



Never stop thinking

# EMBEDDED MULTIPROCESSOR SYSTEM-ON-CHIP FOR ACCESS NETWORK PROCESSING

A THESIS  
SUBMITTED TO THE DEPARTMENT OF INFORMATICS  
AND THE EXAMINATION BOARD  
OF TECHNISCHE UNIVERSITÄT MÜNCHEN  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE  
IN  
COMPUTATIONAL SCIENCE AND ENGINEERING

By  
Mohamed A. Bamakhrama

1<sup>st</sup> Examiner: Prof. Dr. Hans Michael Gerndt  
2<sup>nd</sup> Examiner: Prof. Dr. Arndt Bode  
Supervisor: Dr.-Ing. Jinan Lin

Munich on December 14, 2007

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

December 14, 2007  
Mohamed A. Bamakhrama



*In the name of God, the most graceful, the most merciful*

© Copyright by Mohamed A. Bamakhrama 2007  
All Rights Reserved

# Acknowledgment

First of all, all the thanks and praises are to God for all what he gave to me. Then I would like to pay all the gratitude to my family whom without their continuous encouragement and support, I would not be able to achieve what I have done so far.

I would like to thank my supervisors, Dr. Jinan Lin at Infineon Technologies and Prof. Dr. Hans Michael Gerndt at the university, for providing me with the opportunity to work on a very interesting topic for this master's thesis. This thesis was enabled by their continuous support and brilliant guidance. I also would like to thank Prof. Dr. Arndt Bode for accepting to be the second examiner.

All of my colleagues in the Advanced Systems and Circuits department gave me the feeling of being at home. Thank you Dr. Xiaoning Nie, Stefan Maier, Stefan Meier, Mario Steinert, Dr. Ulf Nordqvist, and Chao Wang for all the help and being wonderful colleagues.

I feel gratitude towards all the people in Infineon Communications Solutions unit who provided me with the support and guidance throughout this thesis. Many thanks to Sandrine Avakian, Stefan Linz, Yang Xu, Dmitrijs Burovs, Beier Li, and Mars Lin.

I also would like to thank Ralf Bächle from the Linux Kernel community for his valuable advice and the interesting discussions.

I am very grateful to Shakeel UrRehman from the ASIC Design and Security team at Infineon and Abdelali Zahi from Risklab Germany for their continuous advice and proofreading this thesis.

Last, but absolutely not least, a big “thank you” to all of my friends for being there all the time. We really had a very nice time together.

*To the soul of my father...*

# Abstract

Multicore systems are dominating the processor market; they enable the increase in computing power of a single chip in proportion to the Moore's law-driven increase in number of transistors. A similar evolution is observed in the system-on-chip (SoC) market through the emergence of multi-processor SoC (MPSoC) designs. Nevertheless, MPSoCs introduce some challenges to the system architects concerning the efficient design of memory hierarchies and system interconnects while maintaining the low power and cost constraints. In this master thesis, I try to address some of these challenges: namely, non-cache coherent DMA transfers in MPSoCs, low instruction cache utilization by OS codes, and factors governing the system throughput in MPSoC designs. These issues are investigated using the empirical and simulation approaches. Empirical studies are conducted on the *Danube* platform. Danube is a commercial MPSoC platform that is based on two 32-bit MIPS cores and developed by Infineon Technologies AG for deployment in access network processing equipments such as integrated access devices, customer premises equipments, and home gateways. Simulation-based studies are conducted on a system based on the ARM MPCore architecture. Achievements include the successful implementation and testing of novel hardware and software solutions for improving the performance of non-cache coherent DMA transfers in MPSoCs. Several techniques for reducing the instruction cache miss rate are investigated and applied. Finally, a qualitative analysis of the impact of instruction reuse, number of cores, and memory bandwidth on the system throughput in MPSoC systems is presented.

# Kurzfassung

Multicore Systeme dominieren inzwischen den Prozessormarkt. Sie ermöglichen den Anstieg der Rechenleistung eines einzelnen Chips mit der gleichen Geschwindigkeit, in der nach Moores Law die Anzahl der Transistoren wächst. Eine ähnliche Entwicklung kann bei Systemen auf einem Chip (System-on-Chip, SoC) mit dem Auftreten von Multiprozessor-SoCs (MPSoC) beobachtet werden. MPSoCs stellen jedoch neue Herausforderungen an die Entwickler solcher Architekturen. Sie liegen insbesondere im effizienten Design von Speicherhierarchien und der Verbindungseinrichtungen mit niedrigem Energieverbrauch und niedrigen Kosten. In dieser Masterarbeit versuche ich einige dieser Herausforderungen zu thematisieren: Nicht Cache-kohärente direkte Speicherzugriffe (direct memory access, DMA) in MPSoCs, schlechte Ausnutzung des Instruktions-Caches durch das Betriebssystem (OS) und Faktoren, die den Gesamtdurchsatz eines MPSoCs bestimmen. Diese Punkte werden mit empirischen und simulativen Ansätzen untersucht. Die empirischen Versuche werden auf der *Danube* Plattform durchgeführt. Danube ist eine kommerzielle MPSoC Plattform basierend auf zwei 32-bit MIPS Prozessorkernen, die von der Infineon Technologies AG für den Einsatz in Zugangsnetzen (Access Networks) entwickelt wurde. Simulativ wird ein System basierend auf der ARM MPCore Architektur untersucht. Ziel dieser Arbeit ist das erfolgreiche Implementieren und Testen neuer Hard- und Softwaremethoden, um die Leistung von nicht Cache-kohärenten DMA-Transfers in MPSoCs zu steigern. Weiterhin werden einige Techniken zur Reduzierung der Cache-Fehlzugriffsrate untersucht und angewandt. Abschließend zeigt eine qualitative Analyse die Auswirkungen der Wiederverwendung von Instruktionen im Cache, der Anzahl der Prozessorkerne und der Speicherbandbreite auf den Durchsatz eines MPSoCs.

# Abbreviations

<b>ASE</b>	Application Specific Extensions
<b>BSP</b>	Board Support Package
<b>CLI</b>	Command Line Interface
<b>CMOS</b>	Complementary Metal Oxide Semiconductor
<b>CMP</b>	Chip Multiprocessing
<b>CMT</b>	Chip Multithreading
<b>CPE</b>	Customer Premises Equipment
<b>CPI</b>	Cycles Per Instruction
<b>CPU</b>	Central Processing Unit
<b>DDR</b>	Double-Data Rate
<b>DMA</b>	Direct Memory Access
<b>DMAC</b>	DMA Controller
<b>DRAM</b>	Dynamic RAM
<b>DSL</b>	Digital Subscriber Line
<b>DSP</b>	Digital Signal Processing
<b>EJTAG</b>	Enhanced Joint Test Action Group
<b>GB</b>	Giga-Byte
<b>IAD</b>	Integrated Access Device
<b>ICD</b>	In-Circuit Debugger
<b>ICMP</b>	Internet Control Message Protocol
<b>ILP</b>	Instruction Level Parallelism
<b>I/O</b>	Input/Output
<b>IP</b>	Internet Protocol

<b>IPC</b>	Instructions Per Cycle
<b>IRQ</b>	Interrupt Request
<b>ISA</b>	Instruction Set Architecture
<b>KB</b>	Kilo-Byte
<b>LAN</b>	Local Area Network
<b>MB</b>	Mega-Byte
<b>MIMD</b>	Multiple Instructions, Multiple Data
<b>MPSoC</b>	Multi-Processor System-on-Chip
<b>NFS</b>	Network File System
<b>NIC</b>	Network Interface Card
<b>OS</b>	Operating System
<b>RAM</b>	Random Access Memory
<b>RDRAM</b>	Rambus DRAM
<b>RISC</b>	Reduced Instruction Set Computer
<b>RTL</b>	Register Transfer Level
<b>SDR</b>	Single Data Rate
<b>SDRAM</b>	Synchronous DRAM
<b>SMT</b>	Simultaneous Multithreading
<b>SoC</b>	System-on-Chip
<b>SRAM</b>	Static RAM
<b>TCP</b>	Transmission Control Protocol
<b>TLP</b>	Thread-Level Parallelism
<b>VLIW</b>	Very Long Instruction Word
<b>VoIP</b>	Voice Over IP

# Contents

<b>Acknowledgment</b>	<b>iv</b>
<b>Abstract</b>	<b>vi</b>
<b>Kurzfassung</b>	<b>vii</b>
<b>Abbreviations</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Objectives . . . . .	4
1.3 Organization . . . . .	5
1.4 Structure of the Thesis . . . . .	5
<b>2 Background and Related Work</b>	<b>7</b>
2.1 Chip Multiprocessing . . . . .	7
2.2 Hardware Multithreading . . . . .	8
2.3 Multithreaded Chip Multiprocessors . . . . .	9
2.4 Symmetric Shared-Memory Multiprocessors . . . . .	9
2.5 Multi-Processor System-on-Chip . . . . .	10
2.6 Memory Hierarchy . . . . .	11
2.7 Direct Memory Access . . . . .	14
2.8 Memory Consistency and Coherence . . . . .	16
2.9 System Interconnect . . . . .	18

2.9.1	Shared-Medium Interconnect . . . . .	18
2.9.2	Switched-Medium Interconnect . . . . .	19
2.9.3	Hybrid Interconnect . . . . .	19
2.10	Processor Performance Counters . . . . .	20
2.10.1	Previous Work . . . . .	20
2.10.2	Proposed Solution: MIPSProf . . . . .	22
<b>3</b>	<b>Evaluation Platforms</b>	<b>25</b>
3.1	Danube Platform . . . . .	25
3.1.1	Features . . . . .	25
3.1.2	Architecture . . . . .	26
3.1.3	Processor Subsystem . . . . .	27
3.1.4	Danube Evaluation Framework . . . . .	29
3.2	ARM11 MPCore Architecture . . . . .	30
3.3	VaST CoMET <sup>®</sup> Platform . . . . .	32
<b>4</b>	<b>DMA and Cache Coherence</b>	<b>33</b>
4.1	Background . . . . .	33
4.2	DMA Operation . . . . .	34
4.3	Linux DMA Driver . . . . .	35
4.4	Original DMA Solution . . . . .	37
4.5	Proposed Solution # 1: Software-Based Solution . . . . .	39
4.6	Proposed Solution # 2: Hybrid Solution . . . . .	40
4.7	Testing and Validation . . . . .	40
4.8	Results and Analysis . . . . .	44
4.9	Conclusions . . . . .	45
<b>5</b>	<b>OS Instruction Cache Utilization</b>	<b>46</b>
5.1	Background . . . . .	46
5.2	Previous Work . . . . .	47
5.2.1	Hardware Solutions . . . . .	47
5.2.2	Software Solutions . . . . .	49

5.2.3	Hybrid Solutions . . . . .	50
5.3	Optimizations for Linux . . . . .	51
5.3.1	Cache Locking . . . . .	51
5.4	CPU Subsystem Behavior . . . . .	53
5.4.1	Effect of Cache Size Reduction . . . . .	53
5.4.2	Effect of the Absence of Critical Word First Filling . . . . .	54
5.4.3	Effect of Cache Locking and Instruction Pre-fetching . . . . .	55
5.5	Conclusions . . . . .	57
<b>6</b>	<b>System Throughput in MPSoCs</b>	<b>58</b>
6.1	Background . . . . .	58
6.2	Evaluation Methodology . . . . .	59
6.3	System Architecture . . . . .	60
6.4	Results and Analysis . . . . .	60
6.4.1	Effect of Memory Bandwidth . . . . .	61
6.4.2	Effect of Number of Cores . . . . .	62
6.4.3	Effect of Instruction Reuse . . . . .	63
6.5	Conclusion . . . . .	64
<b>7</b>	<b>Summary and Outlook</b>	<b>66</b>
	<b>Bibliography</b>	<b>67</b>

# List of Tables

2.1	Comparison of different memory types based on access time and price	12
3.1	Danube features . . . . .	26
3.2	Cache configuration for the MIPS cores in Danube . . . . .	28
3.3	Danube evaluation framework - Hardware components . . . . .	30
3.4	Danube evaluation framework - Software components . . . . .	30
3.5	Cache configuration for the MP11 cores used in the simulation . . . . .	32
4.1	Un-cached loads and stores in the original DMA implementation . . . . .	38
4.2	Events measured during the profiling . . . . .	42
4.3	Events measured concurrently during the profiling . . . . .	42
4.4	Number of fragments per packet that are sent from the ping host . . . . .	43
4.5	Overhead due to the new DMA solution . . . . .	44
5.1	Original DMA solution cache accesses and misses . . . . .	46
6.1	Parameters configuration for the memory bandwidth simulations . . . . .	60

# List of Figures

1.1	Processor performance improvement between 1978-2005 . . . . .	2
2.1	Multicore architecture examples . . . . .	8
2.2	Comparison of hardware threading types with CMP . . . . .	9
2.3	General classification of parallel systems . . . . .	10
2.4	Memory hierarchy example . . . . .	13
2.5	DMA read operation . . . . .	15
2.6	DMA write operation . . . . .	15
2.7	Memory consistency problem in the presence of DMA . . . . .	16
2.8	Shared-medium vs. switched-medium interconnects . . . . .	19
2.9	Proposed kernel module design . . . . .	23
3.1	Danube architecture . . . . .	26
3.2	Danube application example . . . . .	27
3.3	Experimental environment . . . . .	29
3.4	ARM11 MPCore architecture . . . . .	31
4.1	DMA descriptors chain . . . . .	35
4.2	DMA read usage through RX direction of Ethernet driver . . . . .	36
4.3	DMA write usage through TX direction of Ethernet driver . . . . .	37
4.4	DMA read and write logic . . . . .	38
4.5	Descriptors access pattern within the DMA Driver . . . . .	39
4.6	Improvement in the RX handler of ping due to the new solution . . . . .	44
4.7	Improvement in the TX handler of ping due to the new solution . . . . .	45

5.1	Proposed locking mechanism . . . . .	52
5.2	Effect of cache size reduction on the total number of cycles . . . . .	54
5.3	Effect of reducing the cache size on I-cache misses . . . . .	55
5.4	Effect of absence of critical word first filling . . . . .	55
5.5	Effect of the instruction prefetching on sequential hazard-free code streams . . . . .	56
6.1	MPCore-based System architecture . . . . .	60
6.2	Effect of memory bandwidth on system IPC . . . . .	62
6.3	Effect of number of cores on system IPC . . . . .	63
6.4	Effect of instruction reuse on system IPC . . . . .	64

# Chapter 1

## Introduction

### 1.1 Background

Doubling CPU performance every 18 months has been the trend in the hardware market for the last 50 years. Moore's law has fueled such a phenomenon for a long time but today there are many challenges which make this trend more difficult to follow. These challenges are imposed by the physical limitations. The first one is represented in the so-called *speed of light limit* which imposes a minimum length for the wires within the chip in order to avoid hazards [20]. The second challenge is the fabrication technology. Typical fabrication CMOS gate lengths at the time are in the range of 130-45 nm where experiments showed that it is possible to achieve lengths around 6 nm [21], Nevertheless, chip design and manufacturing under such a scale becomes extremely difficult and costly as well. The third challenge is power dissipation. Power has been a major problem for a long time and the "straightforward" approach to solve this problem was to lower the voltage with each new generation of chips since the dynamic power dissipation depends proportionally on the voltage:

$$P = \alpha.C.V^2.f \tag{1.1}$$

where:

$P$ : The dynamic power dissipation

$C$ : The effective capacitance

$V$ : The operational voltage

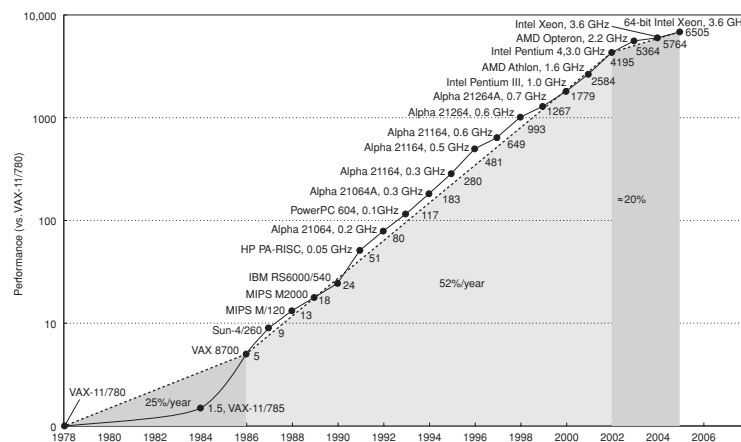
$f$ : The clock frequency

$\alpha$ : The switching activity factor

Equation 1.1 shows that the power also depends on the frequency. With the chip frequencies rising and approaching around 4 GHz at the time [78], the gain out of reducing the voltage is becoming quite small.

A fourth challenge is the leakage current which became a major concern with the advent of nano-scale technologies since it became so large that in some cases it is almost equal to the operational current [46].

Further challenges arise also from the architecture complexity and data intensive applications. According to [61], architects have reached the limit in extracting the computing power from single core processors. Increasing the performance of single core processors will require either enlarging them or putting more sophisticated logic for extracting more instruction-level parallelism (ILP) or thread-level parallelism (TLP) through superscalarity, very long instruction word (VLIW) architectures, and simultaneous multithreading (SMT) [32, 10].



In summary: To keep Moore's law valid, we need a new approach which can fuel the increase in CPU performance in the future. One possibility is to utilize the large number of transistors that can already be put into a single chip today and switch into parallel processing approach. Here the idea is to put more than one CPU within the same chip rather than trying to increase the amount of logic that can be put into a single CPU. The CPU in such systems is usually called a *processor core* or simply *core* and the resulting system is called a *multicore*<sup>1</sup> system.

Multicore systems represent a revolution in many aspects in computing since they require new paradigms, techniques, and tools for efficiently utilizing the computing power available on such chips. This revolution is seen by many researchers as a similar one to the RISC revolution in 1980s [30]. Currently, multicore systems are already deployed in the market and many chips are available such as Intel *Montecito*, AMD *Opteron*, Sun *Niagara* [57, 5, 48]. However, these chips are targeted toward servers/workstations market. The embedded market in which the constraints are primarily related to power consumption and low cost, has also realized the potential of multicore systems. Vendors have already started to release chips based on embedded multicore architectures. Such chips are usually called *multiprocessor system-on-chips (MPSoCs)* since they combine both of multiprocessing and system-on-chip paradigms.

