in the application space and process B's descriptors are updated to reflect the change. Execution then returns to process B to complete the write step.

The previous examples showed how the memory copy operation can be utilized inside of the OS and between applications. In the following, the memory copy operation is introduced in detail. There are several implementations of the memory copy operation, the most optimized are hand-written in assembly for the platform that is going to execute such a code. Moreover, the granularity of the copy also varies from implementations and depends on the size of the registers of the processor. This makes this operation platform dependent and difficult to port to another platform. Figure 2.8 presents an example of simple implementation in C of a memory copy function, which performs a copy with a 1 byte granularity (this implementation is the one used in the Linux OS). Other implementations exists that perform copies on words (4 bytes) or double words (8 bytes) granularities. Figure 2.9 presents the assembly code of the C code in Figure 2.8 generated using GCC [84] for an Intel Pentium IV.

The memory copy operation exists in different "flavors": moving data from one memory location to another or coping data between two different memory locations (i.e., keeping the original data). Moreover, there are different implementations for these "flavors": ISO C and FreeBSD. Functions that start by `mem` are aligned with the ISO C standard (IEEE Std 1003.1 [32]), while functions starting with `b` are part of the Berkeley Software Distribution (BSD) implementations [25]. The ISO C standard family of functions are:

- `void *memccpy(void *s1, const void *s2, int c, size_t n);`
  The `memccpy` function copies bytes from memory area `s2` into `s1`, stop-

```
/**
 * Copy one area of memory to another
 * @dst: Where to copy to
 * @src: Where to copy from
 * @size: The size of the area.
**/
void *memcpy(void *dst, const void *src, size_t size)
{
  char *d=(char *)dst;
  char *s=(char *)src;

  while (size--)
    *d++ = *s++;

  return dst;
}
```

Figure 2.8: C implementation of the `memcpy` function, byte granularity

ping after the first occurrence of `c` (converted to an unsigned char) has been copied, or after `n` bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of `c` in `s1`, or a null pointer if `c` was not found in the first `n` bytes of `s2`.

- `void *memchr(const void *s, int c, size_t n);`
  The `memchr` function returns a pointer to the first occurrence of `c` (converted to an unsigned char) in the first `n` bytes (each interpreted as an unsigned char) of memory area `s`, or a null pointer if `c` does not occur.

- `int memcmp(const void *s1, const void *s2, size_t n);`
  The `memcmp` function compares its arguments, looking at the first `n` bytes (each interpreted as an unsigned char), and returns an integer less than, equal to, or greater than `0`, according as `s1` is lexicographically less than, equal to, or greater than `s2` when taken to be unsigned characters.

- `void *memcpy(void *s1, const void *s2, size_t n);`
  The `memcpy` function copies `n` bytes from memory area `s2` to `s1`. It returns `s1`.

- `void *memmove(void *s1, const void *s2, size_t n);`
  The `memmove` function copies `n` bytes from memory areas `s2` to `s1`. Copying between objects that overlap will take place correctly. It returns `s1`.

- `void *memset(void *s, int c, size_t n);`

```
memcpy:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
.L2:
        leal    16(%ebp), %eax
        decl    (%eax)
        cmpl    $-1, 16(%ebp)
        jne     .L4
        jmp     .L3
.L4:
        movl    -4(%ebp), %eax
        movl    %eax, %edx
        movl    -8(%ebp), %eax
        movb    (%eax), %al
        movb    %al, (%edx)
        leal    -8(%ebp), %eax
        incl    (%eax)
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L3:
        movl    8(%ebp), %eax
        leave
        ret
```

Figure 2.9: Intel assembly implementation of the previous C `memcpy` function

The `memset` function sets the first `n` bytes in memory area `s` to the value of `c` (converted to an unsigned char). It returns `s`.

For the BSD family these include:

- `void bcopy(const void *s1, void *s2, size_t n);`
  The `bcopy` function copies `n` bytes from memory area `s1` to `s2`. It returns `s2`. Overlapping addresses are handled correctly.

- `int bcmp(const void *s1, const void *s2, size_t n);`
  The `bcmp` function compares byte string `s1` against byte string `s2`, returning `0` if they are identical, `1` otherwise. Both strings are assumed to be `n` bytes long. The `bcmp` function always returns `0` when `n` is `0`.

- `void bzero(void *s, size_t n);`
  The `bzero` function places `n` null bytes in the string `s`.

2.4 The Xilinx Virtex Family

As presented in Section 2.1, the solutions presented in this dissertation were prototyped using the Xilinx Virtex FPGAs. Prototyping in real hardware provides an estimate of hardware resources utilized and the raw performance of the proposed solution. Therefore, this section introduces the details of the chosen platforms of the Xilinx family.

The FPGA is often used for prototyping purpose due to its reconfigurable nature. Xilinx [101] is one of the major providers of such devices and developed, in particular, the ML310 [56], the XUP [106] and the ML410 [57] platforms that were utilized in this dissertation. In the following, the description of the tool suite as well as the details of these platforms are presented.

Xilinx provides the tool-chain to program and debug the mentioned platforms, mainly the Integrated Software Environment (ISE) [34] and the Embedded Development Kit (EDK) [24]. The ISE controls all aspects of the design flow for a particular design. The EDK is an integrated software solution for designing embedded processing systems. This pre-configured kit includes a tool suite as well as all the documentation and the intellectual propriety cores that can be used for designing Xilinx FPGAs with embedded PowerPC (PPC) processors. Moreover, it also provides the hardware description language (HDL) Xilinx Synthesis Technology (XST), the ModelSim Xilinx Edition III [58] simulation environment and the Joint Test Action Group (JTAG) (IEEE 1149.1 standard) programming interface. These tools are used when working with either Xilinx platforms.

Both the ML310 and the XUP platforms are constituted by one Virtex-II Pro XC2VP30 FPGA with two PPC 405 cores, FPGA fabric, a 512 MB (on the ML310) and 2 GB (on the XUP) DDR SDRAM main memory, a CompactFlash card, a 10/100 Ethernet network interface card and other I/O capabilities. When paired with the ISE and EDK and its catalog of intellectual propriety cores, both the ML310 and XUP can be used to rapidly prototype, verify system designs and software applications using either stand-alone code or targeting an OS. In order to configure the platform, Xilinx also provides the standard JTAG connectivity for debugging and a serial port connection for communication. The schematic of the ML310 and of the XUP platforms are presented in Figure 2.10 and Figure 2.11, respectively.

The ML410 (the schematic of the ML410 platform is presented in Figure 2.12) is an embedded development platform based on the Xilinx Virtex-4 XC4VFX60 FPGA, also hosting two PPC405 processors, FPGA fabric, a

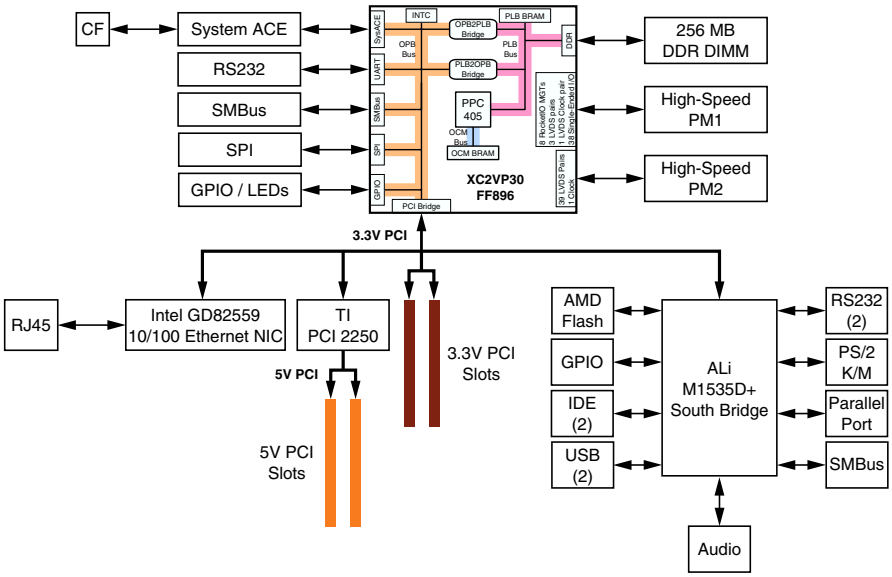


Figure 2.10: Xilinx ML310 schematic [101]

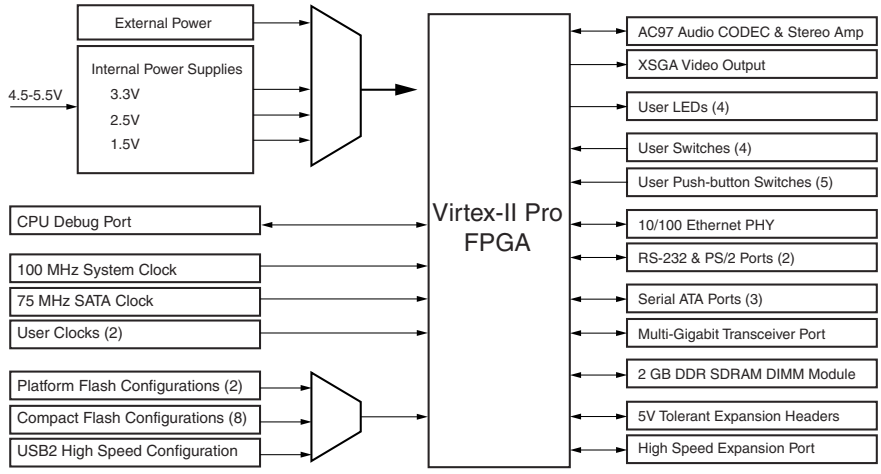


Figure 2.11: Xilinx XUP schematic [101]

256 MB DDR2 SDRAM main memory, a 10/100/1000 Ethernet network interface card and other I/O capabilities. This platform can also be paired with the ISE and EDK in order to easily allow applications to execute either stand-

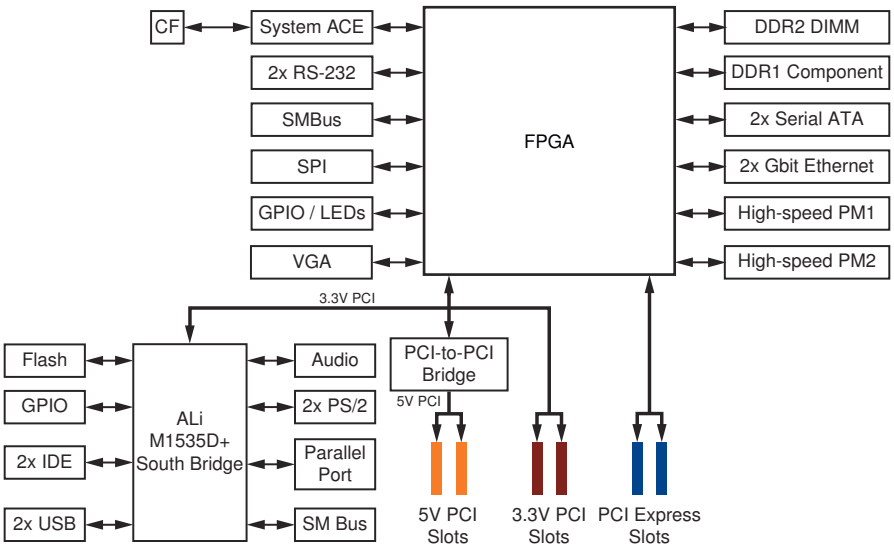


Figure 2.12: Xilinx ML410 schematic [101]

alone or with an OS.

In order to effectively debug the designs utilized in the mentioned platforms, the ModelSim Xilinx Edition III [58] was chosen. It provides a complete HDL simulation environment that verifies the functional and timing models of the design and the hardware description language source code. The HDL utilized in this dissertation is very high speed integrated circuit hardware description language (VHDL) [89], which is commonly used as a design language for FPGA due to its fairly general-purpose. The key advantage of VHDL when used for systems design is that it allows the behavior of the required system to be described (modelled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

Both prototyping platforms have a FPGA, as explained previously. Therefore, in the following it is presented its basic internal organization. A FPGA in general is a reconfigurable device. Like the name states, it is a gate array which is programmable by applying an electric field. Once a FPGA is programmed, the device behaves like real hardware. FPGA of different manufacturers differ in general layout. The Xilinx FPGA mainly consists of input/output blocks (IOB), configurable logic blocks (CLB) and routing resources. Figure 2.13 depicts an abstract view of the Xilinx FPGA internal components. Each IOB can be configured to be an input, output or both (tri-state). A CLB consists of

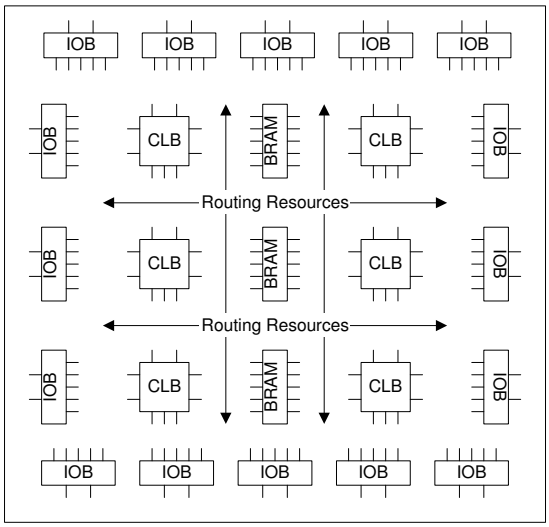


Figure 2.13: Abstract overview of the Xilinx FPGA internal components

four slices, each slice having two memory blocks, which can be used as look-up-table (LUT), random access memory (RAM) memory or read only memory (ROM) memory. Moreover, a slice also contains two sequential blocks which can be configured as a flip-flop or as a latch. The Xilinx FPGA also provide additional block RAM (BRAM) that are true dual port memory blocks, offering fast and discrete access to large blocks of memory. Note that BRAM are situated outside of a CLB and have a large capacity (18 kbits) compared to the memory blocks inside each slice (16 bits). Apart from these common features, the Virtex-II Pro and Virtex-4 have two PPC blocks.

Two important intellectual propriety cores for this dissertation are the single and dual port RAM and the content addressable memory (CAM) cores. Both intellectual propriety cores are generated using LogiCORE [101]. The RAM core (both single and dual port) is generated based on the user-specified width and depth and is composed of single or multiple BRAMs, for both Virtex-II Pro and Virtex-4 FPGAs. Taking advantage of BRAM's true dual port memory block, the implementation of a dual port RAM core [103] can be easily and efficiently performed. Moreover, by just using one port of the BRAMs it is possible to implement a single port RAM core [102]. Figure 2.14 depicts the schematic of a dual port RAM core. The CAM core [104] is also based on user-specified width and depth. The user can select either a LUT used as 16-bit shift register (SRL16) or a BRAM implementation. A CAM

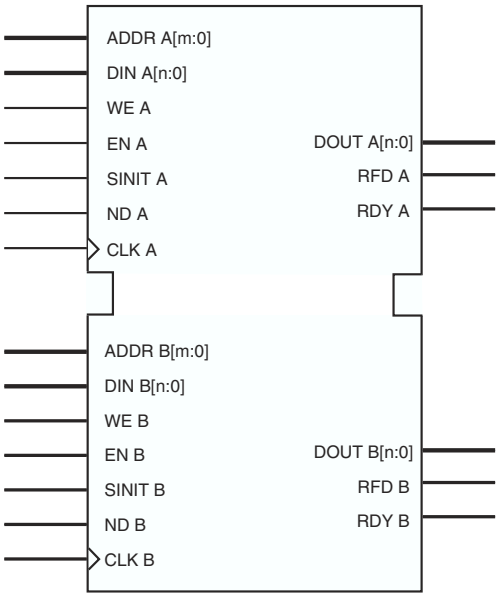


Figure 2.14: Schematic of dual port RAM core [101]

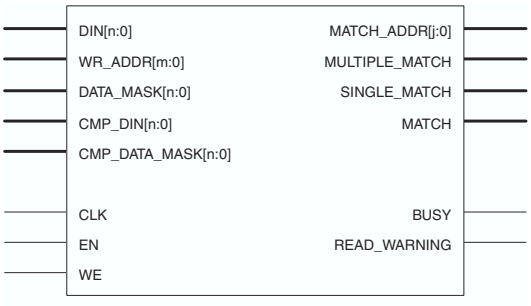


Figure 2.15: Schematic of the CAM core [101]

core implemented with SRL16 primitives has a single clock latency on its read operation and 16 clock cycles latency on its write operation. However, a CAM core implemented with BRAM primitives has a single clock latency on its read operation, and two clock cycles latency on its write operation. Figure 2.15 depicts the schematic of the CAM core.

2.5 Simics Simulator

The solutions presented in this dissertation were also implemented in the Simics simulator. The numbers gather when prototyping in the Xilinx Virtex platforms were afterwards utilized to correctly model the proposed solutions under the Simics simulator.

Simics [48] is a system level instruction-set simulator. Whereas an emulator is focused on executing a program as quickly and accurately as possible, a simulator, in addition, is designed from the ground up to gather information on the execution and in general be a flexible tool. Moreover, Simics is efficient as it is designed to run simulations very fast and still gather a great deal of information during runtime.

Simics is a system level simulator, meaning that it models a target computer at the level that an OS acts. Thus, Simics models the binary interfaces to buses, interrupt controllers, disks, video memory, etc. This means that Simics can run anything that the target system can, i.e., arbitrary workloads. Simics can boot unmodified OS kernels from raw disk dumps.

Simics is an instruction-set simulator, meaning that it models the target system at the level of individual instructions, executing them one at a time. This is the lowest level of the hardware that software has access to. Simulating at this level allows Simics to be system level, yet still permits an efficient design.

Simics fully virtualizes the target computer, allowing simulation of multiprocessor systems as well as a cluster of independent systems, and even networks. The virtualization also allows Simics to be cross platform. The end uses for Simics include program analysis, computer architecture research, and kernel debugging. The analysis support includes code profiling and memory hierarchy simulation (i.e., cache hierarchies). Debugging support includes a wide variety of breakpoint types. The support for system level simulation allows OS code to be developed and analyzed.

In this dissertation, Simics was chosen because it easily allows to create new hardware to be included in the simulated system and it already provides a cache model (a comparative study with other simulators is introduced in Section 4.2). Although Simics does not model any cache system part of the simulated machine (due to speed of simulation), there are cache models available that can be inserted on the simulation. As it uses its own memory system to obtain high speed simulation, including the cache model would slow the simulation down. Therefore, the memory in Simics is always up to date and

the accesses are always atomic. The cache model basically includes the penalties associated with each access, depending on where the data/instructions are located in the memory hierarchy.

2.6 Summary

In this chapter, the general concepts and implementation tradeoffs of caches were introduced. Caches are used to reduce the average access time to main memory and, therefore, try to reduce the "processor-memory performance gap". By carefully choosing the associativity, the write policies, the cache size vs distance to the processor and the replacement policy, the hit rate of the cache can be improved, bringing better performance to the whole system.

Moreover, several memory data movements performed in software were explained in detail. As memory data movements move or copy data from one memory location to another, they are bounded by the access time to main memory and the "processor-memory performance gap". As the processor speeds are increasing faster than the memory access times, the impact of these memory data movements are expected to increase in the future. Therefore, the next chapter introduces the concept and design of the proposed solutions to reduce the impact of memory data movements in a computer system, utilizing the concepts presented on this chapter.

As the proposed solutions were prototyped utilizing the Xilinx tools, the platforms and the intellectual property cores were also explained in this chapter. Moreover, the performance of the proposed solutions were evaluated utilizing Simics full-system simulator, therefore, a high level description of the simulator was also introduced in this chapter.

Chapter 3

Cache-Based Memory Copy Hardware Accelerator

The previous chapters introduced the memory copy operation as a bottleneck in today's uniprocessor and multiprocessor platforms for several applications. Furthermore, the solutions proposed in literature are either platform dependent or involve increased complexity which prevents them to be widely applicable. Consequently, a simple and platform-independent solution needs to be sought after to solve the memory copy bottleneck. Moreover, the presence of caches in processors is common and a trend was identified that for an increasing number of (including embedded) processors incorporate these caches. Therefore, a "smart" solution could take advantage of the presence of caches.

First, this chapter describes the memory copy operation performed in the traditional manner (in Section 3.1). Then, a cache-based solution is derived from the previous observations to improve the performance of the memory copies (in Section 3.2). Subsequently, Section 3.3 presents the design of the indexing table able to support copies of cache line granularity connected to a simple direct-mapped cache. Moreover, an indexing table able to handle word granularity copy and able to be connected to a set-associative cache is also presented. When discussing the indexing table operation it becomes clear that there is a need for a load/store unit, which is also described in this section. Section 3.4 presents a general implementation discussion and, finally, Section 3.5 summarizes this chapter.

3.1 Observations

Chapter 1 demonstrated that the approaches presented in related works do not provide a general or efficient enough solution to the memory copies bottleneck. The DMA-based approaches provide only limited solutions due to the high overhead introduced to explicitly initialize the devices. Software techniques are either OS dependent, or not valid for all cases. Furthermore, today's network cards commonly offloads the checksum, which removes the one part of per-byte overhead (the other part is the memory copy operation). They also coalesce interrupts to reduce per-packet costs. Thus, today copying costs account for a relatively larger part of processor utilization than previously, and therefore relatively more benefit is to be gained in reducing them. Moreover, from the cache studies presented in Chapter 1, there is cache behavior that can be taken advantage of. The fact that the payload presents spacial locality and that almost all the incoming (received) data is subsequently read by the processor. Therefore, in this chapter a solution is derived from the observations of the traditional way to perform a memory copy in software.

The memory copy operation performed in the traditional way in software involves the utilization of many loads and stores by the processor itself to accomplish the operation. As an example, consider the C code presented in Figure 2.8 for the `memcpy` function, where the copy is performed in "pieces" of one byte. However, the demand for performance (by using bigger "pieces" which reduces the number of loads and stores) and the need for flexibility (using smaller "pieces" in order to perform copies of small sizes) is a difficult tradeoff. Moreover, due to the copy being performed in "pieces", the number of loads and stores involved in a memory copy operation is high. Furthermore, the number of instructions executed depends not only on the granularity of the copy (size of the "piece") but also on the size of the original (to-be-copied) data[1]. It is worth mentioning that there has been extensive research on how to optimize this operation, as presented in Section 1.3.

The previously mentioned loads and stores are executed on processors that nowadays have caches attached (as introduced in Chapter 2). Therefore, the execution of the memory copy in the presence of caches in software results in any of the following unwanted scenarios:

- Large number of instructions: As the data is copied "piece-by-piece", the number of instructions needed to load a "piece" to the processor, per-

[1]The term "original data" is referring to the data that needs to be copied whilst the term "copied data" refers to the data located at the destination after the memory copy operation.

form the copy and store the result back in the cache, keeps the processor tied up with the execution of this operation.

- Eviction of original and copied data from the cache: The interleaving of loading and storing of data can lead to situations in which: 1) the original data is being overwritten by the copied data; or 2) the copied data is being overwritten by the original data being loaded. In case 1), the need of loading again the original data, incurs in longer delays. In case 2), the probability of the copied data being needed again after the memory copy operation is rather high (otherwise, why would a memory copy be needed?). Consequently, this would mean that after the memory copy operation has finished, the copied data may have to be loaded again into the cache.

- Eviction of data not involved in the memory copy operation from the cache: The loading of the original data and the storing of the copied data can cause the data previously stored in the cache (not involved in the memory copy operation) to be evicted due to the possible mapping to the same cache lines. If this occurs for data needed again after performing the memory copy operation, it has a large detrimental effect since later on this data must be read again from the main memory.

- Data duplication in the cache: The cache contains both the original data and the copied data. This would mean that the data (content-wise) is present twice in the cache.

Moreover, it is up to the programmer to ensure that, if the source and destination addresses of a memory copy operation overlap, a particular interface is used (using the ISO C standard family of functions as an example, if the addresses do not overlap a `memcpy` function can be utilized, otherwise the `memmove` function should be used). Without making this assessment prior to the execution of the memory copy operation, bytes copied to an intended destination address that overlaps the source address can overwrite and corrupt the original data. A simple illustration of how this might occur is given in Figure 3.1. As can be seen, the destination address begins at address 104, which also happens to be an address within the source address. These data overwrites can be avoided in a couple of ways: i) determine the overlap and transfer first those bytes in the source address that overlap with the desired destination address; or ii) determine the overlap and transfer bytes from the source address to the destination address in reverse order. Note that an overlap in the opposite direction (copying a source address to a destination address lower in main

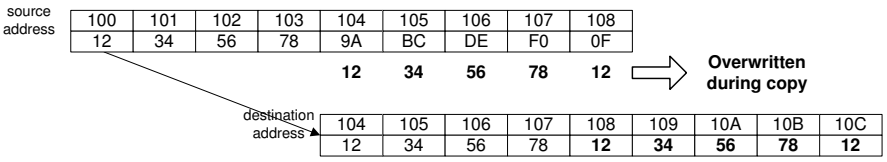


Figure 3.1: Address overlapping on a memory copy operation

memory) poses no problem for the standard memory copy operation that transfers data in ascending address order. However, the first case suffers additional overhead in order to calculate, store, and pass different source, destination, and size variables for multiple memory copy operations, whereas the second solution must do this only once. Still, the overhead involved in the determination of the overlapping and deciding on which software interface to use can never be avoided.

Even though these situations do not pertain the actual memory copy operation, they are the direct result of it. Therefore, the following section describes the cache-based memory copy hardware accelerator that reduces the impact of the mentioned scenarios. In order to make the explanation easier to follow, it is firstly assumed that the original data is already present in the cache. In the following sections, the practical implementation is presented, removing this initial assumption.

3.2 The Concept

As caches store the most recently used data (as explained in Section 2) the data to-be-copied by a memory copy should be already present within the cache. Therefore, by initially assuming that the original data is already present in the cache (in order to make the explanation easier to follow), the memory copy operation can then be performed by redirecting the destination addresses to the cache locations containing the original data. Consequently, without losing generality and without delving into the many details, a memory copy operation is reduced to a single action of entering the destination address into an indexing table and creating a pointer to the original data stored in the cache. When the copied data needs to be accessed, this can be easily performed by extracting the corresponding pointer from the indexing table and subsequently accessing the cache using this pointer. An illustration of this method is depicted in Figure 3.2. The advantages of the cache-based memory copy hardware accelerator

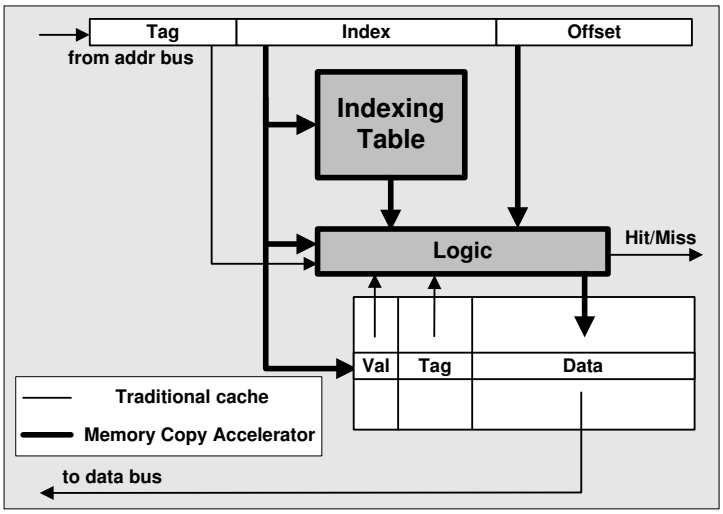


Figure 3.2: The cache-based memory copy hardware accelerator

is two-fold:

- The memory copy operation is performed in a much shorter time, due to reduction of the numerous amount of loads and stores. In turn, the latencies associated with its execution is greatly reduced.
- The ensuing access(es) to all copied data can be performed much faster, because it simply extracts the corresponding pointer from the indexing table.
- The earlier described cache pollution does not exist as it is not possible for the copied data to overwrite cache locations containing the original data or vice-versa.
- The memory copy operation performed by the cache-based memory copy hardware accelerator can efficiently deal with overlapping of the original and copied data memory locations, that would take a software solution many more cycles.

As the indexing table is accessed in parallel with the cache access, there will be hits/misses as data is searched both in the indexing table and in the cache. Table 3.1 summarizes these different cases. As an indexing table entry is a pointer to the original data in the cache, even if the source and destination addresses overlap, there will be no impact on the copy performed using the

cache-based memory copy hardware accelerator. In reality, the overlapping of the source and destination addresses correspond to case 1) of Table 3.1, where the original data is in the cache and the copy exists in the indexing table. Moreover, the storing of the copied data to the main memory is deferred to a later time in the cache-based memory copy hardware accelerator. In particular, when either the original or copied data locations are being overwritten or when the original data location is evicted from the cache, this must be detected and the appropriate measures must be taken to store the copied data to the main memory (introduced in detail in Section 3.3.2). Furthermore, the pointer in the indexing table must be invalidated. This justifies why case 2) of Table 3.1 cannot occur.

