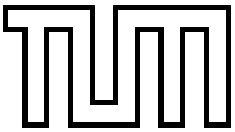

TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Integrierte Systeme

A cycle-accurate coprocessor prototyping platform for system-on-chip

Silvio Dragone

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. Hans-Georg Herzog
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Andreas Herkersdorf
2.	Univ.-Prof. Dr.-Ing. Georg Sigl

Die Dissertation wurde am 24. Juni 2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 29. Mai 2014 angenommen.

For my wife and my son

Acknowledgments

First of all, I am very grateful to my advisor, Professor Dr. Andreas Herkersdorf for his guidance not only with my thesis, but also for his support during my first years at IBM, as my manager.

I am grateful to the IBM Research GmbH, Zurich Research Laboratory, for providing me a fascinating, open and pleasant working environment. Especially I thank my managers, Dr. Antonuis Engbersen and Dr. Christoph Hagleitner, for their continuous support and the many fruitful discussions. I am greatly indebted to them and the rest of the group members, in particular Dr. Gero Dittmann, Dr. Patricia Sagmeister, Dr. Maria Gabrani, Dr. Andreas Döring, and Dr. Jan Van Luntren (random order). Of course there are a lot of colleagues from the Laboratory whom I also would like to thank, especially my office mates Mark Verhappen and Dr. Florian Auernhammer. They made my daily office life a lot more enjoyable.

Sincere thanks go to Philip Roß, Daniel Engeler and Clemens Lombriser, who worked hard on implementing and improving the hardware and software environment. Their questions often made me realize that the answers I thought I had needed a revision.

Further thanks go to the staff of the publications department of the IBM Research GmbH, in particular to Charlotte Bolliger, for reviewing and improving all my publications, and her help with this thesis.

Einen wesentlichen Beitrag zum Erfolg dieser Arbeit haben meine Eltern geleistet. Ohne Ihre Unterstützung, Ihr Verständnis und Ihren Rückhalt hätte ich nie Gelegenheit gehabt, solch eine Aufgabe zu beginnen, geschweige denn, sie erfolgreich abzuschliessen.

Abstract

The semiconductor technology still continues to offer an increasing number of transistors on a single die. However, the ASIC designers do no keep up with that pace because of shorter development time and faster Product Life Cycles (PLC). One possible solution to overcome this productivity gap is to reuse already developed functional cores, so-called Intellectual Properties (IP) or to make use of platform-based System-on-Chip (SoC) solutions. However, the share of the verification step in the development time is about 70% to 80%. This factor demands faster verification methods. The rapid prototyping method offers the benefits of being faster than formal verification and simulation at the same level of detail, when applied to pre-fabricated IP blocks or coprocessors. It also is able to work already with a real system and therefore, further software development can be sped up. However, the disadvantages of the rapid prototyping method are the high costs, the short lifetime, and the generally different timing behavior of the prototype and the future product in a particular CMOS technology.

In this thesis we introduce the Cycle-accurate Coprocessor Prototyping (CyCoP) platform. The platform aims to cycle-accurately emulate coprocessor behavior with its master processor. We show that CyCoP has the ability, with only two simple and highly configurable modules, to cycle-accurately emulate the behavior between a software application and a hardware coprocessor. The implementation of the software application onto the two modules neither modifies its behavior and size, nor its original design flow.

With the CyCoP platform we show how timing information can be obtained from the prototype to provide a cycle-accurate emula-

tion. Therefore, we introduce a new process synchronization algorithm which is adapted to the prototyping conditions.

Because of the modular concept of CyCoP, the platform is hardware independent and therefore it can easily be adapted to any other hardware environment. This has the advantage that users are vendor independent and can change the hardware at any time in the prototyping phase, if, for example, they want to use a more recent hardware.

To show the feasibility of the CyCoP concept, the platform has been implemented in real hardware. Two case studies, one for a tightly-coupled and one for a loosely-coupled coprocessor, demonstrate the feasibility of our concept and the quality of the results.