MPSoC systems present some challenges to system architects. These challenges include efficient memory hierarchy design, scalable system interconnect, new programming paradigms, etc... Some of these problems exist also in uncore systems. However, the lack of extensive experience with embedded multicore chips represents both a challenge and an opportunity at the same time for the architects since they need to come up with novel architectures that satisfy all the constraints. One important market for multicore chips is network processors. The tremendous bandwidth available today in network connections require some innovative solutions in the protocol processing engines to fully utilize the available bandwidth. Manufacturers have already started shipping products based on multicore chips such as Intel *IXP* family, Cavium *Octeon* family, and Freescale *C-Port* family [43, 60, 36].

---

<sup>1</sup>I have chosen to use *multicore* instead of *multi-core* to reflect the fact that these systems are becoming the norm. See [47] for more information about the usage of hyphen in English.

One particular product is the *Danube* chip from Infineon Technologies AG [2]. Danube is a *dual-core* chip based on 32-bit MIPS processors. According to the technical specifications [2]:

Danube is Infineon's next generation ADSL2+ IAD single-chip solution. It comprises the highest integration of VoIP and ADSL for high-end and cost-optimized ADSL2/2+ IADs. It enables the most effective and scalable realization of VoIP applications. It is offered with a complete development kit consisting of reference designs, (...), and an application software stack based upon the Linux operating system.

In a typical application, the two cores in the Danube are organized as:

1. **CPU 0:** A 32-bit MIPS 24KEc core used to control the overall operation of the system.
2. **CPU 1:** A 32-bit MIPS 24KEc core with Digital Signal Processing Application Specific Extensions (DSP ASE) and it is used only for VoIP processing.

## 1.2 Objectives

The purpose of this thesis is to:

1. Explore and analyze the existing architectural solutions for MPSoC systems especially the ones concerning memory hierarchy and system interconnect using empirical and simulation approaches.
2. Implement some novel concepts in hardware and software to improve the performance of these solutions.

Based on that, the thesis was split into three concrete tasks:

1. **Direct memory access (DMA) and cache coherence:** Software-based solutions for cache coherence are attractive in terms of power consumption for MPSoC systems [54, 56]. The aim here is to analyze and enhance the

existing software solutions for non-cache coherent DMA transfers in MPSoCs. The Danube platform is used for conducting the experiments and testing the new solutions.

2. **Operating system instruction cache utilization:** OS codes have poor instruction cache utilization [4, 77]. The goal here is to investigate and improve the instruction cache utilization for the Linux Kernel 2.4 series running on Danube.
3. **System throughput of MPSoCs:** A qualitative analysis of the impact of instruction reuse, number of cores, and memory bandwidth on the overall system IPC is presented. The analysis is obtained through simulating a system based on ARM MPCore architecture using VaST CoMET<sup>®</sup> platform [7, 72].

### 1.3 Organization

The thesis was conducted in Infineon Technologies headquarters in Munich, Germany between May and December 2007 within the Advanced Systems and Circuits (ASC) department. The supervisor from Infineon Technologies is Dr.-Ing. Jinan Lin from the Protocol Processing Architectures (PPA) team. The principal examiner from the university is Prof. Dr. Hans Michael Gerndt from the Lehrstuhl für Rechnertechnik und Rechnerorganisation/Parallelrechnerarchitektur and the second examiner is Prof. Dr. Arndt Bode from the same department.

### 1.4 Structure of the Thesis

Chapter 2 presents a review on selected topics from computer architecture that are used throughout the thesis. Chapter 3 outlines the evaluation platforms used for conducting the experiments. Chapter 4 presents the DMA and cache coherence task and the implemented solutions along with the obtained results. In Chapter 5, the instruction cache utilization problem is presented along with the proposed solutions and results. Chapter 6 presents a qualitative analysis of the impact of instruction

reuse, number of cores, and memory bandwidth on the overall system throughput. Chapter 7 draws the summary of conclusions and suggestions are outlined regarding possible further research work on the topic.

# Chapter 2

## Background and Related Work

This chapter aims to introduce the reader to the prerequisite topics in computer architecture and operating systems that are used throughout the thesis.

### 2.1 Chip Multiprocessing

A *multicore processor* is one that combines two or more independent processors into a single chip. The cores might be identical (i.e. *homogeneous*) or hybrid (i.e. *heterogeneous*). Cores in a multicore chip may share the cache(s) on different levels. They may also share the interconnect to the rest of the system. Each core independently implements optimizations such as pipelining, superscalar execution, and multithreading. A system with  $N$  cores is effective when it is presented with  $N$  or more threads concurrently. Multicore systems are multiprocessor systems that belong to MIMD (Multiple-Instruction, Multiple-Data) family according to Flynn's taxonomy [25]. Another common name used to describe multicore processors is *Chip Multiprocessors (CMP)*. This name emphasizes the fact that all the cores are present on a single physical chip.

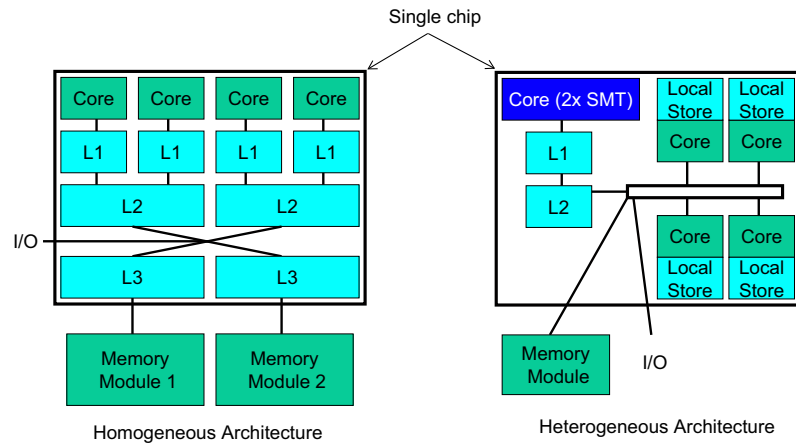


Figure 2.1: Multicore architecture examples. Source: [80]

## 2.2 Hardware Multithreading

Hardware Multithreading exploits *thread-level parallelism (TLP)* by allowing multiple threads to share the functional units of a single processor in an overlapping fashion. According to [32, 22], it can be implemented in three different ways:

1. **Coarse-grained:** CPU switches to a new thread when a thread occupying the processor blocks on a memory request or other long-latency request.
2. **Fine-grained:** CPU switches between threads on each cycle, causing the execution of multiple threads to be interleaved. This interleaving is usually done in a round-robin fashion, skipping any threads that are stalled at that time.
3. **Simultaneous Multithreading (SMT):** Adds multi-context support to multiple-issue, out-of-order processors. Unlike conventional multiple-issue processors, SMT processors can issue instructions from different streams on each cycle for improved ILP. SMT helps to eliminate both *vertical* and *horizontal* waste. Vertical waste is introduced when the processor issues no instructions in a cycle, where horizontal waste is introduced when not all issue slots can be filled in a cycle.

Figure 2.2 highlights the difference between the different types of hardware threading and CMP.

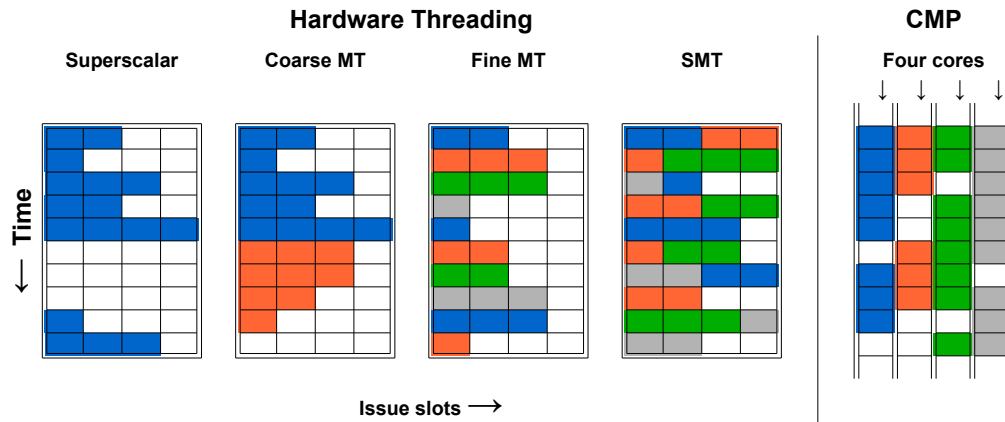


Figure 2.2: Comparison of hardware threading types with CMP. Source: [32]

## 2.3 Multithreaded Chip Multiprocessors

According to [22], a new emerging type of processors is the so-called *multithreaded chip multiprocessors*. In this type of processors, multiple processor cores are integrated into a single chip as in CMP and each core implements hardware multithreading (MT). Sun Microsystems calls this paradigm *Chip Multithreading (CMT)* [42]. Examples of CMT processors include IBM POWER5 and Sun Niagara [45, 48].

## 2.4 Symmetric Shared-Memory Multiprocessors

*Symmetric*<sup>1</sup> *Multiprocessors (SMP)* are shared-memory multiprocessor systems with the following characteristics:

1. Global physical address space
2. Symmetric access to all the main memory from any processor

They belong to the *uniform memory access (UMA)* family according to the general classification of parallel systems. SMPs were originally implemented with each processor being on a separate chip. With the advent of mega-transistors chips, it became

<sup>1</sup>The word *Symmetric* here refers to the symmetry in memory access latency. It should not be confused with *Symmetric Multiprocessing OS* in which a single OS image runs on all the cores. For more info, please refer to [66, 32]

possible to integrate all the processors into a single chip. SMP systems represent the most popular type of multicore systems today [32, 10].

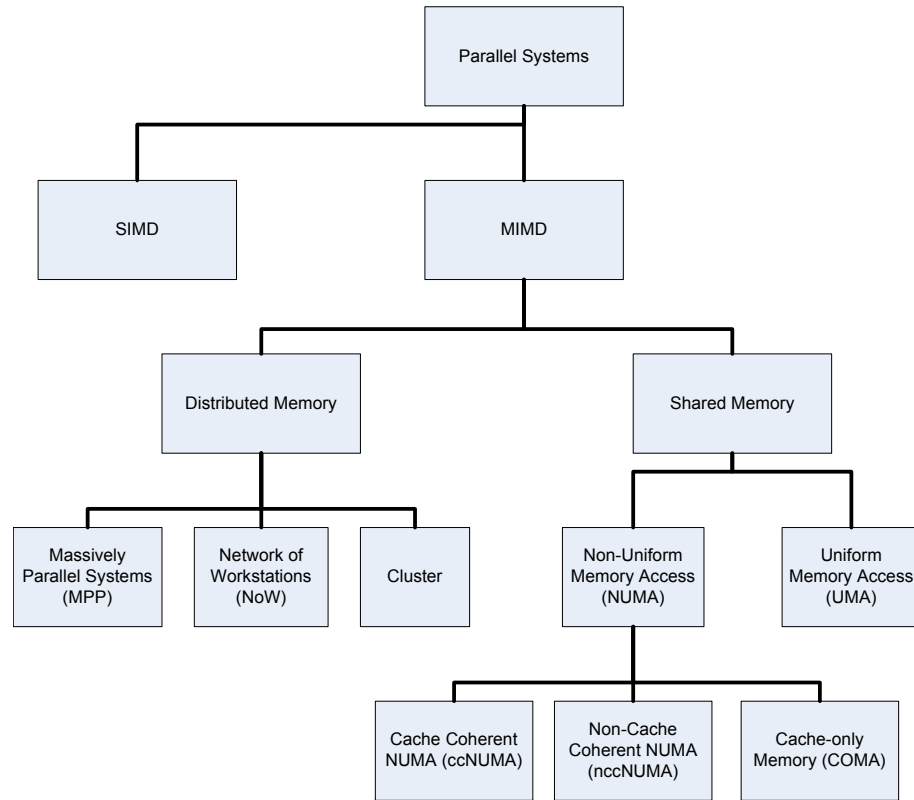


Figure 2.3: General classification of parallel systems. Source: [26]

## 2.5 Multi-Processor System-on-Chip

According to [44], MPSoCs represent the new solution targeted towards embedded computing to overcome the performance limitations of single processor microcontrollers. MPSoCs exhibit some general properties that can be summarized as follows:

- **Heterogeneous processing elements:** MPSoCs usually combine general processors, application processors, and accelerators
- **Application-oriented:** They are targeted towards a specific application field (e.g. protocol processing, video processing, security, etc...)

- **Low power:** Since the main application field for these chips is embedded computing, low power is one of the primary factors that drive the design of such chips.

## 2.6 Memory Hierarchy

Almost all the modern computer systems incorporate some kind of memory hierarchy. Such a hierarchy is needed to cope with the different speeds by which the different components in a computer system operate. For example, registers are very fast but are limited in capacity ( $\sim 1$  KB) where the hard disk is very huge in capacity ( $\sim 100$  GB) but very slow to access. Moreover, CPU speed increases by 55% annually where memory speed increases only by 10% annually leading into what is known as the *memory wall* [81]. Thus, a solution is needed to cope with such a growing disparity in speed. The solution is to provide some kind of *intermediate* memory between the CPU and the main memory which is faster than the main memory and a little bit slower than the CPU registers. It serves to bridge the gap between the CPU and the main memory. This intermediate memory is known as *cache*. Cache helps to hide the latencies incurred in accessing the main memory. To be able to achieve that, cache operation is based on a phenomenon observed in most of the programs. This phenomenon is known as *locality of reference*. Locality of reference refers to the fact that a memory location which is accessed now would be very probably accessed again in the near future or that the adjacent locations would be very likely accessed in the near future. Locality can be split into two types [63]:

1. **Temporal locality** (locality in time): If an item is referenced, it will tend to be referenced again soon.
2. **Spatial locality** (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Thus we can take advantage of the principle of locality of reference by implementing the memory of a computer as a *hierarchy*. A memory hierarchy consists of

Memory technology	Typical access time	\$ per GB in 2004
SRAM	0.5-5 ns	\$4000-\$10,000
DRAM	50-70 ns	\$100-\$200
Magnetic disk	5,000,000-20,000,000 ns	\$0.5-\$2

Table 2.1: Comparison of different memory types based on access time and price. Source: [63]

multiple levels of memory with different sizes and speeds. As the distance from the CPU increases, both capacity and access time of memories increase.

Table 2.1 shows a comparison between the different types of memory. It can be observed that the main memory access time is about 100 times greater than the CPU register access time. The access time gets even larger for external hard-disk access. Accessing the external hard-disk is around 10,000,000 times slower than accessing the CPU internal registers. This table shows that a memory hierarchy is very necessary for modern computer systems in order to get a decent performance. A level closer to the processor is generally a subset of any level further away. Memory hierarchy might contain multiple levels, but data is copied between only two adjacent levels at a time.

Figure 2.4 shows a general abstraction of the memory hierarchy used in modern computers. A small but very fast cache memory is placed on the chip inside the CPU and it is called *CPU cache*<sup>2</sup>. This cache is built usually with SRAM technology and it can be accessed usually at a speed equivalent to CPU register access time. It is important to notice that the actual first level of memory is the CPU registers. Most of modern CPUs contain two caches: one instruction cache and one data cache. Moreover, some CPUs implement more than one level of caching between the CPU and the main memory. For instance, the Intel *Core Duo* processor [19] implements two levels of caches called *L1* cache and *L2* cache with *L2* cache being shared between the two cores.

Data transfer between the different levels of memory occurs in terms of *blocks* or

---

<sup>2</sup>Cache principle is not restricted only to the CPU. Caches are used in many subsystems within a computer system (e.g. *Disk cache*)

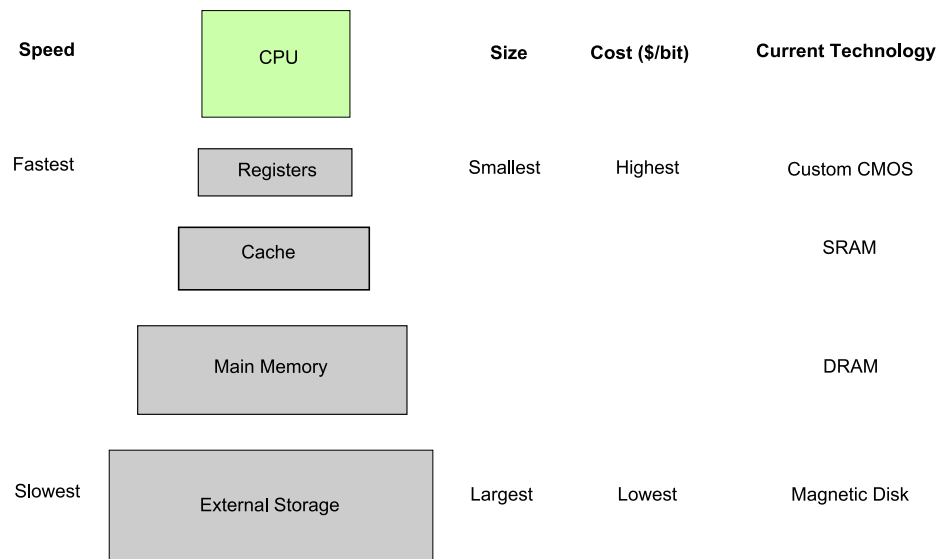


Figure 2.4: Memory hierarchy example

*lines*. The block size might be different between each two levels. Now, suppose that the CPU requested a given address; if this address is present in the cache, then we say that we have a *hit*. Otherwise, we have a *miss*. One important goal for the system architect and programmer is to maximize the *hit ratio* which is the fraction of successful memory accesses. Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. *Hit time* is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss. The *miss penalty* is the time to replace a block in the upper level with the needed block from the lower level plus the time needed to deliver this block to the processor.

One important issue which arises here is that since we have much less cache lines than what does the memory contain, a policy is needed to determine how we map the memory lines into cache lines. There exist three general strategies for cache mapping:

1. **Direct Mapping:** In this scheme, each memory location is mapped to exactly one location in the cache. This means that a block of main memory can only be brought into the same line of cache every time.
2. **Fully-Associative Mapping:** A block of main memory may be mapped into

any line of the cache and it is no longer restricted to a single line of cache.

3. **Set-Associative Mapping:** In order to overcome the complexity of fully-associative cache, cache is divided into a number of sets. Each set contains a number of lines (e.g a 2-way set associative cache has 2 lines per set). Under this scheme, a block of memory is restricted to a specific set of lines. A block of main memory may map to any line in the given set. It represents a compromise between direct mapping and fully-associative mapping.

## 2.7 Direct Memory Access

*Direct Memory Access (DMA)* is a technique used to offload the costly I/O operations from the CPU into a “third party” processor known as *DMA controller*. Prior to DMA, whenever a hardware device wanted to send or receive data from the memory, the CPU had to initiate the transfer and perform the copy between the device and the memory. This means that the CPU was kept busy during the whole process and if we recall from Section 2.6 that I/O devices and memory are much slower than CPU, this means that CPU is kept idle most of the time just waiting for the I/O devices and the memory. With DMA, the CPU initiates the transfer and offloads it to a specialized hardware called *DMA controller (DMAC)*. Then the DMAC starts performing the copy operation between the device and the memory. Meanwhile, the CPU can go back to normal useful processing while the I/O operations are handled by the DMAC. Upon the completion, DMAC signals the CPU indicating that the operation has completed. Obviously, this represents a huge performance saving since the CPU time is not wasted in waiting for the slow peripheral or memory devices.