Case	Indexing Table	Cache	Comments
1	hit	hit	Original data in cache and copy in the indexing table
2	hit	miss	Cannot occur
3	miss	hit	Original data in cache and no copy in the indexing table
4	miss	miss	No original data in cache and no copy in the indexing table

Table 3.1: Hit/Miss combination in the cache and indexing table

3.3 The Design

The previous section presented the ideal concept of the cache-based memory copy hardware accelerator. However, a practical implementation needs to consider the cache organization details and its impact on the accelerator concept.

In the cache-based memory copy hardware accelerator, the indexing table is tightly-coupled with a cache. The most simple cache organization is a direct-mapped cache, as introduced in Chapter 2. Direct-mapped caches are extremely popular in embedded systems due to their low power per access [108]. Therefore, the first design of the cache-based memory copy hardware accelerator connects the indexing table to this particular cache organization (presented in Section 3.3.1). Moreover, the first design also considers

that the original data is always present in the cache (a custom load/store unit to handle the necessary loads and stores is presented in Section 3.3.2).

In the cache-based memory copy hardware accelerator, each entry of the indexing table points to the cache line that contains the original data. Therefore, for the first design, a cache line granularity to perform the memory copy operation was chosen. This size provides a good tradeoff between performance and hardware resources utilized, as it will be shown later. Moreover, this granularity can cover all memory copy operation scenarios (including the less ideal ones). These scenarios entail those memory copies that transfer data smaller then a cache line, which sizes are not a multiple of a cache line size or those that are not cache line aligned:

1. The case of a copy being smaller then a cache line size can be solved by using a software implementation of the memory copy operation without utilizing the cache-based memory copy hardware accelerator. Due to the small size of the copy, the penalty incurred by performing the copy in software is limited (considering an assembly hand-written optimized copy algorithm for the chosen platform).

2. The case where the source and the destination addresses are miss aligned by the same amount can be detected and dealt with. Detecting such a case can be easily performed by looking at the offsets of the source and destination addresses. The solution is to perform the copies of complete cache lines utilizing the cache-based memory copy hardware accelerator and the remainder of the words again using a software implementation. In this way, the penalty incurred remains small compared with the gains of using the cache-based memory copy hardware accelerator (as shown later).

3. In the case where the source and destination addresses are miss aligned (not by the same amount), the indexing table supporting cache line granularity copy can no longer be directly applied. Therefore, alternatives that range from simple padding (by either an intelligent programmer or a smart compiler) to rewriting part of the application code, can be utilized.

The latter case requires a deep knowledge of the application being executed and, therefore, it is not the most efficient option. The amount of application profiling, in order to determine each memory copy source and destination addresses, and re-writing the application to ensure the alignment of those addresses makes this solution time-consuming. Moreover, padding the addresses

in order to align them may fragment the memory and waste this valuable resource. As the indexing table able to support cache line granularity copy does not efficiently cover all spectra of the possible copy operation scenarios, an indexing table design able to support word granularity copy is also introduced in this chapter (in Section 3.3.3).

As presented in Chapter 2, there is a trend towards set-associative caches due to their reduced miss rate. Therefore, this chapter presents the necessary changes to the indexing table to support set-associative caches in Section 3.3.4.

3.3.1 Indexing Table Supporting Cache Line Granularity Copy

The memory copy operation performs a copy of size `size` from a source address `src` to a destination address `dst`. The way a memory copy is performed using the indexing table connected to a cache is to access the indexing table with the index part of the `dst` address and to write the index part of the `src` address, the tag part of the `dst` address and a valid bit in the entry accessed. The execution of a memory copy of cache line granularity, then becomes the filling of the indexing table entry. The time needed to perform this filling is dependent on the implementation of the indexing table and is addressed in Chapter 4.

If there is a read hit in the indexing table (calculated based on the tag part of the address requested by the processor, the tag part of the `dst` address stored in the indexing table and the valid bit also stored in the indexing table), the index part of the `src` address stored in the indexing table (i.e, the pointer to the cache entry) is provided to the cache. It is worth mentioning that, if there is a miss on the indexing table, there will be no penalty in the performance of the system, as the indexing table and the cache are accessed in parallel. This implies that on a miss in the indexing table, the cache is already being accessed and returns the data in the same amount of time. On a read hit on the indexing table, however, the accelerator requires one more clock cycle in order to retrieve the correct address from the indexing table to provide it to the cache. Figure 3.3 depicts the indexing table design for cache line granularity copy. Therefore, the following fields constitute the indexing table:

- The “Val Bit” field stores the validity of an indexing table entry;
- The “Tag DST” field stores the tag part of the `dst` address;
- The “Index SRC” field stores the index part of the `src` address;

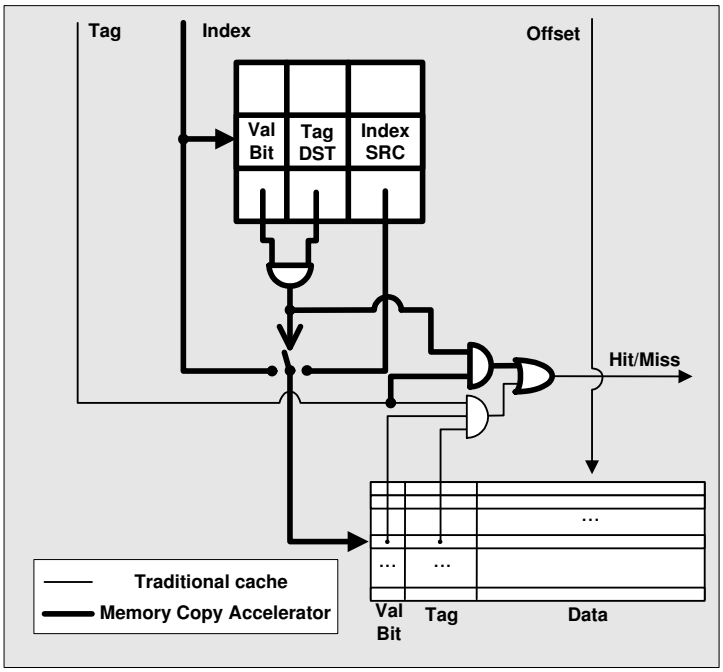


Figure 3.3: The indexing table design for a cache line granularity copy

When filling the indexing table (the index part of the destination address is used to access the indexing table), the following actions are performed:

- Load the original data to the cache, if not present yet (using the custom load/store unit presented in Section 3.3.2). If the cache is a write-back cache the data evicted from the cache also needs to be written to the main memory.
- If the cache has any data from the destination addresses and the original and destination addresses do not overlap, these cache lines need to be invalidated, in order to keep consistency between the cache and the indexing table. However, if the original and destination addresses do overlap, then there is no need to invalidate the common cache lines, as it would invalidate original data from the cache (that afterwards needs to be loaded again to perform the memory copy operation). If the cache is a write-back cache, when invalidating the cache lines, the data needs also to be written back.

On a read request (the index part of the address requested by the processor

is used to access the indexing table), the following actions are performed:

- If the valid bit is set and if the tag part of the address requested by the processor is the same as the tag stored on the table (a read hit on the indexing table which implies also a read hit on the cache), use the address provided by the indexing table to access the cache;
- If the valid bit is not set and/or if the tag part of the address requested by the processor is not the same as the tag stored on the indexing table (a read miss on the indexing table), use the address requested by the processor to access the cache. If there is also a read miss on the cache, more steps need to be performed, which are discussed in the next section.

For a write request (the indexing table is accessed using the index part of the address provided by the processor), the following actions are performed:

- If the valid bit is set and if the tag part of the address provided by the processor is the same as the tag stored on the entry of the indexing table (a write hit in the indexing table):
  1. Use the address provided by the indexing table to access the cache;
  2. Perform the storing of the copied data (the details on how to perform the actual copy is introduced in the next section);
  3. Invalidate the indexing table entry;
  4. Load the requested data to the cache and change its value (this step is not necessary if the cache is a write-allocate cache).
- If the valid bit is not set and/or if the tag part of the address provided by the processor is not the same as the tag stored on the entry of the indexing table (a write miss in the indexing table):
  1. Use the address provided by the processor to access the cache and the main memory;
  2. Write data to the cache and the main memory;
  3. If there also is a write hit on the cache, more steps need to be performed, which are discussed in the next section.

As mentioned before, when either the `src` or `dst` addresses are being overwritten or when one of the `src` addresses is evicted from the cache, this must be detected and the appropriate measures must be taken to store the copied data to the main memory. This is performed by the custom load/store unit introduced in the next section.

3.3.2 Load/Store Unit

The previously described mechanism assumed that the direct-mapped cache already contains the necessary to-be-copied data. As this is not always true, there is a need to load/store data to/from the main memory from/to the cache. Therefore, a load/store unit was developed. It is situated between the cache and the main memory and communicates control signals to direct the behavior of the cache, the main memory, and the indexing table. The load/store unit is mainly an FSM with two different paths for the read and write operations. Furthermore, certain states were specifically introduced to handle the situations particular to the presence of the indexing table. Figure 3.4 depicts the FSM of the load/store unit.

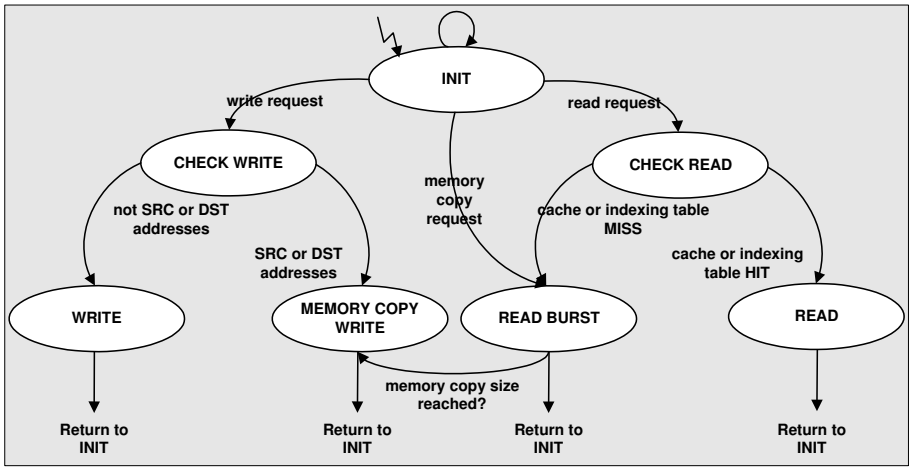


Figure 3.4: Load/store unit finite-state machine

On a read operation, if the address requested by the processor exists in cache and/or in the indexing table (a read hit on the cache and/or in the indexing table) the data is immediately provided to the processor (this is performed by the READ state of the FSM). If there is a read miss on the cache (which implies a miss on the indexing table too), the new cache line being brought to the cache may evict a `src` address from the cache (bringing the new cache line to the cache is performed by the READ_BURST state of the FSM). If so, the load/store unit has to perform the storing of the copied data to main memory (performed by the MEMORY_COPY_WRITE state of the FSM). This implies accessing the indexing table to find the `dst` address that corresponds to the evicted `src` data, write this `src` data to the `dst` address in main memory and

invalidate the indexing table entry.

If a memory copy is being executed, the load/store unit is responsible for loading the `src` data from the main memory from size zero (because all data for the memory copy is already in cache) up to the `size` of the memory copy (this is performed by the READ_BURST state of the FSM), and write-back any previous data that these loads may evict from the cache, if it is a write-back cache (performed by the MEMORY_COPY_WRITE state of the FSM).

On a write operation three different situations can occur: a write to a `src` address; a write to a `dst` address; and a write to a normal address (i.e., addresses that are not `src` or `dst` addresses). For the last case, the load/store unit is responsible for the regular storing of data to the main memory, as a traditional load/store unit would perform (this is performed by the WRITE state of the FSM). If the write address is a `src` address, the indexing table is looked up to find the corresponding `dst` address. The `src` data is written-back to the main memory to the `dst` address and the correspondent entry of the indexing table is invalidated (performed by the MEMORY_COPY_WRITE state of the FSM). After these steps, the systems behaves as a standard write hit (as the data to be modified/written to is already in the cache). If the write address is a `dst` address, the load/store unit, besides performing the same steps as executed for a write to a `src` address, additionally has to load to the cache the copied data (that was just written in the main memory), if the cache is not a write-allocate cache, in order to be modified (this is performed by the READ_BURST state of the FSM).

It is also worth mentioning that the load/store unit benefits from the functionality of nowadays memories, that allow bursts of data to be read. The load/store unit generates addresses until a limit is reached (limit being either the size of a memory copy or a cache line) which implies a delay to access the first word equal to the read latency of the main memory and the next requested word is provided every clock cycle. Another advantage is that the `size` of a memory copy is known in advance which enables the possibility of requesting all the necessary data and only paying once the initial main memory read latency.

The combination of the cache (independent of its organization), the indexing table (independent of copy granularity it supports) and the load/store unit constitute what is referred in this dissertation as the cache-based memory copy hardware accelerator.

3.3.3 Indexing Table Supporting Word Granularity Copy

As mentioned in Section 3.2, the cache line granularity copy can be less efficient in certain cases (in particular when the source and destination addresses are miss aligned not by the same amount). Therefore, an indexing table able to support word granularity copy is presented, once it supports such cases.

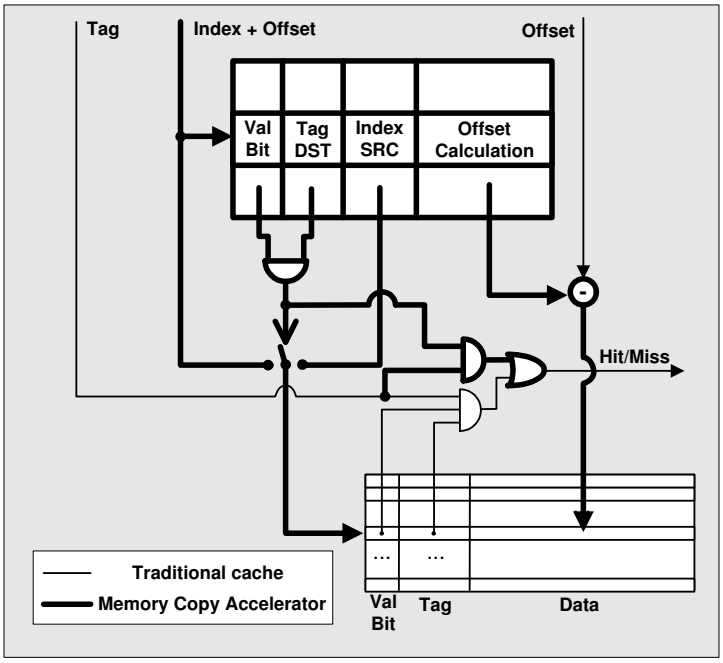


Figure 3.5: The indexing table design for word granularity copy

The index part of the `src` address stored in the indexing table can, in reality, refer to a previous or subsequent cache line depending on the word requested. Therefore, a new field of the indexing table (referred to as "Offset Calculation" field in Figure 3.5), is introduced. This field stores the difference between the offsets of the `dst` and `src` addresses in the indexing table, which allows to calculate the correct address to access the cache. Besides, an entry of the indexing table is now referring to one word instead of one cache line. Therefore, in order to access the correct word, it is needed the offset part of the address to access the indexing table. Summarizing, the indexing table now has to have as many entries as the number of words stored in the cache.

The processor requests an address constituted by a `req tag`, a `req index`

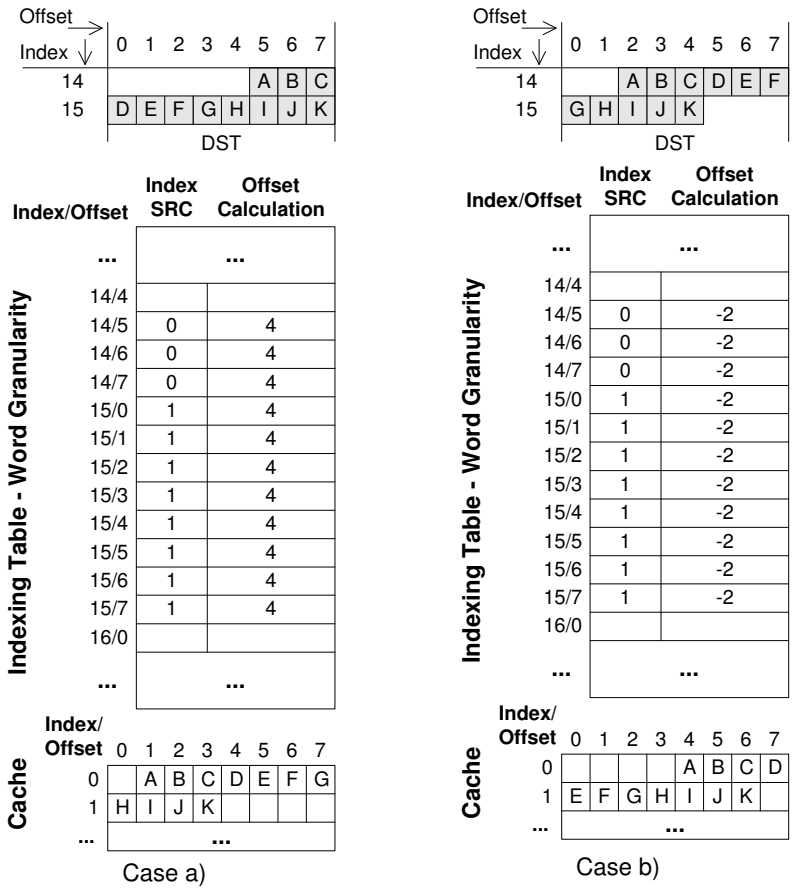


Figure 3.6: Examples to demonstrated the offset calculation

and `req offset`. If the requested address corresponds to a copied value, the indexing table provides the index part of the `src` address, stored in the "Index SRC" field, and the difference between the offsets of the `dst` and `src` addresses, stored in the "Offset Calculation" field. The algorithm to provide the correct address to the cache, i.e., the `index cache` and `offset cache`, is dependent on the cache line size (in the algorithm presented below, a cache line refers to `n` words[2]):

```
cal = req offset - Offset Calculation;
```

[2]For the reminder of this dissertation the default value of `n`, i.e., a cache line size is 8 words.

```
if cal > n - 1 then
    index cache = Index SRC + 1;
    offset cache = cal - n;
elsif cal < 0 then
    index cache = Index SRC - 1;
    offset cache = cal + n;
else
    index cache = Index SRC;
    offset cache = cal;
end if;
```

A couple of situations need also to be considered. In the case the calculation rolls over (i.e., if it crosses the tag boundary), there will be no implication on the system, as only the index part of the address is used and the tag part of the address is not used to access the cache. In this case, index 0 is accessed as it refers to the subsequent cache line where the requested word is (this is the intended behavior). Another situation is that this solution requires one more clock cycle in the critical path (i.e., providing the data to the processor) compared with the indexing table that supports cache line granularity copy. This is due to an extra addition to calculate the word being accessed, however this extra time can be hidden on the cache access time (depending on the cache design). Moreover, the expected benefits of the solution (i.e., being able to perform memory copies with word granularity much faster then the software approach) should cover this extra penalty.

Consider four examples to illustrate the algorithm where the cache line size is `8` words. The first two examples are based on Case a) from Figure 3.6 ("Offset Calculation": offset `dst` - offset `src = 5 - 1 = 4`), while the remainder two examples are base on Case b) of the same Figure ("Offset Calculation": offset `dst` - offset `src = 2 - 4 = -2`).

Example 1: The processor requests `req index 15`, `req offset 2` (this means the copy of the word `F`). From Case a) of Figure 3.6, the "Index SRC" field of the indexing table will return `1` and the "Offset Calculation" field returns `4`. This means the address provided to the cache is: `index cache = 0` and `offset cache = 6`, corresponding to word `F`, as expected.

```
cal = 2 - 4 = -2 < 0 =>
=> index cache = 1 - 1 = 0;
=> offset cache = -2 + 8 = 6;
```

Example 2: The processor requests `req index 14`, `req offset 6` (this means the copy of the word `B`). From Case a) of Figure 3.6, the "Index SRC"

field of the indexing table will return `0` and the "Offset Calculation" field returns `4`. This means the address provided to the cache is: `index cache = 0` and `offset cache = 2`, corresponding to word `B`, as expected.

```
cal = 6 - 4 = 2 =>
=> index cache = 0;
=> offset cache = 2;
```

Example 3: The processor requests `req index 14`, `req offset 7` (this means the copy of the word `F`). From Case b) of Figure 3.6, the "Index SRC" field of the indexing table will return `0` and the "Offset Calculation" field returns `-2`. This means the address provided to the cache is: `index cache = 1` and `offset cache = 1`, corresponding to word `F`, as expected.

```
cal = 7 - (-2) = 9 > 7 =>
=> index cache = 0 + 1 = 1;
=> offset cache = 9 - 8 = 1;
```

Example 4: The processor requests `req index 15`, `req offset 2` (this means the copy of the word `I`). From Case b) of Figure 3.6, the "Index SRC" field of the indexing table will return `1` and the "Offset Calculation" field returns `-2`. This means the address provided to the cache is: `index cache = 1` and `offset cache = 4`, corresponding to word `I`, as expected.

```
cal = 2 - (-2) = 4 =>
=> index cache = 1;
=> offset cache = 4;
```

With this simple algorithm and using also the offset part of the address provided by the processor, the problem of alignment can be solved and copies of word granularity can be performed.

3.3.4 Indexing Table Supporting Set-Associative Caches

Until now the indexing table was connected to a direct-mapped cache. On the other hand, this type of caches have increasingly been replaced by set-associative caches, due to their performance improvement. However, when including the cache associativity, the available number of cache entries decreases. As each entry of the indexing table points to a cache line[3], the number

[3]In order to also support word granularity copy with set-associative caches, the "Offset Calculation" field can also be included in the indexing table. In this dissertation, however, the

of entries of the indexing table have to be the same as the cache entries. Moreover, as the size of the copy depends on the number of entries of the indexing table and on the cache line size, the maximum size of the copy is then reduced. Therefore, an indexing table able to support associativeness is a requirement.

A solution is to use more bits to access the indexing table. Instead of just using the index part of the address requested/provided by the processor, new bits are included, which number depends on the number of ways of the cache the indexing table is connecting to. This technique will create more entries in the indexing table (a new entry per way), i.e., each index can refer to as many entries as many ways the cache has. When filling the indexing table, the entry corresponding only to the index part of address (the new bits are zero), is used first, and only if that entry is taken than the following entry is used. This has a performance impact on performing a copy as the time to fill the indexing table is now dependent on the number of ways the cache has. However, the bigger size of the copy (i.e., more entries of the indexing table are used) the more time the processor would spend executing a memory copy in software. Therefore, the increase in time to fill the indexing table is still much smaller then the software version.

As there is an increase in the number of entries of the indexing table to check on a read or write request (as there is an entry per way), performing a sequential search is not the most efficient way. A solution is to use a CAM core for the "Tag DST" field of each index (i.e., each index refers to as many entries as many ways the cache has) and a RAM core for the other fields of the indexing table (a description of the CAM and RAM cores was presented in Section 2.4). This means that there will be as many CAMs as the indexes, each CAM with as many entries as the ways of the cache. By searching this field, the first step to determine if it is a hit in the indexing table can be performed efficiently in one clock cycle. The content of the "Tag DST" field, accessed based on the index part of the address provided/requested by the processor, is afterwards (second step to determine a read hit on the indexing table) feed in parallel to the "Val Bit" and the "Tag + Index SRC" fields to determine the index of the `src` in the cache. A read hit in the indexing table will then take two clock latency (one clock cycle for the CAM of the "Tag DST" field plus one clock cycle for both "Val Bit" and "Tag + Index SRC" fields). Again, as the indexing table and the cache are accessed in parallel there is no penalty on a read miss on the indexing table. The indexing table connected to a 4-way associative cache is depicted in Figure 3.7.

indexing table connected to a set-associative cache is only able to support cache line granularity copy.

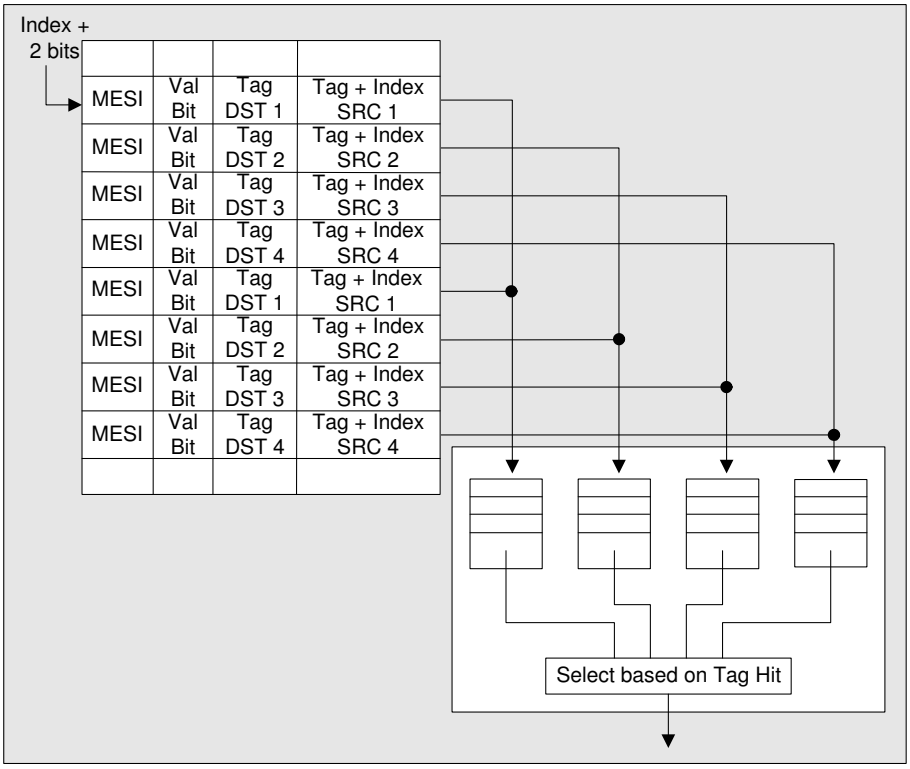


Figure 3.7: The indexing table design for a 4-way associative cache

As introduced in Section 2.2.4, a cache coherence protocol keeps information on the state of each cache line. These states are depending on the presence of the cache line in only one or more than one caches in a multiprocessor system and on the content of the cache line being different or not from the one in the main memory. Therefore, a cache supporting a cache coherence protocol includes some bits per cache line to store its state. The number of bits stored depends on the type of protocol implemented. As introduced in Section 2.2.4, the MESI cache coherence protocol is widely utilized. The MESI protocol supports four states: Modified, Exclusive, Shared and Invalid, which implies the need of two bits to encode each state.

As the indexing table stores a pointer to the original data in the cache, it effectively extends the size of the cache by including entries with the copied data. These entries should also hold a state, as the system supports cache coherence protocol. Therefore, in order to support the MESI cache coherence protocol, the indexing table also needs to include a field where the state of

each cache line is kept. That is referred as “MESI” field in Figure 3.7 and whenever there is an access to the “Val Bit” and “Tag + Index SRC” fields also the “MESI” field is accessed to determine/update the state of the entry.

3.4 Communication Issues and Cost Estimative

In this section, general implementation issues in designing the cache-based memory copy hardware accelerator are discussed. One of the first issues to address is the communication between the accelerator and the processor. Mainly two options are available from the hardware point of view: i) bank of registers to transfer the input parameters; and ii) assigning a particular address to each input parameter. The first option is the traditional way to implement the communication between the accelerator and the processor. The last option is possible only if the addresses used to transfer the parameters are reserved (the program cannot use these reserved addresses except to communicate with the accelerator, i.e., any load or store to any reserved address will result in accessing the accelerator) and therefore is not the most common one. From the software point of view there are also two options to transfer the input parameters to the accelerator: i) a memory-mapped device-driver; and ii) an instruction-set architecture extension. The first option utilizes a particular file of the OS that contains the memory mapping of addresses. Any write or read to a particular location of this file will result in accessing the accelerator. To use this option there is also a need to implement either a polling or interrupt mechanism to start and end the execution of the accelerator and resume the execution of the program. The instruction-set architecture extension can implement a single instruction to substitute the software memory copy calls. Therefore, the use if this new instruction will transfer the necessary parameters to the accelerator and start the execution of the memory copy operation. It is worth mentioning that any combination of software and hardware options is possible.