Zusammenfassung

Die Halbleiterindustrie verkleinert kontinuierlich die Strukturgrösse der Transistoren und kann deshalb weiterhin eine wachsende Anzahl von Transistoren auf einem einzelnen Chip anbieten. Die ASIC-Designer können jedoch mit dieser Geschwindigkeit nicht Schritt halten, da die Systeme stets komplexer werden und somit relativ gesehen, die Entwicklungszeiten immer kürzer werden. Auch der Produkt-Lebenszyklus (PLC) verkürzt sich kontinuierlich. Mögliche Lösungen zur Überwindung dieser Produktivitätslücke sind die Wiederverwendung von bereits entwickelten funktionelle Kernen, so genannte Intellectual Properties (IP), und/oder die Nutzung von Plattformbasierenden System-on-Chip (SoC) Lösungen. Jedoch liegt der Hauptanteil während der Entwicklungszeit in der Verifikation, welche etwa 70% bis 80% davon ausmacht. Dieser Faktor verlangt daher vor allem auch schnellere Verifikationsmethoden. Die Rapid-Prototyping-Methode bietet den Vorteil, dass sie auf derselben Detaillierungsstufe schneller ist als die Formale Verifikation und die Simulation, insbesondere auf vorgefertigte IP Blöcke oder Coprozessoren angewandt. Die Methode ermöglicht es auch, dass Entwickler bereits mit einem echten System arbeiten können und daher die parallele Softwareentwicklung beschleunigt wird. Allerdings sind die Nachteile der Rapid-Prototyping-Methode, die der hohen Anschaffungskosten, die kurze Lebensdauer und in der Regel auch das unterschiedliche Zeitverhalten des Prototyps und des zukünftigen Produkts in einer bestimmten CMOS Technologie.

In der vorliegenden Arbeit stellen wir die Cycle-Accurate Coprocessor-Prototyping (CyCoP) Plattform vor. Die Plattform ist in der Lage das Verhalten von einem Coprozessor mit seinem Hauptprozessor

Zyklen-genau zu emulieren. Wir zeigen, dass CyCoP die Fähigkeit hat, mit nur zwei einfachen und flexiblen Modulen Zyklen-genau das Verhalten zwischen einer Software-Anwendung und einem Hardware-Coprozessor zu emulieren. Die Implementierung der Software-Anwendung auf die beiden Module verändert weder deren Verhalten und Grösse, noch deren ursprünglichen Design-Flow.

Mit der CyCoP-Plattform zeigen wir, wie die Timing-Informationen aus dem Prototyp entnommen werden kann, um eine Zyklen-genaue Emulation zu ermöglichen. Dafür stellen wir einen neuen Prozess-Synchronisations-Algorithmus vor, der für die Prototyping-Umgebung angepasst ist.

Durch das modulare Konzept des CyCoPs ist die Plattform hardware-unabhängig und sie ist daher leicht auf andere Hardware-Umgebungen adaptierbar. Dies hat den Vorteil, dass die Anwender unabhängig von spezifischen Anbietern sind und dass sie jederzeit innerhalb der Prototypenphase die Hardware austauchen können, wenn z. B. eine neuere Hardware verwendet werden soll.

Um die Machbarkeit des CyCoP-Konzepts zu demonstrieren, ist die Plattform auf einer realen Hardware-Umgebung umgesetzt worden. Zwei Fallstudien, eine für eng gekoppelte und eine für lose gekoppelte Coprozessor, demonstrieren die Durchführbarkeit unseres Konzepts und belegen die Qualität der Ergebnisse.

Contents

List of Figures

List of Tables

List of Acronyms

ADL	Architecture Description Language
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
BFM	Bus Functional Model
CLB	Configurable Logic Block
CoP	CoProcessor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CSP	Communicating Sequential Process
CyCoP	Cycle-accurate Coprocessor Prototyping
DAC	Data Access Compare
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DUT	Device Under Test
FFT	Fast Fourier Transformation
FIFO	First In First Out
FPIC	Field Programmable Interconnect Component
FPGA	Field Programmable Logic Array
FPR	Floating-Point Register
FPU	Floating-Point Unit
FSM	Finite-State Machine
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU is Not Unix
GPP	General-Purpose Processor
GPR	General-Purpose Register