In general, DMAC provides two core operations: *read* and *write*. Figures 2.5 and 2.6 show the simplified sequence of actions taken during these operations.

Modern DMA controllers (e.g. ARM PrimeCell<sup>®</sup> [6]) provide a wide range of operations. These operations include memory-to-memory, memory-to-peripheral, peripheral-to-memory, and peripheral-to-peripheral transactions. Typically, DMAC provides its functionality through *DMA channels*. These channels determine the interconnection

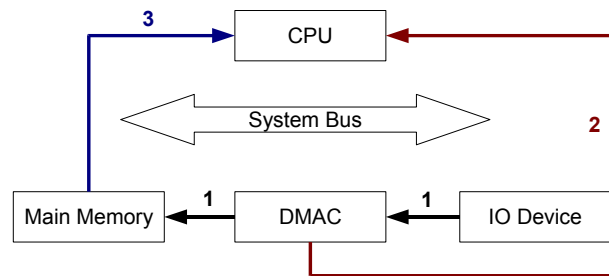


Figure 2.5: DMA Read Operation: **(1)** The first step is to transfer the data from the I/O device (e.g. network controller) via DMAC into the main memory. **(2)** After that, DMAC issues an interrupt to the CPU signaling the reception of data (e.g. packet). **(3)** Finally, CPU accesses the main memory and processes the received data.

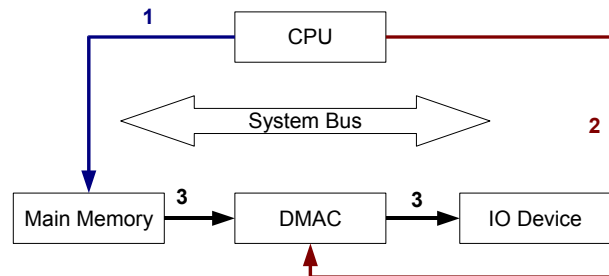


Figure 2.6: DMA Write Operation: **(1)** The first step here is taken by the CPU which writes the data into the main memory. **(2)** After that, the CPU signals the DMAC to start the transaction. **(3)** Finally, DMAC transfers the data from the main memory into the I/O device.

between the different modules within the system via the DMAC. The channels assignment can be fixed or configurable. Furthermore, DMAC is usually split internally into an RX controller and a TX controller with both of them operating in parallel.

However, DMA comes at a cost: the presence of another data producer/consumer (i.e. DMAC) in the system implies the need for maintaining *memory consistency*. To illustrate this problem, let us have a look on Figure 2.7. Suppose that the following operations are executed in order on a given address:

- 1: CPU writes the value **Y** into the cache without updating the main memory
- 2: DMAC transfers the old value **X** from the main memory into an external device

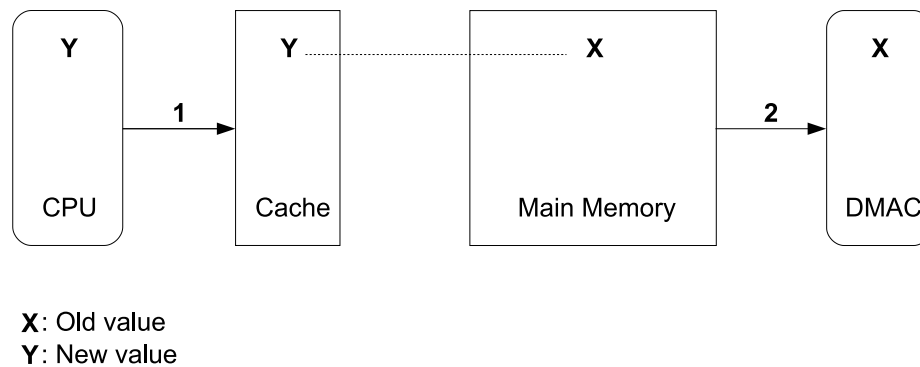


Figure 2.7: Memory consistency problem in the presence of DMA: Here is a situation where lack of coherence in the memory system might yield the whole system useless

It is obvious that the DMAC will be transferring a stale value of the memory address. This leads us to the definition of two important properties of memory systems called *coherence* and *consistency*.

## 2.8 Memory Consistency and Coherence

Any memory system which incorporates more than one memory data producer/consumer (i.e. processor) should maintain two important principles in order to keep the calculations correct and avoid situations as the one illustrated in Figure 2.7. These two important principles are:

1. *Coherence*: Defines what values can be returned by a read.
2. *Consistency*: Determines when a written value will be returned by a read.

Another refined version of the above definition can be stated as follows: *Coherence* defines the behavior of reads and writes to the same memory location, while *consistency* defines the behavior of reads and writes with respect to accesses to other memory locations.

However, these definitions are somehow vague. A more concrete definition is found in [32]. It states that a memory system is *coherent* if:

1. A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated and no other writes to X occur between the two accesses.
3. Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors. Namely, the system should exhibit *write serialization*. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

*Consistency* is however more complex. It specifies constraints on the order in which memory operations become visible to the other processors. It includes operations to the same location and to different locations. Therefore, it subsumes coherence. The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.

One last remark is that these two principles apply on accesses to main memory and caches as well. Cache coherence problem exists also in multiprocessor systems where each CPU has its own cache and coherence has to be ensured also on the level of caches. Nevertheless, the general concepts are the same as above except for the implementation. For more detailed discussion about memory consistency and cache coherence, please refer to [32].

## 2.9 System Interconnect

System Interconnect refers to the medium which connects the different parts of a system. In this context, we restrict ourselves to the *on-chip interconnect* that connects the processors, memory controllers, DMACs, and other peripheral interfaces in a SoC [44, 14]. A new trend in the interconnect technology is to use internetworking ideas (e.g. OSI model) to implement the on-chip network. This led to the introduction of *on-chip networks (OCNs)*—also referred to as *network-on-chip (NoC)* [13, 32]—that are used for interconnecting microarchitecture functional units, register files, caches, and processor and IP cores within a chip or multichip modules.

According to [32, 59, 49, 13], on-chip interconnect can be classified into three main categories:

1. **Shared-Medium Interconnect**
2. **Switched-Medium Interconnect**
3. **Hybrid Interconnect**

### 2.9.1 Shared-Medium Interconnect

In shared-medium interconnect, all devices share the interconnect media. An example of such an architecture is the *traditional system bus*. In bus architectures, all the devices are connected via the bus. Only one message is allowed to be sent at a time and this message is broadcasted to all the devices in the bus. Access to the bus is controlled via an *arbiter* which decides which device gets the access to the bus medium upon request. One obvious bottleneck with the bus architecture is the limited scalability. Some prominent industrial standards that are deployed in many products at the time are ARM *Advanced Microprocessor Bus Architecture (AMBA)*, STMicroelectronics *STBus*, and IBM *CoreConnect* [8, 70, 34].

## 2.9.2 Switched-Medium Interconnect

Switched-medium interconnects address the scalability bottleneck incurred in shared-medium interconnects. Figure 2.8 shows the difference between shared-medium and switched-medium interconnects.

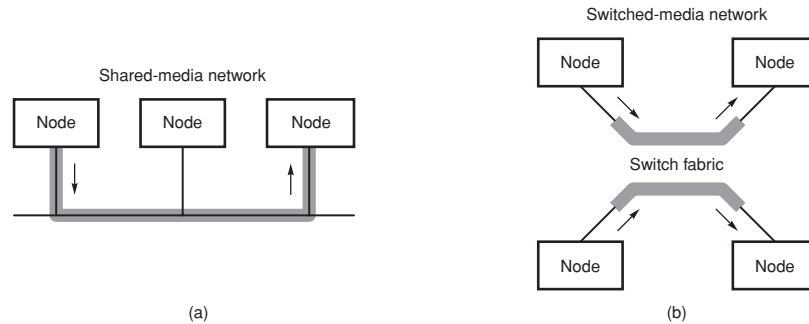


Figure 2.8: Shared-medium vs. switched-medium interconnects. Source: [32]

Switched-medium interconnects can be classified into two main categories [13]:

1. **Direct networks:** known also as *point-to-point* networks. In such architecture, each node connects directly to a limited number of neighboring nodes. Direct interconnect networks are popular for building large-scale systems because the total communication bandwidth also increases when the number of nodes in the system increases.
2. **In-direct networks:** or *switch-based* networks offer an alternative to direct network for scalable interconnection design. In these networks, a connection between nodes must go through a set of *switches*. Switches dynamically establish communication between sets of source-destination pairs. One prominent example of such interconnects that is used in multicore architectures is the *crossbar* switch [32].

## 2.9.3 Hybrid Interconnect

Usually, such designs combine both of the traditional shared buses and switched interconnects in a hierarchal manner in order to provide some kind of trade-of between

bandwidth and energy efficiency.

## 2.10 Processor Performance Counters

Almost all the modern CPUs come with a set of performance counters. These counters provide the programmer with a powerful utility to profile and evaluate the performance of the working system through counting the different events occurring within the system. Quantities that are typically provided by the performance counters include—but not limited to—instructions executed, total cycles, instruction/data cache accesses and misses, and branches and branch mis-predictions.

These quantities provide the starting point for the architect who is evaluating an existing system to judge its performance.

Due to the need for a comprehensive evaluation of the Danube system performance, the performance counters present in the MIPS core are used. This approach provides much more accurate results and has very low overhead compared to sampling or code instrumentation.

### 2.10.1 Previous Work

Here we present a survey of the most popular profiling APIs at the time which allow the programmer to use the performance counters in a systematic and modular way.

#### PerfCtr

PerfCtr is a Linux driver which allows the user to use performance monitoring counters [64]. According to its documentation, it adds support to the Linux Kernel (2.4.16 or newer) for using the performance counters. PerfCtr works by issuing an interrupt on each processor periodically. After each interrupt, the performance counters are read and the *virtual* counters associated with each Linux process are updated accordingly.

It supports many architectures including all Intel Pentium processors, AMD processors, PowerPC, etc... Nevertheless, MIPS architecture is still not ported for this module. Therefore, PerfCtr could not be used directly to profile the Danube.

## **OProfile**

According to [62], *“OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.”*

OProfile is supported on a variety of architectures including MIPS32. Nevertheless, the problem was that it requires a kernel version that is  $\geq 2.6.11$ . Since the Linux kernel running on Danube is based on 2.4 series, OProfile was also discarded from the list of possible choices.

## **PAPI**

Performance API or PAPI is a project which aims to provide a portable standard API for using hardware performance counters. According to [35], *“PAPI provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level PAPI interface deals with hardware events in groups called EventSets. EventSets reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another.”*

PAPI was not supported for Linux@MIPS and it was eliminated from the choices' list.

## **Sysprof**

According to [65], *“Sysprof is a sampling CPU profiler for Linux that uses a kernel module to profile the entire system, not just a single application. Sysprof handles shared libraries and applications do not need to be recompiled. In fact they don't even have to be restarted.”*

Sysprof is available only for i386 and x86-64 architectures thus making it useless for evaluating the Danube.

### 2.10.2 Proposed Solution: MIPSProf

As all the available tools at the time for profiling Linux Kernel using the performance counters were either not supported for MIPS architecture and/or not ported to the Linux Kernel series that is running on Danube and due to the need to measure specific points and segments within the kernel code itself and even in some cases portions of some kernel functions, the counters were used manually by inserting the measurement code at the measurement points. However, this decision faced a challenge in terms of the efficiency of development since the events to be counted were at the beginning hard-coded within the measurement code. Soon, it was realized that there is a need for providing a more flexible solution in order to be able to change the measured events during run-time thus saving the compilation time needed for changing the events each time.

To accomplish this task under Linux kernel, a special kernel module was developed which allows the user to use the counters interactively. The main reasons which lead to this decision are:

1. Access to the performance counters under MIPS32 architecture is privileged to the kernel mode
2. The kernel does not have access to the C library so we do not have access to standard functions like `printf()` and `scanf()`

One attractive alternative was to use the `/proc/` file system under Linux. According to [15], `/proc/` file system provide an access point to the kernel data structures. Using `copy_from_user()` function from the kernel API, it is possible to pass data (e.g. the counted events) through a `/proc/` file system entry into our proposed kernel module and set the performance counters accordingly. This idea gave us the flexibility to use the performance counters at any point within the kernel code. Moreover, the counted events now can be changed during run-time without the need to recompile the whole kernel.

Figure 2.9 shows the general design of the proposed kernel module to collect the measurements. The module was implemented in GNU C and integrated into both of

the kernel source tree provided by the board support package (BSP) and the kernel image running on the Danube to facilitate collecting the measured data.

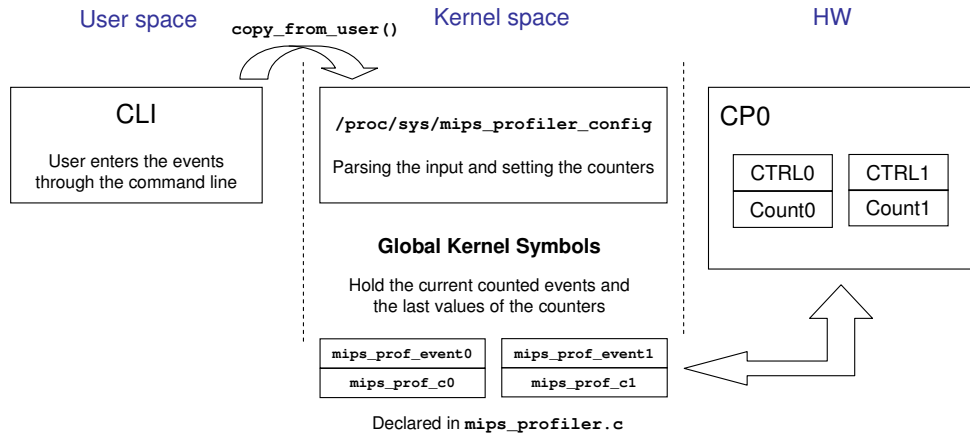


Figure 2.9: Proposed kernel module design

Code listings 2.1 and 2.2 show the assembly macros used for profiling the kernel and an example of how these macros can be used within the kernel.

One important note regarding the macros in Listing 2.1 is that there must be a separation of at least *two* instructions between `MIPS_PROF_START` and `MIPS_PROF_STOP` to avoid execution hazards [38]. Another alternative is to use the `EHB` (*Execution Hazard Barrier*) instruction introduced in MIPS32 Release 2 ISA which eliminates the need for manual spacing to prevent hazards by stopping instruction execution until all execution hazards have been cleared.

Listing 2.1: Assembly macros used for profiling the kernel

```

#define MIPS_PROF_START(event0, event1)    \
    asm volatile ( ".set    mips32r2    \n" \
                  ".set    noreorder   \n" \
                  "mtc0   %z0, $25, 1 \n" \
                  "mtc0   %z1, $25, 3 \n" \
                  "mtc0   %z2, $25    \n" \
                  "mtc0   %z3, $25, 2 \n" \
                  ".set    mips0       \n" \
                  : : "Jr" (0), "Jr" (0), \
                  "Jr" (( event0 << 5 ) | 0x1F), \
                  "Jr" (( event1 << 5 ) | 0x1F))

#define MIPS_PROF_STOP(c0, c1)    \
    asm volatile( ".set    mips32r2    \n" \
                 ".set    noreorder   \n" \
                 "mfc0   %0, $25, 1  \n" \
                 "mfc0   %1, $25, 3  \n" \
                 ".set    mips0       \n" \
                 : "=r" (c0), "=r" (c1))

#define MIPS_PROF_PRINT(event0, cntr0, event1, cntr1)\
    printk( KERN_INFO "Event %d: %d, Event %d: %d\n", \
           event0, cntr0, event1, cntr1)

```

Listing 2.2: Example of the profiling macros usage within the kernel

```

/**
Prerequisites:
- mips_profiler.h is included
- The following variables are declared as ‘extern’:
    extern int mips_prof_event0, mips_prof_event1;
    extern unsigned long mips_prof_c0, mips_prof_c1;
*/

MIPS_PROF_START(mips_prof_event0, mips_prof_event1);
...
MIPS_PROF_STOP(mips_prof_c0, mips_prof_c1);
MIPS_PROF_PRINT(mips_prof_event0, mips_prof_c0, \
               mips_prof_event1, mips_prof_c1);

```

# Chapter 3

## Evaluation Platforms

In this chapter, the reader is introduced to the platforms used for analyzing and investigating the different tasks. Danube platform is used for studying the non-cache coherent DMA transfers and the OS instruction cache utilization issues while an ARM MPCore based system modeled and simulated using VaST Systems CoMET<sup>®</sup> platform is used for analyzing the system throughput in MPSoCs.

### 3.1 Danube Platform

Danube is a single-chip solution for ADSL2/2+ with integrated 2-channel analog codec targeted towards access network processing equipments such as Internet access devices (IADs), broadband customer premises equipments (CPEs), and home gateways [3].

#### 3.1.1 Features

Danube features the cutting-edge technologies in ADSL and VoIP [3, 2]. It is based on two 32-bit MIPS cores. Danube also features a 32-bit multithreaded protocol processing engine (PPE) which provides high performance protocol processing with low silicon cost plus user programmability. Danube also is capable of running the Linux OS. Table 3.1 highlights the main features of the Danube platform.

VoIP and ADSL	
ITU-T G.992.1/3/5 (ADSL/2/2+)	Embedded VoIP derived from VINETIC family
Two integrated codecs	Integrated voice compression support
Line Echo Cancellation (LEC) support	T.38 Fax Relay support
Physical Interfaces	
Two 10/100/200 MII/Reverse and MII/TMII	32-bit PCI 2.2 bus
UART for RS-232 with HW Flow Control	EJTAG/JTAG Debug interface
AC97 Codec interface	USB 2.0 host/device
16/8-bit NOR/NAND Flash memory	16-bit SDR/DDR DRAM
IOM-2 (Time division multiplexing)	SPI with DMA support
32 GPIOs, 24-bit serial LED controller	Multi Media Card Interface (SD/MMCI)
ADSL2+ Analog Hybrid interface	Two Analog SLIC interfaces

Table 3.1: Danube features

### 3.1.2 Architecture

Figure 3.1 shows the general architecture of the Danube platform.