The traditional way to implement the communication interface in the accelerator is through the register bank. Assigning addresses to each parameter to transfer might result in reserving too many addresses for the communication (depending on the number of parameters to pass). Moreover, reserving addresses can only be performed if the programmer has complete control of the application including the addresses generated by the compiler (in order to ensure that the reserved addresses are not used except to communicate with the accelerator). As it will be shown later, the option used in prototyping the cache-based memory copy hardware accelerator presented in this dissertation

is the later one. As the software utilized was hand-written and completely under control of the programmer, there was no need to implement a register bank to transfer parameters. For the software, the traditional communication mechanism is the use of a memory-mapped device-driver. This option can be easily implemented and does not require access to the processor hardware, as the case of the instruction-set architecture extension. However, as it will be shown later, the penalty of using such a communication mechanism is not negligible. Therefore, when utilizing a simulator to demonstrate the performance benefits of the cache-based memory copy hardware accelerator, the instruction-set architecture extension was used (as it can be easily implemented in a simulator).

Another issue to consider is the hardware cost of the cache-based memory copy hardware accelerator (for each cache organization and indexing table type). Moreover, the address bus size used to connect the cache and the indexing table also have impact on the hardware costs. As presented previously, the indexing table stores the tag and index parts of the addresses used in the memory copy. The number of bits stored have impact on the size of the memories used to implement the indexing table. The bigger the number of bits stored the more hardware resources are necessary.

In order to estimate the size of the indexing table for different caches organizations, a detailed study was performed. Assuming a cache size of 1 MB (a typical size for caches nowadays) and varying the cache line size and associativity, the hardware resources are estimated for both the indexing table and the cache. Figure 3.8 depicts the increase (in percentage) of the hardware resources used in the indexing table compared with the ones used only in the cache. Assuming now a direct-mapped cache with a cache line size of 32 bytes and varying the cache size and the address bus size, again the hardware resources are estimated for both the indexing table and the cache. Figure 3.9 depicts the increase (in percentage) of the hardware resources used in the indexing table compared with the ones used only in the cache.

The main conclusions reached from a careful analysis of Figures 3.8 and 3.9 are:

- The increase in the cache line size reduces the percentage of hardware resources utilized on the indexing table compared with the cache (from 20% down to 2.5%). This is expected as one cache line now contains more data, so one entry of the indexing table points to more data.

- The increase on associativity does not have a big impact on the percentage of hardware resources utilized on the indexing table compared with

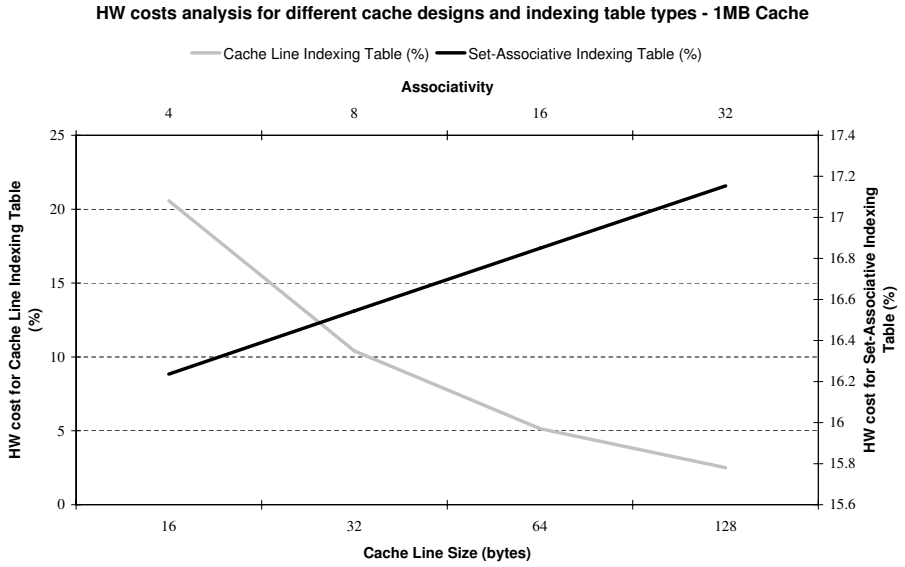


Figure 3.8: Hardware costs for different cache designs and indexing table types

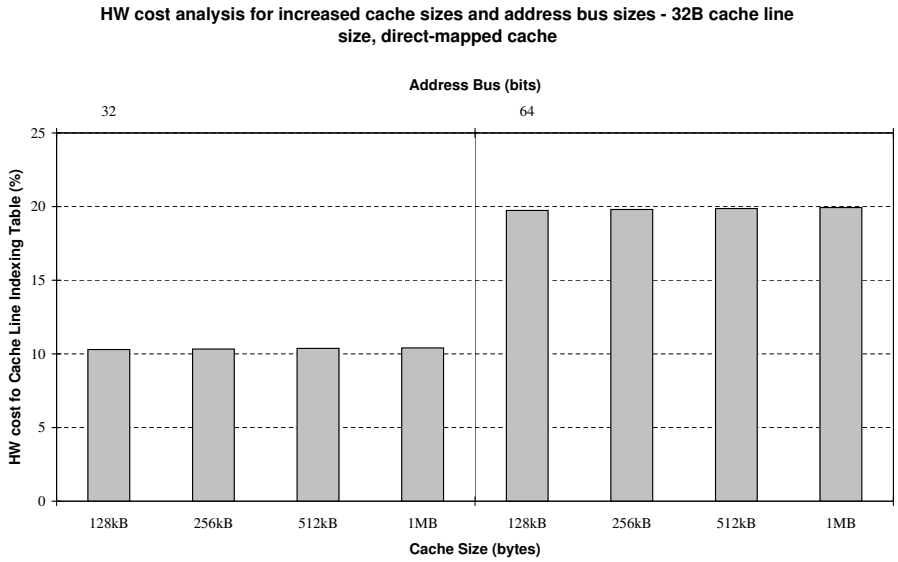


Figure 3.9: Hardware costs for different cache sizes and address bus sizes

a cache (values around 17%). The reason for this is that in order to support associativity, the cache itself increases the number of bits stored in

this way reducing the impact of the bigger number of bits also stored in the indexing table.

- The increase of the cache size does not have a big impact on the percentage of hardware resources utilized on the indexing table compared with a cache (values around 10%). This is expected as the number of bits stored increase both in the indexing table and the cache.
- Finally, the analysis of increasing the address bus size demonstrates that the size of the bus does increase the percentage of hardware resources utilized on the indexing table compared with a cache (from 10% to 20%). The reason for this is the increase on the number of bits stored (in particular due to the size of the tag, that typically increases when the address bus size is increased).

The increase in the percentage of hardware resources utilized on the indexing table compared with a cache for the different situations is mitigated due to the effective cache size increases for a program executing a large number of copies. Moreover, due to the effective increase of the cache size, it is also expected that the hit rate of the cache also increases.

3.5 Summary

This chapter presented the concept of the cache-based memory copy hardware accelerator, as a combination of a cache (independent of its organization), an indexing table (independent of the copy granularity it supports) and a load/store unit. Moreover, the design of the indexing table able to support cache line and word granularity copies and set-associative caches was discussed. The cache-based memory copy hardware accelerator does not incur in performance penalty on a read hit on the cache and read miss on the indexing table. However, on a read hit on the indexing table there is a need of one more clock cycle to access the copied data. Furthermore, the options to communicate with the accelerator (from the software and hardware points of view) were presented as well as a detailed study was performed to estimate the necessary hardware resources need to implement the cache-based memory copy hardware accelerator.

The next chapter introduces the methods used to demonstrate the benefits of the cache-based memory copy hardware accelerator. The details of platforms used to execute the synthetic benchmarks and take performance measurements are introduced.

Chapter 4

Uniprocessor Platform

The cache-based memory copy hardware accelerator introduced in the previous chapter is implemented in real hardware using reconfigurable technologies, i.e., a FPGA, and incorporated into a simulator running several benchmarks to determine the performance gains of the proposal.

This chapter introduces the platforms utilized to demonstrate the cache-based memory copy hardware accelerator. Section 4.1 describes the prototyping platform and the implementation tradeoffs for the cache-based memory copy hardware accelerator. Section 4.2 introduces the simulator platform and presents the simulation parameters. Finally, Section 4.3 summarizes the main points presented in this chapter.

4.1 Prototyping Platform

The concepts presented in Chapter 3 were implemented on hardware. Two options for digital systems implementation can be found: application specific integrated circuit (ASIC) or FPGA. The design flow of an ASIC device involves a wide variety of complex tasks, including placement and physical optimization, clock tree synthesis, signal integrity analysis, and routing using different tools suite. When compared to ASIC devices, the design flow of FPGA devices is very simple and is accomplished, by the majority of the FPGA providers, with a single software tool. Therefore, potentially it is possible to reduce the complexity of the design process as well as significantly reduce cost. Figure 4.1 presents a comparison of the design flow of both technologies.

Therefore, the FPGA was chosen as digital support for prototyping the

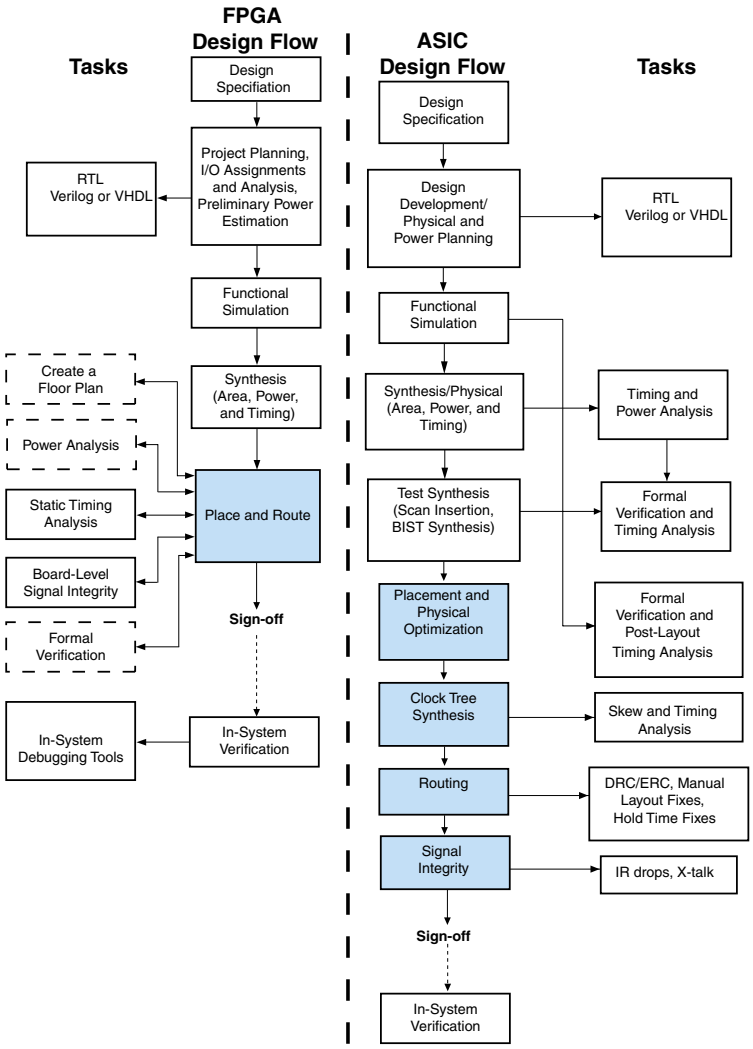


Figure 4.1: ASIC vs FPGA design flow [3]

cache-based memory copy hardware accelerator. Moreover, the FPGA tools suite does provide a simulation environment that enables the verification of the functional and timing models of the design, as introduced in Section 2.4. This simulation environment provides an estimative of area utilization as well as timing, an important part in analyzing the performance of a design. As the focus of this dissertation is on performance analysis, the simulation environment of the digital support chosen, in this case FPGA, is enough. The

data provided by this environment is sufficient to provide a estimative of the cache-based memory copy hardware accelerator raw performance. This data is utilized afterward to perform a system performance analysis using the simulator presented in Section 2.5.

As introduced in Section 2.4, Xilinx [101] is one of the major providers of FPGA devices. Moreover, the Virtex-II Pro FPGA was within the most widely deployed devices from Xilinx in the beginning of this work. By that time, Virtex-4 FPGA had just became available and, therefore, the tools suite to support the design in these devices were not mature enough. The first implementation of the cache-based memory copy hardware accelerator was then performed in a Virtex-II Pro. However, during the course of the work, some limitation of the device chosen became clear (as introduced in the next section) that justified moving to a Virtex-4 FPGA. During this time, the tools suite had time to mature and were able to provide the necessary performance and ease of use required for the work performed in this dissertation.

4.1.1 Virtex-II Pro Platform

The cache-based memory copy hardware accelerator is implemented on the simulation environment targeting a XUP [106] platform, containing a Virtex-II Pro XC2VP30 FPGA with two PPC405 cores, although only one is used (the details of the platform was introduced in Section 2.4).

The PPC located within the Virtex-II Pro FPGA contains two interfaces that are capable of accessing memory: the on-chip memory (OCM) interface and the processor local bus (PLB) interface. These interfaces have different architectures, timings, and protocols, which affect their relative performance. The OCM is a dedicated interface between the PPC and BRAMs in the FPGA. Some key features of this interface are:

- It provides fast access to a fixed amount of instruction and data memory space;
- It provides an instruction-side OCM (ISOCM), 64-bit data bus and a data-side OCM (DSOCM), 32-bit data bus;
- It provides independent clock inputs for the ISOCM and DSOCM and for the PLB.

The PLB is the main processor bus; some key features of this bus are:

- The instruction cache unit (ICU) and data cache unit (DCU) masters provide the interface between the processor and the PLB;
- The ICU/DCU masters attach to the PLB through separate address, read data, and write data buses with a plurality of transfer qualifier signals;
- The data buses are 64 bits wide and the address buses are 32 bits wide;
- It is capable of doing an 8-word cache line transfer;
- The PLB arbiter controls the access to the PLB slave devices attached to the bus.

Figures 4.2 illustrates how the processor interfaces with the OCM and the PLB buses.

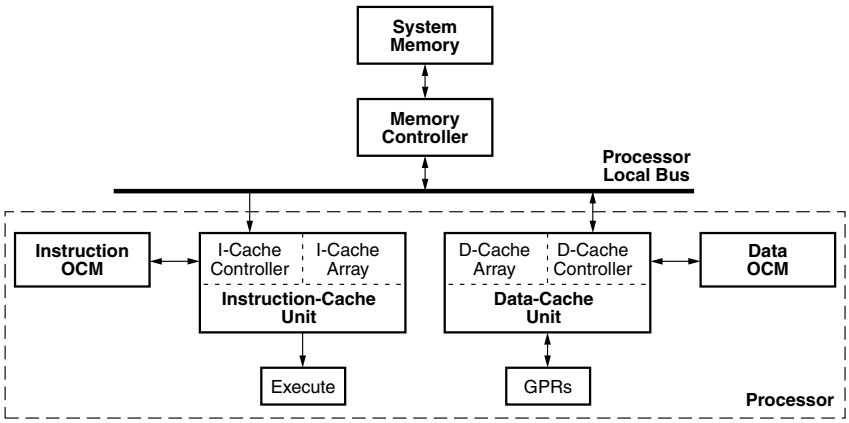


Figure 4.2: The PPC memory system [101]

The PLB has a very large address space compared to the OCM; altogether, the PLB can address 4 GB of memory. The DSOCM and ISOCM interfaces can physically address up to 16 MB of BRAM, however, the amount of BRAM available for OCM is limited by the number of BRAMs in the FPGA being used.

The PLB operating frequency is dependent on the maximum operating frequency of the PLB arbiter and the number of peripherals that are connected to it. The PLB is restricted to operate at an integer ratio (1 to 16) of the processor frequency. In the same way, the OCM frequency is dependent on the amount of memory that is connected to it (typically BRAM). Additionally, the routing

between the OCM controllers and the BRAM uses general FPGA routing resources. Therefore, the larger the memory attached to the interface, the slower the interface may run. The OCM's operating frequency, like that of the PLB, must be in integer ratio (1 to 3) to the processor frequency. The DSOCM, ISOCM, and PLB are all independent of each other and can operate at different frequencies.

The PLB is a decoupled bus, meaning that its address, read data, and write data buses are not coupled to one another. Therefore, an address cycle can overlap with data cycles, and a read cycle can overlap with a write cycle. The PLB can do this because all masters have their own address, read data, write data, and transfer qualifier signals. Bus slaves also have address, read data, and write data buses, but these buses must be shared. In contrast to the PLB, the OCM interface is a coupled bus. Like the PLB, there are separate read and write data buses for the ISOCM and DSOCM, but each address cycle is immediately followed by (coupled to) a corresponding data transfer. Because the OCM controllers are dedicated interfaces, decoupling the buses would not increase their bandwidth.

The PLB is a shared bus and can support eight masters and eight slaves, in the Virtex-II Pro FPGA. The PPC has two masters on the PLB: the DCU and the ICU. All devices connected to the PLB must share the bandwidth that is available on the bus. Obviously, other masters on the bus can interfere when the processor would like to access data or instructions. The OCM, however, has two dedicated memory interfaces: the Data-Side and Instruction-Side. Therefore, the processor never has to wait for data or instructions because another device is accessing them.

The fact that the PLB must share its bandwidth with many masters and slaves makes it a nondeterministic bus. This means that the timing is variable. For instance, if the processor is executing a function in which the data/instructions are not already in the cache, the time that function takes to execute is dependent on how much traffic is on the PLB. Because the OCM is a dedicated interface, it is a deterministic bus. A given set of instructions being fetched from ISOCM will always take the same amount of time, assuming the instructions do not require data that is stored in main memory. Table 4.1 summarizes the main characteristics of both the OCM and PLB.

In general, the OCM and the PLB are by nature very different bus interfaces, both having its advantages and disadvantages. Furthermore, the typical OCM utilization is very similar to the PPC cache, i.e., they both help offload traffic from the PLB. It has similar performance to the PPC cache when both

OCM	PLB
Dedicated Interface	Shared Bus
Deterministic Bus	Nondeterministic Bus
Coupled Bus	Decoupled Bus
Memory Size: 16 MB	Memory Size: 4 GB

Table 4.1: OCM vs PLB comparison

are operating at the same frequency. As the OCM has an access time equivalent to a standard cache access [100], the DSOCM is chosen to connect the cache-based memory copy hardware accelerator (i.e., the data cache, the indexing table and the load/store unit) to the processor. As only the data side is implemented (the cache-based memory copy hardware accelerator concerns only the data part), BRAMs are utilized on the PLB to store the instructions part of a program. A linker script is used to assign the data to the BRAMs memory on the OCM and the instructions to the PLB BRAMs memory. Figure 4.3 depicts the described system.

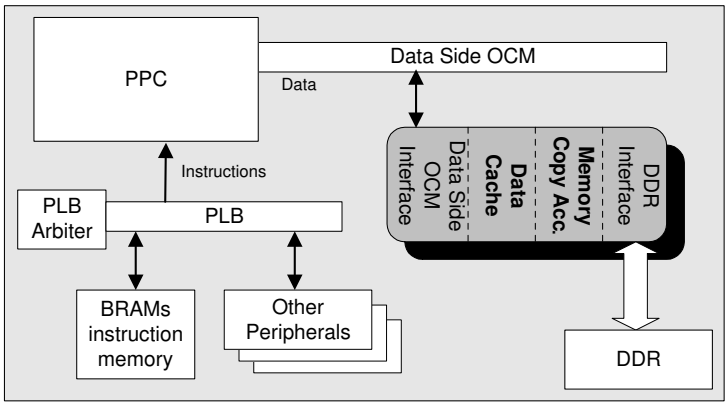


Figure 4.3: System used to prototype

The interface that the DSOCM provides [105] has 22 bits address bus (bits 0 to 7 and 30 to 31 are reserved by the processor), 32 bits of separate read and write data bus and 4 bits of byte write bus. The byte write bus is used to identify, on a write, how many bytes are being transferred. Figure 4.4 depicts the DSOCM interface to the processor and to peripherals. The `DSCNTLVALUE`,

the DSARCVALUE and the TIEDSOCMDCRADDR are control registers.

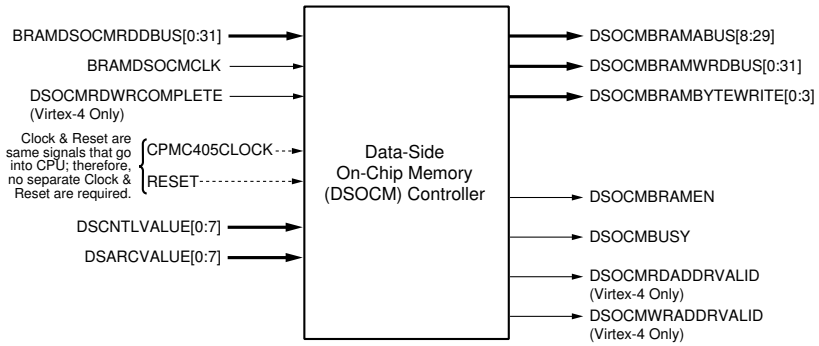


Figure 4.4: DSOCM controller interfaces [101]

The DSOCM bus of Virtex-II Pro is expected to provide the data that the PPC requested in two clock cycles[1]. The cache-based memory copy hardware accelerator requires a non fixed amount of time (most probably bigger than two clock cycles) to perform a memory copy. The solution implemented in this dissertation is to disable the clock of the PPC (keeping the clock of all other parts of the system running) while the accelerator is executing the memory copy. Using this approach, the PPC will stop its execution, giving the necessary time for the accelerator to finish the memory copy operation. As the DSOCM allows a idle time of two clock cycles, the first clock cycle is used to determine if the PPC is requesting a memory copy operation or not. If it is, then the clock of the PPC is disabled, and the memory copy is executed in the accelerator. When the accelerator is performing the copy of the last cache line (which is known as the size of the copy is known in advance), the clock of the PPC is enabled again, in order to take advantage of the second clock cycle to stabilize the PPC. Although this is not the best practice, it proved to be stable enough to perform the necessary verification of the cache-based memory copy hardware accelerator implemented in the Virtex-II Pro FPGA.

As mentioned before, the indexing table works in conjunction with a cache. Therefore, the PPC internal caches (both the instruction and data caches of 16 kB, two-way associative) are disabled and a direct-mapped cache is implemented. This is necessary because the cache-based memory copy hardware

[1]The previous chapter introduced an indexing table able to support word granularity copies. This solution incurred in an extra clock cycle due to an addition in the critical path to calculate the word being accessed. However, as stated, this extra clock cycle can be hidden under the 2 clock cycle delay imposed by the PPC.

accelerator needs to have access and control over the cache. The cache implemented is a 32 kB direct-mapped, write-through, write allocate cache with 32 bytes cache lines (this is one of the simplest implementation, as explained in Section 2.2.4). This implies that, from the 22 bits of the address (provided by the DSOCM), 10 bits are for the index (to address 1024 entries of one word each), 3 bits for the offset (to access one word per entry) and the remainder 9 bits are for the tag part of the address. The cache implementation is depicted in Figure 4.5 and is constituted mainly by the BRAMs available in the FPGA fabric to store the tag, the valid bit and the data (respectively in the tag-array, the valid-array and the data-arrays, as introduced in Section 2.2.1). As each data-array stores one word in each entry, to implement a 32 kB cache, eight of such data-arrays are required. This implies that the processor can only access one word and not one byte in this cache implementation.

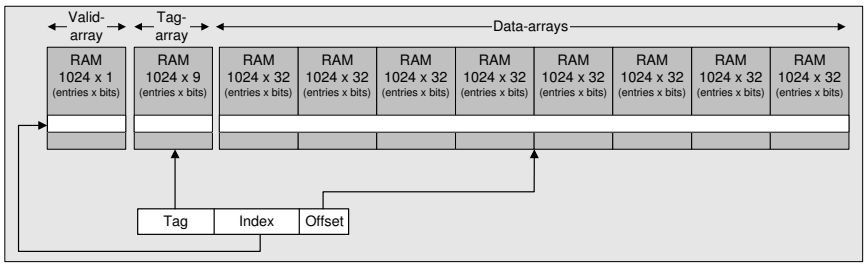


Figure 4.5: Cache implementation

The tool utilized to implement each cache array, i.e, each RAM core, is LogiCORE (as introduced in Section 2.4). The number of BRAMs required depends on the value of the data width utilized. The number of slices required depends on the way the memory depth is constructed which in turn depends on the data width and the implementation of any necessary decoding and multiplexing. For some memory depths, extra logic is required to decode the address and multiplex the outputs. CLBs can be used to provide this functionality. Moreover, the use of BRAMs as implemented by LogiCORE allows the possibility to initialize the contents of the tag-array, the valid-array and the data-arrays. Therefore, the assumption that the data is present in the cache at the moment a memory copy operation is executed, can be easily implemented on the Virtex-II Pro FPGA. Therefore, the limitations of this implementation are:

- The DSOCM bus on the Virtex-II Pro does not provide a signal to sign the completion of an operation, therefore, the clock of the PPC is dis-

abled during execution of a copy operation;

- The data to-be-copied is assumed to be present in the cache at the time of executing a memory copy operation; therefore, the contents of the implemented cache are initialized according to the code to be executed.

The cache and the indexing table able to support cache line granularity copy are implemented in VHDL and synthesized for the Virtex-II Pro XC2VP30 FPGA. The ModelSim XE-III [58] simulation environment is utilized to verify the HDL source code and the functional and timing models of the designs. Table 4.2 presents the estimation of the percentage of resources needed.

	Total Available	**Cache**	**Indexing Table (CL)**
Slices	13696	46 (<1%)	421 (3%)
Flip-Flops	27392	39 (<1%)	482 (1%)
LUTs	27392	79 (<1%)	597 (2%)
IOBs	556	74 (13%)	96 (17%)
BRAMs	136	18 (13%)	28 (20%)
Gclk	16	1 (6%)	5 (31%)

Table 4.2: Resource estimation on the Virtex-II Pro XC2VP30 FPGA

4.1.2 Virtex-4 Platform

In order to solve the previous limitations, a second implementation of the cache-based memory copy hardware accelerator is performed in the simulation environment targeting the ML410 platform, containing a Virtex-4 XC4VFX60 FPGA, with two PPC405 cores, although only one is used (the details of the platform were introduced in Section 2.4).

The ML410 platform, as it is based on a Virtex-4 FPGA, has a different interface between the PPC and the DSOCM. This difference is reflected in two signals provided by the interface to sign the completion of an operation (the signals, `DSOCMRDWRCOMPLETE` and `DSOCMRDADDRVALID`, are depicted in Figure 4.4). These signals are necessary in order to allow the cache-based memory copy hardware accelerator to start and stop executing a memory copy operation, as the accelerator requires a non fixed amount of time to perform

the operation (the time needed depends on the `size` of the memory copy). Utilizing the completion signals, the approach used in the Virtex-II Pro FPGA to allow for variable execution time of the peripheral connected on the OCM, i.e., disable the PPC clock while executing a memory copy operation is not used anymore, making the design in a Virtex-4 FPGA more stable than the one implemented in the Virtex-II Pro.

Besides the differences due to the use of the completion signals in the Virtex-4 FPGA, the design of the cache itself is also changed when implemented in the Virtex-4 FPGA compared with the implementation on the Virtex-II Pro FPGA. The main difference between the cache implemented in the Virtex-4 and a standard one (or the one implemented in the Virtex-II Pro FPGA depicted in Figure 4.5) is the use of dual-ported memory in the cache directory (i.e., in the valid-array and in the tag-array), which allows for checking if data is in the cache, while loading data to a different address (performed by the load/store unit), when executing a memory copy.

The load/store unit is based on the design of [20] and [22]. The first [20] design moved the main memory on a Virtex-II Pro from the PLB to the OCM bus, performing the necessary signal translation between both buses. The work of [22] further extended the design to the Virtex-4 FPGA. The load/store unit utilizes the data interface to the main memory on the Virtex-4 FPGA and includes the necessary FSM extensions to support a memory copy operation.

The cache, the indexing table able to support cache line granularity copy and the load/store unit were implemented in VHDL and synthesized for the Virtex-4 XC4VFX60 FPGA. The ModelSim XE-III [58] simulation environment was utilized to verify the HDL source code and the functional and timing models of the designs. Table 4.3 presents the estimation of the percentage of resources needed.

4.1.3 Cache-Based Memory Copy Hardware Accelerator Implementation in Xilinx

In the previous subsections, the differences in the design of the cache-based memory copy hardware accelerator due to the platforms constrains (i.e., Virtex-II Pro or Virtex-4) were presented. However, the common design tradeoffs (i.e., those not dependent on the being implemented on the Virtex-II Pro or Virtex-4 but depending on using Xilinx FPGA as prototyping platform) are introduced in this subsection.