GPT	General-Purpose Timer
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuit
ICE	In-Circuit Emulation
ILP	Instruction-Level Parallelism
IP	Intellectual Property
IPC	Instructions Per Cycle
IR	Intermediate Representation
ISA	Instruction-Set Architecture
ISS	Instruction-Set Simulator
KPN	Kahn Process Network
MAC	Multiply-Accumulate
MIPS	Million Instructions Per Second
MW	Middleware
NGC	Xilinx netlist file with constraint information
OS	Operating System
PC	Program Counter
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PLC	Product Life Cycle
PPC	PowerPC
RAM	Random-Access Memory
RBA	Roll-Back Algorithm
RISC	Reduced Instruction-Set Computer
RTL	Register Transfer Level
SDF	Synchronous Data Flow
SDRAM	Synchronous Dynamic Random-Access Memory
SIA	Semiconductor Industry Association
SLE	Source Level Emulator
SIMD	Single-Instruction, Multiple-Data
SoC	System-on-Chip
SPU	Synergetic Processor Unit
SRAM	Static Random-Access Memory
SW	Software
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

In this chapter we motivate the use and the design of prototyping platforms for coprocessors in general. Therefore, we introduce the coprocessor concept and analyze its issues in the Section 1.2. In Section 1.3 we look into the available verification concepts and introduce related work in prototyping platforms. We combine the findings in Section 1.4 and point out the challenges for the design of a cycle-accurate coprocessor prototyping platform. Finally, Section 1.5 gives an overview of the remaining chapters of this thesis.

1.1 Motivation

The performance requirements for embedded systems grow steadily. At the same time the market asks for systems that require less space and less power at higher clock frequencies, and the systems have to be more reliable. In return the Time-to-Market for a new product becomes shorter and shorter as the success of a product—and therefore also its total revenue—depends mainly on its launch. Thus, the available development time for an embedded system, as well as the Product Life Cycle (PLC) is shortened. The development of embedded systems, which is characterized by growing system complexity and the above mentioned requirements, can only be handled with structured procedures and (semi-)automated tools.

The application domain for embedded systems spans every area of our daily life in which electronic equipment supports our daily business. In the multimedia domain, embedded systems are integrated, e.g., into digital cameras, tablets and DVD players. In the telecommunication domain, they are everywhere from smartphones to the switchboards. Also the car industry utilizes embedded systems all over, e.g. in engine control, break control, or the navigation system.

The industry plans to integrate entire embedded systems, which are realized nowadays as board-level-systems, into a single silicon die (refer to Figure 1.1). Such silicon dies are also known as *System-on-Chip* (SoC). According to Chang et al. [1], SoCs are defined as a complex Integrated Circuit (IC) that integrates the major functional elements of a complete end-product into a single chip or chip set. SoCs have several advantages compared with board-level-systems. SoCs require less space as they mainly consist of only one single chip. Therefore they are also lighter and have lower power dissipation.

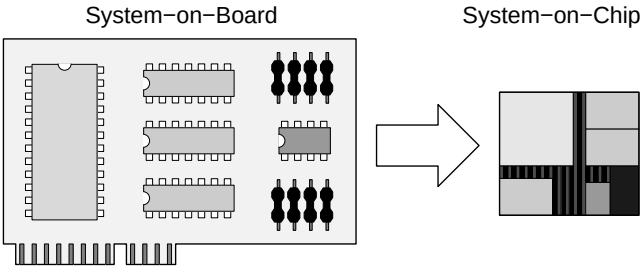

Figure 1.1: Evolution of the System-on-Chip (SoC) [1]

In 1999, Application Specific Integrated Circuits (ASIC) contained on average 150 kb of gates or of 400 kb of memory. The design consisted of about 10^4 lines of VHDL code. In 2005, it contained about 500 kb of gates and 1 Mb of memory. Such ASICs were designed with 10^5 lines of code. Economically, ICs that are five times bigger and more complex are possible [2]. This phenomenon is called the *productivity gap*, i.e. there is a steadily growing gap between what is technologically can be designed and what is actually realized in praxis owing to the short development time (Figure 1.2). Furthermore, both the PLC and the available development time become shorter. These

contradictions can only be resolved if the development efficiency can be increased significantly. This can be achieved mainly through abstraction, automation, and reuse.