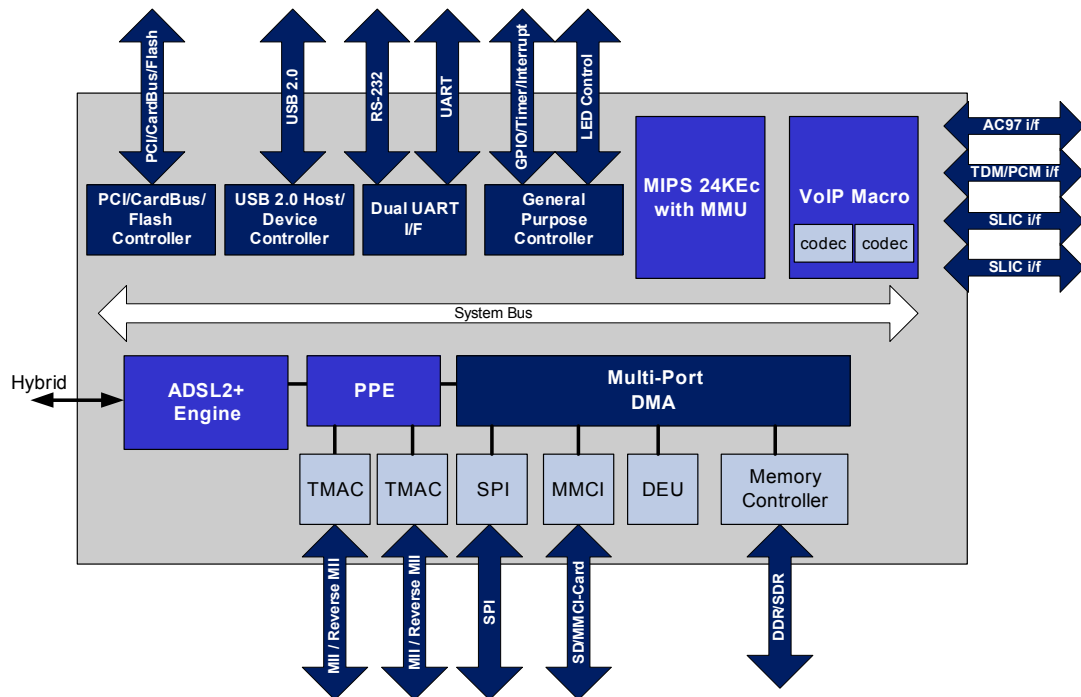


Figure 3.1: Danube architecture (Source: [2])

Figure 3.2 shows a typical application example for Danube. In this example, Danube is used at the core of an IAD with wireless LAN and VoIP functionalities. It functions as a connectivity point for all the networking equipments at home or small offices<sup>1</sup>.

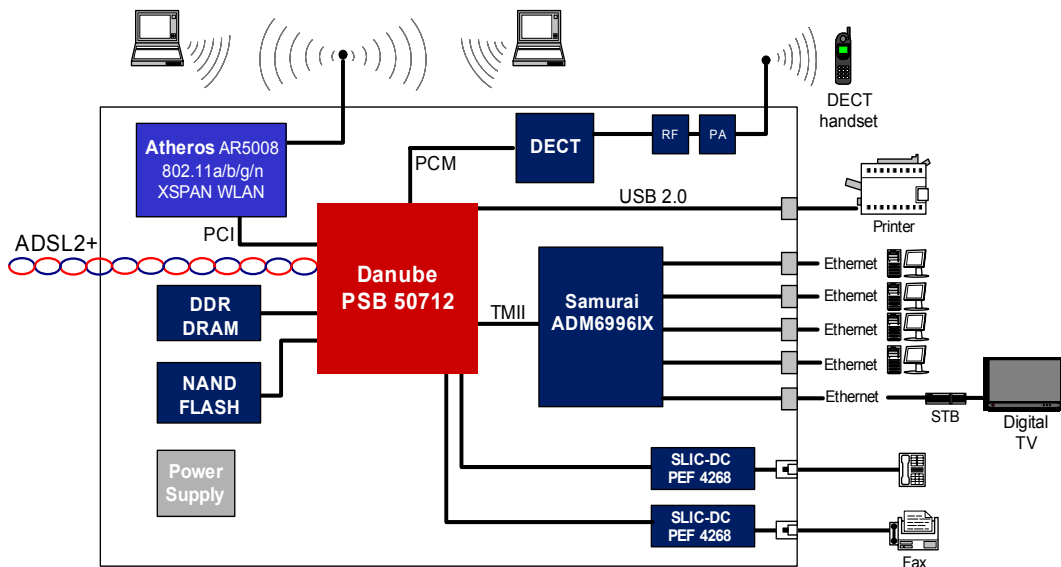


Figure 3.2: Danube application example: IAD + WLAN + VoIP (Source: [2])

### 3.1.3 Processor Subsystem

The processor subsystem in Danube is composed of *two* cores. Both cores are based on the MIPS32 24KEc RISC core from MIPS Technologies and they are connected to the rest of the SoC via a crossbar architecture. In a typical application (e.g. IAD), they are organized as:

- **CPU 0:** A MIPS32 24KEc core controlling the networking tasks and the overall operation of the system

<sup>1</sup>Such networks are usually called *Access Networks* since they connect subscribers to their immediate service provider.

- **CPU 1:** A MIPS32 24KEc core enhanced with the Digital Signal Processing Application Specific Extensions (DSP ASE) and it is used only for VoIP processing

Each core features [38, 39]:

- Support for MIPS32 Enhanced Architecture (MIPS32 Release 2)
- 32-bit address path
- 64-bit data path to the caches
- Eight stages pipeline
- Two separate instruction and data caches with 16 KByte per each
- Cache lines are virtually indexed, physically tagged
- Cache replacement policy is based on *least recently used (LRU)* strategy
- The cache controller supports early restart
- Ability to manipulate the cache via **CACHE** instruction
- Supports cache locking per line: One restriction dictates that the user can *not* lock all the ways. In such a case, a collision will replace one of the locked ways
- EJTAG-based debug
- Clock frequency = 333 MHz

Further specifications of the cache configuration are given in Table 3.2.

	Cache size	Cache line size	No. of ways	No. of sets
<b>Instruction</b>	16 KB	32 B	4	128
<b>Data</b>	16 KB	32 B	4	128

Table 3.2: Cache configuration for the MIPS cores in Danube

### 3.1.4 Danube Evaluation Framework

In this section, the test environment which was used to evaluate the Danube system and develop the new solutions is presented.

#### Hardware Environment

In order to be able to conduct the experiments successfully and efficiently, a comprehensive testing environment is needed. Setting up such an environment was the first task to be accomplished in order to be able to interact with the system.

A private LAN was designed and implemented in order to allow the interaction between the different components used during the experiments. Figure 3.3 shows the network setup that was used for the experiments and development.

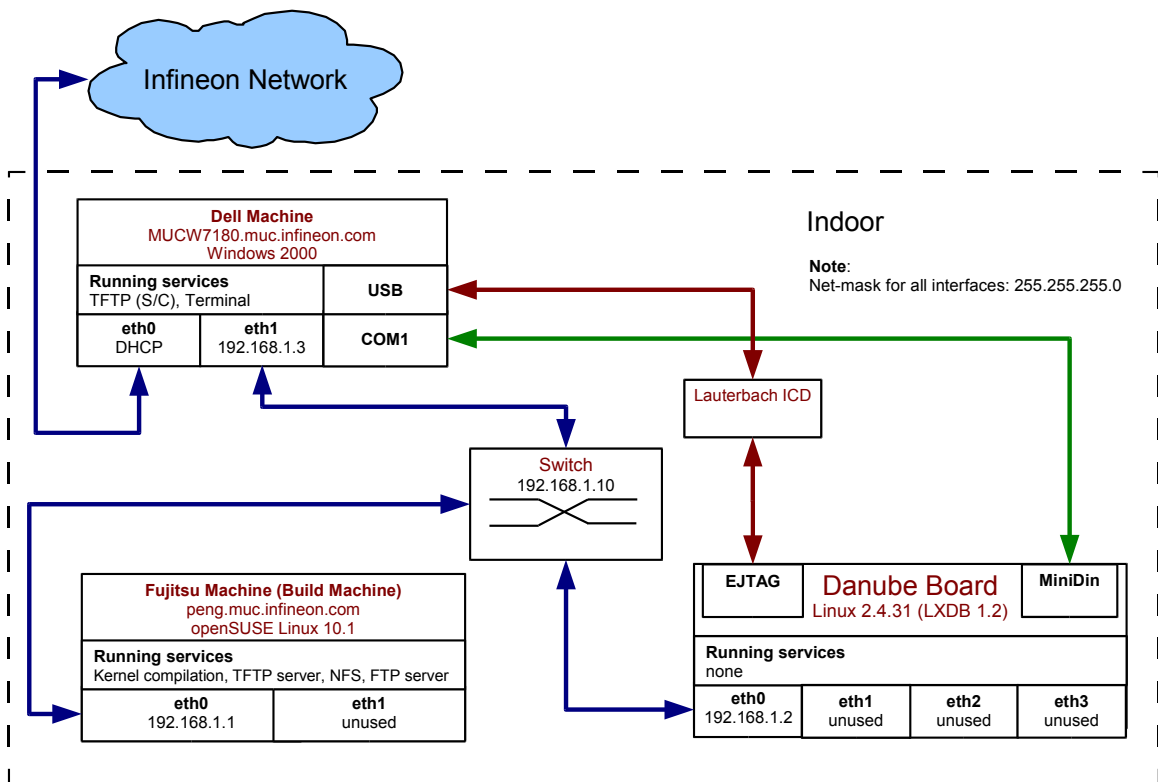


Figure 3.3: Experimental environment

Table 3.3 presents the hardware used for the evaluation.

Item	Description
Evaluation Board	Danube Reference Board (EASY 50712) [1]
In-circuit Debugger	Lauterbach Trace32 [27]

Table 3.3: Danube evaluation framework - Hardware components

### Software Environment

Table 3.4 summarizes the software tools and libraries used throughout the project:

Item	Description
OS	Infineon Linux Distribution (LXDB) 1.2 <sup>1</sup>
Kernel Configuration	DANUBE G0432V30_BSP
File-system	NFS-mounted file-system
Boot loader	U-boot 1.0.5.1
Shell	BusyBox 1.00
Compiler	Infineon LXDB 1.2 based on GNU GCC 3.3.6

Table 3.4: Danube evaluation framework - Software components

## 3.2 ARM11 MPCore Architecture

ARM11 MPCore is an embedded multicore architecture developed by ARM [9]. It can be configured to contain between one up to four cores based on ARM11 microarchitecture [7, 9]. Figure 3.4 shows the general design of the MPCore architecture.

The ARM MPCore processor features [7]:

- Up to four 32-bit MP11 cores
- *Snoop Control Unit (SCU)* responsible for maintaining coherence among MP11 cores L1 data caches
- A distributed interrupt controller with support for legacy ARM interrupts

---

<sup>1</sup>Based on Linux Kernel 2.4.31

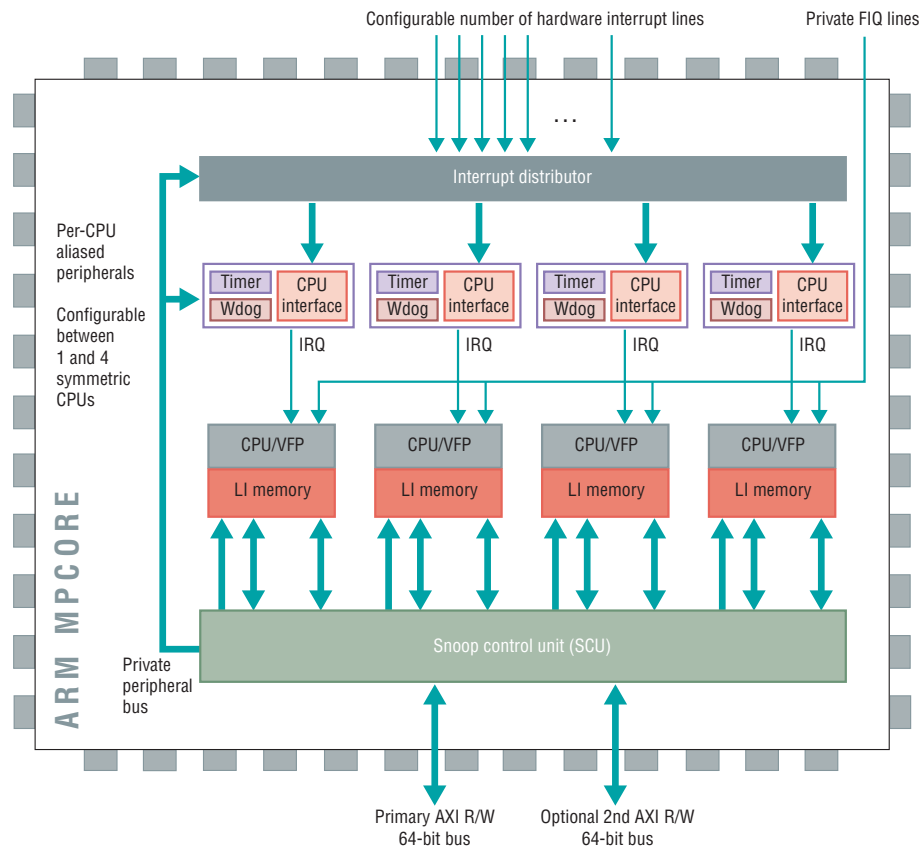


Figure 3.4: ARM11 MPCore architecture. Source: [28]

- A private timer and a private watchdog per MP11 CPU
- Support for private *fast interrupts* (FIQ)
- *Advanced Extensible Interface* (AXI) high-speed AMBA level two interfaces

Each MP11 core features:

- ARMv6K architecture-based multiprocessor-capable instruction set architecture
- Eight-stages pipeline
- Branch prediction with return stack
- Support for instruction prefetching

- Cache controller supports critical word first filling
- Cache replacement policy is based on round-robin strategy
- Two separate instruction and data caches. Each can be between 16 KB up to 64 KB
- 32-bit instruction interface and 64-bit interface to the data cache
- Hardware support for data cache coherence
- *Vector Floating-Point (VFP)* co-processor support
- JTAG-based debug

The cache parameters used in the simulated systems are highlighted in Table 3.5.

	Cache size	Cache line size	No. of ways	No. of sets
<b>Instruction</b>	32 KB	32 B	4	256
<b>Data</b>	32 KB	32 B	4	256

Table 3.5: Cache configuration for the MP11 cores used in the simulation

### 3.3 VaST CoMET<sup>®</sup> Platform

CoMET (Co-Design Methodology and Engineering Toolset) is a system engineering tool from VaST Systems [72, 74]. CoMET is intended to be the environment for the concurrent design of hardware and software. It enables the creation of a SW simulation-based virtual system prototype (VSP) of a SoC. CoMET features [72]:

- *Virtual processor models (VPM)* for most of the popular commercial processors
- Cycle-accurate simulations
- Fast simulations with up to 200 MIPS
- Interfaces to blocks developed using SystemC and Mentor Graphics Modelsim

# Chapter 4

## DMA and Cache Coherence

### 4.1 Background

As explained in Section 2.7, DMA is an essential component in today's computing systems. However, cache coherence is a problem that needs to be tackled efficiently. In general purpose processors targeted towards servers and workstations, a snooping-based cache coherence protocol implemented in hardware is usually used to ensure the coherence. Nevertheless, there is no consensus on the usage of hardware solutions for cache coherence in the resource-constrained MPSoCs [56, 55, 50]. [54] shows that software-based solutions are attractive in terms of power consumption. The aim here is to analyze the existing software solutions for non-cache coherent DMA transfers in MPSoCs and try to optimize it. To accomplish that, Danube platform is used. Danube contains an advanced multi-port DMA controller [2] and software-based solutions are used for ensuring the coherence of DMA transfers.

For proper operation, the CPU and DMAC need to communicate successfully. The communication can be done *per each transaction* or it can be done in such a way that the CPU prepares a *group of transactions* and passes them to the DMAC for batch processing. The latter has the advantage of reducing the overhead on the CPU.

In order to be able to perform any DMA transaction, some information is needed. These information can be abstracted as:

1. **Source Address:** The current location of the data
2. **Destination Address:** Where the data should be written
3. **Data Length:** Number of words to be written
4. **Synchronization Information:** Normally, the transfer status is communicated to synchronize the overall operation (e.g. pending, on-going, completed, failed, etc...)

It is important to note here that the DMAC should be configured so that it is aware of any byte-ordering issues (i.e. little-endian vs. big-endian). It is also worth noting that the DMAC operates on *physical* addresses.

In the Danube platform, communication is accomplished via special data items called *DMA descriptors* [75]. These descriptors carry the needed information to initiate any DMA operation. Each descriptor in Danube can fit in *one* cache line. There are two types of descriptors:

- **RX:** Used for DMA read operations
- **TX:** Used for DMA write operations

## 4.2 DMA Operation

In many of the modern designs, DMAC utilize a *descriptor's linked list* to increase the throughput and reduce the CPU overhead. CPU generates a descriptor chain that is passed in turn to the DMAC to process it. DMAC after processing the whole chain or a segment of it (depending on the DMAC configuration) will signal the CPU indicating the completion of the transaction. CPU after that decides what to do next (i.e. pass another descriptor, disable the DMA channel, etc...). Figure 4.1 shows an abstract view of the DMA chains and how they are processed.

Chains can be implemented in a variety of ways. For example, PrimeCell<sup>®</sup> DMAC from ARM [6] has eight special registers associated with each channel. These registers are called *linked list item (LLI)* registers. Each register of these holds a pointer to the

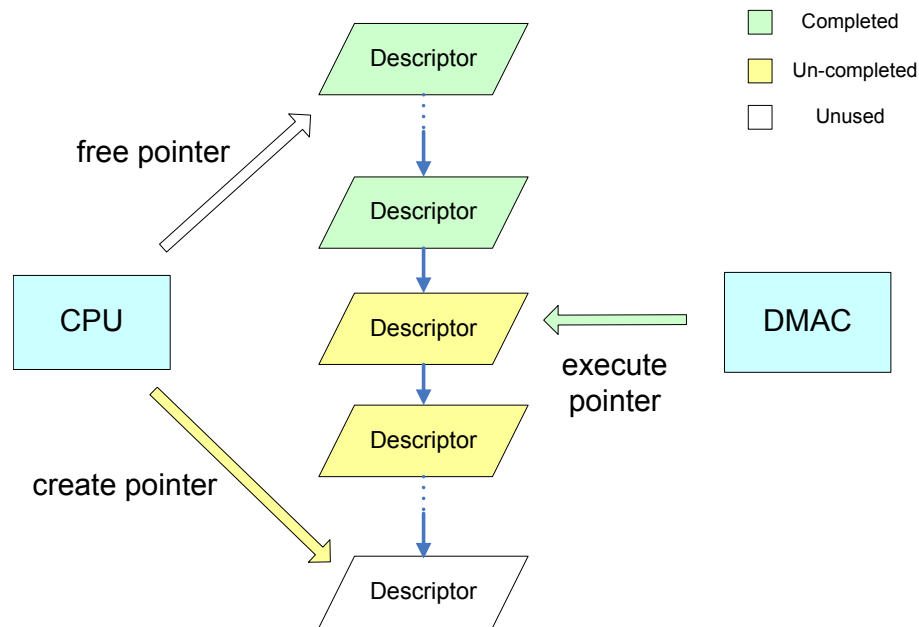


Figure 4.1: DMA descriptors chain

next descriptor. DMAC starts processing the LLI registers and it stops when it first encounters an LLI register that is set to “0”. Danube uses a similar principle which is based on a sequential buffer stored in memory containing the descriptors chain with both of the CPU and the DMAC being able to process the DMA descriptors in parallel as shown in Figure 4.1. Putting the chains in memory has an advantage which is saving the chip area needed to store the chain inside the DMAC.