The PPC is running at 100 MHz in all experiments presented in this dis-

	Total Available	**Cache**	**Indexing Table (CL)**
Slices	25280	301 (1%)	5475 (21%)
Flip-Flops	50560	68 (<1%)	1294 (2%)
LUTs	50560	531 (1%)	7987 (15%)
IOBs	352	129 (36%)	129 (36%)
BRAMs	232	18 (7%)	120 (51%)
Gclk	32	1 (3%)	5 (15%)

Table 4.3: Resource estimation of the Virtex-4 XC4VFX60 FPGA

sertation, however the PPC is able to run at 300 MHz. The reason behind choosing a frequency lower then the maximum available is due to the use of a cache on the OCM. In a standard implementation of a processor and its cache, both are running at the same frequency. In the prototyping platform, the cache implemented is connected to the OCM, as introduced in the previous sections, becoming an external cache running at the same frequency as the OCM, the bus it is connected to. Moreover, the frequency achievable by the OCM is dependent on the amount of BRAMs connected to it (as explained in the Section 4.1.1, the bigger the amount of BRAMs connected to the OCM the slower the bus can run). As the cache and the indexing table are implemented on this bus, it can not achieve the 300 MHz of the PPC. Therefore, in order to have the PPC, the cache and the indexing table on the OCM, running at the same frequency the value of 100 MHz was chosen.

In order to perform a memory copy, there is a need to pass the parameters `src`, `dst` and `size` to the accelerator. A possible way is to reserve particular addresses to pass the parameters and enable the accelerator to recognize these addresses (other options were presented in Section 3.4). The data on the data buses has the values of each parameter and each reserved address in the address buses enables a buffer to store the value of the parameter. Therefore, these values can be utilized after, if needed be. Subsequently, to perform the parameters communication between the software and the accelerator, the memory copy calls of any program are substituted by[2]:

[2]The reserved addresses in this implementation are 0xA1FFFFF4 to pass the `size` parameter, 0xA1FFFFF8 to pass the `src` parameter, 0xA1FFFFFC to pass the `dst` parameter and 0xA1FFFFF0 to start the execution of the accelerator.

```
size = (int *)0xA1FFFFF4;
*size = // bytes to copy
src = (int *)0xA1FFFFF8;
*src = // src address
dst = (int *)0xA1FFFFFC;
*dst = // dst address
start = (int *)0xA1FFFFF0;
*start = 0x1; // start the accelerator
```

For a direct-mapped cache, the number of cache lines in the cache and the number of entries of the indexing table have to be the same (as each entry in the indexing table points to a cache line). Besides, the indexing table needs to be looked up for the cases when a `src` address is being evicted from the cache or when a `src` or `dst` addresses are being overwritten. This can be accomplished by using "brute force approach" of performing a sequential search in the indexing table. However, such an approach would result in a delay dependent on the size of the indexing table and, consequently, not realistic. Therefore, the "Lookup" field is introduced in the indexing table (in Figure 4.6, Figure 4.7 and Figure 4.8).

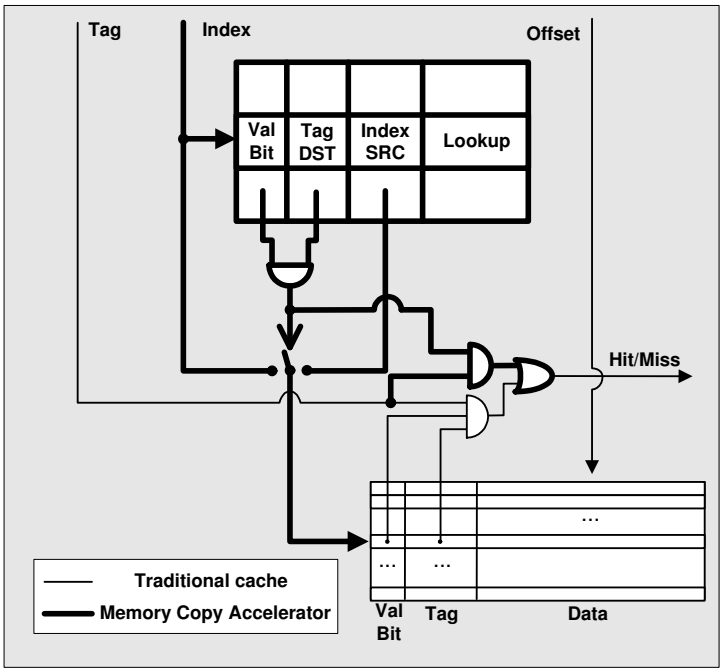


Figure 4.6: The indexing table implementation for a cache line granularity copy

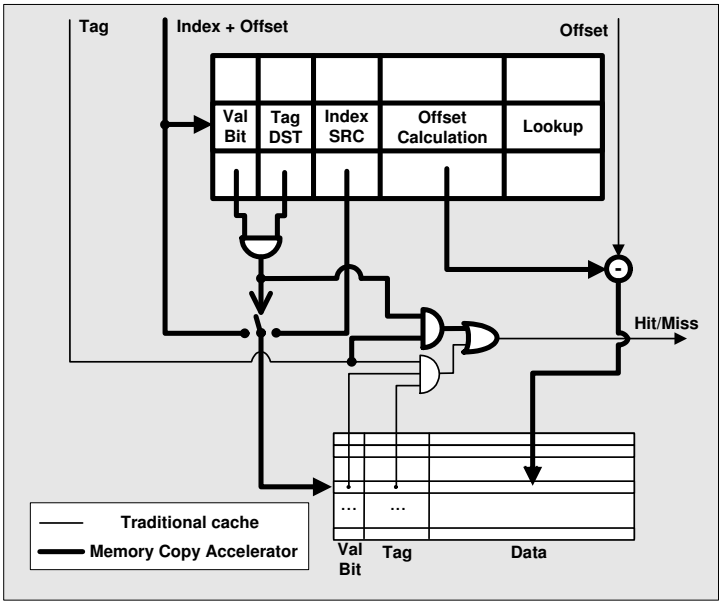


Figure 4.7: The indexing table implementation for word granularity copy

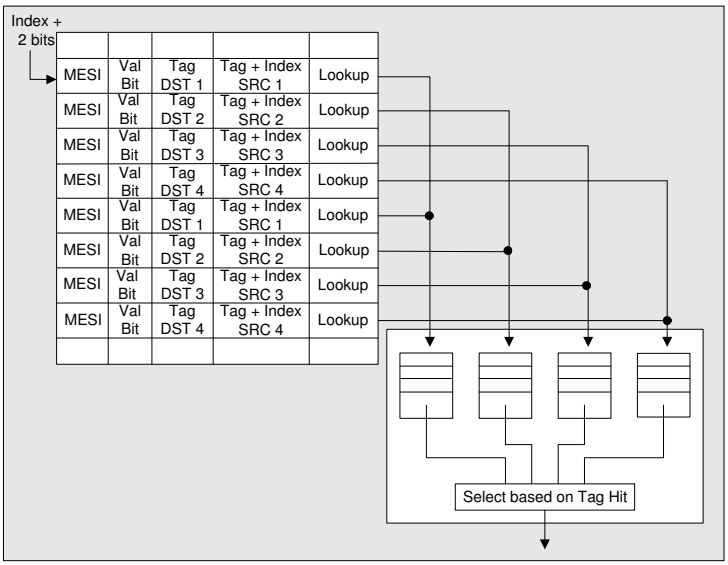


Figure 4.8: The indexing table implementation for a 4-way associative cache

The "Lookup" field is actually constituted by a CAM core (introduced in Section 2.4). As the performance of the CAM can deeply impact the performance of the cache-based memory copy hardware accelerator, the implementation using BRAM is chosen as it provides the smallest number of cycles latency on a write operation. The CAM implementation chosen has a single clock latency on its read operation, and two clock cycles latency on its write operation. As the CAM, containing the index part of the `src` address, is on the critical path of the cache-based memory copy hardware accelerator, the performance of a memory copy operation is bounded to the write time of this memory, i.e., two clock cycles. Therefore, a lookup in the indexing table for the mentioned cases behaves like the following:

1. On a standard read miss in the cache, a new cache line is loaded. Due to possible mapping to the same cache line, this may evict a `src` address from the cache. In order to identify such a case, the index part of the address requested by the processor (which corresponds to the new cache line being loaded) is used to search the "Lookup" field of the indexing table. If this address is present in the "Lookup" field, the CAM returns the entry that has a match (which corresponds to the index part of the `dst` address). This is then used to access the "Val Bit", the "Tag DST" and the "Index SRC" fields of the indexing table. The output of the "Val Bit" field is used to determine if this is a valid entry. If it is, this means that the cache line being loaded will evict a `src` address from the cache and that the storing of the copied data needs to be performed. The output of the "Index SRC" field provides the corresponded entry of the cache to be copied (the index part of the `src` address). The index part of the `dst` address (given by the entry of the CAM) combined with the output of the "Tag DST" field constitute the complete `dst` address (there is no need for the offset part of the address because a complete cache line is being loading). This address plus the data read from the cache is written to the main memory by the load/store unit and the entry of the indexing table is invalidated (i.e., the "Val Bit" field is unset).

2. On a write to a `src` address, the index part of the address provided by the processor is used to search the "Lookup" field of the indexing table to determine if the `src` address is present. If it is, the CAM returns the entry that has a match (which corresponds to the index part of the `dst` address). This is then used to access the "Val Bit", the "Tag DST" and the "Index SRC" fields of the indexing table. The output of the "Val Bit" field is used to determine if this is a valid entry. If it is, this means

that it is a write to a `src` address and that the storing of the copied data needs to be performed. The output of the "Index SRC" field provides the corresponded entry of the cache to be copied (the index part of the `src` address). The index part of the `dst` address (provided by the CAM) combined with the output of the "Tag DST" field constitute the complete `dst` address (there is no need for the offset part of the address because a complete cache line is being written). This address plus the data read from the cache is written to the main memory by the load/store unit and the entry of the indexing table is invalidated (i.e., the "Val Bit" field is unset).

3. On a write to a `dst` address, the index part of the address provided by the processor is used to access the indexing table, and determine if the "Val Bit" field is set and if the "Tag DST" is the same as the tag provided by the processor. If this situation happens, there is a write to a copied value and, therefore, the storing of the copied data needs to be performed. The index part of address provided by the processor is then used to directly access the "Index SRC" field of the indexing table. The output of the "Index SRC" field provides the corresponded entry of the cache to be copied (the index part of the `src` address). The address provided by the processor, which is the `dst` address, and the data read from the cache is written to the main memory by the load/store unit and the entry of the indexing table is invalidated (i.e., the "Val Bit" field is unset).

It is important to mention that case 1) is performed for every cache read miss. Moreover, cases 2) and 3) are performed in parallel (as there is a need to determine if the write is to a `src` address, a `dst` address or neither). As case 2) first accesses the "Lookup" field and case 3) first accesses the "Val Bit" and the "Tag DST" fields, cases 2) and 3) can be performed at the same time, in order to determine which type of writes is being requested (i.e., the `src` address, the `dst` address or neither) and not impose additional delay.

A comparison between the number of entries and the number of bits stored for each entry (for the cache, the indexing tables able to support cache line and word granularity copy and set-associative caches) is also presented. The direct-mapped cache requires 10 bits per entry (for 1024 entries) on the cache directory plus 32 bytes (256 bits) to store the data. For the indexing table able to support cache line granularity copy, 20 bits are stored for each entry (for the same 1024 entries). As the indexing table supporting word granularity copy is accessed using the index plus the offset parts of the address (10 bits for the index part plus 3 bits for the offset part), it accesses 8192 entries and

stores 29 bits per entry. Finally, for the support of set-associative caches (in particular for a 4-way associative cache), the indexing table is accessed using 12 bits (4096 entries), it requires a new entry per way, and stores 31 bits.

Comparing the number of bits needed for the cache with the indexing tables supporting cache line and word granularity copies, and 4-way associative cache, there is an increase in the number of bits used in the indexing table. For the indexing table supporting cache line granularity copy this increase is only 7.5% and 12% for the 4-way associative cache. However, for the indexing table supporting word granularity copy the increase in hardware resources is bigger, in this particular case of 84%. On the other hand, by using the indexing table, the effective size of the cache is increased, as both `src` and `dst` are always accessible. These results confirm the initial expectations presented in Section 3.4.

4.2 Simulation Platform

As introduced in Section 2.2.6, in order to provide a bigger addressing space than the physical memory available, nowadays processors (and the PPC included) benefit from the presence of the TLB. This hardware enables the virtual-to-physical address translation needed to provide the user with a virtual memory.

The OS developed for the XUP and ML410 platforms (the MontaVista Linux 2.4.22 [60]) requires the use of the TLB present on the PPC. However, only the PLB bus is able to access the TLB. As the OCM bus does not have access to it, there is no support of virtual-to-physical address translation on this bus, as reported by [22]. Therefore, the available OS for the platforms cannot be used as is. One can find other implementations of the Linux OS that do not require access to the TLB, however using one of these would require porting such implementation to the platform, which is a tedious operation. Another option, would be to migrate the cache-based memory copy hardware accelerator to the PLB, which would have impact on the performance of the cache-based memory copy hardware accelerator, due to the handshaking protocol on this bus. Therefore, in order to perform a complete system analysis (including the Linux OS) of the previously described accelerator, a simulator is used. Moreover, as presented in the beginning of this chapter, the simulation environment of the prototyping platforms provides the necessary raw information (area and timing estimates) to, afterwards, be feed into the instruction-set simulator. Therefore, in order to identify the simulator that can provide the

necessary features to evaluate the cache-based memory copy hardware accelerator, a comparative study between the most used simulators is performed.

The IBM full-system simulator [8], internally referred to as “Mambo”, is an execution-driven simulator, that facilitates the experimentation and evaluation of a wide variety of system components for PPC 970. The “Mambo” is a complete simulation infrastructure that enables systems and software developers to run a variety of data collection and analysis tools to gather multiple types of system metrics at varying levels of granularity. The simulator’s functional fidelity and runtime performance allows a full OS, such as Linux, to be run interactively in simulation. In this manner, the simulator allows applications that require inter-process or complex OS interactions to execute in a complete system environment. In addition to this full OS mode, the simulator also provides a “stand-alone” environment for self-contained applications, in which the simulator intercepts and marshals the application’s system calls to the underlying host to optimize execution.

SimOS [76] is another complete machine simulation environment designed for the efficient and accurate study of both uniprocessor and multiprocessor computer systems. SimOS simulates computer hardware in enough detail to boot and run commercial OS. SimOS provides models of the MIPS R4000 and R10000 and Digital Alpha processor families. In addition to the processor, it simulates caches, multiprocessor memory buses, disk drives, ethernet, consoles, and other devices commonly found on these machines. By simulating the hardware typically found on commercial computer platforms, the simulator is able to easily port existing OS to the SimOS environment.

The SimpleScalar [80] tools suite is a system software infrastructure used to build applications for program performance analysis, detailed micro-architectural modelling, and hardware-software co-verification. Using the SimpleScalar tools suite, users can build modelling applications that simulate real programs running on a range of modern processors and systems. The tools suite includes sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction-sets. The tools suite includes a machine definition infrastructure that permits most architectural details to be separated from simulator implementations. Complex instruction-set emulation (e.g., x86) can be implemented with or without microcode, making the SimpleScalar tools particularly useful for modelling CISC instruction-sets.

Simics [48] is a system level instruction-set simulator. It allows for device modelling and to execute unchanged OS. Moreover, it provides caches and memory hierarchy models and a cycle accurate measurement of the simulated system (more details of this simulator were introduced in Section 2.5).

Table 4.4 compares the mentioned simulators in the features important for this dissertation, and clear identifies Simics as having all the requirements for the intended work.

	IBM Mambo	**SimOS**	**SimpleScalar**	**Simics**
Allows to develop new hardware?	YES (although is not open-source)	YES	YES (not easily)	YES
Simulates memory hierarchy?	YES	YES	YES	YES
Allows cycle accurate measurement?	YES	YES	YES	YES
Allows to boot OS?	YES	YES	YES (not easily)	YES
Provide up-to-date support?	NO	NO	NO	YES
Last Update	2006	1998	2004	2008

Table 4.4: Simulators comparison

Therefore, the Simics full-system simulator is used to model the cache-based memory copy hardware accelerator. Moreover, the knowledge gained when implementing the cache-based memory copy hardware accelerator in the prototyping platforms (mainly in terms of latency) is used to correctly model the accelerator in the Simics simulator. Simics models the world as a series of disjoint events, where each event occurs at some specific time. Events can query and modify the simulated state, and post new events. However, Simics simulates a processor at the instruction-set level, the lowest level that is readily visible to software. This lets it run unmodified binary code for the complete software stack of a system.

A Simics device model models the real hardware and is transparent to the software, so that the software does not notice the difference between the model and the real hardware. The model provides the functional behavior of the hardware as seen by the software. Since Simics usually controls both parties

in any transactions at the hardware level, hardware-hardware interfaces can usually be heavily simplified. For example, transactions on a memory bus or hardware interconnect are usually modelled as atomic, rather than composed of individual steps where buses are arbitrated, addresses written, data read, etc. Another example is the standard Simics cache model: the caches hold no data, they just keep track of what items are in the cache, and inform the processor model of the delay associated with each memory access. The actual data is acquired from the simulator memory, regardless of what cache level the item was found in.

`g-cache` is the standard Simics cache model. It handles one transaction at a time in a flat way: all needed operations (write-back, load, etc.) are performed in order and at once. The cache returns the sum of the stall times reported for each operation. As the goal is to study the timing behavior of the transactions (as they go through the indexing table and the cache), then the transactions should have differentiated times depending where they finish (on which level the transaction finds the data), which implies the possibility of stalling the processor. To be able to stall the processor, any model should be connected to the timing-model interface. This type of simulation modifies the execution, since for example interrupts will be influenced by the timing provided by the model.

In order to perform a performance analysis of the cache-based memory copy hardware accelerator, besides the cache model there is a need to also model the indexing table. Simics provides a sample device that can easily be modified to the purpose of this dissertation. The model of the cache-based memory copy hardware accelerator also has to be able to stall the processor, therefore, the timing-model interface also has to be used. Moreover, because the indexing table is directly connected with a cache, another interface needed is the interface to the cache model. Therefore, all memory transactions go through both the indexing table and the cache model.

As presented in Section 3.4, to allow the communication of the necessary `src` and `dst` addresses and the `size` to the accelerator, two different ways were studied: a memory-mapped device-driver and an instruction-set architecture extension. In the first case, the parameters are communicated to the cache-based memory copy hardware accelerator by accessing the OS `/dev/mem` file. This special file, accessed through a memory-mapped device-driver [86], allows read and write accesses to the virtual memory address space for the current process, as it is seen by the OS. Before issuing a read or write operation, the `/dev/mem` file is looked up to determine the relevant address in virtual

memory, which impose some delay in passing the necessary parameters.

For the instruction-set architecture extension, a single instruction is implemented, and used to substitute the software function calls. Simics provides a sample decoder module that can easily utilized to extend the instruction-set architecture of the processor simulated. This module identifies an opcode (that has to be an un-used opcode of the processor instruction-set architecture) and utilizes the registers used in the instruction to pass the parameters. Moreover, this instruction, besides communicating to the accelerator the necessary parameters, performs the necessary checks of the boundary conditions (is the size of the copy smaller or bigger than the accelerator can support?; are `src` and `dst` addresses cache line aligned?). All memory copies function calls were substituted by the either of the two ways to access the accelerator in all the benchmarks used to verify the cache-based memory copy hardware accelerator.

The cache-based memory copy hardware accelerator is modelled together with a standard Pentium 4 processor at 2 GHz running the standard Linux 2.4 kernel. The L1 cache design has a separate instruction and data cache. Both caches are write-through, write-allocate direct-mapped cache with 512 cache lines each one with 64 bytes. The total size of the caches is then 32 kB (corresponding to the maximum size of a copy) and the replacement policy is LRU. The L2 cache is a unified instruction and data cache, 8-way associative write-back cache with 128 bytes cache line and 2 MB size. The cache hit penalty (both read and write for both instruction and data caches) is two clock cycles.

As mentioned in Section 4.1, the use of a CAM to search the indexing table bounds the performance of the accelerator. As the CAM takes two clock cycles to be written to, the time to fill the indexing table, as well as the time needed to handle the write to the `src` and the `dst` addresses and to handle a cache miss that may evict a `src` address from the cache, are bounded by the CAM time. Moreover, for all these cases, the total delay should also include the time to access the L2 cache (as all cases requires the storing of the copied data in the L2 cache). For a read hit on the indexing table, as the access is performed directly (there is no need to wait for the CAM match) the delay is just one clock cycle plus the read hit time of the L1 data cache. On a read miss on the indexing table, there is no penalty as the cache and the indexing table are accessed in parallel.

As presented in Section 2.2.6, in order to increase the size of the address space of the applications compared with the physical space available, the processor utilizes the virtual-to-physical address translation mechanism.

CPU	
Type	Pentium 4
Frequency	2GHz
CPI	1
Operating System	Linux 2.4
Caches	
L1 I/D Caches	32 kB, 64B cache line, direct-mapped, write--through, write-allocate
L1 Hit Time	2 clk
L2 Unified Cache	2 MB, 128B cache line, 8-way, write-back
L2 Hit Time	15 clk
Cache-based Memory Copy Hardware Accelerator	
Fill Indexing Table Time	2 clk per cache line copied
Indexing Table Read Hit Time	1 clk + L1 cache read hit time
Time to handle cases 1, 2 and 3 of Section 4.1	2 clk
Write-back the evicted data	L2 access time
Main Memory	
Type	SDRAM DDR2 400 MHz
Avg. Access time	240 clk

Table 4.5: Simulation parameters

The cache-based memory copy hardware accelerator fills the indexing table, stating with the first destination address and sequentially fills the consecutive ones until the size of the copy is reached. The destination addresses only are guaranteed to be consecutive if they are virtual addresses (consecutive virtual addresses can map to several non consecutive block of physical addresses). Therefore, the indexing table needs to use virtual addressing. Moreover, in order to have consistency between the indexing table and the cache, this imposes

that the cache has also to be at least virtually indexed. The use of virtual tags is not required, however it might reduce the delay (if the cache lookup takes less time then the TLB lookup).

Finally, the main memory is modelled with an average latency of 240 clock cycles which corresponds to an average access time of 80 nsec of a DDR2 SDRAM 400MHz. The simulation parameters are described in Table 4.5.

4.3 Summary

This chapter introduced the platforms used to implement the cache-based memory copy hardware accelerator introduced in the previous chapter. The details of each platform and of the implementation of the cache-based memory copy hardware accelerator were presented. The implementation constrains that drive the performance of the cache-based memory copy hardware accelerator were also introduced. In particular, the need of a new field in the indexing table, the "Lookup" field, is introduced. Moreover, the impact of this field in the complete design is also discussed. The resource utilization on the prototyping platforms were presented, as well as a study of the number of bits stored in the indexing table and in the cache (to determine the increase in hardware resources due to the utilization of the indexing table). Finally, the modelling of the cache-based memory copy hardware accelerator in the simulator was also presented.

The next chapter presents the results of the benchmarks utilized to demonstrate the benefits of the proposal and its performance gains. Synthetic benchmarks were executed in the prototyping platform, while the performance study was performed in the simulator.

Chapter 5

Results of the Uniprocessor Platform

This chapter presents the results of the benchmarks utilized to demonstrate the benefits of the cache-based memory copy hardware accelerator and its performance gains. Synthetic benchmarks are executed in the previously introduced prototyping platforms, while the performance study, utilizing both the STREAM [52] and the LMbench [54] benchmarks and the receiver side TCP/IP benchmark is performed in the previously introduced simulator. The results present the raw performance of the cache-based memory copy hardware accelerator when prototyped in the Xilinx platforms and provided the necessary input to correctly model the accelerator in the simulator. The performance study in the simulator demonstrated speedups up to 4.61 times.

Section 5.1 presents the raw performance of the indexing table able to support cache line granularity copy, and Section 5.2 introduces the results with the load/store unit. Section 5.3 presents the raw performance for the indexing table able to support word granularity copy. Section 5.4 presents the performance study performed with the simulator and finally, Section 5.5 summarizes the results presented in this chapter.

5.1 Indexing Table Supporting Cache Line Granularity Copy

This section first presents and compares the waveforms (generated by the ModelSim XE-III [58] hardware simulator) of the indexing table supporting cache line granularity copy and of the optimized hand-coded in assembly software

implementation of a memory copy operation. Subsequently, this section presents the results for the best and worst cases in software and compares them with the worst case utilizing the accelerator, for an increasing number of cache lines, for both clock cycles measurement and throughput. The implementation of the indexing table supporting cache line granularity copy is performed on a Virtex-II Pro, as introduced in Section 4.1.1.

In order to validate the design and determine the raw performance of the cache-based memory copy hardware accelerator, the execution of a single memory copy operation of 4 cache lines performed with the accelerator is depicted in Figure 5.1 (the setup time is not shown for clarity). Moreover, in order to compare it with the software approach, the waveform of a software memory copy operation of 32 bytes (corresponding to one cache line; in interest of clarity the waveform for 4 cache lines is not included) is depicted in Figure 5.2. It is worth mentioning that the software code is hand-written in assembly for PPC and that the source and the destination addresses are aligned for both the execution with the accelerator and without.

For the software implementation of the memory copy operation, there is a period to calculate if the addresses overlap and if there are bytes or words to perform the memory copy operation on. This time is 74 clock cycles and the total time to perform a memory copy of one cache line in software is 143 clock cycles. For a memory copy of 4 cache lines (or 128 bytes) the software implementation takes 374 clock cycles (not depicted in interest of clarity). Performing the memory copy operation using the accelerator, there is also a setup time of transferring the `src` and `dst` addresses and the `size` of 28 clock cycles. On a copy of one cache line the accelerator takes 2 clock cycles, consequently the total time to perform a memory copy of one cache line is 30 clock cycles. Therefore, on a copy of one cache line, the cache-based memory copy hardware accelerator performs 79% better than the software implementation. For a copy of 4 cache lines, the accelerator takes 36 clock cycles, which corresponds to a reduction of execution time of 90% compared with the software. Table 5.1 presents the number of clock cycles and the percentage of improvement that a copy of 1, 5 and 1024 cache lines (which corresponds to the maximum the accelerator allows) take executing in software and with the accelerator. Generalizing, a memory copy performed with the accelerator takes 28 clock cycles of setup time plus 2 clock cycles per cache line.