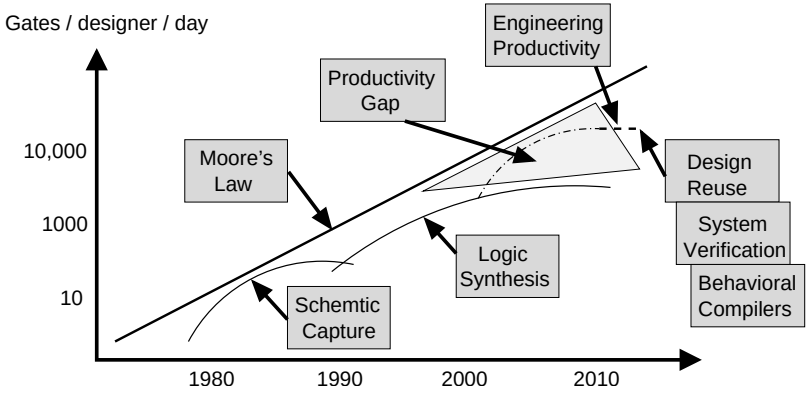

Figure 1.2: Productivity Gap (From [2])

For the year 2015 the roadmap of the Semiconductor Industry Association (SIA) [3] predicts a processor with 7 billion transistors and a memory (DRAM) capacity of 68.7 Gb [4]. The SIA therefore recommends a minimal gate length of 30 nm for the underlying technology. The challenge is to make full use of the offered performance and capacity, as well as to satisfy the need for fault-free systems.

One possible solution to this problem is *design reuse* of already developed and verified *Intellectual Properties* (IP). The IPs are claimed to be the enabler of the SoC revolution [1]. By employing such components, a complex system can be designed in shorter time, as the redesign of every component would increase the development time. However, while IP reuse and SoC platforms let a system developer to compile a design rapidly, they do not guarantee that the composition, including the software application, works correctly.

If the capacity of digital systems continues to grow at the rate of Moore's law, i.e. by a factor four every three years, also the time spent at the verification step will increase. The verification step has a share of about 70% [5] to 80% [6] in the project development time. This factor demands more formal verification methods, but in praxis the

simulation sill uses the bigger part of the verification. For embedded systems, the hardware/software co-simulation is decisive in the verification process. As the application areas for the embedded systems often are real-time domains, more and more software functions are replaced by special-purpose hardware, i.e. coprocessors. This happens because pure software implementations may not achieve the desired performance or use too much power. This trend results in a higher fault probability at the hardware/software interface. To guaranty the correct functionality of the system, the software has to be tested for conflicts with the hardware. The problem is how the functionality of the hardware/software interface can be verified sufficiently.

The formal verification method is the most accurate as it demonstrate the correctness of a system with an exact mathematical proof. However, automated verification of real-world systems [7] still is subject to research and is currently prevented by the complexity from covering the system constraint exhaustively. Simulation techniques are used for functional and timing validation of a system and rely on the execution of software models of the Design Under Test (DUT) on powerful computers. The simulation time mainly depends on the abstraction level of the models and ranges from one cycle per second for gate-level simulations up to 10^7 cycles per second for the algorithmic level. Thus, simulation speedups are achieved by using simplified models of the system. The simulation fails to verify correctness of a system sufficiently either owing to long runtime or due to simplification. Working up to 10^5 times faster than simulation, the FPGA-based emulation allows one to significantly extend the verification coverage [8]. Logic emulation closes the wide verification gap between simulation and actual silicon steps. Working close to the actual speed, the rapid prototyping provides extensive, “live” test coverage. However, prototyping platforms come together with expensive electronic equipment and inconvenient handling.

The main motivation for this work was to find a method to rapidly prototype any combination of processors with their corresponding coprocessors. The flexibility of the platform has to offer the possibility of using it in different stages of the verification step and the concepts have to be configurable and rapidly adoptable to any environment. Furthermore, the emulated system should not only run at almost the same speed as the final product, but also have exactly the same real-

time behavior.