### 4.3 Linux DMA Driver

Before performing any measurements or design modifications, it is extremely important to understand the existing system. Figure 4.2 shows the actual execution flow within the DMA driver through the RX direction of the Ethernet driver. The core operation needed in case of receiving a packet is the DMA *read*. The figure shows a layered view of what a packet must go through when it arrives to Danube. The most important layer for us is the *Device Driver* layer and in particular `dma-core.c::dma_device_read()`. This function contains the core functionality of

handling completed DMA read transactions.

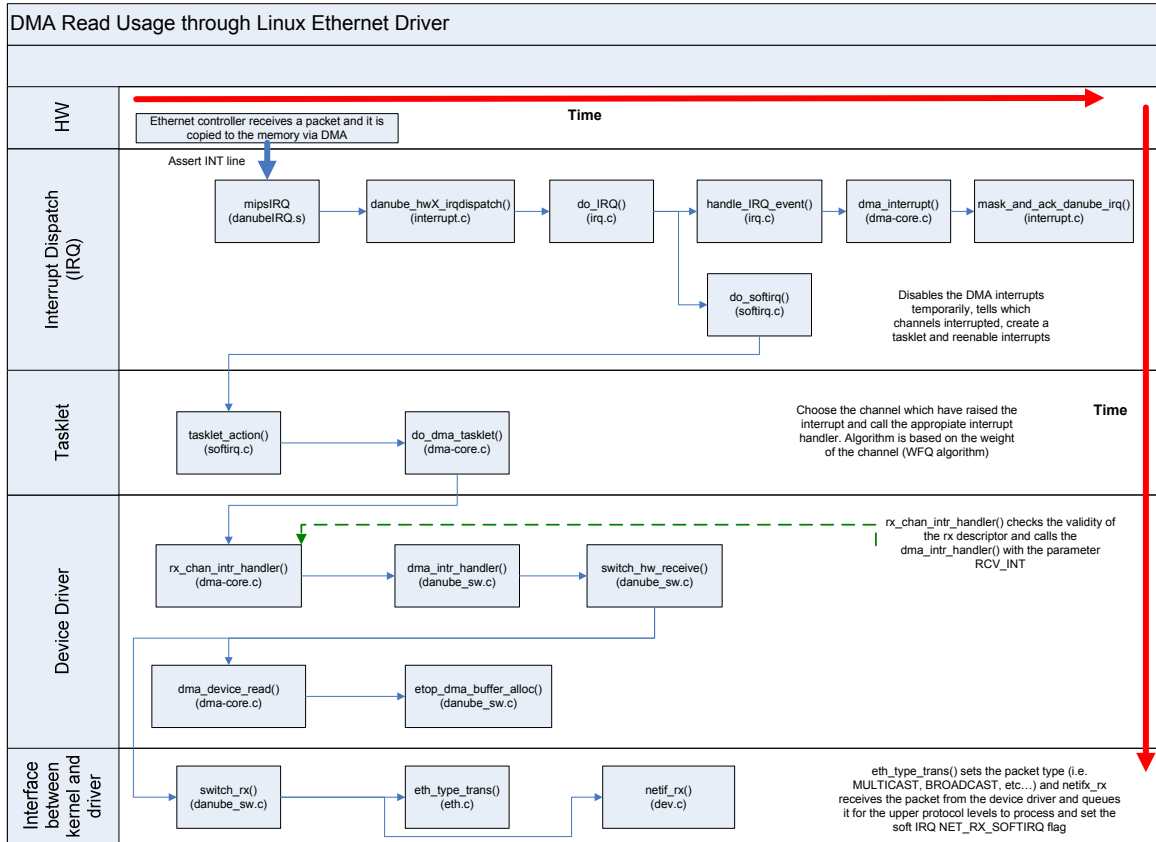


Figure 4.2: DMA read usage through RX direction of Ethernet driver

The packet journey starts when the DMAC signals an interrupt to the CPU indicating the completion of the packet transfer from the NIC to the memory. This triggers the interrupt handler which disables all the interrupts temporarily, determines which channel interrupted, creates a corresponding *tasklet*<sup>1</sup>, and finally re-enables the interrupts again.

In a similar fashion, the core DMA function for initiating DMA write transactions

<sup>1</sup>*Tasklet* in Linux Kernel terminology is a mechanism to implement the non-critical deferrable portions of interrupt handlers through a scheduled kernel routine that runs in interrupt context with all interrupts enabled and it is scheduled to run later at a system-determined safe time. Its main advantages are (1) the reduction of time in which the CPU runs with interrupts disabled and (2) serialization (i.e. two CPUs can NOT execute the same tasklet at the same time). See [15, 82] for further information.

is `dma-core.c::dma_device_write()` and the corresponding layered view is shown in Figure 4.3.

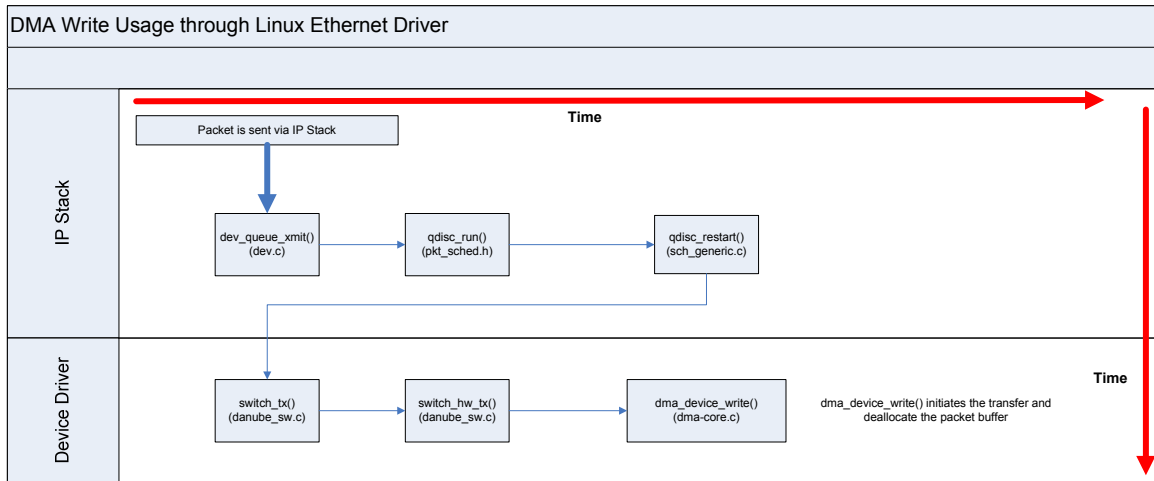


Figure 4.3: DMA write usage through TX direction of Ethernet driver

## 4.4 Original DMA Solution

To guarantee the integrity of the descriptors chains under Danube, they were kept in the un-cached memory space. This meant that for each DMA operation, all the accesses to the descriptors are full accesses to the main memory. This solution requires also invalidating the transferred data from the CPU cache. The latter requirement is not a major concern due to the fact that the DMA subsystem is used by three peripheral subsystems only within the whole system. The most important one is the Ethernet driver. In this driver, the packet usually goes in one direction (i.e. either TX or RX). So the data is not read after being written by the CPU in case of TX and in case of RX the data buffer allocated for the packet is usually invalidated from the cache immediately after allocating it hence avoiding any chances for data discrepancies.

Table 4.1 provides a summary on the total number of un-cached loads and stores within the original implementation of the core DMA operations.

	<code>dma_device_read()</code>	<code>dma_device_write()</code>
Un-cached Loads	4	9
Un-cached Stores	2	5

Table 4.1: Un-cached loads and stores in the original DMA implementation (per function call)

Figure 4.4 shows the flowchart representation of the DMA read and write operations.

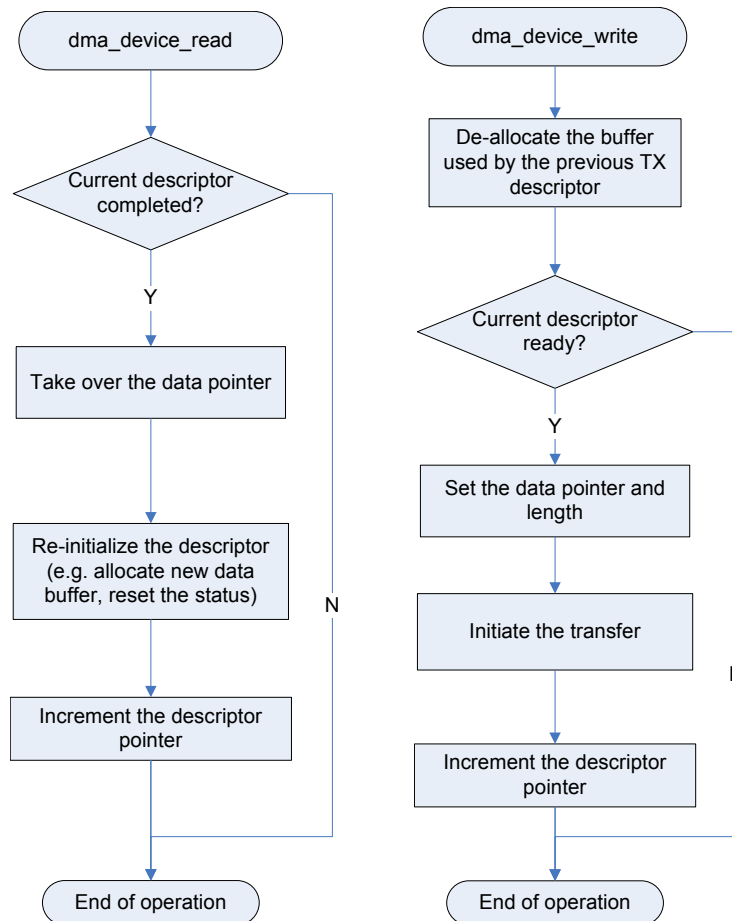


Figure 4.4: DMA read and write logic

## 4.5 Proposed Solution # 1: Software-Based Solution

By analyzing Table 4.1, it is easy to observe that the DMA driver has more reads to the descriptors than writes. Looking into the driver implementation, we observe that multiple read accesses are performed to access the descriptors and these read accesses interleave with *fewer* write accesses. We also found that these accesses read some fields from the descriptors (with all these un-cached reads being in series without any write in-between) and then it updates the descriptors (i.e. write access). Figure 4.5 illustrates this access pattern.

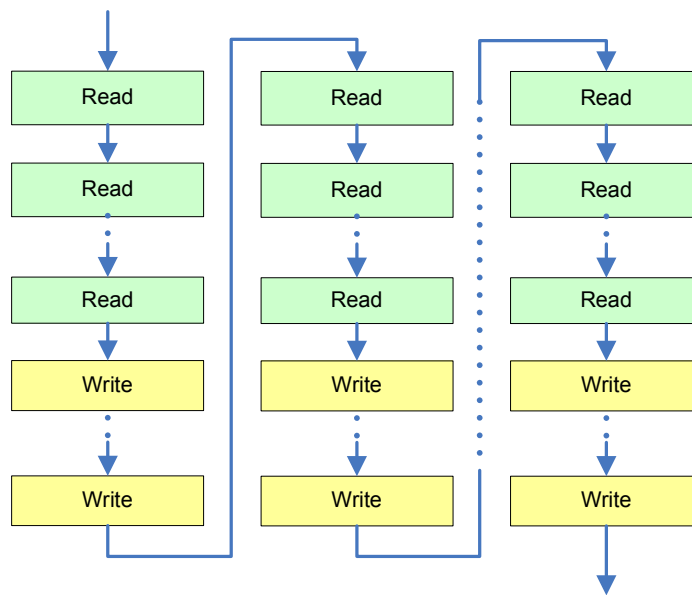


Figure 4.5: Descriptors access pattern within the DMA Driver

The new concept is to turn all these accesses into cached accesses via putting the descriptors in the cached memory space (thus eliminating the memory access penalty). Nevertheless, to guarantee the validity of the descriptors, all the write accesses should be followed by a flush and invalidate operation to evict them from the CPU cache and all the remaining read operations that are not followed by a write at the end of the function execution flow should be followed by an invalidate operation as shown in Algorithm 1.

The new solution requires manual inspection of the driver code to determine the possible execution flows and insert the appropriate operations accordingly. The implemented kernel module (see Section 2.10.2) was used in order to determine the exact number of un-cached operations and the total gain achieved from the new solution.

---

**Algorithm 1** Proposed solution for reducing un-cached memory accesses

---

**Require:** Descriptors allocated in the cached memory space

```

1: for all Functions accessing the descriptors do
2:     for all Descriptors accesses do
3:         if Access is a READ then
4:             if Last access within the function execution flow? then
5:                 Insert a cache invalidation after the read
6:             else
7:                 Do nothing
8:             end if
9:         else
10:            Insert a flush and invalidate operation after the write
11:        end if
12:    end for
13: end for

```

---

## 4.6 Proposed Solution # 2: Hybrid Solution

The second solution aims to overcome some of the shortcomings of the first solution described in Section 4.5 by providing hardware support through enhancing the DMAC logic. Since the algorithm devised for this solution is currently being filed as a patent by Infineon [52], no details can be provided about the algorithm. However, it is sufficient to say that successful testing and emulation of this algorithm has been conducted to verify its functionality and correctness.

## 4.7 Testing and Validation

In order to validate the new solutions, they must be tested under representative workloads. Since Danube platform is intended for usage in access network processing,

the choice was to use `ping` as the main test application. The reasons for choosing `ping` are:

1. Utilizes major parts of the Linux IP stack
2. Already implemented (no need to design custom benchmarks)

`ping` was used to send ICMP ECHO requests from `peng.muc.infineon.com` host shown in Figure 3.3. Different packet sizes and sending rates were used to study the performance of Danube.

In order to evaluate the DMA module, the measurements were split into two types:

1. **RX (Receiver side)**: involves the reception of the packet
2. **TX (Transmitter side)**: involves re-transmission the packet

Within the RX, the measurements were conducted at three different points selected based on the hierarchy of the system:

1. `dma-core.c::dma_device_read()`

Measured from `danube_sw.c::switch_hw_receive()`. Measurement was simply done by surrounding the function call by the measurement macros.

2. `dma-core.c::do_dma_tasklet()`

Measured by putting the macros at the beginning and end of the function itself.

3. Application

Defined as the path from the beginning of `dma-core.c::do_dma_tasklet()` till the end of `icmp.c::icmp_rcv()`.

Within the TX, two measurement points were used:

1. `dma-core.c::dma_device_write()`

Measured from `danube_sw.c::switch_hw_tx()` in a similar fashion to `dma_device_read()`.

2. Application

Covers the total time needed to execute `icmp.c::icmp_echo()`. The measurements macros were placed at the beginning and end of this function.

It is worth noting that each point is included in the next point (e.g. For RX, DMA read is included in DMA tasklet and the tasklet itself is included in the application). Each measurement was repeated several times ( $\sim 10$ -15) to overcome any caching and transient effects. Table 4.2 shows the measured quantities in each case.

Quantity	CP0 Event #	CP0 Counter
Cycles	0	0/1
Instructions completed	1	0/1
Branches	2	0
Branch mis-predictions	2	1
Instruction cache accesses	9	0
Instruction cache misses	9	1
Data cache accesses	10	0
Data cache misses	11	0/1
Stalls	18	0
Un-cached loads	33	0
Un-cached stores	33	1
Instruction cache miss stall cycles	37	0
Data cache miss stall cycles	37	1
Un-cached load stall cycles	40	0
Load to use stalls	45	0

Table 4.2: Events measured during the profiling

It is important to mention that during the measurements, only *two* events were measured at a time. This is because the MIPS core provides two counters only. Table 4.3 shows the quantities that are measured together (i.e. in the same run) due to their close inter-dependence. The rest are measured alone.

Counter 0	Counter 1
Cycles	Instructions completed
Branches	Branch mis-predictions
Un-cached loads	Un-cached stores
Instruction cache misses	Instruction cache miss stall cycles
Data cache misses	Data cache miss stall cycles

Table 4.3: Events measured concurrently during the profiling

Another important note is regarding the `ping` host. The host fragments all the packets that are larger than 1KB. Table 4.4 shows the number of fragments per packets that are sent from the host. From Danube point of view, DMA subsystem treats each fragment as a separate transaction.

ICMP Data Size (in bytes)	# of Packets	# of DMA reads	# of DMA Writes
64	1	1	1
128	1	1	1
256	1	1	1
512	1	1	1
1024	1	1	1
2048	2	2	2
4096	3	3	3
8192	6	6	6
16384	12	12	12

Table 4.4: Number of fragments per packet that are sent from the `ping` host

## 4.8 Results and Analysis

In this section, the results obtained via the software solution proposed in Section 4.5 are presented. Using the new proposed solution, all the un-cached loads and stores were completely eliminated from the core DMA operations. Table 4.5 outlines the overhead due to the new solution.

	DMA read	DMA write
D-cache misses	1	2
Instructions	4	8

Table 4.5: Overhead due to the new DMA solution

Figures 4.6 and 4.7 show the overall improvement in the RX and TX handlers of the application due to the new solution.