Figure 5.3 depicts the performance (in terms of throughput and clock cycles) of a memory copy executed in software and with the accelerator, for different sizes. As expected, the benefit of the cache-based memory copy hard-

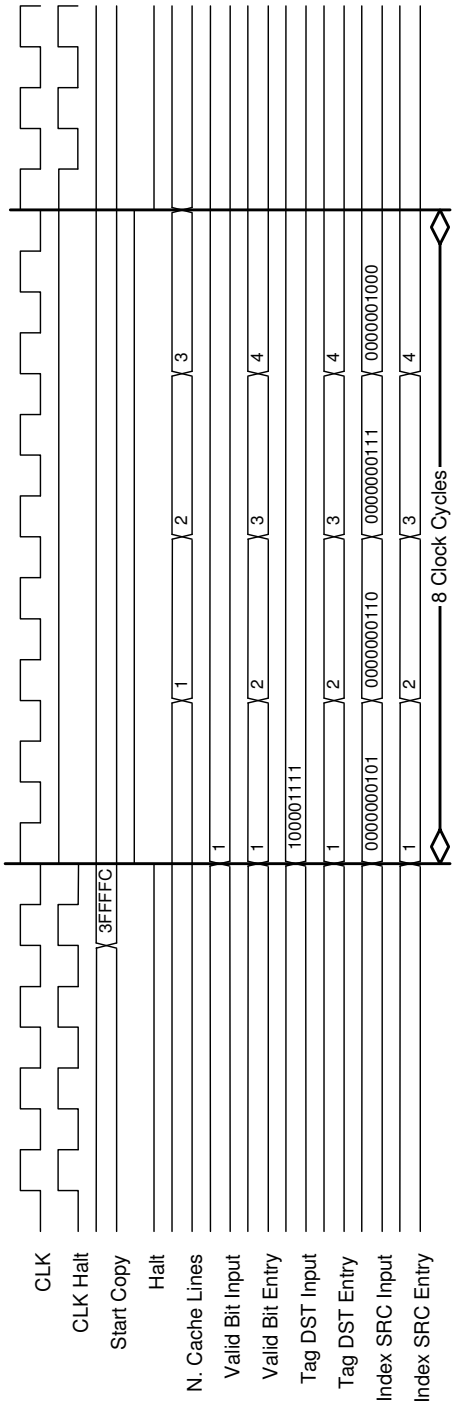


Figure 5.1: Waveform of a copy of 4 cache lines with the accelerator

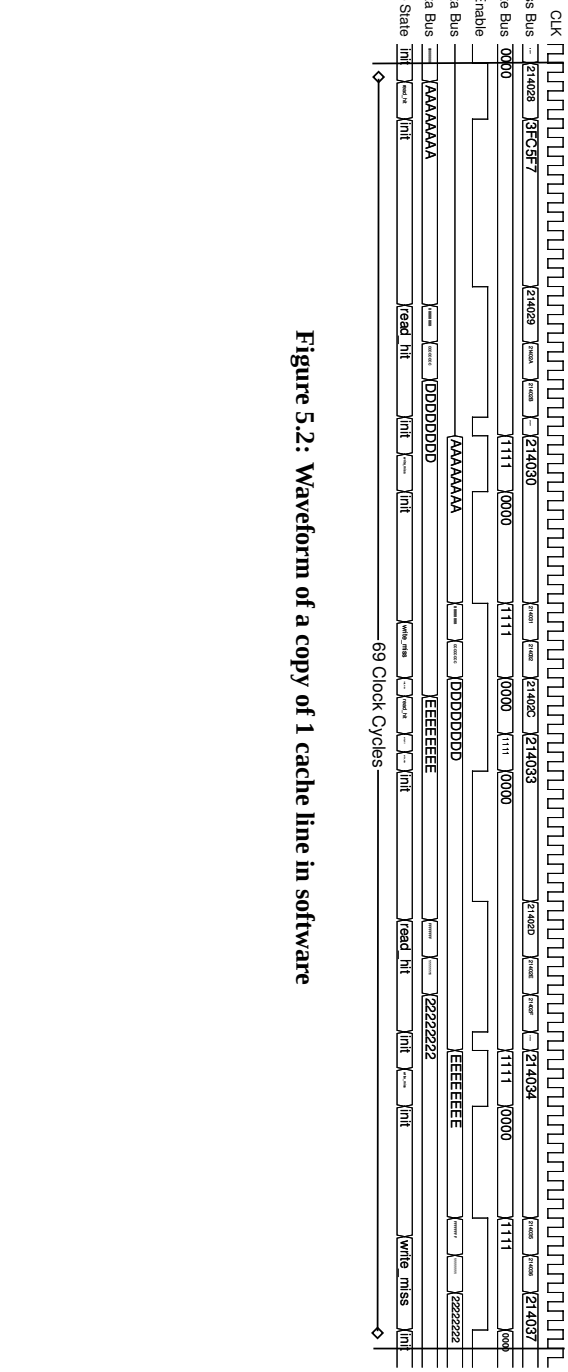


Figure 5.2: Waveform of a copy of 1 cache line in software

	Copy of 1 Cache Line	**Copy of 5 Cache Lines**	**Copy of 1024 Cache Lines**
SW memory copy	143 clk	419 clk	70730 clk
Acc. memory copy	30 clk (79%)	38 clk (91%)	2076 clk (97%)

Table 5.1: Performance of memory copies for cache lines

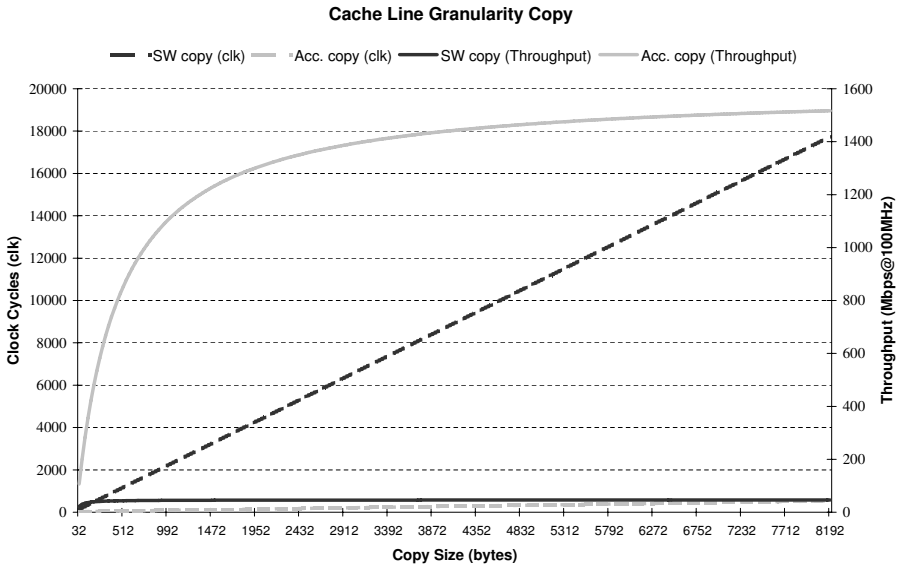


Figure 5.3: Memory copy throughput for cache lines

ware accelerator increases with the size of the copy.

A comparison was also performed between the proposed cache-based memory copy hardware accelerator with AltiVec [2] solution. In [2], the authors present values for a copy of 160 bytes around 586 Mbps (the picture presented does not allow to determine exact values, and no values are referred in the text). As the PPC runs at 750 MHz in [2], this corresponds to around 210 clock cycles. The PPC used with the cache-based memory copy hardware accelerator is running at 100 MHz and a memory copy can be performed in 38 clock cycles, for the same 160 bytes (5 cache lines). This is approximately 80% better than [2], in terms of clock cycles, although the throughput achieved is only 421 Mbps (due to the difference in the PPC clock).

5.2 Load/Store Unit

To validate the design, first it is presented the waveforms of executing a memory copy of 4 cache lines with the optimized hand-coded in assembly software (in Figure 5.4) and, next, the same 4 cache lines are copied using the accelerator (in Figure 5.5), both with subsequent reads and writes to the original and copied data. This implementation of the indexing table is able to support cache line granularity copy and is implemented in the Virtex-4 FPGA.

Afterwards, in order to show the advantages of the cache-based memory copy hardware accelerator, a synthetic benchmark based on a real application is created. It is based on the data collected by [23] (and presented in Appendix A), which describes the procedure to reassemble a frame of the Bluetooth protocol in the Linux OS. Bluetooth reassembles a frame in order to create a frame the same size as a Ethernet frame (1500 bytes), that is afterwards processed through TCP/IP. The Bluetooth performs this by executing the memory copy operation 4 times with size 339 bytes plus one of 151 bytes. As these numbers are not cache line aligned, 4 memory copies with a size of 352 bytes plus one of 96 bytes (corresponding of `4x11` cache lines `+ 1x3` cache lines) were performed. The Bluetooth protocol uses consecutive `src` addresses to reassemble one frame, which means that the data has to be loaded from the main memory for every memory copy (in the accelerator case). For the `dst` addresses, the Bluetooth protocol also uses consecutive addresses, which means there is no need to write-back to the main memory (in the accelerator case). For the accelerator, the complete synthetic benchmark takes 906 clock cycles while the software version takes 6263 clock cycles. This implies that, for this synthetic benchmark, the accelerator achieves a speedup of approximately 7 times compared with the software version.

Subsequently, the results for the best and worst cases in software are presented and compared with the worst case of utilizing the accelerator, for an increasing number of cache lines. Figure 5.6 depicts the worst case scenario for the cache-based memory copy hardware accelerator, which is no data required for the memory copy is in cache (all the data has to be loaded from the main memory) and the memory copy is overwriting previously performed memory copy (every cache line has to be written-back to the main memory). The best case scenario for the software version is having all the required data in cache, while the worst case is not having it.

As expected, the benefit of the accelerator increases with the increase of the size of a memory copy. For a memory copy of only one cache line, there

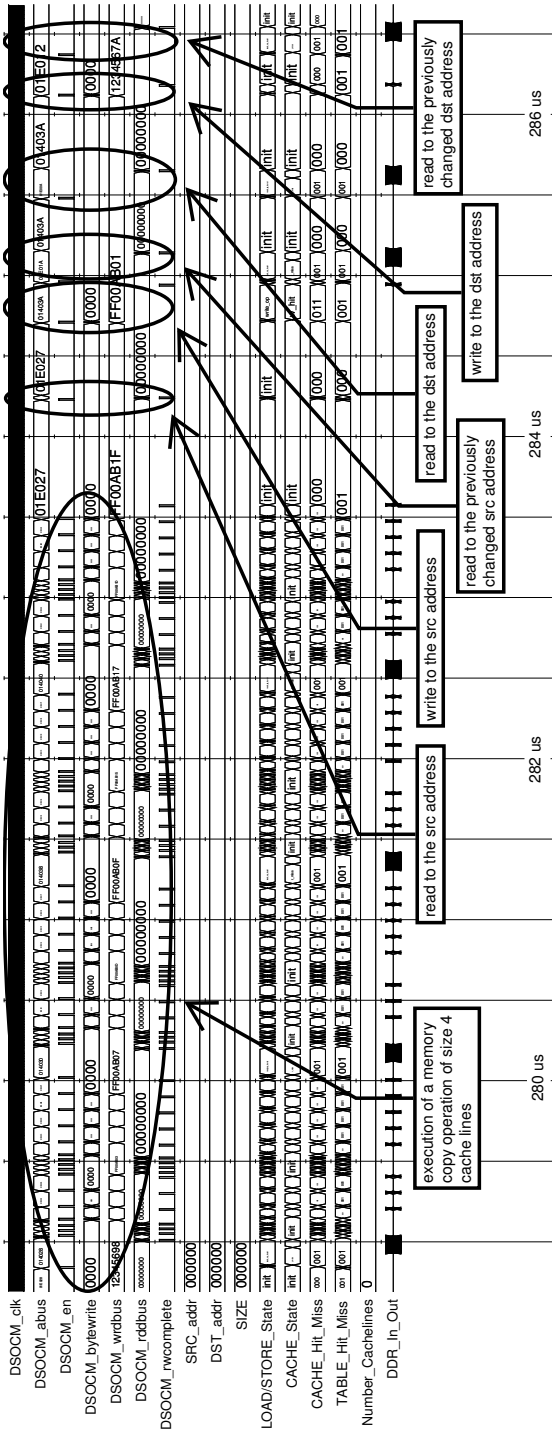


Figure 5.4: Waveform of a copy of 4 cache lines in software

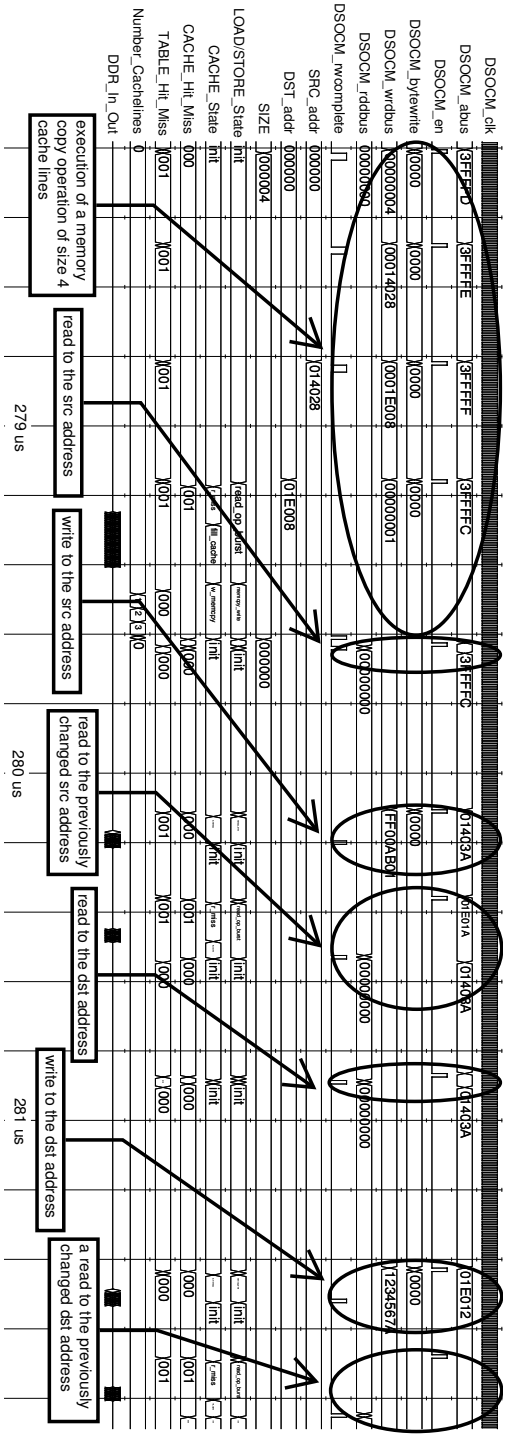


Figure 5.5: Waveform of a copy of 4 cache lines with the accelerator

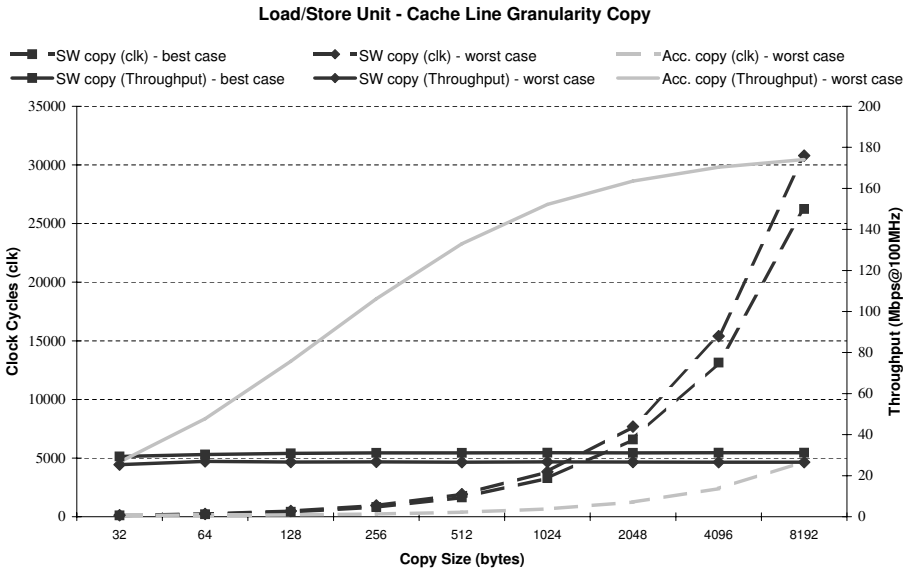


Figure 5.6: Memory copy throughput with the load/store unit

is no clear benefit of the worst case of utilizing the accelerator compared with either the best or worst case software solutions. However, for bigger sizes the accelerator is capable of reducing the memory copy latencies by 82% (for a copy of 8192 bytes or 256 cache lines) compared with the best case for the software implementation.

5.3 Indexing Table Supporting Word Granularity Copy

This section presents the synthetic benchmarks used to demonstrate the benefits of using the indexing table able to support word granularity copy implemented in a Virtex-II Pro FPGA. For the cache line granularity copy, the accelerator needs 28 clock cycles to communicate with the parameters and 2 clock cycles to actually fill the indexing table (this time is bounded by the time it takes for the CAM to be written, referred in Section 4.1). For the word granularity copy the same 2 clock cycles are needed, however as an entry in the indexing table now refers to a word and not to a cache line, this is the time needed to fill a word in the indexing table.

Table 5.2 presents the number of clock cycles and percentage of improvement that a copy of 1, 8, 40 and 8192 words take when executed in software

	Copy of 1 Word	**Copy of 8 Words**	**Copy of 40 Words**	**Copy of 8192 Words**
SW memory copy	89 clk	143 clk	419 clk	70730 clk
Acc. memory copy	30 clk (66%)	44 clk (69%)	108 clk (74%)	16412 clk (77%)

Table 5.2: Performance of memory copies for words

and with the accelerator. Generalizing, a memory copy performed with the accelerator takes 28 clock cycles of setup time plus 2 clock cycles per word. Figure 5.7 depicts the performance utilizing the accelerator compared with the software implementation of memory copy for different number of words (represented in bytes instead of words). As expected, the benefit of the cache-based memory copy hardware accelerator increases with the size of the copy.

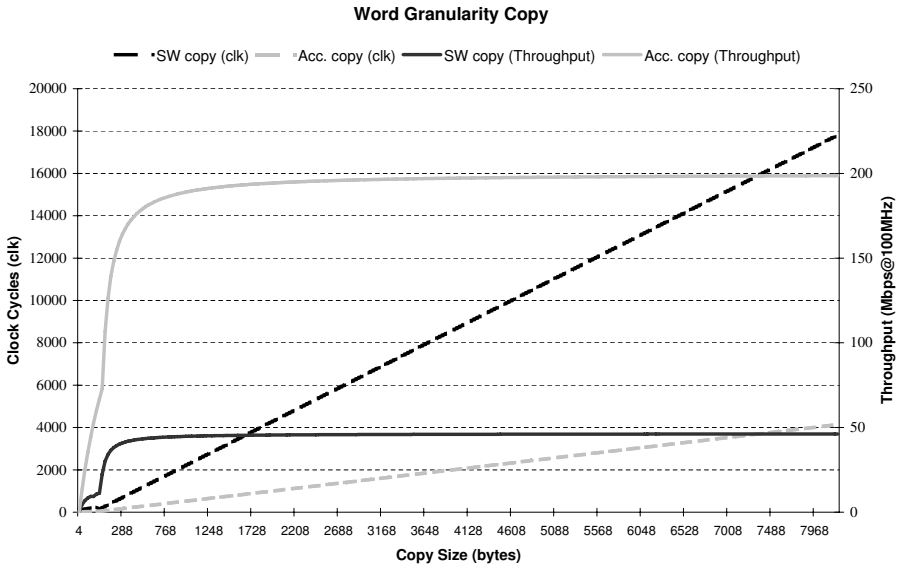


Figure 5.7: Memory copy throughput for words

Again, a comparison was performed between the cache-based memory copy hardware accelerator with the AltiVec [2] solution. A memory copy can be performed in 108 clock cycles using the cache-based memory copy hardware accelerator, while the AltiVec solution takes 210 clock cycles, for the

same 160 bytes. This is approximately 48% better than [2], in terms of clock cycles, although the throughput achieved is only 148 Mbps (due to the difference in the PPC clock frequencies).

5.4 Performance Study

In order to determine the performance of the cache-based memory copy hardware accelerator, the simulator described in Section 4.2 is utilized. The input parameters of the simulator are based on the information gathered in the previous section, mainly latency information. First, it is presented the results of using a memory-mapped device-driver to access the cache-based memory copy hardware accelerator, and secondly, the results using the instruction-set extension are discussed.

5.4.1 Memory-Mapped Device-Driver

In order to determine the raw throughput that the accelerator can provide, the STREAM [52] benchmark and the LMbench [54] were used, as both benchmarks test the memory copy performance utilizing different algorithms. STREAM benchmark compares several copy kernels: `copy_32`, `copy _64`, `copy_32x2`, `copy_32x4` and `glibc memcpy`. The first four kernels perform copies on loops of either integers (32 bits) or doubles (64 bits), using two or four intermediary variables. The `glibc memcpy` uses the standard `glibc` algorithm implemented in the Linux OS, which is optimized in assembly. A call to the accelerator is also included in this benchmark in order to compare it with the mentioned kernels.

The LMbench consists on several applications although only one is used to measure memory bandwidth (the `bw_mem`), and it provides the average throughput and execution time at which a processor can move data. The benchmark is executed in software (which utilizes the `glibc bcopy` function) and afterwards it is executed calling the accelerator. The execution times and throughput are gathered for both cases. This benchmark is used to measure the impact of changing the memory latency, the cache line size and the processor frequency on the performance of the accelerator and on the `glibc bcopy` function.

The STREAM benchmark provides the average throughput measured over 1000000 executions, the process is repeated 10 times and the best time is displayed. It provides the throughput for copies of arrays of 800 bytes (to test the

L1 cache throughput). The STREAM benchmark does not use the copied data (the copied data is not written-back to the main memory) and executes several copies in order to average the results. Therefore, the accelerator has, for each repetition, to write-back the previously performed copy, fetch the necessary data from the main memory into the cache and fill the indexing table, which is the worst-case scenario for the accelerator.

The LMbench, on the other hand, estimates the necessary number of iterations that provide an accuracy of 95% of the execution time [54]. Looking at the number of iterations, the number is higher on LMbench than on STREAM, which leads to the conclusion that LMbench is more accurate then STREAM. This is the reason why STREAM was chosen to compare the accelerator to other copy algorithms and the LMbench to further study the impact of changing the memory latency, the cache line size, and the processor frequency.

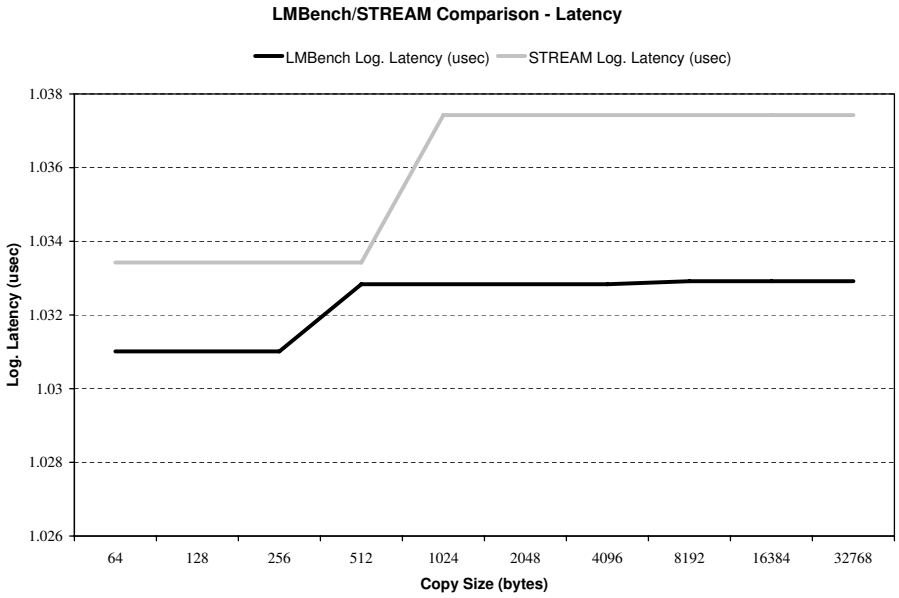


Figure 5.8: Average latency of LMbench and STREAM benchmarks

The first analysis, depicted in Figures 5.8 and 5.9 present a comparison of the average latency and throughput of both benchmarks utilizing the accelerator for different copy sizes. The steps in Figure 5.8 are due to the report of the measurement. As the benchmarks are executed on top of an OS, their report depends on the state of the system. Therefore, the reported values can change slightly for each execution. In order to minimize this unwanted influence,

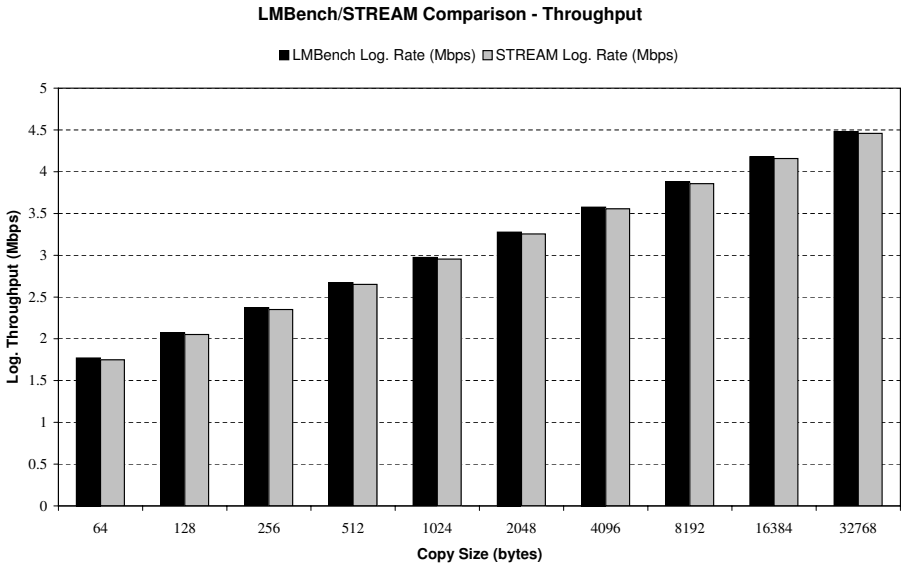


Figure 5.9: Average throughput of LMbench and STREAM benchmarks

the benchmarks are executed 10 times and the average values are presented. Moreover, in absolute times, these steps correspond to a difference in latency of 45 nsec for LMbench and of 100 nsec for STREAM therefore negligible. As explained earlier, LMbench is more accurate than STREAM thus it is not surprising the average throughput to be slightly higher than the one presented by STREAM.

The subsequent analysis is performed by comparing the accelerator with other copy algorithms, such as the STREAM kernels. Figure 5.10 depicts the average execution time and Figure 5.11 depicts the average throughput comparisons. It is clear from the figures that for copies smaller than 512 bytes (4 cache-lines) the accelerator presents a penalty compared with the `glibc memcpy` algorithm. However, as soon as the sizes of the copies increase the benefit of using the accelerator becomes evident, and can reach, for copies of 32 kB, an average execution time speedup of approximately 121 times. In Section 5.2, it was presented 82% reduction of the execution time (comparing the accelerator execution time and the `glibc memcpy` execution time) for copies of 256 cache lines, which corresponded, on the prototyping platform, to a copy of 8 kB. For the same size of a copy, it is now possible to reach a reduction of the execution time of 94.5%. The reason for this increase in performance is due to the size of a cache line. The implementation performed

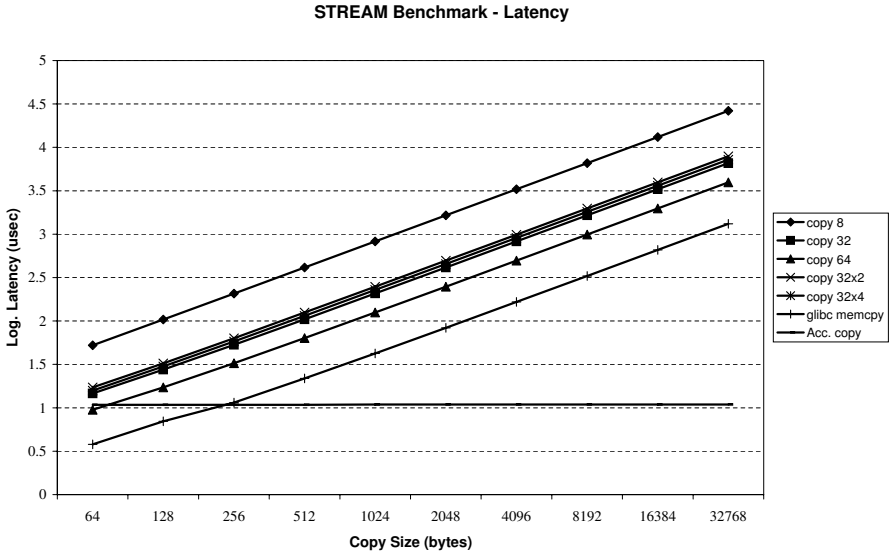


Figure 5.10: Average execution time for STREAM benchmark

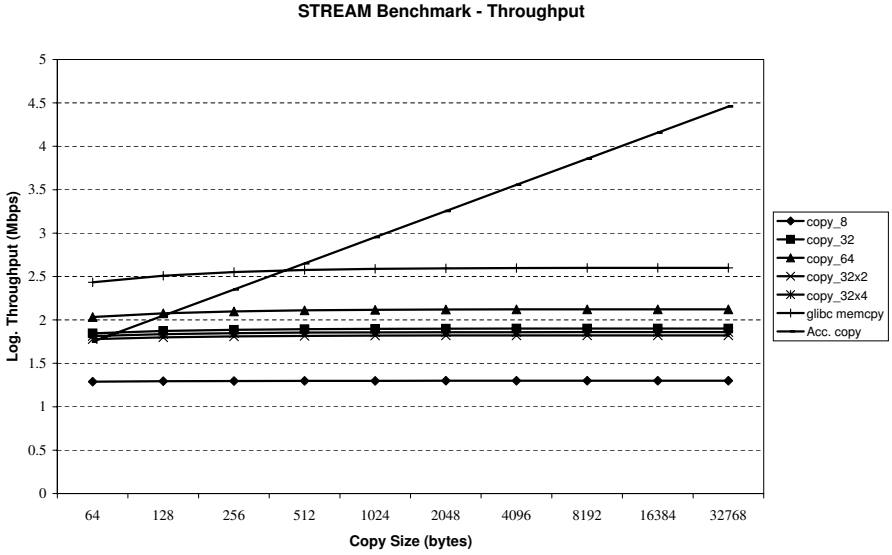


Figure 5.11: Average throughput for STREAM benchmark

in the prototyping platform has a cache line of 32 bytes, while in with the simulator it is 64 bytes. In summary, the accelerator is able to roughly double the average throughput for every double copy size and it provides roughly a

constant latency independent of the copy size. The throughput provided by the `glibc memcpy` decreases with the size of the copy (increasing the copy size from 64 to 128 bytes will increase the average throughput by 16% while increasing the copy size from 16 kB to 32 kB will only increase the throughput by 0.08%). On latency, the `glibc memcpy` increases the latency by roughly 45% for every doubling the copy size.