1.2 Coprocessor Classification

Coprocessors (CoP) can be as varied as the applications they serve, and consequently, there are many ways to classify them. The first and possible most logical is along functional lines; i.e., math CoPs, graphics CoPs, string or text CoPs, and so on. Within a function it is possible to separate them into general-purpose versus special-purpose implementations. At a lower level, hardware protocol may categorize an implementation as tightly-coupled or loosely-coupled.

Hansen introduces in his Ph.D. thesis [9] in very detail the concepts and issues of CoPs, which are still valid for today. Nevertheless, we identify some characteristics of CoPs according to the hardware and software issues. We seek to categorize them and identify types or styles of CoPs architectures. Once classes are identified, we can define the challenges for a CoP prototyping platform.

1.2.1 Software Issues

We define the CoP as a device that replaces software routines. One of the primary considerations would be the integration of the CoP with the rest of the software. The presence of a CoP in the SW manifests in different forms. The CoP can be seen in the instruction's operation-code (opcode), or at the functional level, or not at all. These are all determined by the nature of the various computational forms and depend on whether the CoP assists at the instruction level, the function level (e.g. a subroutine), or the entire algorithm level (e.g. a program).

Software Interface

As an example of a CoP that assists at the instruction level, consider floating-point arithmetic. Floating-point instructions normally require two unique operands, generating a change to one operand or possibly producing a third unique result. As the instructions of the CPU are issued synchronous to the program flow, the interaction with

a CoP to assist the floating-point computation must be quick and efficient.

As an example of a CoP at the function or subroutine level, consider image processing. Often it is necessary to compute the convolution of an image with some filter. For each pixel, its value designated as $V(x, y)$, the Gaussian convolution of radius r is given in Equation 1.1

$$V'(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} C_{i,j} V(x + i, y + j) \tag{1.1}$$

This computes V', a new value for V. It is essentially a blurring step to eliminate visual noise of frequency less than distance r. Given the values for x, y, i, j, and r, the CoP could compute the pixel value V' without any other interaction or direction from the host.

As an example of a CoP that implements a complete program, consider a complete image processing system. In addition to the filtering function mentioned above, a complete image processing system must include functions for contrast enhancement, noise rejection, edge extraction, edge enhancement, and various transformations. The full system might have CoPs for each of these functions.

Compiler Issues

For CoPs that are seen at the instructions level, compiler issues become significant. First, the compiler needs to understand and recognize the execution model of the CoP. If the CoP consumes several cycles for its basic operations, a compiler optimization to intersperse instances of CoP commands with other non-data-dependent and non-related code allows the CPU to execute concurrently. Without optimization, significant performance improvements may be lost. As a means of accommodating a first-order level of concurrency—the so-called Instruction-Level Parallelism (ILP)—the hardware protocol may allow the CPU to proceed with other instructions after the issuance and before the completion of CoP instructions. To the extent that sequential instructions are independent or do not require the CoP resources, parallel execution can occur.

The best compiler solution would allow the inclusion of a CoP in a software system and yet not require the compiler be modified at all to

accommodate it. That is unlikely if performance is a consideration. A widespread approach is to model the target processor architecture in a dedicated Architecture Description Language (ADL) and to generate the compilers automatically from the ADL specifications [10]. For C compiler generation, however, most existing tools are limited either by the manual retargeting effort or by redundancies in the ADL models that lead to potential inconsistencies. *Configurable Processor Cores* like the ARM [11], MIPS [12], Xtensa [13] and PowerPC [14] families can be optimized by the user via addition of custom instructions. In this case, semi-custom compiler systems, such as a modified GCC [15] can be used, and retargeting is implemented by making new instructions available to the compiler in the form of intrinsic. It exist also the class of retargetable compliers, like the CoSy [16]. These are highly flexible, easy-targetable compilers, which are based upon their highly modular design and extensible intermediate representation (IR). The configurability and retargetability make them a particularly effective environment for exploration of compiler effects on possible architecture variations, thus enabling true hardware/software co-design.