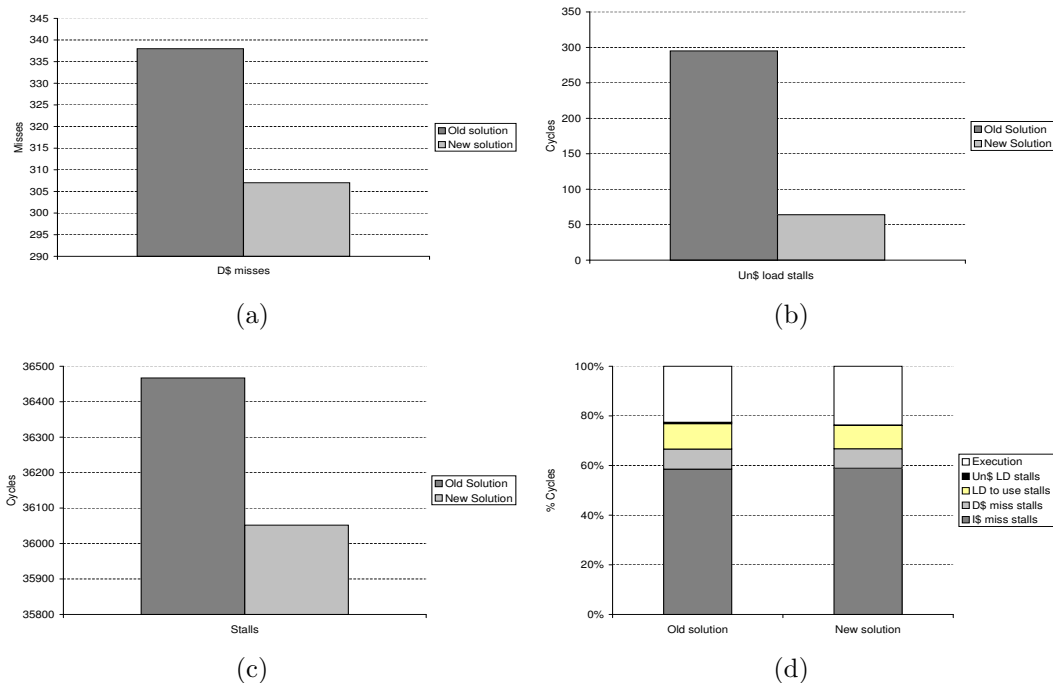


Figure 4.6: Improvement in the RX handler of ping due to the new solution, packet size = 1066 bytes: The reduction in: D-cache misses =  $\sim 9.17\%$ , un-cached load stalls =  $\sim 78\%$ , total stalls =  $\sim 1.13\%$ , and the improvement in the execution cycles =  $\sim 4.0\%$ .

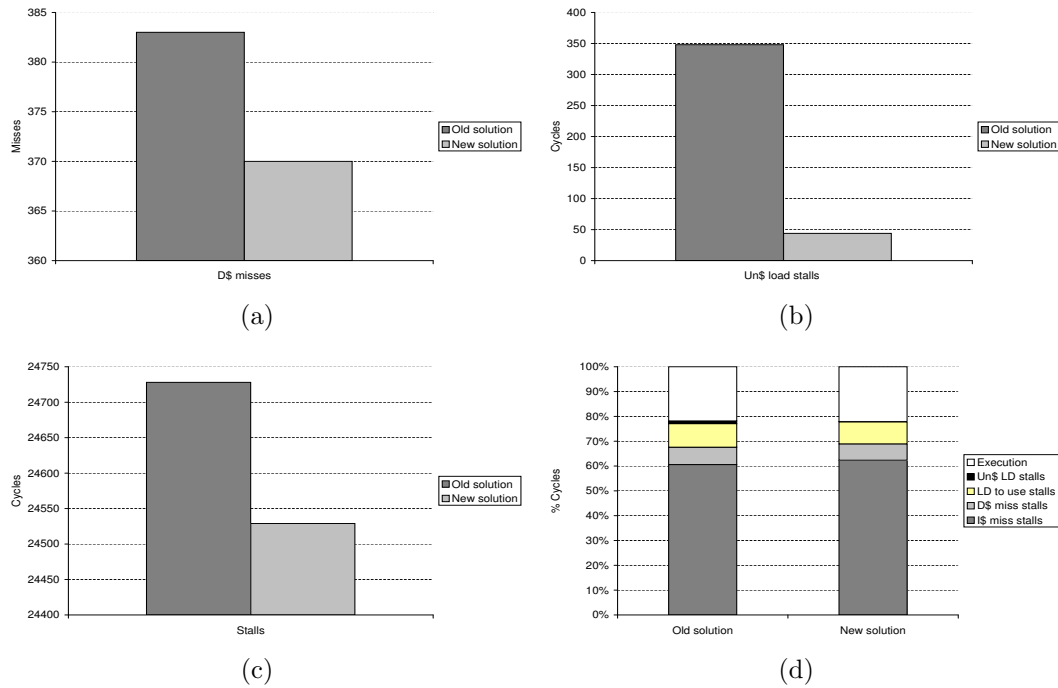


Figure 4.7: Improvement in the TX handler of `ping` due to the new solution, packet size = 1066 bytes: The reduction in: D-cache misses =  $\sim 3.3\%$ , un-cached load stalls =  $\sim 87\%$ , total stalls =  $\sim 0.4\%$ , and the improvement in the execution cycles =  $\sim 1.24\%$ .

## 4.9 Conclusions

CPU and DMAC can be incorporated in one system without the need for a dedicated cache coherence snooping logic. Software-based solutions can be used to maintain the data integrity. Such solutions are attractive for low-cost embedded systems since it saves some of the chip area needed for the cache coherence HW. The optimization implemented in Section 4.5 results in better CPU utilization. Nevertheless, the overall improvement is dependent on the application and its usage of the DMA operations. A further improvement which can be implemented is the optimization of buffer allocation/deallocation functions used within the core DMA operations. These buffer manipulation functions account for around 60-70% of the total execution time of the core DMA operations. Moreover, 60-70% of the total cycles are stall cycles due to instruction cache misses. This observation motivated us to investigate the instruction cache utilization which is discussed in Chapter 5.

# Chapter 5

## OS Instruction Cache Utilization

### 5.1 Background

During the measurements which were conducted in the course of the DMA issue, an interesting phenomenon was observed: For some applications (e.g. Linux IP stack), up to 70% of the total cycles are stall cycles due to instruction cache misses caused by the OS code although that the instruction cache miss rate is only in the range between 6.07% and 6.20% as shown in Table 5.1. This was an intriguing observation since it shows that the core is badly utilized while executing the OS code. However, such an observation is not a new one. It was first noted by [4] and confirmed a decade later by [77]. Their conclusion was that OS codes exhibit a bad instruction cache utilization. What makes it worse now is the increasing gap between CPU and memory which results in huge miss penalties [10, 32].

Module	I-cache accesses	I-cache misses
<b>DMA Read</b>	758	49
<b>Tasklet</b>	2207	146
<b>Application</b>	11276	685
<b>DMA Write</b>	1025	106
<b>Application</b>	7237	449

Table 5.1: Original DMA solution cache accesses and misses

Previous research concluded that OS codes exhibit bad instruction cache hit rate due to the nature of the code. OS codes usually are organized as large number of small modules with each module exhibiting low locality<sup>1</sup>. Furthermore, there are a lot of function calls within the OS to request services from different modules. This leads to a lot of jumps within the address space which eventually results in *cache pollution*.

There are two main parameters which control this issue according to [32]:

1. **Instruction cache miss rate:** This parameter depends on *both* of the software and architecture. Thus it can be modified through hardware and software based solutions.
2. **Instruction cache miss penalty:** This parameter is completely dependent on the architecture and the memory technology used. Thus a hardware-based solution is needed to change this quantity.

## 5.2 Previous Work

OS instruction cache utilization is a well-studied area [4, 77, 16, 76]. Many hardware and software based solutions has been devised. In this section, a short survey of the existing solutions is presented.

### 5.2.1 Hardware Solutions

#### **Tweaking the cache parameters: Larger caches and associativity**

The effect of increasing the cache size and associativity has been extensively studied by researchers. [32] provide a detailed discussion on when larger caches are better and when larger and more complex caches can increase the hit time. Fedorova et al. in [22] provide a detailed analysis of the impact of larger caches on multicore and CMT processors especially for L2 cache.

---

<sup>1</sup>*Locality* in this context is defined to be the ratio of I-cache accesses to I-cache misses.

## Instruction prefetching

Instruction prefetching is a technique that has been deployed since 1970s [67]. Nevertheless, the tremendous changes in CPU and memory speeds and the advent of multiprocessors impose a new challenge. [69, 33, 68] provide a comprehensive survey on the existing instruction prefetch techniques for single core CPUs and CMPs. They classify the prefetching techniques into two main categories:

1. **Sequential instruction prefetchers** According to [69], sequential misses account for 40-60% of the total misses for typical workloads. Thus it is quite important to eliminate these misses. Luckily, sequential misses can be easily eliminated using a *next- $N$  lines* prefetcher (known also as *fall-through* prefetcher). This type of prefetchers as its name implies prefetches the next  $N$  lines starting from a given starting address. The starting address can be simply the current active line or a *prefetch lookahead distance*. The starting address should be selected in such a way that the memory latency is hidden as much as possible.
2. **Non-sequential instruction prefetchers** This type of prefetcher tries to eliminate the misses resulting from transitions to distant lines. Such misses occur a lot in applications composed of small functions or applications exhibiting frequent changes in control flow. OS codes exhibit both (many small functions with high control flow change rate) [51]. This type of prefetchers can be further classified according to [69, 33] into two types:
  - **History-based:** Uses a history table to predict the prefetched line.
  - **Execution-based:** On the contrast of history based prefetchers, execution based prefetchers scout ahead of the main thread of execution and prefetch the cache lines that constitute the predicted path of execution.

## Multilevel caches

Multilevel caches have been deployed in almost all recent processors designs. [22] studied the relationship between IPC and L1 and L2 caches sizes (L1 private, L2

shared). It concluded that L2 cache size has the highest effect on IPC in CMPs and CMTs.

### **Increase of external memory bandwidth**

This solution is useful in case the external memory bandwidth is smaller than the CPU bandwidth. However, this solution requires more pins on the chip which means extra cost [32].

### **Advanced memory technology usage**

According to [29], using RDRAM instead of SDRAM can save up to 33% of the access time. However, this solution requires extra licensing fees from Rambus Inc.

## **5.2.2 Software Solutions**

### **Enhanced code layout**

This technique relies on generating a profile of the OS code and re-organize the code layout according to that profile in such a way that reduces the address collisions. Disadvantages of such an approach include the need for source code modifications (i.e. insertion of locking code), its dependence on the specific cache configuration, the need for tailoring it for different platforms, and the fact that OS profiling can be difficult to obtain in embedded systems.

### **Reduction of function inlining**

Function inlining is a technique available in some programming languages such as C [37]. It aims to reduce the overhead of function calls by replacing the call with the body of the function itself. Inline functions are used a lot within the Linux kernel [17, 18], nevertheless, inlining is a double-edged sword since it makes the code larger which translates into more cache misses. [31, 79, 58] have discussed this issue extensively on the Linux Kernel mailing list. It has been shown that a performance gain of up to 3% can be achieved via simply eliminating the `inline` construct [31].

### Cache fair scheduling and cache oblivious algorithms

This approach tries to employ cache-friendly algorithms within the OS. A prominent candidate for utilizing such algorithms is the scheduler. [23, 24] proposed a cache-fair scheduling algorithm for CMT processors that reduces the variation in application performance on multicore chips by up to a factor seven. Such an algorithm takes into the account the cache miss rate for each thread while scheduling.

## 5.2.3 Hybrid Solutions

### On-chip memories

On-chip memories emerged as an attractive solution to reduce the cache miss rate since they have much lower latency than the external memory. They have been used in IBM Cell processor to store the program and data in each of the Synergistic Processing Elements (SPU) [30]. They have also been used in Intel 80-tiles chip to integrate the memory with the processors into a single chip via 3-D stacking [78]. This solution reflects a trend in putting the memory closer to the CPU in order to reduce the memory latency. Scratchpad RAM is one particular example of on-chip memories. It is used for temporary storage of calculations, data, and other work in progress. Such a type of memory is seen by some researchers as an efficient alternative to L1 cache [12]. Some MIPS processors can be configured to provide a scratchpad memory. One prominent example of a MIPS-based system using scratchpad memory is the Sony PlayStation 2 [32].

One difficulty that emerges with scratchpad memories is the need for programmer and compiler awareness. The programmer must know what portions of the application should go to the scratchpad. Such a knowledge in many cases is difficult to obtain in advance while development and hence requires profiling the application to find the "hot spots"<sup>2</sup>.

---

<sup>2</sup>*Hot spots* are defined as the addresses containing the most used instructions.

## Cache locking

Some cache controllers provide the ability to *lock* certain lines within the cache. In this way, the programmer has the ability to lock certain lines which contain the hot spots of the application. Disadvantages of such an approach include the need for source code modifications (i.e. insertion of locking code), its dependence on the specific cache configuration, the need for tailoring it to different platforms, and the fact that OS profiling can be difficult to obtain in embedded systems.

## 5.3 Optimizations for Linux

As outlined in Section 5.2, many solutions have been devised to overcome the problem of poor I-cache utilization by the OS code. However, due to the project's scope and time restrictions, the choices that can be used for Linux running on Danube were limited to *software solutions*. Taking into consideration that MIPS core provides the ability to lock cache lines (see Section 3.1.3), cache locking was the first method to be implemented on Danube.

### 5.3.1 Cache Locking

The approach here is to find the *most used* functions within the kernel, and then *lock* these functions in the cache using the cache locking feature provided by the core. The procedure can be outlined as follows:

1. Find the most used functions within the kernel
2. Lock the cache lines containing these functions

Finding the most used functions within the kernel requires profiling the kernel. As mentioned in Section 2.10, none of the available profiling tools at the time is capable of profiling the Linux Kernel 2.4 series on MIPS. Thus, we are limited in two choices:

1. Locking the DMA core functions and observing the resulting effect

2. Using the profiling module provided via Lauterbach in-circuit debugger to get a system-wide profile

The second choice turned out not to be useful because the profiling module in Lauterbach ICD is based on *stop-and-go* sampling. This profiler was not able to capture the IP stack behavior and it showed that 97% of the total time was spent in the `cpu_idle()` routine. Hence the decision was taken to lock the cache lines containing the core DMA routines.

To get a speedup out of cache locking, the locking must take place outside the main program loop. Figure 5.1 illustrates this idea. The cost for fetching and locking the cache lines can only be amortized if the lock is done once at the initialization phase of the kernel. Otherwise, the total cost will be equivalent to that of the existing solution since the memory fetches will take place each time.

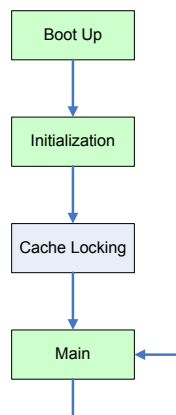


Figure 5.1: Proposed locking mechanism

As a result, the decision was to perform the lock early during the initialization of the kernel. The most interesting routines where the locking can be done are:

1. `arch/mips/kernel/head.S::kernel_entry()`
2. `init/main.c::start_kernel()`
3. `arch/mips/kernel/setup.c::setup_arch(&command_line)`

However, the testing results showed that I-cache misses are not changing after the locking. Thus changing the lock location across the kernel initialization flow [53] was tried. Nevertheless, no improvement was observed. By referring to MIPS errata [40], it turned out that the core revision has a bug in the RTL implementation that affects the cache locking mechanism. However, MIPS provided a workaround to overcome this problem. After applying the workaround, the problem persisted: no improvement after locking the DMA core routines in the cache. This pushed the author to look for support from the Linux community.

After discussing the issue on LinuxMIPS mailing list with Ralf Bächle (the maintainer of Linux@MIPS), it turned out that Linux kernel is using a lot of cache invalidations and this will remove any locked cache lines. According to Ralf [11]:

Linux generously uses Index cacheops and so would also blow away wired cache lines and that would need to be prevented.

After consulting MIPS Technologies support team, they verified the same conclusion [71]: *Linux kernel is using invalidation operations which remove any locked cache lines.*

Based on the above results, cache locking could not be used for improving the performance on Linux running on Danube.

## 5.4 CPU Subsystem Behavior

Several experiments were conducted to analyze the CPU subsystem behavior and performance. The aim of these experiments is to study the effect of some architectural decisions or optimizations on the overall performance.

### 5.4.1 Effect of Cache Size Reduction

The aim here is to study the effect of reducing the cache size on the I-cache miss rate. To be able to reduce the cache size, the `CACHE` instruction has been used to lock the cache lines into a non-existent address. Locking the lines into a non-existent address avoids their eviction by the Linux kernel. Different cache sizes have been

tried including 16, 13, 10, 7, and 4 KB. Figure 5.2 shows the effect of reducing the cache size on the total number of cycles.

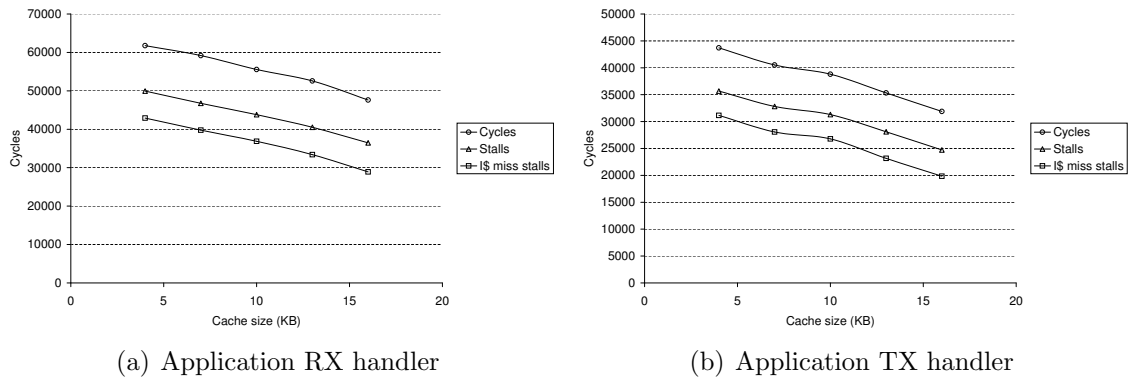


Figure 5.2: Effect of cache size reduction on the total number of cycles, packet size = 1066 bytes

Figure 5.3 shows the effect of cache reduction on the number of I-cache misses. An interesting observation here is that for a 75% reduction in cache size, I-cache misses increase by 48% of its original value, but total cycles increase by 29% only. This means that the instruction cache can be reduced in order to use some of the chip area for implementing a scratchpad memory without hurting the performance significantly.

#### 5.4.2 Effect of the Absence of Critical Word First Filling

Figure 5.4 shows the effect of the absence of critical word first filling on the memory latency. Memory subsystem is based on DDR333 running @ 166 MHz with a 16-bit external interface to the memory chip and a CAS latency of 2.5 cycles. As can be seen, the latency increases by 24% if the requested word lies at the end of the cache line.