In order to understand the impact of changing some parameters of the system, the LMbench kernel was used for copy sizes up to 32 kB. Table 5.3 presents the average throughput and execution time of the baseline scenario, which corresponds to a memory latency of 240 clock cycles, a cache line size of 64 bytes and a processor frequency of 2 GHz.

	Baseline Scenario		Freq = 6 GHz		Lat = 180 clk		Cache line = 128 B	
	SW	Acc.	SW	Acc.	SW	Acc.	SW	Acc.
Throughput (Gbps)	0.025	30.28	0.025	30.23	0.033	40.36	0.025	30.28
Execution Time (msec)	1.316	0.011	1.317	0.011	0.983	0.008	1.316	0.011

Table 5.3: Impact of changing several parameters of the system

The processor frequency was increased to 6 GHz, an increase of 66% to simulate the impact of future processors (with higher frequencies) on the copies algorithms. In order to have correct simulations, it was also need to increase by the same amount the modelled latencies of the accelerator (the cache, the indexing table and the main memory) because the model is based on the number of cycles of the processor. Therefore, the penalty of filling the indexing table is now of 4 clock cycles instead of the previously modelled 2 clock cycles, and the penalty of a write to either the `src` or `dst` addresses is also 4 clock cycles. For the cache a hit penalty (both read and write for both instruction and data caches) is now 4 clock cycles and the main memory latency is now 480 clock cycles. Table 5.3 presents the correspondent results. The increase in frequency does not have an impact for either the `glibc bcopy` or the accelerator, because the copy algorithms are not computing-intensive but memory-intensive.

In order to model the impact of future generations of memories, the impact

of decreasing the memory latency to 180 clock cycles (60 ns, a decrease of 25%), was simulated. Table 5.3 also presents the correspondent results. As expected there is an average reduction of 25% on both the average throughput and execution time.

And finally, the cache line size was increase to 128 bytes, an increase of 50%. Table 5.3 presents the correspondent results. It would be expected to have a increase in the accelerator average throughput and execution time, because it is actually using a bigger block to perform the copy. However, because the penalty of fetching data from the main memory is dominant, the benefit of increasing the cache line size is not visible.

It is important to mention the impact that accessing the accelerator has on these results. When the memory latency is decreased there is a decrease on the accelerator's average throughput and execution time. However, when the cache line size is increased, there is no change on the average throughput or execution time, when it should. The reason for this behavior is due to the way the accelerator is accessed. As explained before, the accelerator is accessed through a memory-mapped device-driver, so in order to access it there is the penalty of performing system calls and memory copies to communicate with the OS. Therefore, reducing the memory latency will reduce the time to perform the necessary memory copies and consequently reduce the access time of the accelerator (as depicted in Table 5.3). However, increasing the cache line size has no visible increase on the average throughput or execution time. The conclusion is that the accelerator is bounded by the access time and not by the copy size. This justifies the implementation of an instruction-set architecture extension.

5.4.2 Instruction-Set Architecture Extension

This section presents the results of the cache-based memory copy hardware accelerator, now using an instruction-set architecture extension. The same STREAM benchmark is used to study the performance.

Table 5.4 depicts the results reported by the benchmark itself and Table 5.5 depicts the statistics reported by the accelerator. The speedup on the execution time of the STREAM benchmark, when utilizing the accelerator compared with utilizing the `glibc memcpy` algorithm is 1.2 times while the reduction of the average execution time is 45%. As expected the number of writes to either the `src` or the `dst` addresses are small due to the write-back of the copy previously performed before each new iteration.

STREAM Memory Benchmark		
Total memory required = 16 MB.		
Number of runs = 10		
1st level cache Working on Arrays of 800 B.		
Function	Rate (Mbps)	Avg. Time (ns)
copy 32	212.31	14.25
copy 64	294.38	13.43
copy 32x2	140.58	19.33
copy 32x4	150.15	30.89
glibc memcpy	600.37	2.66
Accelerator	722.34	2.21

Table 5.4: STREAM benchmark results

STREAM Benchmark	
Number of Copies	100000
Number of Read Hits in the Indexing Table	18
Number of Writes to the `src` Address	180
Number of Writes to the `dst` Address	44

Table 5.5: Memory copy statistics for the STREAM Benchmark

As the STREAM benchmark is a synthetic benchmark, the cache-based memory copy hardware accelerator is also evaluated with a more realistic workload. The receiver side of the TCP/IP stack is bounded by memory copies (as mentioned in Section 1). Therefore, a user-space implementation of this part of the stack is used to evaluate the accelerator. Several packets are captured by sending a file of 10 MB over a TCP connection. The file is divided into 8418 packets of which 92.8% have 1260 bytes (the typical packet size ranges between 800 bytes and 1500 bytes). The packets captured are fed into the receiver side of the TCP/IP stack in user-space, measured the average number of cycles per packet and repeated the process 10 times.

The execution time of the receiver side of the TCP/IP stack using the `glibc memcpy` algorithm took on average 54972 cycles/packet while with the utilization of the accelerator the execution time was reduced to 11930 cycles/packet. It is worth mentioning that this numbers are given by the application itself and not measured in the simulator (the application returns an approximation due to interference of the OS). In order to have more accurate results, the execution time and number of instructions executed are also measured with the simulator. Table 5.6 presents the statistics gathered by the application, the simulator and by the cache-based memory copy hardware accelerator itself.

From Table 5.6, a speedup of 4.61 times can be calculated (corresponding to a 78% reduction in execution time), when the accelerator is used. Besides, there is also a reduction of 44% of the number of instructions executed (because no loads and stores are used) and a higher cache hit rate compared with the software version (because the `src` data is fetch from the main memory to the cache while executing the memory copy).

The receiver side of the TCP/IP stack does not include the reading of the copied data by the application that is receiving this data (as it can be noticed by the zero read hits in the indexing table in Table 5.6). In order to correctly evaluate the cache-based memory copy hardware accelerator, a read of every packet received is included in order to mimic the normal behavior of an application. Table 5.6 also presents the statistics for this case. The simulator reports a speedup of 2.96 times (corresponding to a 66% reduction in execution time). The reason for this decrease is the increase in execution time of the receiver side of the TCP/IP stack, as seen by the number of instructions executed (in this case, the accelerator only reduces by 26% the number of instructions compared with the software case). Therefore, the clear benefits of the cache-based memory copy hardware accelerator are diluted over the total execution time, however still achieves a higher cache hit rate than the software.

Standard TCP/IP stack		
Application Stats	SW Execution Time	54972 cycles/packet
	Acc. Execution Time	11930 cycles/packet
Simulator Stats	SW # Instructions	25256762
	SW Execution Time	232 msec
	SW L1$ Read Hit	92.12%
	SW L1$ Write Hit	89.53%
	Acc. # Instructions	13968064
	Acc. Execution Time	51 msec
	Acc. L1$ Read Hit	98.05%
	Acc. L1$ Write Hit	96.02%
Accelerator Stats	# Copies	8414
	# Read Hits Index.	0
	# Writes SRC Addr.	262
	# Writes DST Add.	481685
TCP/IP stack with reads		
Application Stats	SW Execution Time	98693 cycles/packet
	Acc. Execution Time	57097 cycles/packet
Simulator Stats	SW # Instructions	534685789
	SW Execution Time	268 msec
	SW L1$ Read Hit	92.77%
	SW L1$ Write Hit	90.24%
	Acc. # Instructions	180120983
	Acc. Execution Time	90 msec
	Acc. L1$ Read Hit	97.82%
	Acc. L1$ Write Hit	96.18%
Accelerator Stats	# Copies	8414
	# Read Hits Index.	4299
	# Writes SRC Addr.	22686
	# Writes DST Addr.	1194569

Table 5.6: Memory copy statistics for the TCP/IP stack

Synthetic benchmark of 32 copies	
SW Execution Time	457291 cycles
Acc. Execution Time	13905 cycles
# Copies	32
# Read Hits Index.	0
# Writes SRC Addr.	31
# Writes DST Addr.	0
Synthetic benchmark of 32 copies with reads	
SW Execution Time	1162650 cycles
Acc. Execution Time	897626 cycles
# Copies	32
# Read Hits Index.	479
# Writes SRC Addr.	0
# Writes DST Addr.	7943

Table 5.7: Memory copy statistics for the synthetic benchmark

In order to evaluate the impact of changing the number of packets processed by receiver side of the TCP/IP stack, the experiment is repeated with a smaller file. The execution time of the receiver side of the TCP/IP stack using the accelerator compared with utilizing the `glibc memcpy` algorithm (measured by the simulator) is 5.68 times speedup, where the accelerator kept the same average cycles per packet (the software time increased). In the case that a read is included in each packet, the speedup is 3.48 times, for reasons explained in the paragraph before.

To evaluate the benefits of the accelerator in the case that several copies are performed but the data is not read immediately after, a synthetic benchmark that performs 32 memory copies to consecutive addresses is created. The number of copies performed is the maximum allowed by the cache-based memory copy hardware accelerator for the simulated setup. The measurement evaluates the performance of the accelerator without and with reads of the data after all 32 copies have been performed. Table 5.7 depicts the results.

If no reads are performed, meaning the execution time of the application is much smaller, the benefit of the cache-based memory copy hardware accelerator is bigger, as expected. The speedup for the case where no reads are

performed is 32.8 times and where reads are performed drops to 1.29 times as the execution time increased by 60%.

5.5 Summary

This chapter presented the results of the benchmarks used to demonstrate the benefits of the cache-based memory copy hardware accelerator on the prototyping platforms. The indexing table is able to perform a copy in 28 clock cycles plus 2 clock cycles per cache line or word (depending on the granularity of the indexing table utilized). Moreover, accessing the cache-based memory copy hardware accelerator through a memory-mapped device-driver has impact on the performance of the system, as demonstrated by the performance study on the simulator. When the access is performed through an instruction-set architecture extension, a speedup ranging from 2.96 to 4.61 times for the receiver side TCP/IP benchmark is achieved, when utilizing the cache-based memory copy hardware accelerator compared with the software implementation of the benchmark. Moreover, there is also a reduction on the number of instructions executed ranging from 26% to 44%, due to removing the loads and stores needed to execute the memory copy operation in software.

The next chapter extends the cache-based memory copy hardware accelerator to support multiprocessor platforms.

Chapter 6

Multiprocessor Platform

The previous chapters introduced the cache-based memory copy hardware accelerator (with the indexing table able to support cache line and word granularity copy and its load/store unit), the platforms used to demonstrate its benefits and the performance gains. This chapter introduces the applicability of the cache-based memory copy hardware accelerator to a multi-core processor (utilizing the indexing table implementation for set-associative caches). The utilization of the accelerator can provide up to 50% reduction on the average number of cycles executed per instruction for one of the cores in the system (which takes into account the presence of the other cores of the system) for the best case.

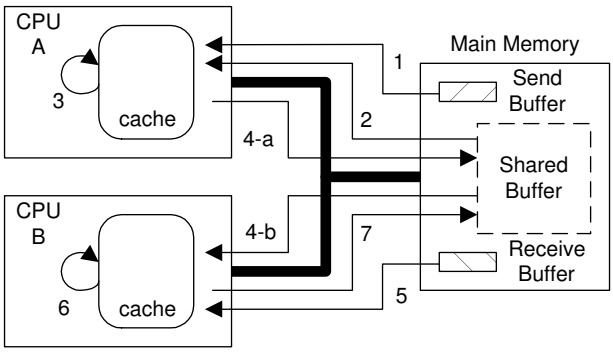
This chapter is organized as follows. In Section 6.1, the typical implementation of a message passing communication model for a multi-core processor is presented, and it presents the applicability of the cache-based memory copy hardware accelerator to the message passing communication model. Section 6.2 introduces the analytical analysis used to demonstrate the accelerator feasibility and Section 6.3 presents the results. Finally, Section 6.4 summarizes this chapter.

6.1 Observation

As discussed in Chapter 1, multi-core processors have become a commodity and, therefore, they have been deployed in loosely-coupled systems, which implement a message passing communication model. This communication model is based on two main operations, namely `send` and `receive` (or its

variations, i.e., various forms of blocking and non-blocking operations). Only the structures (header queues and payload buffers) involved in the message passing send/receive are allocated in a shared address space during the initialization process. The header queues hold information about the messages (e.g., type, size, tag, etc.), and also the payload for short messages. The payload buffers contain the payload for large data messages. There are three data exchange mechanisms: eager, rendezvous and get. Which mechanism is used in a particular instance is determined by the implementation, depends on the size of the memory data movement and its performance depends on the underlying platform. The referred mechanisms mainly rely on copying the data to and from the header queues and the payload buffers and they use the `memcpy` function to accomplish it.

A typical implementation of the mentioned mechanisms to send and receive data over a message passing communication model, is depicted in Figure 6.1. The sending process copies the message along with other information required for message matching, to the shared memory area (Shared Buffer in Figure 6.1). The receiver process can then match the tags of the posted receives and, accordingly, copy the correct message to its own buffer (Receive Buffer in Figure 6.1). Although this approach involves minimal setup overhead, it requires the use of at least two copy operations that will keep the core performing the memory copy busy for the duration of the copy.



1 - Bring the Send Buffer into the cache A, if it is not there;
2 - Bring the Shared Buffer into the cache A, if it is not there;
3 - Write the Send Buffer into the Shared Buffer in cache A;
4-a - Update the content of the Shared Buffer in the main memory;
4-b - Bring the Shared Buffer into the cache B, if it is not there;
5 - Bring the Receive Buffer into the cache B, if it is not there;
6 - Write the Receive Buffer into the Shared Buffer in cache B;
7 - Update the content of the Shared Buffer in the main memory.

Figure 6.1: A typical message passing protocol

The cache-based memory copy hardware accelerator can be utilized to alleviate the bottleneck imposed by the memory data movements, mainly due to the memory copies involved in the send/receive message passing communication model. It takes advantage of having a data L1 cache tightly-coupled to each core, a shared L2 cache for all cores, and a bus-based interconnect network, that provides the necessary functionality for the cache coherence protocol. Therefore, the indexing table implementation presented in Section 3.3.4 to support set-associative caches and the MESI protocol is used and is connected to a L2 cache.

Figure 6.2 depicts a multi-core processor with 3 cores with a message passing communication model. The cache of each core can have cache lines

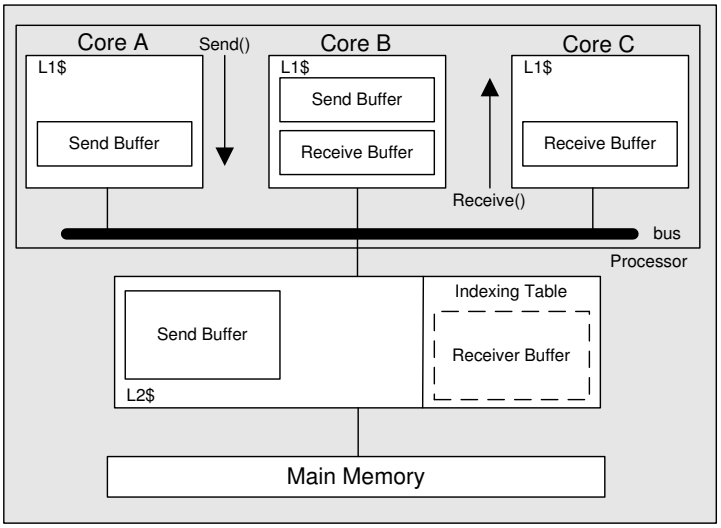


Figure 6.2: Multi-core processor with 3 cores

belonging to either the Send or Receive Buffers involved in the `send` and `receive` operations of the message passing communication model. When a `send` operation is issued by one core, all the cores in the system need to write-back any cache line in its L1 cache that belongs to the Send or the Receiver Buffer. This can be done by changing the state of the cache line (part of the cache coherence protocol) in order to trigger a write-back of the cache line to the L2 cache. On the execution of the `send` operation, the L2 cache loads the necessary data, if it is not there yet, and fills the indexing table. When the `receive` operation is issued by a particular core, the cache line of the Receiver Buffer it refers to, will cause a miss in its L1 cache. This will trigger

a request of the data on the bus that will be served by the L2 cache. The requested cache line is loaded from the L2 cache, using the indexing table, into the L1 cache that requested it. This implies a change in the cache line state to Shared in the L1 cache and to Modified in the L2 cache. When a cache line in the L1 cache, part of the Receiver Buffer, changes status from Modified to any other state, the L2 cache needs to write it back to the main memory. If another core requests a cache line that is in the indexing table, the L2 cache serves the request and provides the cache line, changing the state of its the cache line to the appropriated one.

In order to analyze the benefits that the cache-based memory copy hardware accelerator can have in a multi-core processor, the queue theory [26] was used to perform an analytical analysis of the system. The queuing theory applied was based on the work of [50] and [51]. The input parameters for the model of the cache-based memory copy hardware accelerator are based on the accelerator implementation on the prototyping platforms and on the simulation results presented in Chapters 4 and 5. The next section describes this analysis.

6.2 Analytical Analysis of the System

In the simplest form, any processing system can be considered to be composed of a number of servers with a specific service time. These servers can be any shared resources with a queue, such a bus or a memory. The queuing theory correctly models the performance of such servers in a complex system and, therefore, can be utilized to model a processing system. The request rate to each server in the system is assumed to follow a Poisson distribution, which allows the use of relatively simple equations to depict queue sizes and queue delays[1]. The service time of each server is assumed to be either constant or exponential (both analysis are performed). Therefore, the modelled system falls in Kendall's notation [26] M/D/1 (Poisson input, deterministic/constant service time, one server) or M/M/1 (Poisson input, exponential/Poisson service time, one server).

The ideal performance of a processing system is measured in cycles per instruction (CPI) when executed with an infinite cache ($CPI[\infty]$), i.e., when the L1 cache acts as if there were no cache misses and thus no fetch penalties. However, the actual performance of a system ($CPI[system]$) with a memory

[1] Studies by [39] and [41] present traces which closely approximate a Poisson distribution. For the cases where the traces deviate from a Poisson distribution, there is no simple method to evaluate the impact of the memory hierarchy.

hierarchy is considerably less due to the stalls and delays caused by the cache misses and thus the fetching actions. In reality, a processor and its L1 cache are the sources that generate the outstanding misses to the memory hierarchy, and therefore also responsible by the fetching actions. This additional delay, measured as cycles per instructions, is typically known as the finite cache penalty (FCP). These two parameters are added together to obtain the actual processing system performance:

$$CPI[system] = CPI[\infty] + FCP. \tag{6.1}$$

A problem in all analytical queuing analysis is to determine the queue delays at each server. In particular, the open queue theory requires that the number of requests at each server cannot be fixed at any point in the system. In fact, all queues must theoretically be capable of unlimited length. However, in a real multi-processor system, each processor typically permits only a fix number of outstanding misses to exist within the memory hierarchy at any time. The reason is that the outstanding misses stall the processor which cannot continue processing until the fetches (due to the misses) are fulfilled. This means that the maximum requests for fetching is fixed for all the queues within a memory hierarchy at any instant of time. Of course, the number of requests could be fewer than the maximum at any instant, only the maximum is fixed. Therefore, there is a negative feedback process in the memory hierarchy which in practice, guaranties small and self-limited queues. Consequently, there is no need to complicate the model using closed queue theory to model the system. Moreover, the accuracy of the open queue theory was compared with a closed queue theory and the results showed a 3% to 10% maximum difference (these results are demonstrated on [50] and [51]). The accuracy of the open queue theory was also measured to be approximately 3% compared with data gathered by real systems measurements.

Another way of viewing the mentioned negative feedback is the following. The FCP delays depend on the request rates for the memory hierarchy, which are inversely proportional to the CPI - the smaller the CPI is, the faster the instructions are processed, thereby generating more memory hierarchy requests. The increase in memory hierarchy requests create larger queue delays, which then increase FCP value and the CPI value, reducing the queue delays, and so on and so forth. Thus, the analysis require an iterative calculation, but it will converge in a maximum of 10 interactions.

In a system with a multi-level cache hierarchy, each main memory access can incur in a different delay determined by which level of the hierarchy contains the desired access at that moment. The FCP delay, is then, the weighted

sum of the hits at each level multiplied by the delay per hit at each level. The FCP equation for a uniprocessor with tandem caches (3 level hierarchy) is:

$$\begin{aligned} FCP &= (mr_1 - mr_2)T_2 + (mr_2 - mr_{main})T_{main} \\ &\quad + mr_{main}T_{main} \\ &= mr_1[(1 - \frac{mr_2}{mr_1})T_2 + \frac{mr_2}{mr_1}T_{main}] \end{aligned} \tag{6.2}$$

where the mr_k are the miss rates and T_k are the fetching times of each cache level, $k = \{1, 2, main\}$.

In a memory hierarchy in which misses can occur at various levels, the probability of a miss reaching any given cache level will vary as determined by the given miss rates for each cache. This is expressed as the visitation probability (V), which is simply the hit probability per L1 cache miss request. From Equation (6.2), the visitation probability at each level can be determined to be simply:

$$\begin{aligned} V_2 &= 1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2}{mr_1} \end{aligned} \tag{6.3}$$

To determine the fetching time of each level (i.e., the value of T_k, with $k = \{1, 2, main\}$), it is assumed that each cache is simply an individual server. In a simple open queue system, the queue length at an individual server is determined by the utilization (U), which itself depends on the request rate (R) and the service time (S) of the server. For a memory hierarchy, the visitation probability has to be included in the utilization of each server. For a uniprocessor with tandem caches (3 level hierarchy), the utilization of each server is then:

$$\begin{aligned} U_2 &= V_2 \times S_2 \times R \\ U_{main} &= V_{main} \times S_{main} \times R \end{aligned} \tag{6.4}$$

where the request rate is given by $R = mr_1/(CPI[system] \times T_{CPU})$ (i.e., the request rate is inversely proportional to $CPI[system]$ providing the negative feedback process in the memory hierarchy and the reason for an iterative calculation). Assuming a constant service time, the total delay for the uniprocessor with tandem caches (3 level hierarchy) case is:

$$\begin{aligned} T_2 &= V_2 \times S_2 \times (1 + \frac{0.5U_2}{1 - U_2}) \\ T_{main} &= V_{main} \times S_{main} \times (1 + \frac{0.5U_{main}}{1 - U_{main}}) \end{aligned} \tag{6.5}$$

And the queue size is:

$$\begin{aligned} Q_2 &= \frac{U_2 - 0.5U_2^2}{1 - U_2} \\ Q_{main} &= \frac{U_{main} - 0.5U_{main}^2}{1 - U_{main}} \end{aligned} \tag{6.6}$$

As the targeted cache is a write-back L1 cache, when a cache miss requires the replacement of a cache line, this cache line must be written to a higher cache level. This is referred as cast-out (CO), and typical values range from 20% to 40% (as reported by [50] and [51]). The use of the cache-based memory copy hardware accelerator influences this parameter. If the `src` address is not present in the cache at the moment of filling the indexing table, it has to be fetched from the main memory. The same situation happens if a miss in the L1 cache also misses in the L2 cache. Both situations will probably evict some cache lines from L2 cache, which will increase the CO.

A memory hierarchy that provides the critical word first is assumed. If the processor requires another word in the cache line that is being transferred at the moment, the processor must wait until it is available. This is referred as trailing-edge (TE) delay, which typically ranges from 10% and 30% (as reported by [50] and [51]). With the use of the cache-based memory copy hardware accelerator, it is necessary to also consider the case of a first read to a copied address, that will always miss in all L1 caches. These accesses have go to the L2 cache, and if another word is requested while the cache line is being transferred, there will be an increase of the TE delay.

With the usage of the cache-based memory copy hardware accelerator, the copied data is also accessible in the L2 cache, as well as the data that was already in the cache before the copy was performed. This means that both `src` and `dst` addresses are available through either the indexing table or the cache itself, without evicting any cache lines. This implies a decrease in the original L2 miss rate (mr_2).

Besides, due to the increase of the cast-outs (CO) of the L2 cache, there will also be an increase in the visitation probability (V) of the main memory. This increase in the V of the main memory due to the presence of the accelerator (that increases the CO of the L2 cache) is modelled through a factor of eviction (FE). Therefore, the visitation probabilities (V) presented in Equa-

tion (6.3), becomes:

$$\begin{aligned} V_2 &= 1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2 + FE \times mr_2}{mr_1} \end{aligned} \tag{6.7}$$

The final model is based on Equation (6.2), where T_k is given by Equation (6.5), U_k is given by Equation (6.4) and V_k is given by Equation (6.7), with $k = \{1, 2, main\}$. It must be noted that the accelerator is not modelled as another server in the system. However, its impact is modelled in the values attributed to the L2 miss rate (mr_2), the visitation probability of the main memory (V_{main}), the cast-outs (CO) and the trailing-edge (TE) delay.

The impact of using the accelerator in a uniprocessor with a L1 cache, a L2 cache and a main memory is studied in the next section, to evaluate the model. Summarizing, the L2 cache and the main memory service times and the cache line sizes utilized in this analytical analysis are the same as the ones used in Simics simulator (in Section 4.2), the miss rates of both caches (mr_1 and mr_2) are also provided by Simics simulator (in Chapter 5) when executing the TCP/IP benchmark and the values of the cast-out (CO) and trailing-edge (TE) delay are given by [50] and [51] (based on real application measurements). The factor of eviction (FE) is related with the accelerator and, because there is no measurements for it, a wide enough range of values are evaluated to determine the impact of utilizing the accelerator.

In a multi-core processor like the targeting one (Figure 6.2), it is also necessary to analyze the impact of shared resources in the system, namely the bus. The utilization of the bus depends of several situations:

- It depends on the number of misses generated by the three cores and their L1 cache, which affects the amount of data transferred on the bus;
- It depends on the cache coherence protocol, i.e., a read miss in one L1 cache that can hit in any other L1 cache and a write that either forces the invalidation or the update of the data in the others L1 cache.