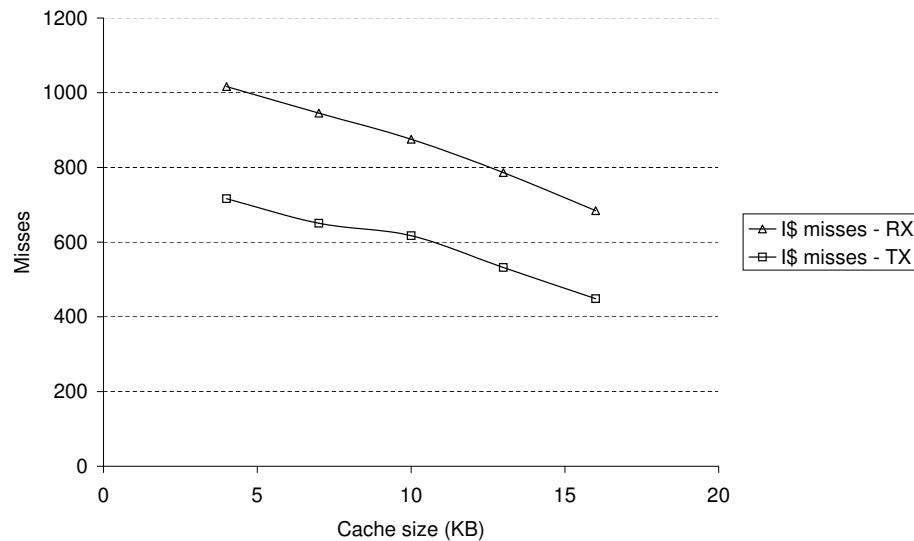


Figure 5.3: Effect of reducing the cache size on I-cache misses, packet size = 1066 bytes

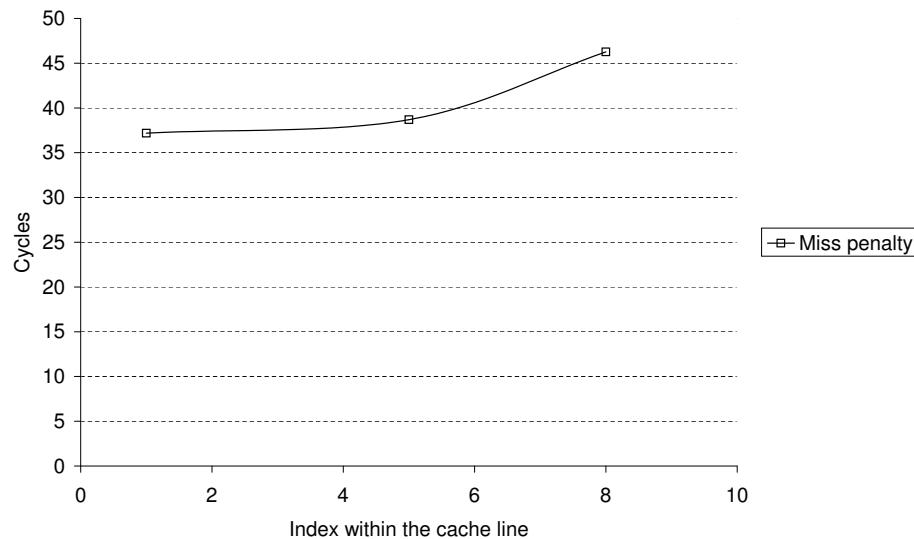


Figure 5.4: Effect of absence of critical word first filling

### 5.4.3 Effect of Cache Locking and Instruction Pre-fetching

Figure 5.5 is obtained via locking an instructions sequence in the cache and executing it while all interrupts are disabled to prevent the intervention of the Linux Kernel (i.e. through evicting the locked cache lines). All the instructions within the sequence do

not cause pipeline hazards. Nevertheless, the pipeline will stall at the beginning of each cache line due to the need to perform an I-cache fill operation.

Figure 5.5 can be interpreted in two ways:

1. The potential gain out of an instruction pre-fetcher on the performance of a sequential code that has no instructions that can cause pipeline hazards
2. The potential gain out of cache locking on the performance of a given code provided that all the control/system instructions (e.g. branch, jump, call, return, etc...) in the code will change the execution flow into instructions within the locked lines

One can easily see the potential gain that can be obtained from the above techniques when used for the proper workloads<sup>3</sup>.

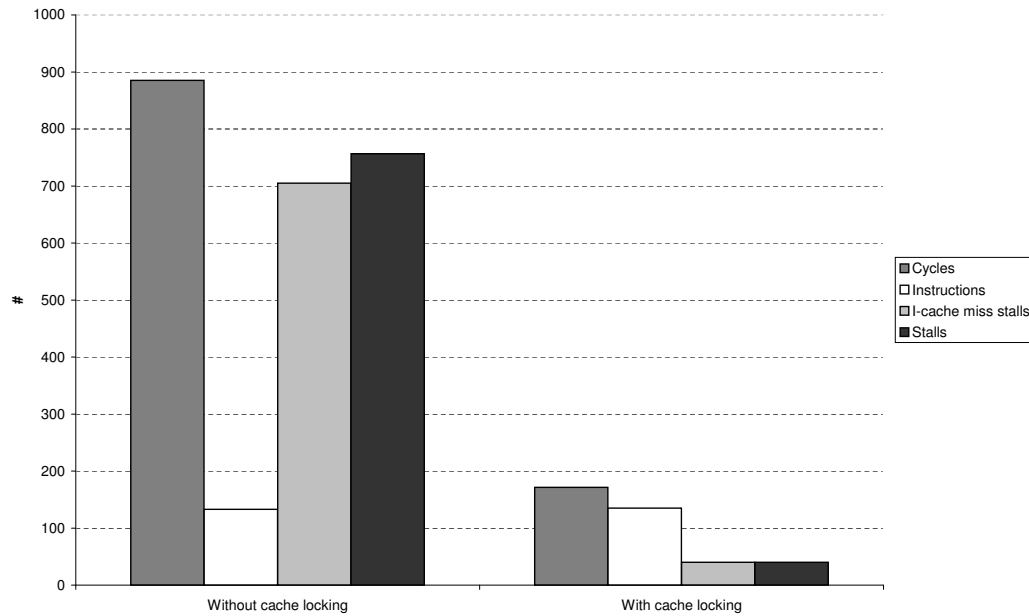


Figure 5.5: Effect of the instruction prefetching on sequential hazard-free code streams

<sup>3</sup>The remaining I-cache miss stalls after locking are due to a cache miss induced by the measurement macro.

## 5.5 Conclusions

Instruction cache miss rate for OS codes is a significant issue. More effort should be invested in overcoming this issue due to its significant effect on the system performance. The increasing processor-memory gap and the contention for memory caused by the advent of MPSoCs will increase the effect of such an issue dramatically. Cache locking has been tested on the aim of reducing the I-cache miss rate. The technique did not yield any gain due to the restrictions imposed by the Linux port to MIPS. It might be of interest to embedded systems designers to enable such a feature in the future for the Linux Kernel port to MIPS. Suggestions to reduce the effect of the I-cache miss rate include using a multithreaded processor (e.g. MIPS32 34KE family [41]). Another possibility will be the addition of an instruction prefetcher and a scratchpad memory. Finally, several experiments showed that the instruction cache size can be reduced without hurting the performance significantly. Critical word first filling can reduce the cache miss penalty by up to 25%. An important conclusion is that I-cache utilization is an important issue that can affect the performance of unicones significantly. This phenomenon can be of a greater effect on multicore systems. This led us to investigate the effect of instruction reuse, number of cores, and the memory bandwidth on the overall system IPC. The result is a qualitative analysis that is presented in Chapter 6.

# Chapter 6

## System Throughput in MPSoCs

### 6.1 Background

As explained in Chapter 5, instruction cache miss rate can play an important role in determining the overall performance of the system. Looking into the performance analysis which we have done for Danube, the following remarks can be outlined:

- **Locality in OS and protocol processing codes is bad:** Locality is defined in Chapter 5 as:

$$Locality = \frac{Instruction\ cache\ accesses}{Instruction\ cache\ misses} \quad (6.1)$$

From Table 5.1, it can be easily seen that the locality for the Linux Kernel 2.4 series running on MIPS32 24KEc is in the range between 16.12 - 16.46 instruction/miss which can be interpreted as 0.5 miss per cache line<sup>1</sup>.

- **Memory latency is the bottleneck:** As explained in Section 2.6, processor-memory gap is increasing. Thus memory bandwidth must be increased in order to overcome the long delays incurred by accessing the memory.

Using these two remarks and the fact that future designs are shifting into MPSoC paradigm, we want to gain a better understanding of the factors that control the

---

<sup>1</sup>Recall that the instruction cache line size in Danube is 32 bytes (eight instructions)

system performance.

## 6.2 Evaluation Methodology

To be able to perform a comprehensive analysis, we need to determine the factors and relationships that affect the overall system throughput for MPSoCs. For the purpose of this work, we use the following quantities:

- **Instruction Reuse** (*Reuse*): The number of times an instruction is executed before being evicted from L1 cache.
- **Number of Cores** (*N*)
- **Core IPC** (*CIPC*): The IPC of a single core
- **System IPC** (*SIPC*):  $SIPC = N \times CIPC$
- **Memory Bandwidth** (*BW*): The maximum number of instructions that can be delivered by the memory subsystem in a single CPU cycle to all of the cores and it is measured in instructions/cycle.
- **Instruction Cache Line Length** (*L*)

### 6.3 System Architecture

Figure 6.1 shows the system architecture that incorporates the ARM MPCore processor<sup>2</sup>. This system is modeled and simulated under the CoMET environment. Table 6.1 highlights the different configurations used during the simulations.

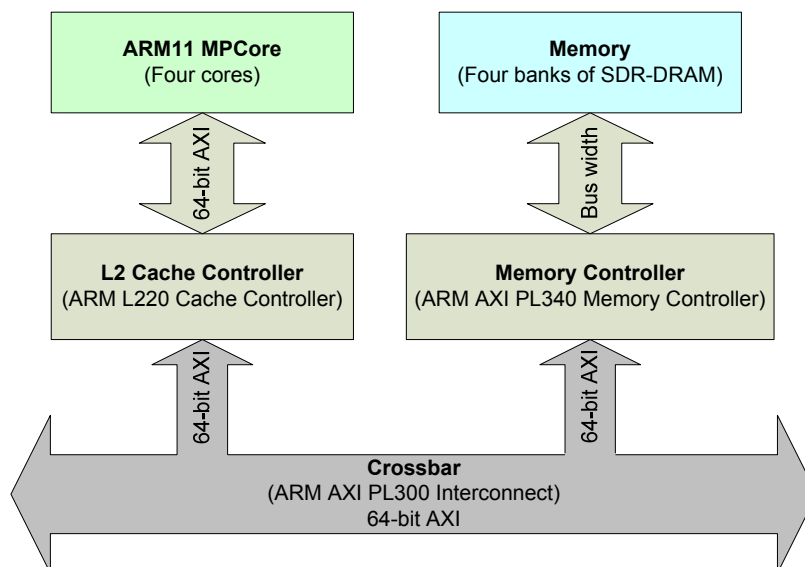


Figure 6.1: MPCore-based System architecture

$BW$ (in I/C)	2	1	0.5	0.25
Core frequency	100 MHz	100 MHz	100 MHz	100 MHz
Memory frequency	100 MHz	50 MHz	25 MHz	50 MHz
Memory bus width	64-bit	64-bit	64-bit	16-bit

Table 6.1: Parameters configuration for the memory bandwidth simulations

### 6.4 Results and Analysis

The experiments involved studying the effect of the following three factors:

<sup>2</sup>Bus width between the memory and the memory controller is varied during the simulations. See Table 6.1

1. **Instruction Reuse** (*Reuse*)
2. **Number of Cores** (*N*)
3. **Memory Bandwidth** (*BW*)

The analysis aims in determining the effect of these three factors on the performance of overall system under the following assumptions:

- Instructions executed on the cores are neither data transfer instructions (e.g. load, store, etc...) nor control/system instructions (i.e. jumps, branches, etc...). Moreover, all of the executed instructions do not cause pipeline hazards. This assumption aims to guarantee that all the traffic on the system bus is only for fetching the instructions.
- Each cores executes instructions from a private region in the memory (i.e. no shared code). Nevertheless, all instructions are located within a single memory bank.
- Each core has its private L1 instruction cache
- L2 cache is disabled (i.e. L220 cache controller is working in bypass mode).
- No interrupts are generated during the execution flow

Although these assumptions are somehow not realistic, they simplify the analysis significantly. An extensive and thorough analysis can be carried out in a future study, using the groundwork prepared by this thesis.

The test scenarios utilize custom benchmarks that are written to comply with the previous assumptions. CoMET Target Software Programming Interface (TSPI) [73] is used to profile the benchmarks.

### 6.4.1 Effect of Memory Bandwidth

Figure 6.2 shows the following trend: for low memory bandwidth and low instruction reuse, multicore systems do not provide any performance gains. The “straightforward” solution to overcome this issue is by providing higher bandwidth. Another

solution is to increase the instruction reuse via using some sort of on-chip memory where frequently accessed code is kept. For ARM architectures, this kind of memory is called *Tightly-Coupled Memory (TCM)*. It can be configured as two independent instruction TCM (ITCM) and data TCM (DTCM) with sizes configurable between 1 KB up to 4 MB.

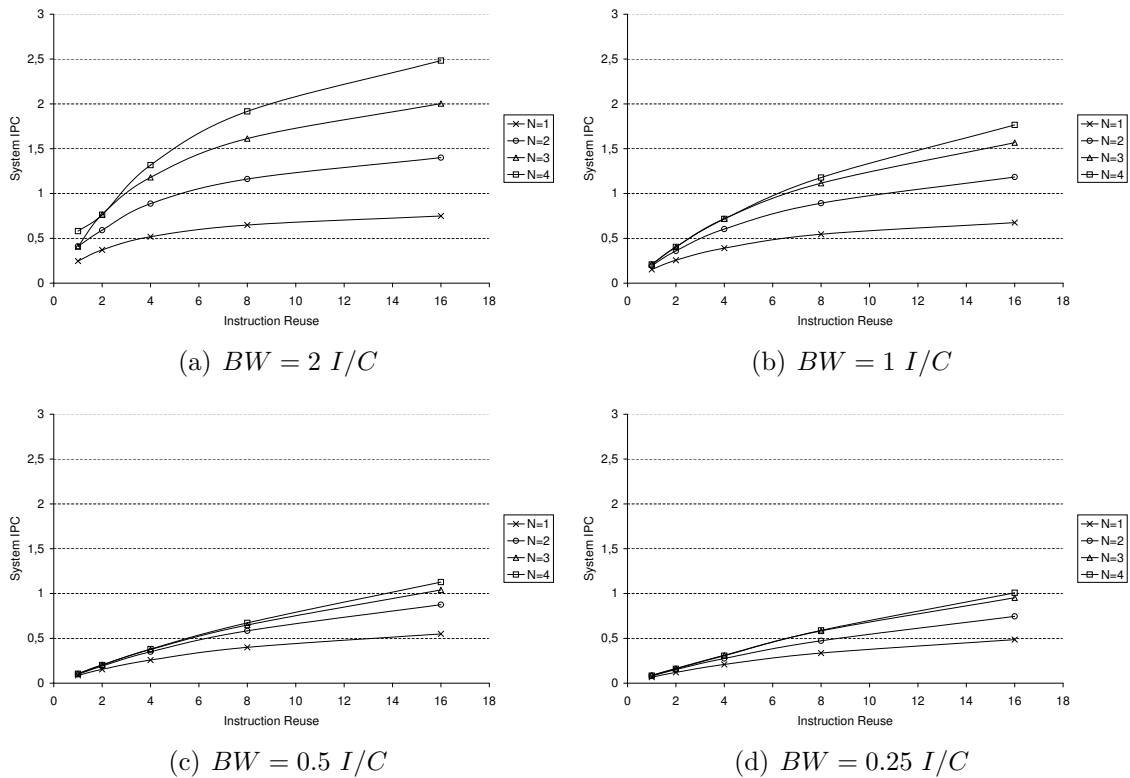


Figure 6.2: Effect of memory bandwidth on system IPC

### 6.4.2 Effect of Number of Cores

Figure 6.3 gives the following conclusion: For low memory bandwidth ( $BW = 0.25$  and  $BW = 0.5$ ), increasing the number of cores or the instruction reuse does not provide a significant speedup. As the bandwidth increases, the effect of multicore becomes more apparent.

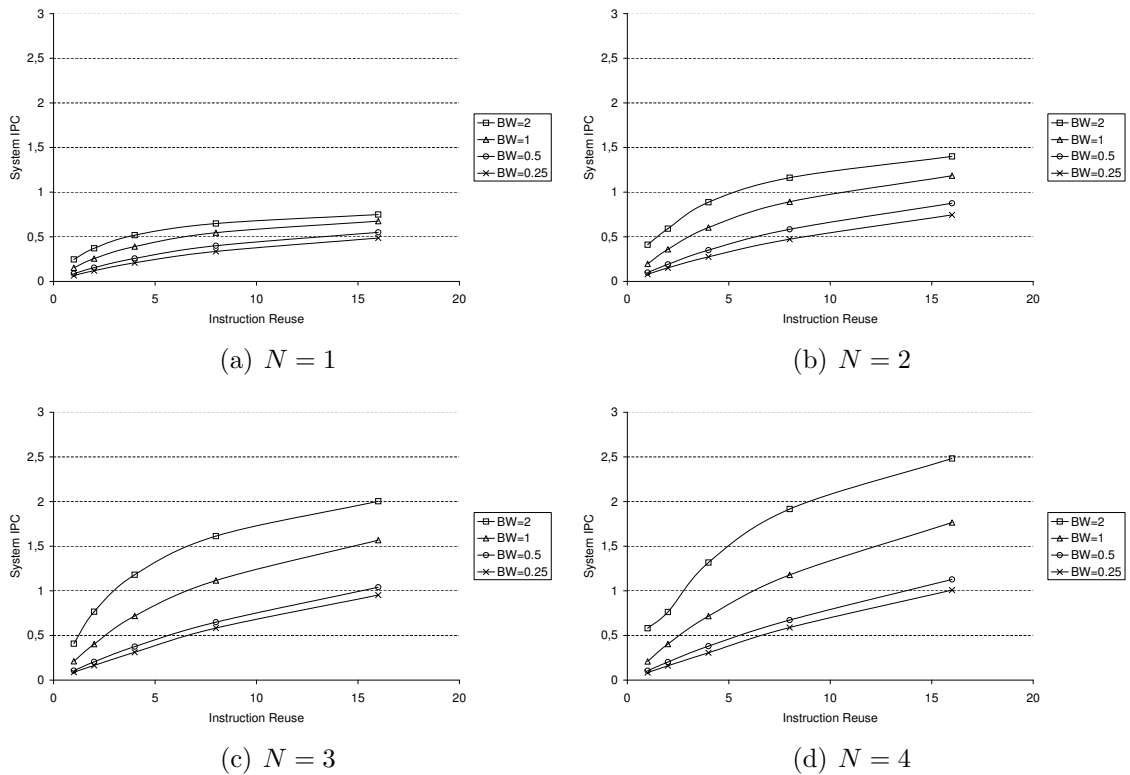


Figure 6.3: Effect of number of cores on system IPC

### 6.4.3 Effect of Instruction Reuse

Figure 6.4 shows an interesting phenomenon: For high instruction reuse (i.e. good locality), increasing the memory bandwidth and the number of cores provides a nice speedup. However, if the instruction reuse is low, then multicore and even higher memory bandwidth does not provide any significant speedup. This is justified by the fact that the core needs to access the main memory quite often for fetching the instructions which leads to processor stalls most of the time. If we recall from Section 6.1 that in the Linux Kernel, IP stack instruction reuse is around 2, then it becomes quite evident that bottleneck is the software itself. One way to overcome this problem is through using minimal OS images that can be fit into a low latency on-chip memory.

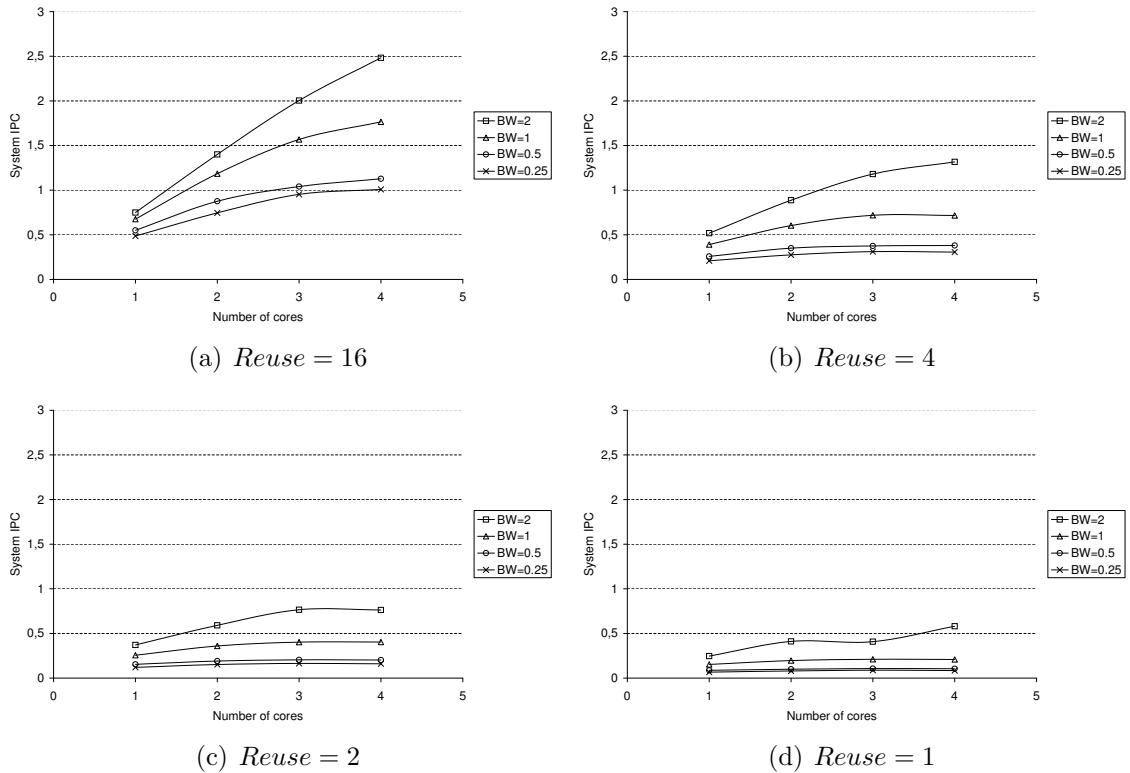


Figure 6.4: Effect of instruction reuse on system IPC

## 6.5 Conclusion

Software characteristics (represented in the instruction reuse) and system characteristics (number of cores and memory bandwidth) have a great effect in determining the overall system throughput. In order for MPSoC systems to achieve a modest performance, memory bandwidth should be increased to keep up with the number of cores. One way to accomplish this is through using switched network-on-chip accompanied with on-chip memories. Switched NoCs provide an attractive and scalable solution since the total system bandwidth increases with increasing the number of nodes. Instruction reuse is a limiting factor that is determined by the type of the applications running on the system. For low locality codes (e.g. operating systems, protocol processing codes, and device drivers), on-chip memories can be an attractive solution to reduce the software effect on the system throughput. Future work can

focus on performing the type of analysis conducted in Section 6.4 on different architectures (e.g. with L2 caches, different interconnects, etc...) while running real-world software codes which exhibit different localities.

# Chapter 7

## Summary and Outlook

The objective of this thesis was the investigation of the non-cache coherent DMA transfers in MPSoCs, the low instruction cache utilization by the Linux Kernel 2.4 series, and the effect of some hardware and software characteristics on the system throughput in MPSoCs. To accomplish that, both of the empirical and simulation approaches were used. Danube platform developed by Infineon Technologies AG was used to study the first two issues while a simulated system based on ARM MPCore architecture was used to study the last one.

The primary objectives have been met. Two new solutions have been implemented and tested to improve the performance of the non-cache coherent DMA transfers. One solution is a pure software solution while the other one is a hybrid one combining software and hardware changes. Techniques for reducing the instruction cache miss rate for the Linux Kernel 2.4 series have been investigated and applied. Cache locking could not be used due to the implementation restrictions of the Linux port to MIPS. A qualitative analysis of MPSoC systems throughput was conducted. Such analysis gives meaningful insights to the factors that governs MPSoCs performance.

Issues that need to be tackled are the poor instruction cache utilization for operating systems and protocol processing codes that run on MPSoCs. Such an issue turns to be of a great impact on the system performance because of the increasing memory latency. Future work can focus on exploring solutions to increase the memory bandwidth and reduce the effect of low cache utilization codes.

# Bibliography

- [1] Infineon Technologies AG. *Preliminary Tool Brief: EASY 50712: Danube Reference Board, ADSL2/ADSL2+ IAD Gateway*, 2006.
- [2] Infineon Technologies AG. *Product Brief: Danube: ADSL2/2+ IAD-on-Chip Solution for CPE Applications, PSB 507x2*, 2006.
- [3] Infineon Technologies AG. *DANUBE Family - PSB 50702, PSB 50712*, Retrieved on August 1, 2007. <http://www.infineon.com/applications/danube>.
- [4] Anant Agarwal, John L. Hennessy, and Mark Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems (TOCS)*, 6(4):393–431, 1988.
- [5] AMD. *Product Brief: Quad-Core AMD Opteron™ Processor*, Retrieved on October 16, 2007. [http://www.amd.com/us-en/Processors/ProductInformation/0,,30\\_118\\_8796\\_15223,00.html](http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_15223,00.html).
- [6] ARM. *PrimeCell® DMA Controller (PL080): Technical Reference Manual*, August 31, 2004. Revision: r1p3.
- [7] ARM. *ARM11 MPCore Processor: Technical Reference Manual*, August 11, 2006. Revision: r1p0.
- [8] ARM. *AMBA Overview*, Retrieved on November 22, 2007. <http://www.arm.com/products/solutions/AMBAHomePage.html>.
- [9] ARM. *ARM11 MPCore*, Retrieved on November 23, 2007. <http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html>.

- [10] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18:2006–183, 2006.
- [11] Ralf Bächle. *Re: cacheops.h & r4kcache.h*, Retrieved on October 24, 2007. <http://www.linux-mips.org/archives/linux-mips/2007-08/msg00015.html>.
- [12] R. Banakar, S. Steinke, B.S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. *Proceedings of the tenth international symposium on Hardware/software code-sign*, pages 73–78, 2002.
- [13] L. Benini and G. De Micheli. Networks on Chips: a New SoC Paradigm. *Computer*, 35(1):70–78, 2002.
- [14] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, and M. Diaz-Nava. Component-Based Design Approach for Multicore SoCs. *Proceedings of the Design Automation Conference (DAC)*, 2002.
- [15] Marco Cesati and Daniel P. Bovet. *Understanding the Linux Kernel*. O’Reilly, 2003.
- [16] J.B. Chen and B.N. Bershad. The Impact of Operating System Structure on Memory System Performance. *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 120–133, 1994.
- [17] Jonathan Corbet. *The cost of inline functions*, Retrieved on August 23, 2007. <http://lwn.net/Articles/82495/>.
- [18] Jonathan Corbet. *Drawing the line on inline*, Retrieved on August 23, 2007. <http://lwn.net/Articles/166172/>.

- [19] Intel Corporation. *Intel® Core™ Duo Processors*, Retrieved on July 31, 2007. <http://www.intel.com/products/processor/coreduo/>.
- [20] Angel L. DeCegama. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Hardware (vol. 1)*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.
- [21] B. Doris, M. Jeong, T. Kanarsky, Y. Zhang, RA Roy, O. Dokumaci, Z. Ren, F.F. Jamin, L. Shi, W. Natzle, et al. Extreme Scaling with Ultra-thin Si Channel MOSFETs. *Electron Devices Meeting, 2002. IEDM'02. Digest. International*, pages 267–270, 2002.
- [22] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of Multi-threaded Chip Multiprocessors and Implications for Operating System Design. *Proc. of the USENIX 2005 Annual Technical Conf*, pages 17–04, 2005.
- [23] Alexandra Fedorova. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Harvard University, September 2006.
- [24] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Cache-Fair Thread Scheduling for Multicore Processors. Technical report, Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University, October 2006.
- [25] Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
- [26] Michael Gerndt. *Lecture Notes on HPC Architectures*, Retrieved on October 29, 2007. <http://www.lrr.in.tum.de/~gerndt/home/Teaching/SS2007/Hochleistungsarchitekturen/Hochleistungsarchitekturen.htm>.
- [27] Lauterbach Datentechnik GmbH. *LAUTERBACH - TRACE32 ICD In-Circuit Debugger*, Retrieved on October 1, 2007. <http://www.lauterbach.com/icd.html>.

- [28] J. Goodacre and AN Sloss. Parallelism and the ARM Instruction Set Architecture. *Computer*, 38(7):42–50, 2005.
- [29] M. Gries. The Impact of Recent DRAM Architectures on Embedded Systems Performance. *Proceedings of the 26 thEuromicro Conference*, pages 282–289, 2000.
- [30] Michael Gschwind. Chip Multiprocessing and the Cell Broadband Engine. *Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, 2006.
- [31] Stephen Hemminger. *eliminate large inline's in skbuff*, Retrieved on August 23, 2007. <http://marc.info/?l=linux-kernel&m=108310099432154&w=2>.
- [32] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4<sup>th</sup> edition, 2007.
- [33] W.C. Hsu and J.E. Smith. A Performance Study of Instruction Cache Prefetching Methods. *IEEE Transactions on Computers*, 47(5):497–508, 1998.
- [34] IBM. *CoreConnect bus architecture*, Retrieved on November 26, 2007. <http://www.ibm.com/chips/products/coreconnect>.
- [35] Innovative Computing Laboratory (ICL). *Performance API (PAPI)*, Retrieved on September 20, 2007. <http://icl.cs.utk.edu/papi/>.
- [36] Freescale Semiconductor Inc. *C-Port™ Network Processors*, Retrieved on October 27, 2007. <http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=02VS01DFTQ3126>.
- [37] FSF Inc. *Using the GNU Compiler Collection: For gcc version 4.2.2*, Retrieved on October 15, 2007. <http://gcc.gnu.org/onlinedocs/gcc-4.2.2/gcc.pdf>.
- [38] MIPS Technologies Inc. *MIPS32® 24KE™ Processor Core Family Software User's Manual*, December 20, 2005. Revision 01.02.
- [39] MIPS Technologies Inc. *MIPS32™ : Programming the MIPS32® 24K® Core Family*, July 5, 2005. Revision 4.01.

- [40] MIPS Technologies Inc. *MIPS32® 24KE™ Processor Core Family RTL Errata Sheet*, December 22, 2006. Revision 03.10. **Infineon Internal**.
- [41] MIPS Technologies Inc. *MIPS32® 34KE™ Family*, Retrieved on November 29, 2007. <http://www.mips.com/products/cores/32-bit-cores/mips32-34k/>.
- [42] Sun Microsystems Inc. *Improving Application Efficiency Through Chip Multi-Threading*, Retrieved on October 21, 2007. [http://developers.sun.com/solaris/articles/chip\\_multi\\_thread.html](http://developers.sun.com/solaris/articles/chip_multi_thread.html).
- [43] Intel. *Intel® IXP2xxx Product Line of Processors*, Retrieved on October 16, 2007. <http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm>.
- [44] A. Jerraya, H. Tenhunen, and W. Wolf. Guest Editors' Introduction: Multiprocessor Systems-on-Chips. *Computer*, 38(7):36–40, 2005.
- [45] R. Kalla, B. Sinharoy, and JM Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *Micro, IEEE*, 24(2):40–47, 2004.
- [46] NS Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, JS Hu, MJ Irwin, M. Kandemir, and V. Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12):68–75, 2003.
- [47] Donald E. Knuth. *Knuth versus Email*, Retrieved on September 20, 2007. <http://www-cs-faculty.stanford.edu/~knuth/email.html>.
- [48] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.
- [49] R. Kumar, V. Zyuban, and DM Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 408–419, 2005.

- [50] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007.
- [51] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and Improving Operating System Effects in Control Flow Prediction. *SIGARCH Comput. Archit. News*, 30(5):68–80, 2002.
- [52] Dr.-Ing. Jinan Lin. *Protocol Processing Architectures Team, Infineon Technologies AG*. Email: [jinan.lin@infineon.com](mailto:jinan.lin@infineon.com).
- [53] LinuxMIPS. *Linux/MIPS Porting Guide*, Retrieved on October 24, 2007. <http://www.linux-mips.org/wiki/Porting>.
- [54] M. Loghi and M. Poncino. Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs: Snoop-Based Cache Coherence vs. Software Solutions. *Proceedings of the conference on Design, Automation and Test in Europe- Volume 1*, pages 508–513, 2005.
- [55] Mirko Loghi, Martin Letis, Luca Benini, and Massimo Poncino. Exploring the Energy Efficiency of Cache Coherence Protocols in Single-Chip Multi-processors. In *GLSVLSI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 276–281, New York, NY, USA, 2005. ACM.
- [56] Mirko Loghi, Massimo Poncino, and Luca Benini. Cache Coherence Tradeoffs in Shared-memory MPSoCs. *Trans. on Embedded Computing Sys.*, 5(2):383–407, 2006.
- [57] C. McNairy and R. Bhatia. Montecito: a Dual-core, Dual-thread Itanium Processor. *Micro, IEEE*, 25(2):10–20, 2005.
- [58] Ingo Molnar. *[patch 00/2] improve .text size on gcc 4.0 and newer compilers*, Retrieved on August 23, 2007. <http://marc.info/?l=linux-kernel&m=113577055717684&w=2>.

- [59] MD Nava, P. Blouet, P. Teninge, M. Coppola, T. Ben-Ismaïl, S. Picchiottino, and R. Wilson. An Open Platform for Developing Multiprocessor SoCs. *Computer*, 38(7):60–67, 2005.
- [60] Cavium Networks. *OCTEON™ Plus CN58XX Multi-Core MIPS64 Based SoC Processors*, Retrieved on October 16, 2007. [http://www.caviumnetworks.com/OCTEON-Plus\\_CN58XX.html](http://www.caviumnetworks.com/OCTEON-Plus_CN58XX.html).
- [61] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. *ACM SIGPLAN Notices*, 31(9):2–11, 1996.
- [62] OProfile. Retrieved on September 20, 2007. <http://oprofile.sourceforge.net>.
- [63] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier Inc., 3<sup>rd</sup> edition, 2005.
- [64] Mikael Pettersson. *Linux Performance-Monitoring Counters Driver*, Retrieved on September 20, 2007. <http://user.it.uu.se/~mikpe/linux/perfctr>.
- [65] Søren Sandmann. *Sysprof - A System-wide Linux Profiler*, Retrieved on September 20, 2007. <http://www.daimi.au.dk/~sandmann/sysprof/>.
- [66] M. Singhal, N.G. Shivaratri, and N. Shivaratro. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc. New York, NY, USA, 1994.
- [67] AJ Smith. Sequential Program Prefetching in Memory Hierarchies. *Computer*, 11(12):7–21, 1978.
- [68] JE Smith and W.C. Hsu. Prefetching in Supercomputer Instruction Caches. *Supercomputing'92. Proceedings*, pages 588–597, 1992.
- [69] L. Spracklen, Y. Chou, and S.G. Abraham. Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications. *Proc. High-Performance Computer Architecture, IEEE CS Press*, pages 225–236, 2005.

- [70] STMicroelectronics. *STMicroelectronics - STBus Interconnect*, Retrieved on November 22, 2007. <http://www.st.com/stonline/products/technologies/soc/stbus.htm>.
- [71] MIPS Technologies Support. *Case:14614 - MIPS32 24KE core support*, August 22, 2007. **Infineon Internal**.
- [72] VaST Systems. *Datasheet: CoMET - Architecture Development*, 2007.
- [73] VaST Systems. *User Guide: VaST Target Software Programming Interface Functions for Virtual Processor Models*, 2007. **Infineon Internal**.
- [74] VaST Systems. *Architecture & Systems Engineering*, Retrieved on November 23, 2007. <http://www.vastsystems.com/solutions-architecture-systems.html>.
- [75] Infineon Technologies. *Danube, Preliminary User's Manual - Hardware Description*, 2006-07-21 edition, July 2006. Revision 1.0. **Infineon Internal**.
- [76] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 162–174, 1992.
- [77] Josep Torrellas, Chun Xia, and Russell L. Daigle. Optimizing the Instruction Cache Performance of the Operating System. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.
- [78] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, et al. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, 2007.
- [79] Denis Vlasenko. *Large inlines in include/linux/skbuff.h*, Retrieved on October 15, 2007. <http://marc.info/?l=linux-kernel&m=108257574113095&w=2>.

- [80] Josef Weidendorfer. *Lecture Notes on Computer Architecture*, Retrieved on November 28, 2007. <http://www.lrr.in.tum.de/~gerndt/home/Teaching/WS2006/Rechnerarchitektur/MulticoreArchitectures.pdf>.
- [81] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.
- [82] Yongguang Zhang. *Linux Time Management*, Retrieved on October 12, 2007. <http://www.cs.utexas.edu/~ygz/378-02S/lecture11.html>.