The use of a shared bus changes the previously described visitation probabilities in Equation (6.3). The probability of visitation of the bus is 1, because all misses in any L1 cache imply utilizing the bus. The probability of visiting the L2 cache is now not only dependent on the miss rate of each L1 cache ($OwnL1$), but also on the miss rate of other L1 caches ($OtherL1$) that share

the bus. Besides, the number of hits/misses of a particular cache line in the $OtherL1$ is also dependent on the application. [46] presents some typical values for the hit rate in $OtherL1$, which are in the range of 10% to 50%. This parameter will referred as the hit rate in $OtherL1$ ($HRO1$). Therefore, the new visitation probability (V) for each core in the system is:

$$\begin{aligned} V_{bus} &= 1 \\ V_2 &= 1 - HRO1 - \frac{mr_2}{mr_1} \\ V_{main} &= \frac{mr_2 + FE \times mr_2}{mr_1}, \quad \text{where } FE = 0, \text{ if the accelerator is not used} \end{aligned} \tag{6.8}$$

To determine the service time (S) of the shared bus, there is a need to know the amount of data transferred, the bus cycle time and the bus width. A bus that has separate lines for data and control has two different services times:

$$\begin{aligned} S_{control} &= bus\_cycle\_time \times \frac{control\_request\_size}{bus\_control\_width} \\ S_{data} &= bus\_cycle\_time \times \frac{L1\_cacheline\_size}{bus\_data\_width} \end{aligned} \tag{6.9}$$

Therefore, the total utilization (U) of the shared bus, both control (CB) and data (DB), due to misses in any of the three L1 caches ($OwnL1$), in the other L1 caches ($OtherL1$), and due to the cast-outs (CO) is:

$$\begin{aligned} UOwnL1_{DB} &= V_{bus} \times S_{data} \times R \\ UOwnL1_{CB} &= V_{bus} \times S_{control} \times R \\ UOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \times R \\ UOtherL1_{DB} &= 1/3 \times V_{bus} \times HRO1 \times S_{data} \times R \\ UOtherL1_{CB} &= 1/3 \times V_{bus} \times HRO1 \times S_{control} \times R \end{aligned} \tag{6.10}$$

And the corresponding delay (T) for both control (CB) and data (DB) busses, due to misses in any of the three L1 caches ($OwnL1$), in the other L1 caches

($OtherL1$), and due to the cast-outs (CO) is:

$$\begin{aligned}
TOwnL1_{DB} &= V_{bus} \times S_{data} \times (1 + \frac{0.5UOwnL1_{DB}}{1 - UOwnL1_{DB}}) \\
TOwnL1_{CB} &= V_{bus} \times S_{control} \times (1 + \frac{0.5UOwnL1_{CB}}{1 - UOwnL1_{CB}}) \\
TOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \\
&\quad \times (1 + \frac{0.5UOwnL1_{CO}}{1 - UOwnL1_{CO}}) \\
TOtherL1_{DB} &= 0.5 \times V_{bus} \times HRO1 \\
&\quad \times S_{data} \times (1 + \frac{0.5UOtherL1_{DB}}{1 - UOtherL1_{DB}}) \\
TOtherL1_{CB} &= 0.5 \times V_{bus} \times HRO1 \\
&\quad \times S_{control} \times (1 + \frac{0.5UOtherL1_{CB}}{1 - UOtherL1_{CB}})
\end{aligned} \tag{6.11}$$

For the queue sizes (Q) for both control (CB) and data (DB) busses, due to misses in any of the three L1 caches ($OwnL1$), in the other L1 caches ($OtherL1$), and due to the cast-outs (CO):

$$\begin{aligned}
QOwnL1_{DB} &= \frac{UOwnL1_{DB} - 0.5UOwnL1_{DB}^2}{1 - UOwnL1_{DB}} \\
QOwnL1_{CB} &= \frac{UOwnL1_{CB} - 0.5UOwnL1_{CB}^2}{1 - UOwnL1_{CB}} \\
QOwnL1_{CO} &= \frac{UOwnL1_{CO} - 0.5UOwnL1_{CO}^2}{1 - UOwnL1_{CO}} \\
QOtherL1_{DB} &= \frac{UOtherL1_{DB} - 0.5UOtherL1_{DB}^2}{1 - UOtherL1_{DB}} \\
QOtherL1_{CB} &= \frac{UOtherL1_{CB} - 0.5UOtherL1_{CB}^2}{1 - UOtherL1_{CB}}
\end{aligned} \tag{6.12}$$

The trailing-edge (TE) delay due to other L1 caches ($OtherL1$) and L2 cache ($L2$) is given by:

$$\begin{aligned}
TOtherL1_{TE} &= (\frac{L1_cacheline_size}{bus_data_width} - 1) \\
&\quad TE \times bus_cycle_time \\
TL2_{TE} &= (\frac{L2_cache_line_size}{bus_data_width} - 1) \\
&\quad TE \times bus_cycle_time;
\end{aligned} \tag{6.13}$$

Until now it is assumed a constant service time for each server (M/D/1). As this parameter models the time each resource of the system takes, it is interesting to study the impact of changing this value. Therefore, the following assumes an service time of each server to be exponential (M/M/1). Subsequently, the delay given by Equation (6.5) and Equation (6.11), became:

$$\begin{aligned}
T_2 &= V_2 \times S_2 \times (1 + \frac{U_2}{1 - U_2}) \\
T_{main} &= V_{main} \times S_{main} \times (1 + \frac{U_{main}}{1 - U_{main}}) \\
TOwnL1_{DB} &= V_{bus} \times S_{data} \times (1 + \frac{UOwnL1_{DB}}{1 - UOwnL1_{DB}}) \\
TOwnL1_{CB} &= V_{bus} \times S_{control} \times (1 + \frac{UOwnL1_{CB}}{1 - UOwnL1_{CB}}) \\
TOwnL1_{CO} &= CO \times V_{bus} \times S_{data} \\
&\quad \times (1 + \frac{UOwnL1_{CO}}{1 - UOwnL1_{CO}}) \\
TOtherL1_{DB} &= 0.5 \times V_{bus} \times HRO1 \\
&\quad \times S_{data} \times (1 + \frac{UOtherL1_{DB}}{1 - UOtherL1_{DB}}) \\
TOtherL1_{CB} &= 0.5 \times V_{bus} \times HRO1 \\
&\quad \times S_{control} \times (1 + \frac{UOtherL1_{CB}}{1 - UOtherL1_{CB}})
\end{aligned} \tag{6.14}$$

And the corresponding queue sizes:

$$\begin{aligned}
QOwnL1_{DB} &= \frac{UOwnL1_{DB}}{1 - UOwnL1_{DB}} \\
QOwnL1_{CB} &= \frac{UOwnL1_{CB}}{1 - UOwnL1_{CB}} \\
QOwnL1_{CO} &= \frac{UOwnL1_{CO}}{1 - UOwnL1_{CO}} \\
QOtherL1_{DB} &= \frac{UOtherL1_{DB}}{1 - UOtherL1_{DB}} \\
QOtherL1_{CB} &= \frac{UOtherL1_{CB}}{1 - UOtherL1_{CB}}
\end{aligned} \tag{6.15}$$

Finally, the total delay T_{total} for each core in the multi-core processor is given by the summation of Equation (6.5), Equation (6.11) and Equation (6.13) (with the visitation probability given by Equation (6.8)), for a constant service

time. If a exponential service time is used the total delay (T_{total}) is calculated by the summation of Equation (6.14) and Equation (6.13) (with the visitation probability given by Equation (6.8)). The CPI for each core in the multi-core processor is then given by:

$$CPI[system] = CPI[\infty] + mr_1 \times T_{total} \tag{6.16}$$

Again, the impact of using the accelerator in a multi-core processor each with a L1 cache, a shared L2 cache and a main memory is studied in the next section, to evaluate the model. Summarizing, the bus_data_width, $control\_req\_size$, $bus\_control\_width$ and bus_cycle_time are typical values and the hit rate in $OtherL1$ ($HRO1$) is given by [50] and [51] (based on real application measurements).

6.3 Results

First, the uniprocessor case is analyzed, in order to evaluate the model. Figure 6.3 depicts the impact of increasing the mr_2 and the FE values on the FCP, compared with the baseline scenario described in Table 6.1. The default values depicted in the second column correspond to the typical values used in real systems. In particular, the L2 cache and the memory service times and the cache lines sizes are the same as the used in Chapters 4 and 5 for the Simics simulator; the $HRO1$, CO and TE delay are given by [50] and [51] (based on real application measurements) and the cache miss rates were observed when the accelerator was implemented in Simics simulator. The other parameters are typical values. The FE is related with the accelerator and, because there is no possibility of performing measurements to determine its value, a wide enough range of values is evaluated to determine the impact of utilizing the accelerator. The third column presents the parameter's range used to evaluate the system, as these are the parameters that either depend on the application ($HRO1$), or on the use of the accelerator (mr_2 and FE), as introduced in the previous section. As Figure 6.3 depicts the analysis of the uniprocessor case, the $HRO1$ is zero, as there are no other cores in the system when evaluating a uniprocessor.

It is clear from Figure 6.3 that the accelerator can provide benefits for a wide range of mr_2 and FE values and that, in the best case, those benefits can reach a decrease of FCP of 57.1% (the best FCP is when the value of mr_2 is 0.5% and the value of FE is 1% and is compared with the FCP of the baseline

Parameter	Default Value	Range
L2 Cache Service Time (S_2)	15 clk (also used in Simics)	-
Memory Service Time (S_{main})	240 clk (also used in Simics)	-
L1 Cache Miss Rate (mr_1)	10% (returned by Simics)	-
L2 Cache Miss Rate (mr_2)	3% (returned by Simics)	0.5% to 3%
Hit Rate $OtherL1$ ($HRO1$)	20% (given by [50] and [51])	10% to 50%
Factor of Eviction (FE)	0% (wide range of values)	1% to 40%
Cast-Outs (CO)	20% (given by [50] and [51])	-
Trailing Edge (TE) delay	10% (given by [50] and [51])	-
L1_cacheline_size	32 bytes (typical value)	-
L2_cacheline_size	128 bytes (typical value)	-
bus_data_width	8 bytes (typical value)	-
$control\_req\_size$	1 bytes (typical value)	-
$bus\_control\_width$	1 bytes (typical value)	-
bus_cycle_time	1 clk (typical value)	-

Table 6.1: Parameters defining the system modelled

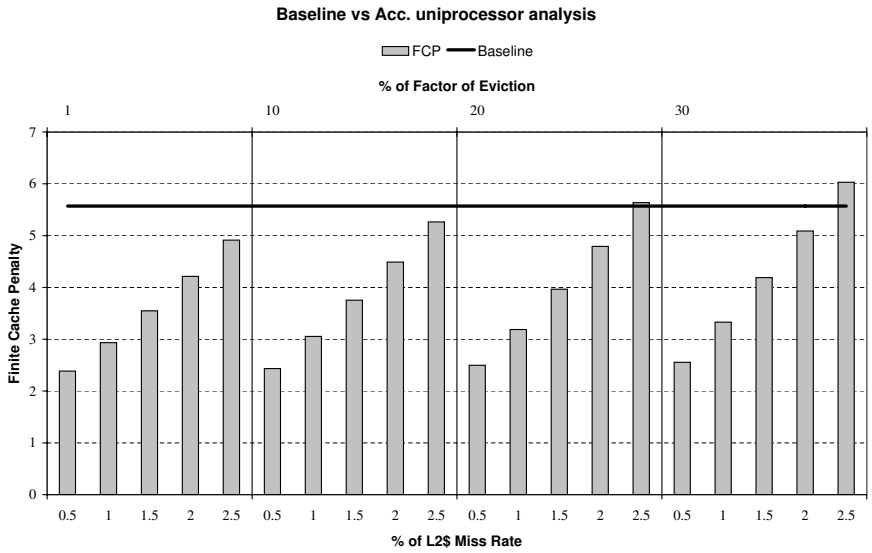


Figure 6.3: Uniprocessor analysis

scenario). However, if the FE and the mr_2 values are too high, then the utilization of the accelerator can actually decrease the performance. On the other hand, this is not a realistic scenario, once the presence of the accelerator will always decrease the value of mr_2. Moreover, the previous section also mentioned the possible influence of the CO value and of the TE delay. Therefore, the value of CO and the TE delay was increased by 20%, which increased the FCP by 1.2% for the first case and 4.5% for the second. Subsequently, these parameters have a small impact in the uniprocessor system.

As mentioned before, one of the parameters of the multi-core processor depends on is the application, modelled through the $HRO1$ value. Therefore, the impact of changing this parameter's values is studied and the benefits of one of the cores in the system (as the model calculates the FCP of one of the cores taking into account the presence of the others) is analyzed. Figure 6.4 depicts the impact of changing the $HRO1$ values, compared with the baseline scenario. As expected, the increase in $HRO1$ value reduces the FCP of the system compared with the baseline scenario.

As demonstrated for the uniprocessor, the usage of the accelerator is modelled mainly through the mr_2 and the FE values. Therefore, the cache-based memory copy hardware accelerator for a multi-core processor is evaluated by analyzing the impact on the FCP that the accelerator have for different para-

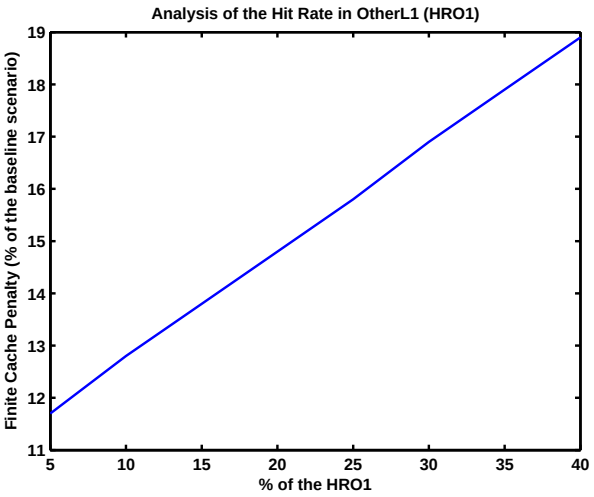


Figure 6.4: Hit Rate in $OtherL1$ ($HRO1$) analysis

meter's values. Figure 6.5 depicts the percentage of FCP decrease compared with the baseline scenario by increasing the mr_2 and the FE values in a multi-core processor, keeping the remainder of the parameters with the default values described in the Table 6.1.

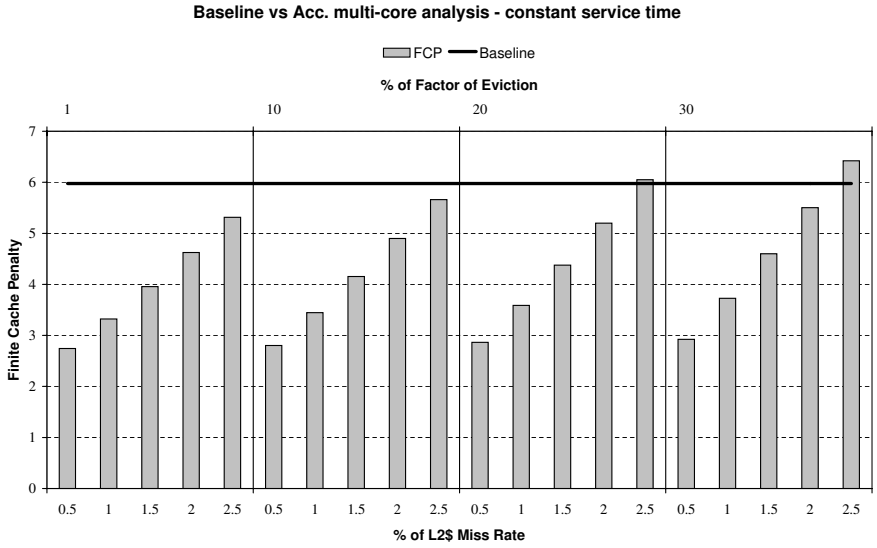


Figure 6.5: Multi-core processor analysis with constant service time

From the previous analysis it is possible to reach a FCP decrease of 54.1% for one of the cores for the best case (the best FCP is when the value of mr_2 is 0.5% and the value of FE is 1% and is compared with the FCP of the baseline scenario). The theoretical worst case is when the value of mr_2 does not decrease and the value of FE is bigger then zero. In such a case, the accelerator will actually have a negative impact on the system. However, as mentioned for the uniprocessor case, this is not a realistic scenario once the presence of the accelerator will always decrease the value of mr_2.

Figure 6.6 depicts the same analysis, now assuming a exponential service time for each server. For this case it is possible to reach a FCP decrease of

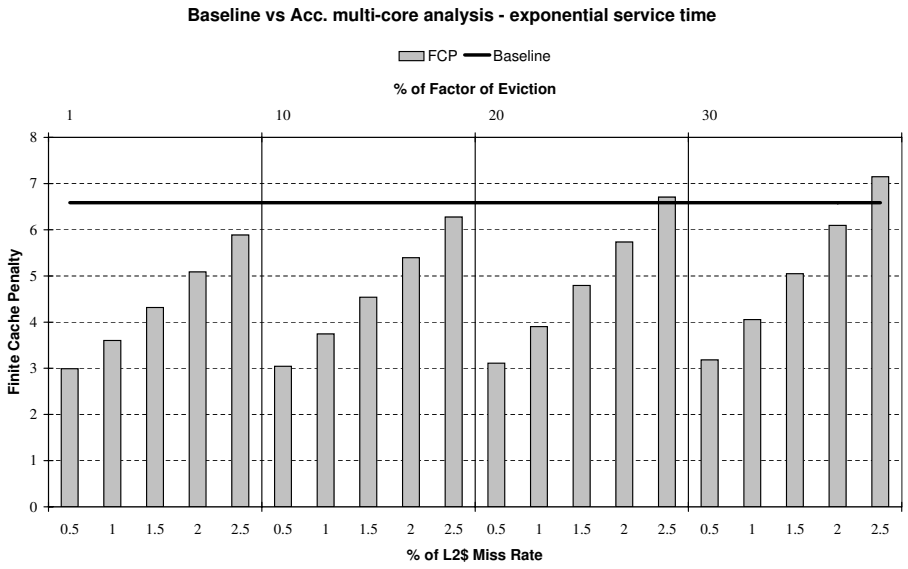


Figure 6.6: Multi-core processor analysis with exponential service time

54.6% for one of the cores for the best case (the best FCP is when the value of mr_2 is 0.5% and the value of FE is 1% and is compared with the FCP of the baseline scenario). Moreover, an exponential service time imposes increase on the baseline FCP by 9.2%, compared with the constant service time. However, the different service times in the servers has a smaller impact on the usage of the accelerator, approximately 7.9%. Moreover, the impact on using an exponential service time for the cache-based memory copy hardware accelerator is bigger for higher mr_2.

As mentioned in the previous section, the other parameters that also have impact in the system are the CO value and the TE delay. However, as demon-

strated for the uniprocessor case, these have small impact on the system.

6.4 Summary

This chapter presented the applicability of the cache-based memory copy hardware accelerator, with the indexing table able to support set-associative caches, to a multi-core processor utilizing a message passing communication model. An analytical analysis based on open-queues theory was used to evaluate the cache-based memory copy hardware accelerator. The parameters that have higher impact on the system's performance are the second level cache miss rate and the cache line eviction. The utilization of the accelerator can provide up to 50% reduction on the average number of cycles executed per instruction for one of the cores in the system (which takes into account the presence of the other cores of the system) for the best case.

Next chapter will conclude this dissertation and present future work.

Chapter 7

Conclusions

This dissertation introduced the memory copy operation and its bottleneck. Subsequently, a cache-based memory copy hardware accelerator that is able to reduce the impact of such operation in a computer system is presented. The accelerator was implemented in reconfigurable hardware to demonstrate its feasibility and performance gains. The performance gains were incorporated in a simulator to further investigate applications that utilize the memory copy operation.

In this chapter, concluding remarks are provided, the major achievements of the investigation are presented and possible future research directions are highlighted. This chapter is organized as follows. In Section 7.1, a summary of the main conclusions of this dissertation is presented. In Section 7.2, the major contributions described in the dissertation are listed and finally, in Section 7.3, several possible future research directions are presented.

7.1 Summary

In Chapter 1 it is argued that memory copies are a bottleneck in several systems (uniprocessor and multiprocessor) and for different standards (networking standards based on TCP/IP and the message passing communication model). Moreover, the proposals in related works to solve this bottleneck still cannot provide the necessary performance in some circumstances or can only be applied to a limited number of cases.

Furthermore, it was identified that the processor's evolution have increasingly taken benefit of the presence of caches, intended to reduce the "processor-

memory performance gap". As caches store the most recently used data, it is only logical to assume that data to-be-copied by a memory copy is already present within the cache. This is especially true when considering that such data often must be processed first. Therefore, a solution to solve the memory copy operation bottleneck taking advantage of the presence of caches is logical to pursue.

In Chapter 2, the general concepts and implementation tradeoffs of caches were introduced. Caches are used to reduce the average access time to main memory and, therefore, try to reduce the "processor-memory performance gap". By carefully choosing the associativity, the write policies, the cache size vs distance to the processor and the replacement policy, the hit rate of the cache can be improved, bringing better performance to the whole system.

Moreover, several memory data movements performed in software were explained in detail. As memory data movements move or copy data from one memory location to another, they are bounded by the access time to main memory and the "processor-memory performance gap". As the processor speeds are increasing faster than the memory access times, the impact of these memory data movements are expected to increase in the future.

As the cache-based memory copy hardware accelerator was prototyped utilizing the Xilinx tools, the platforms and intellectual property blocks were also explained in this chapter. Moreover, the performance of the cache-based memory copy hardware accelerator was also evaluated utilizing Simics full-system simulator, therefore, a high level description of the simulator was also introduced in this chapter.

Chapter 3 presented the concept of the cache-based memory copy hardware accelerator, as a combination of a cache (independent of its organization), an indexing table (independent of the copy granularity it supports) and a load/store unit. Moreover, it discussed the design of the indexing table able to support cache line and word granularity copy and set-associative caches.

The cache-based memory copy hardware accelerator does not incur in performance penalty on a read hit on the cache and read miss on the indexing table, however on a read hit on the indexing table there is a need of one more clock cycle to access the copied data. Furthermore, the options to communicate with the accelerator (from the software and hardware points of view) were presented as well as an analytical study was performed to estimate the necessary hardware resources need to implement the cache-based memory copy hardware accelerator.

Chapter 4 introduced the platforms used to implement the cache-based

memory copy hardware accelerator. The details of each platform and of the implementation of the cache-based memory copy hardware accelerator were presented. The implementation constrains that drive the performance of the cache-based memory copy hardware accelerator were also introduced. In particular, the need of a new field in the indexing table, the "Lookup" field, is introduced. Moreover, the impact of this field in the complete design is also discussed. The resource utilization on the prototyping platforms were presented, as well as a study of the number of bits stored in the indexing table and in the cache (to determine the increase in hardware resources due to the utilization of the indexing table). Finally, the modelling of the cache-based memory copy hardware accelerator in the simulator was also presented.

Chapter 5 presented the results of the synthetic benchmarks used to demonstrate the cache-based memory copy hardware accelerator on the prototyping platform. The indexing table is able to perform a copy in 28 clock cycles plus 2 clock cycles per cache line or word (depending on the indexing table utilized). Moreover, accessing the accelerator through a memory-mapped device-driver has impact on the performance of the system, as demonstrated by the performance study on the simulator. When the access is performed through an instruction-set architecture extension, a speedup ranging from 2.96 to 4.61 times for the receiver side TCP/IP benchmark is achieved, when utilizing the cache-based memory copy hardware accelerator compared with the software implementation of the benchmark. Moreover, there is also a reduction on the number of instructions executed ranging from 26% to 44%, due to removing the loads and stores needed to execute the memory copy operation in software.

Chapter 6 presented the applicability of the cache-based memory copy hardware accelerator, with an indexing table able to support set-associative caches, to a multi-core processor utilizing a message passing communication model. An analytical analysis based on open-queues theory was used to evaluate the cache-based memory copy hardware accelerator. The parameters that have higher impact on the system's performance are the second level cache miss rate and the cache line eviction. The accelerator can provide up to 50% reduction on the average number of cycles executed per instruction for one of the cores in the system (which takes into account the presence of the other cores of the system) for the best case.

The answers to the research questions presented in Section 1.4, can therefore be summarized as follows:

- Can the presence of caches in today's processors be exploited to solve

the memory copy bottleneck? Yes, by introducing a redirection mechanism that links the original data (in the cache) to the copied addresses (in a newly added indexing table), the memory copy operation can be performed in much shorter time. Moreover, the ensuing access(es) to all copied data can be performed much faster, the cache pollution existent in the traditional way of performing a memory copy no longer exists with the cache-based memory copy hardware accelerator. Furthermore, the cache-based memory copy hardware accelerator can deal efficiently with overlapping of the original and copied data memory locations.

- How do the proposed solutions in this dissertation perform compared with existing approaches? The raw performance of the indexing table is able to perform a memory copy in 28 clock cycles plus 2 clock cycles per cache line or word (depending on the granularity of the indexing table utilized). Moreover, for the receiver side of the TCP/IP stack benchmark, the cache-based memory copy hardware accelerator can achieve a speedup ranging from 2.96 to 4.61 times, while reducing the number of instructions executed by 26% to 44%, when compared with the software implementation of the benchmark.

- How can the proposed solutions in this dissertation be adapted to support multiprocessor platforms? An indexing table that is able to support set-associative caches and cache coherence protocols was also presented in this dissertation. Moreover, an analytical analysis based on open-queues theory was presented. It demonstrated that the cache-based memory copy hardware accelerator can provide up to 50% reduction on the average number of cycles executed per instruction for one of the cores in the system (which takes into account the presence of the other cores of the system) for the best case.

7.2 Main Contributions

In this section, the main contributions of the research described in this dissertation are highlighted:

- A hardware unit that performs memory copy operations using an additional indexing table in an existing cache organization, was introduced. The use of an indexing table connected to the cache avoids duplicating data in caches, because the copy (of the original data) is simply represented by inserting an additional pointer to the original data that is

already present in the cache. This pointer allows the copied data to be accessed from the cache. The indexing table also offloads the processor as it is no longer required to perform the copies byte by byte (or the largest data unit the utilized architecture supports). Moreover, it can support the case where the original and the copied addresses overlap, which has a big performance degradation on the software implementation of the memory copy operation.

- An indexing table able to support cache lines and word granularity copy was also presented. The indexing table performs a memory copy operation of one cache line in 30 clock cycles and performs memory copies of cache lines sizes varying from 32 bytes to 256 bytes (which are common values for cache lines), 79% to 93% faster than an optimized software implementation, respectively. Moreover, the indexing table also performs a memory copy of one word and of eight words (one cache line), 66% and 69% faster than an optimized software implementation, respectively.

- A new load/store unit attached to the cache, the indexing table and the main memory controller were presented. The unit is able to autonomously load/store data from/to the main memory to/from cache necessary to perform the memory copy operation, when the data was not present in the cache. Moreover, as the size of the to-be-copied data is known beforehand, the new unit is able to fully utilize the available bandwidth between the cache and the main memory. In addition, this approach allows the load latency to be reduced. A synthetic benchmark based on the reassembly of a Bluetooth frame was utilized to demonstrate that the cache-based memory copy hardware accelerator (the cache, the indexing table and the load/store unit) provides a speedup of approximately 7 times, for the experimented benchmark, compared to an optimized (hand-coded in assembly) software solution. It was also presented a comparison between the worst case utilizing the accelerator with the best case for the software solution that demonstrated the accelerator brings increasing benefits for bigger number of cache lines copies.

- The integration, through a memory-mapped device-driver, of the cache-based memory copy hardware accelerator in a complete computer system was also presented. The results of executing two benchmarks suites, LMbench and STREAM were introduced and demonstrated that the utilization of the accelerator can provide up to 121 times average execution

time speedup for copies of 32kB, when compared with the execution of the benchmarks without the accelerator. The impact of changing the processor frequency, the memory latency and the cache line size was analyzed and the conclusions reached was that the cache-based memory copy hardware accelerator is access bounded and not copy bounded.

- Subsequently, an instruction-set architecture extension by one instruction was introduced. It demonstrated that the cache-based memory copy hardware accelerator is able to provide 4.61 times speedup for the receiver side of the TCP/IP stack and 1.2 times speedup of raw throughput for the STREAM benchmark, when compared with the execution of the benchmarks without the accelerator

- Finally, the cache-based memory copy hardware accelerator was integrated in a multi-core processor supporting message passing communication model. The indexing table able to support set-associative caches and the cache coherence protocol was coupled with a set-associative shared second level cache. The benefits were demonstrated through an analytical analysis based on open queues theory. The parameters that have higher impact on the system's performance are the second level cache miss rate and the cache line eviction. The utilization of the accelerator can provide up to 50% reduction on the average number of cycles executed per instruction for one of the cores in the system (which takes into account the presence of the other processors of the system) for the best case.

7.3 Future Research Directions

The following propose future research directions and improvements to the work presented in this dissertation:

- In this dissertation, a cache-based memory copy hardware accelerator was developed to particularly accelerate the `memcpy` or `bcopy` functions. However, functions that copy memory regions and stop after encountering a particular occurrence (i.e., `memccpy`), or functions that set a particular memory region to a predefined value (i.e., `memset` or `bzero`) are also common. The cache-based memory copy hardware accelerator can easily be adapted to perform these other memory related functions. In order to support such functionalities, the cache-based

memory copy hardware accelerator needs to have a new load/store unit that is able to read the data that is being loaded from the main memory to perform the memory copy (if the data is not in the cache) or being able to read the contents of the cache (if the data is already in the cache) to identify the occurrence of a particular value or to set the memory region to a particular value.

- In this dissertation, a study of the applicability of the cache-based memory copy hardware accelerator in a multi-core processor was performed. However, this study was performed using an analytical analysis. In order to study more accurately the benefits that the cache-based memory copy hardware accelerator can have in a multi-core processor, it is interesting to test the proposed system in a platform. This may be performed by using an extension to the Simics simulator (that includes detailed memory hierarchy simulation for multi-core processors) or by implementing a multi-core processor in a FPGA and connect the cache-based memory copy hardware accelerator. However, by performing this last option it is envisioned to encounter problems due to lack of physical space in the device to implement two L1 caches (one for each PPC in the platform), one shared L2 cache, a shared bus to connect both L1 caches and the L2 cache and finally the cache-based memory copy hardware accelerator.

- In this dissertation, a cache-based memory copy hardware accelerator that effectively reduces the number of accesses to the main memory was presented. Moreover, because the copy addresses are sequential and the size of the copy is known in advance, those accesses to the main memory are sequential and can be performed taking advantage of the burst mode present in today's memories. This implies that the address decoder of the memory controller needs to perform much less decoding operations to write a memory copy performed with the accelerator. Furthermore, because memory accesses are power hungry (in particular due to the decoding operation), it is expected that the cache-based memory copy hardware accelerator can also provide reduction in the power consumption of a system. An interesting research direction would be to study the power consumption associated with a memory copy performed in the traditional software way and compare it with a memory copy performed with the cache-based memory copy hardware accelerator.

- In this dissertation, a load/store unit to load and store the data to/from the main memory was presented. This unit performs the loading of the data to the cache (if the data is not there yet) at the same time the index-

ing table is being filled. However, some processor architectures support prefetching which allows to load the data in advance, much before being needed. Such a functionality is supposed to bring benefits to the cache-based memory copy hardware accelerator, once it might reduce the loading time of the data to the cache before performing the copy. A performance study with an architecture that supports prefetching can demonstrate the benefits to the cache-based memory copy hardware accelerator.

- A technique that has became increasingly popular within multi-processor systems is the transactional memory model. A transaction is a sequence of instructions that are guaranteed to execute and complete only as an atomic unit. Each transaction produces a block of writes which are committed to shared memory only as an atomic unit, after the transaction completes execution. Once the transaction is complete, the hardware must arbitrate system-wide for the permission to commit its writes. After this permission is granted, the processor can take advantage of high system interconnect bandwidths to simply broadcast all writes for the entire transaction out as one large packet to the rest of the system. Snooping by other processors on these store packets maintains coherence in the system, and allows them to detect when they have used data that has subsequently been modified by another transaction and must rollback. A study of a possible integration of the transactional memory model with the cache-based memory copy hardware accelerator is a very interesting research direction to pursue.

Appendix A

Bluetooth Profiling

The Linux OS and the BlueZ [7] stack (the official Bluetooth standard [6] implementation for Linux OS) were installed in two systems the in order to retrieve the profiling information of the Bluetooth standard. The systems used to perform the experiments are the following:

- A desktop Intel Pentium 4 (at 2.80 GHz) running Linux 2.4.22.
- The Xilinx ML310 Embedded Development Platform [56] with a Virtex II Pro FPGA containing two Power PCs 405 (at 300 MHz). The Linux MontaVista 2.4.24 [60] is running only on one of the Power PCs (the other is not used).

One of the most interesting applications (profiles) that can be executed over Bluetooth is 'file transfer'. As expected, 'file transfer' will also utilize part of the TCP/IP stack of Linux. The Bluetooth USB adapter from Conceptronic [17] was used to provide the necessary radio connection between the systems. In order to do a 'file transfer', a file of 50 MB was created and transferred between the Bluetooth devices.

The profiler used to retrieve the profiling information is part of the Linux kernel, and the application that is able to interpret that information is also a kernel build-in application, `readprofile`. To identify the Bluetooth functions, the Bluetooth stack has to be statically linked into the ML310 platform Linux kernel. It is important to mention that the Bluetooth standard is executed on top of the Linux OS, which is not a deterministic application. Depending on the state of the system, the OS will react differently. Therefore, in order to retrieve meaningful data, 20 identical and independent trials of the same

experiment were performed. The trials were performed without rebooting the system and the profiler was restarted for each trial.

In Table A.1, the first 40 rows of the profiling information of one of these trials, sorted by number of ticks (N.Ticks), are presented. The N.Ticks is a value returned by the profiler representing the number of times a particular function was running on the processor when the system was interrupted by the profiler to retrieve the profiling information.

Table A.2 presents the first 40 rows sorted by Normalized Load, which is another value returned by the profiler. It is calculated by dividing the number of ticks recorded by the profiler for a function by the function size (i.e., length of the address space in memory) [97]. Therefore, it is safe to assume that functions rating high in both lists (Table A.1 and A.2) occupy the processor and are, therefore, the most time-consuming.

From Tables A.1 and A.2, eight Bluetooth functions can be identified: `hci_usb_rx_complete`, `l2cap_recv_acldata`, `hci_usb_rx_submit`, `hci_send_to_sock`, `hci_rx_task`, `hci_acldata_packet`, `hci_recv_frame`, `bnep_rx_frame`, and a mixture of OS and TCP/IP functions. Moreover, in the majority of time, the processor is controlling interrupts (e.g., `__sti_end`, `__sti`, which occupied 95% of the time), their execution (e.g., `hc_interrupt`, `speedo_interrupt`, `trident_interrupt`) or copying data within the main memory (e.g., `memcpy`). The interrupt-related functions are the most time-consuming ones, however they are less good candidates to be implemented on an accelerator. The reason for this is because each time an interrupt is issued, the way the OS deals with it is depended of the state of the system. Therefore, the executed code may be slightly different each time the same interrupt is issued.

A comparison of the number of ticks of the Bluetooth functions and the `memcpy` function is depicted in Figure A.1. It is clear that `memcpy` is more time-consuming than any other Bluetooth function.

N.Ticks:	Function Name:
69274	__sti_end
2247	__sti
1136	__save_flags_ptr_end
153	memcpy
150	hc_interrupt
138	speedo_interrupt
127	trident_interrupt
126	XSysAce_RegRead32
81	sohci_submit_urb
68	__copy_tofrom_user
67	XSysAce_RegWrite32
64	dl_done_list
62	hci_usb_rx_complete
61	l2cap_recv_acldata
61	dl_transfer_length
59	tcp_recvmsg
52	hci_usb_rx_submit
48	sys_select
48	kmalloc
47	hci_send_to_sock
45	__kfree_skb
44	hci_rx_task
43	skb_under_panic
41	fget
39	normal_poll
38	do_select
37	__kmem_cache_alloc
36	DoSyscall
35	tty_poll
35	power_save
35	__save_flags_ptr
33	__free_pages_ok
32	tcp_rcv_established
31	hci_recv_frame
31	bnep_rx_frame
30	tcp_poll
30	hci_acldata_packet
29	tasklet_action
28	tcp_v4_rcv

Table A.1: The first 40 rows of the profiler sorted by number of ticks

Normalized Load:	Function Name:
2164.8125	__sti_end
28.0875	__sti
11.3600	__save_flags_ptr_end
1.5000	XSysAce_RegRead32
1.2885	XSysAce_RegWrite32
0.9808	memcpy
0.6029	fget
0.6000	kmalloc
0.5040	trident_interrupt
0.4375	__save_flags_ptr
0.3913	DoSyscall
0.3074	hc_interrupt
0.3017	power_save
0.2315	speedo_interrupt
0.1906	dl_transfer_length
0.1862	tty_poll
0.1616	dl_done_list
0.1449	hci_usb_rx_complete
0.1318	tasklet_action
0.1310	hci_rx_task
0.1230	hci_recv_frame
0.1206	__copy_tofrom_user
0.1160	__kfree_skb
0.1066	hci_usb_rx_submit
0.1048	normal_poll
0.0934	__kmem_cache_alloc
0.0882	hci_acldata_packet
0.0822	hci_send_to_sock
0.0786	l2cap_recv_acldata
0.0714	do_select
0.0714	tcp_poll
0.0694	skb_under_panic
0.0671	sohci_submit_urb
0.0441	sys_select
0.0326	__free_pages_ok
0.0280	bnep_rx_frame
0.0273	tcp_recvmsg
0.0180	tcp_v4_rcv
0.0149	tcp_rcv_established

Table A.2: The first 40 rows of the profiler sorted by Normalized Load

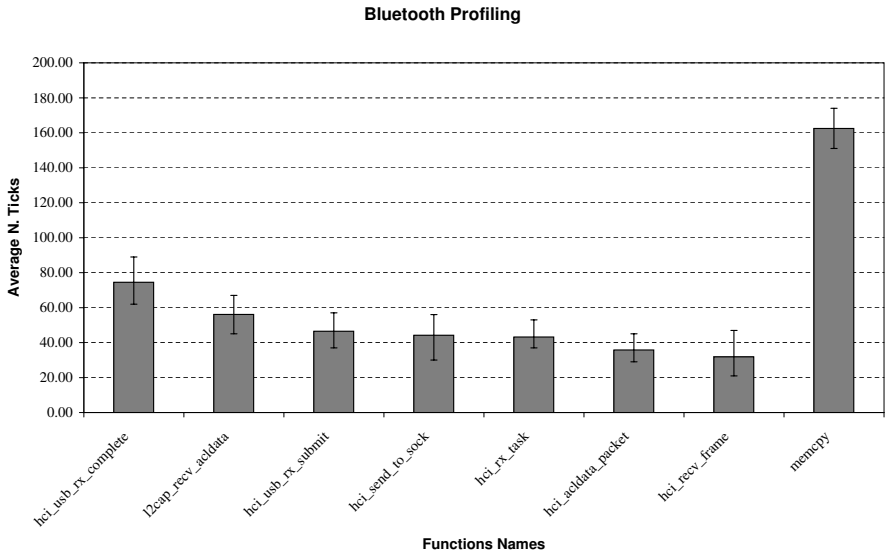


Figure A.1: Comparison of the average N.Ticks for different functions

Bibliography

[1] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[2] Enhanced TCP/IP Performance with AltiVec.
`http://www.freescale.com/AltiVec`.

[3] ASIC to FPGA Design Methodology and Guidelines.
`http://www.altera.com/literature/an/an311.pdf`.

[4] Beowulf Class Cluster Computers.
`http://www.beowulf.org/`.

[5] P. Bhagwat. Bluetooth: Technology for Short-Range Wireless Apps. *IEEE Internet Computing*, 5(3):96–103, May 2001.

[6] Bluetooth specification v1.2.
`http://www.bluetooth.org/spec/`.

[7] BlueZ.
`http://www.bluez.com`.

[8] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo: a Full System Simulator for the PowerPC Architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[9] M.M. Buddhikot, X.J. Chen, W. Dakang, and G.M. Parulkar. Enhancements to 4.4 BSD UNIX for efficient networked multimedia in project MARS. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 326, 1998.

[10] D. Buntinas, G. Mercier, and W. Gropp. Data Transfers between Processes in an SMP System: Performance Study and Application to MPI. In *Proceedings of the 2006 International Conference on Parallel Processing*, pages 487–496, 2006.

[11] V. G. Cerf and R. E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):627–641, May 1974.

[12] C-H. Chang, D. Flower, J. Forecast, H. Gray, B. Hawe, A. Nadkarni, K. K. Ramakrishnan, U. Shikarpur, and K. Wilde. High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. In *Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing*, pages 36–42, 1994.

[13] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End-System Optimizations for High-Speed TCP. *IEEE Communications Magazine*, 39(4):68–74, April 2001.

[14] H. K. Jerry Chu. Zero-copy TCP in Solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[15] Giuseppe Ciaccio. Using a Self-connected Gigabit Ethernet Adapter as a memcpy() Low-Overhead Engine for MPI. In *Proceedings of the Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 247–256, 2003.

[16] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[17] Bluetooth USB Adapter. `http://www.conceptronic.net`.

[18] C. J. Conti, D. H. Gibson, and S. H. Pitkowsky. Structural Aspects of the System/360 Model 85, Part I: General Organization. *IBM Systems Journal*, 7(1):2–15, 1968.

[19] Cray T3D System Architecture. Cray Research, Inc.

[20] B. Donchev, G.K. Kuzmanov, and G. N. Gaydadjiev. External Memory Controller for Virtex II Pro. In *Proceedings of the International Symposium on System-on-Chip*, pages 37–40, 2006.

[21] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network Subsystem Design. *IEEE Network*, 7(4):8–17, July 1993.

[22] F. Duarte, S.D.Breijer, and S. Wong. Supporting the Linux Operating System on the MOLEN Processor Prototype. In *Proceedings of the Annual Workshop on Circuits, Systems and Signal Processing*, pages 205–210, 2007.

[23] F. Duarte and S. Wong. Profiling Bluetooth and Linux on the Xilinx Virtex-II Pro. In *Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 229–235, 2006.

[24] Xilinx EDK. `http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm`.

[25] Free Berkeley Software Distribution (FreeBSD). `http://www.freebsd.org/`.

[26] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory*. John Wiley and Sons, Inc., 1998.

[27] Jim Handy. *The Cache Memory Book*. Academic Press, Inc., 1993.

[28] J. Heinlein, K. Gharachorloo, R. Bosch, M. Rosenblum, and A. Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 18–27, 1997.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[30] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 50–59, 2005.

[31] H.Y. Hwang and J.J.J. Shann. An X86 Load/Store Unit with Aggressive Scheduling of Load/Store Operations. In *Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, pages 496–503, 1998.

[32] IEEE Std 1003.1. `http://www.unix.org/version3/ieee_std.html`.

[33] Internet Systems Consortium.
`http://www.isc.org/`.

[34] Xilinx ISE.
`http://www.xilinx.com/ise/logic_design_prod/webpack.htm`.

[35] David Kammer, Gordon McNutt, Brian Senese, and Jennifer Bray. *Bluetooth Application Developer's Guide: The Short Range Interconnect Solution*. Syngress Publishing, Inc., 2002.

[36] J. Kay and J. Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, December 1996.

[37] Yousef A. Khalidi and Moti N. Thadani. An Efficient Zero-Copy I/O Framework for UNIX. Technical report tr-95-39, Sun Microsystems, Inc., Mountain View, CA, USA, 1995.

[38] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. *ACM SIGCOMM Computer Communication Review*, 25(4):87–98, October 1995.

[39] L. Kleinrock. *Queueing Systems, Vol. I: Theory, Vol. II: Computer Applications*. John Wiley and Sons, Inc., 1975.

[40] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, D. Nakahira J. Chapin, M. Horowitz J. Baxter, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, 1994.

[41] S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, Inc., 1983.

[42] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitive System Performance: Computer System Analysis using Queuing Network Models*. Prentice-Hall, Inc., 1984.

[43] C. Lemuet, W. Jalby, and S. Touati. Improving Load/Store Queues Usage in Scientific Computing. In *Proceedings of the 2004 International Conference on Parallel Processing*, pages 38–45, 2004.

[44] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):358–368, May 2007.

[45] J. S. Liptay. Structural Aspects of the System/360 Model 85Part II: The cache. *IBM Systems Journal*, 7(1):15–22, 1968.

[46] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Comercial Marketplace. *ACM SIGARCH Computer Architecture News*, 24(2):308–317, May 1996.

[47] P. Mackerras. Low-Level Optimizations in the PowerPC Linux Kernels. In *Proceedings of the Linux Symposium*, pages 304–314, 2003.

[48] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[49] S. Makeni and R. Iver. Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads. In *Proceedings of the 2003 IEEE International Workshop on Workload Characterization*, pages 33–41, 2003.

[50] R. E. Matick. Comparison of Analytic Performance Models using Closed Mean-Value Analysis versus Open-Queuing Theory for Estimating Cycles per Instruction of Memory Hierarchies. *IBM Journal of Research and Development*, 47(4):495–517, July 2003.

[51] R. E. Matick, T. J. Heller, and M. Ignatowski. Analytical Analysis of Finite Cache Penalty and Cycles per Instruction of a Multiprocessor Memory Hierarchy using Miss Rates and Queuing Theory. *IBM Journal of Research and Development*, 45(6):819–842, November 2001.

[52] J. D. McCalpin. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers. In *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[53] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 162–167, 2004.

[54] L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. In *Proceedings of the Annual Technical Conference on USENIX*, page 23, 1996.

[55] Message Passing Interface Specification. `http://www.mpi-forum.org/docs/`.

[56] Xilinx ML310 Development Platform. `http://www.xilinx.com/products/boards/ml310/current`.

[57] Virtex-4 ML410 Embedded Platform . `http://www.xilinx.com/products/devkits/HW-V4-ML410-UNI-G.htm`.

[58] ModelSim™ Xilinx Edition III. `http://www.xilinx.com/ise/verification/mxe_details.html`.

[59] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[60] MontaVista. `http://www.mvista.com`.

[61] Tali Moreshet and R. Iris Bahar. Effects of speculation on performance and issue queue design. *IEEE Transactions on Very Large Scale Integration Systems*, 12(10):1123–1126, October 2004.

[62] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache Behavior of Network Protocols. *SIGMETRICS Performance Evaluation Review*, 25(1):169–180, June 1997.

[63] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1589:533–546, 1999.

[64] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, page 174, 2003.

[65] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *Proceedings of the 12th International Conference on Supercomputing*, pages 243–250, 1998.

[66] T. Ohneda, M. Kondo, M. Imai, and H. Nakamura. Design and evaluation of high performance microprocessor with reconfigurable on-chip memory. In *Proceedings of the Asia-Pacific Conference on Circuits and Systems*, pages 211–216, 2002.

[67] L. L. Peterson P. Druschel. Fbufs: a High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium of Operating Systems Principles*, pages 189–202, 1993.

[68] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 15–28, 1999.

[69] C. Papadopoulos and G. M. Parulkar. Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(2):199–216, April 1993.

[70] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 Conference on Parallel Processing*, pages 764–771, 1985.

[71] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *Lecture Notes in Computer Science*, 1388:472–485, 1998.

[72] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. *IEEE Micro*, 24(1):24–31, January 2004.

[73] G. Reignier, S. Makineni, R. Illikkal, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. *IEEE Computer*, 37(11):48–58, November 2004.

[74] RFC 4296 - The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols. `http://tools.ietf.org/html/rfc4296`.

[75] RFC 760 - Internet Protocol.
`http://tools.ietf.org/html/rfc760`.

[76] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: the SimOS Approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995.

[77] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond SoftNet. In *Proceedings of the 5th Annual Conference on Linux Showcase & Conference*, page 18, 2001.

[78] H. Shan and J. P. Singh. A Comparison of MPI, SHMEM and Cache-Coherence Shared Address Space Programming Models on Tightly-Coupled Multiprocessors. *International Journal of Parallel Programming*, 29(3):283–318, May 2001.

[79] SHMEM Specification.
`http://www.sgi.com/products/software/mpt/`.

[80] SimpleScalar.
`http://www.simplescalar.com/`.

[81] P. Steenkiste. Design, Implementation, and Evaluation of a Single-Copy Protocol Stack. *Software-Practice and Experience*, 28(7):749–772, June 1998.

[82] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout. The implementation of the Cm* multi-processor. In *Proceedings of the AFIPS National Computer Conference*, pages 645–655, 1977.

[83] H. Tezuka, F.O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 308–315, 1998.

[84] The GNU Compiler Collection.
`http://gcc.gnu.org`.

[85] The STREAM Benchmark Reference Information.
`http://www.cs.virginia.edu/stream/`.

[86] Factorization of Device Driver Code between Kernel and User Spaces.
`http://pages.cs.wisc.edu/~arinib/report.pdf`.

[87] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2007.

[88] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Moscu Painainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.

[89] VHDL IEEE Std 1076.
`http://www.eda.org/vhdl-200x/`.

[90] V.Groza and R. Abielmona. What next? A Hardware Operating System? In *Proceedings of the 21st IEEE Instrumentation and Measurement Technology Conference*, pages 1496–1501, 2004.

[91] E. Vlachos. Design and Implementation of a Coherent Memory Sub-System for Shared Memory Multiprocessors. Technical report, Computer Architecture & VLSI Systems Laboratory, University of Crete, Crete, 2006.

[92] R. J. Walsh. DART: Fast Application-Level Networking via Data-Copy Avoidance. *IEEE Network*, 11(4):28–38, July 1997.

[93] G. Wigley and D. Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–250, 2001.

[94] G. Wigley and D. Kearney. Research Issues in Operating Systems for Reconfigurable Computing. In *Proceedings of the International Conference Engineering of Reconfigurable Systems and Algorithms*, pages 10–16, 2002.

[95] R. Wolf. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, Inc., 1989.

[96] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, 2000.

[97] M. Wong. Stressing Linux with Real-World Workloads. In *Proceedings of the Linux Symposium*, pages 495–504, 2003.

[98] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural support for Programming Languages and Operating Systems*, pages 219–229, 1994.

[99] Z.Z. Wu and H.C. Chen. Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet. In *Proceedings of the 15th International Conference on Computer Communications and Networks*, pages 245–250, 2006.

[100] PLB vs. OCM Comparison Using the Packet Processor Software. `http://www.xilinx.com/bvdocs/appnotes/xapp644.pdf`.

[101] Xilinx Inc. `http://www.xilinx.com`.

[102] Single-Port Block Memory Core v6.2. `http://www.xilinx.com/ipcenter/catalog/logicore/docs/sp_block_mem.pdf`.

[103] Dual-Port Block Memory Core v6.3. `http://www.xilinx.com/ipcenter/catalog/logicore/docs/dp_block_mem.pdf`.

[104] Content Addressable Memory v5.1. `http://www.xilinx.com/ipcenter/catalog/logicore/docs/cam.pdf`.

[105] Data Side OCM Bus v1.0 (v2.00a). `http://www.xilinx.com/bvdocs/ipcenter/data_sheet/dsocm.pdf`.

[106] Xilinx University Program - V2P. `http://www.xilinx.com/univ/xupv2p.html`.

[107] J. Yang and R. Gupta. Energy-Efficient Load and Store Reuse. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 72–75, 2001.

[108] C. Zhang, F. Vahid, and W. Najjar. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 87–91, 2003.

[109] L. Zhao, L. Bhuyan, R. Iyer, S. Makineni, and D. Newell. Hardware Support for Accelerating Data Movement in Server Platform. *IEEE Transactions on Computers*, 56(6):740–753, June 2007.

[110] L. Zhao, R. Illikkal, S. Makineni, and L. Bhuyan. TCP/IP Cache Characterization in Commercial Server Workloads. In *Proceedings of the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads*, page 0, 2004.

[111] B. Zhou, W. Qiu, and C. Peng. An Operating System Framework for Reconfigurable Systems. In *Proceedings of the 5th International Conference on Computer and Information Technology*, pages 781–787, 2005.

Curriculum Vitæ

Filipa Duarte was born in Coimbra, Portugal, on the 28^{th} of July 1976. In 1991, she started her secondary education at the High School N^o 1 of Aveiro, that she finished in 1994 with the Technical Diploma in Electronics. In September 1994, she started her university studies at the University of Aveiro. In 1999, she finished her Licentiate in Engineering of Electronics and Telecommunications and enrolled in the Master in Engineering of Electronics and Telecommunications at the same university, that she finished in 2002. During her Master, she taught several subjects at both the University of Aveiro or at the Polytechnic School of Technology of Castelo Branco.

In September 2002, she moved to the Netherlands to be a Portuguese trainee at the European Space Research & Technology Center of European Space Agency, working for the Galileo Project. In May 2004, she enrolled in her Ph.D. studies at the Computer Engineering Laboratory, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, under the supervision of Prof. Dr. S. Vassiliadis and Dr. Ir. J.S.S.M. Wong. During her Ph.D. studies, she was a junior researcher in the European Project VISIONS, she was involved in several teaching activities and was invited to be Session Chair at ASAP'07 conference.

Her research interests include: computer architecture, application-specific processors, hardware accelerators, memory hierarchy and reconfigurable computing.

Publications

Conference Proceedings (International):

1. S. Vassiliadis, F. Duarte and S. Wong, "A Load/Store Unit for a *memcpy* Hardware Accelerator", in *Proceedings of 17th IEEE International Conference on Field Programmable Logic and Applications*, (Amsterdam, The Netherlands), August 2007.

2. S. D. Breijer, F. Duarte and S. Wong "An OCM Based Shared Memory Controller For Virtex 4", in *Proceedings of 17th IEEE International Conference on Field Programmable Logic and Applications*, (Amsterdam, The Netherlands), August 2007.

3. F. Duarte and S. Wong, "A *memcpy* Hardware Accelerator Solution for Non Cache-line Aligned Copies", in *Proceedings of the 18th IEEE International Conference on Application-specific Systems, Architectures and Processors*, (Montreal, Canada), July 2007.

4. S. Wong, F. Duarte, and S. Vassiliadis, "A Hardware Cache *memcpy* Accelerator", in *Proceedings of the 2006 IEEE International Conference on Field Programmable Technology*, (Bangkok, Thailand), December 2006.

5. F. Duarte, and S. Wong, "Profiling Bluetooth and Linux on the Xilinx Virtex II Pro", in *Proceedings of the 9th Euromicro Conference on Digital System Design*, (Dubrovnik, Croatia), September 2006.

Conference Proceedings (Local):

6. F. Duarte and S. Wong, "Analysis of a User-space Device-driver for the *memcpy* Hardware", in *Proceedings of the 18th Annual Workshop*

on Circuits, Systems, and Signal Processing, (Veldhoven, The Netherlands), November 2007.

7. F. Duarte and S. Wong, "Bluetooth Profiling on the Xilinx Virtex II Pro", in *Proceedings of the 16th Annual Workshop on Circuits, Systems, and Signal Processing*, (Veldhoven, The Netherlands), November 2005.

8. F. Duarte and S. Wong "Bluetooth Protocol Profiling on the Xilinx Virtex II Pro", in *Proceedings of Architectures and Compilers for Embedded Systems*, (Edegem, Belgium), September 2005.

Publications not directly related to this dissertation:

9. F. Duarte, S. D. Breijer, and S. Wong "Supporting the Linux Operating System on the MOLEN Processor Prototype", in *Proceedings of the 18th Annual Workshop on Circuits, Systems, and Signal Processing*, (Veldhoven, The Netherlands), November 2007.

Acknowledgments

This dissertation is the culmination of four years of hopes combined with disappointments, enthusiasms combined with frustrations, and eventually the result of a life time effort. I most definitely would not have been successful without the support and care of several people that I am thankful to.

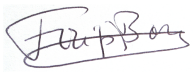
First and foremost, I am grateful to Stamatis Vassiliadis for believing in me and giving me the chance to do my PhD in his research group. It was my privilege to work under his supervision, although for such a short period of time. I will miss him and remember him always.

I would like, equally, to thank Stephan Wong for the meetings that took the research to a higher level, for the help in organizing the research and improving my writing skills. My PhD work would not have been the same without his inputs. I also want to thank Kees Goossens for his comments on the dissertation and propositions.

I would like to thank everybody from the Computer Engineering Laboratory where I had the opportunity to work in a truly international environment, with people coming from all parts of the world, with different working experiences. This "special environment" allowed me to learn the best of people. Specially, I would like to thank the "smoking group", where a lot of scientific and sometimes less scientific discussions happened. The PhD work is also made of these breaks. I am also thankful to Bert Meijs for all his support in guarantying that all the material I needed during the research would be up and running (and if it was not, he made it up and running in no time!) and to Lidwina Tromp for putting pressure in the administrative part of the university whenever it was needed.

Quero agradecer também à pessoa mais importante, ao meu marido, pelas demasiadas vezes em que tratou da casa e da comida, dos meus desapontamen-

tos e frustrações, de mim... Sem ele, o doutoramento nunca teria chegado ao fim no tempo previsto. E finalmente, agradeço aos meus pais, por me terem apoiado durante os longos anos de estudo, especialmente quando decidi vir para o estrangeiro. Paizinho, acabei o curso!



Delft, The Netherlands, 2008

Samenvatting

Een op-cache-gebaseerde hardwarematige versneller voor het verplaatsen van gegevens in het geheugen

Deze dissertatie presenteert een hardwarematige versneller die in staat is grote (ook niet-parallelle) verplaatsingen van gegevens in het geheugen te versnellen en vooral het kopiëren van geheugen, traditioneel uitgevoerd door de processoren. Omdat hedendaagse processoren verbonden of uitgerust zijn met cachegeheugens van variërende groottes (van enkele kilobytes in handheld apparaten tot vele megabytes in desktop computers en grote servers), is het niet meer dan logisch aan te nemen dat de te-kopiëren gegevens reeds aanwezig zijn binnen het cachegeheugen. Zeker als we in ogenschouw nemen dat dergelijke gegevens vaak eerst verwerkt moeten worden. Dit betekent dat de aanwezigheid van cachegeheugens aangewend kan worden om de latentie inherent aan het kopiëren van geheugen significant te verminderen, wanneer een "slimmere" manier om het geheugen te kopiëren wordt gebruikt.

Daarom haalt de voorgestelde versneller voor het kopiëren van geheugen voordeel uit de aanwezigheid van deze cachegeheugens en introduceert een doorverwijzingsmechanisme dat de originele gegevens (in het cachegeheugen) verbindt met de gekopieerde adressen (in een nieuw toegevoegde indexeringstabel). De voorgestelde oplossingen vermijden cache pollution en duplicatie van gegevens en plannen de toegang tot het hoofdgeheugen efficiënt in, aldus wordt de latentie inherent aan het kopiëren van geheugen doeltreffend verminderd. Bovendien ondersteund de voorgestelde versneller kopieën van cache line en word granulariteit, kan het verbonden worden met een direct-mapped of een set-associative cachegeheugen en kan het doelmatig het knelpunt van het kopiëren van geheugen in single core processoren en multi-core proces-

soren, die een message passing communicatiemodel uitvoeren, verminderen.

De voorgestelde oplossingen zijn als conceptversie geïmplementeerd in een FPGA en gebruikmakend van een simulator werden verscheidene benchmarks uitgevoerd om de prestatiewinst van het voorstel te bepalen. In het bijzonder aan de ontvangerskant van de TCP/IP stack zijn de voorgestelde oplossingen in staat snelheidswinsten tussen de 2,96 en 4,61 maal te behalen en het aantal uitgevoerde instructies met 26% tot 44% te verminderen.