CoP operations can be controlled either at compile time or runtime. Even for systems equipped with a hardware CoP, it is often a user-controlled feature of compilation to include explicit CoP instructions in the code generated. Without the instructions, library routines are linked and parameters are passed to the CoP through memory, the user-process stack. With explicit CoP instructions, transfers and operations are defined functions of the hardware and operands may be transferred from registers, memory, ore reused in the CoP. For some systems, if the CoP is absent or disabled, executing the instruction will result in a trap to routines that implement the function.

1.2.2 Hardware Issues

As with software, there are many hardware characteristics that differentiate CoPs. Broadly speaking, there are six classifications, depending on various control and data-related factors. These include:

- the instruction or command paradigm,
- the data transfer protocol,

- the data types support,
- the memory hierarchy interaction,
- the interconnection topology, and
- performance and speed.

There is a correlation between the physical placement of the processor within the SoC hierarchy and these hardware issues. To guide our discussion in the sections that follow, Figure 1.3 illustrates three levels of CoP hierarchy within an SoC.

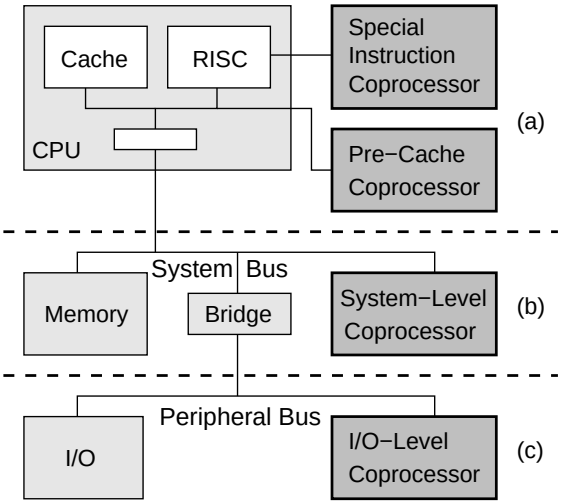

Figure 1.3: Topology of CoPs Interconnection

On the basis of the interconnection topology, CoPs are grouped in three levels:

(a) CPU-level CoPs, including special instruction CoP and pre-cache CoPs,

(b) Memory-level CoPs, and

(c) Input/Output-level CoPs.

At each of the three levels, there are two main things to consider:

- instruction sequencing and/or control, and
- data/operand manipulation.

In the following we discuss inter-level differences and similarities and intra-level properties for control and data for each of the three levels and relate them to the six classifications above.

Instruction and Control Issues

Some CoPs have built-in or hardwired instruction sequences and allow no generalization as provided by an instruction stream. Such CoPs can exist at any level.

For CoPs that do operate off the instruction stream, there are several ways of providing control. The instruction could be:

- fetched by the CPU and seen simultaneously by the CoP, or
- fetched by the CPU and sent to the CoP, either in encoded or decoded form, or
- fetched by the CoP itself.

We call the first of these an *instruction tracker* CoP. As the name implies the CoP tracks or follows the instruction stream as it comes out of storage and decodes and executes those instructions intended for it. The CPU execution unit essentially treats CoP instructions as "no-ops." These are either CPU-level CoPs or memory-level CoPs and interact directly with CPU instruction fetch unit, CPU instruction caches, and main memory. I/O-level CoPs do not use instruction tracking protocols.

We refer to the second instruction-issue method as a *master/slave* protocol. The CPU is the master and the CoP is the slave, and receives direction and begins operation only on command by the CPU. These are usually memory-level CoPs, but in some cases are found at the CPU-level and I/O-level. This method is less tightly-coupled than instruction-trackers and is used for general-purpose CoPs that may operate asynchronously with the CPU and do not interact as closely

with the CPU pipeline, for example. Many I/O-level CoP receive some portion of their control in a master/slave manner.

The third instruction issue paradigm we refer to as *autonomous*, since the CoP has the ability to control its own instruction stream or continue execution under its own control. These types of CoPs are nearly always found at the I/O-level of the interconnection hierarchy. In a multiprocessor system, a second CPU is used to implement certain aspects of an algorithm may be considered an autonomous CoP, e.g. the Cell Broadband Engine [17] from IBM consists of one main PowerPC processor and several so-called Synergistic Processor Units (SPU) which are free programmable SIMD processors. These SPUs can be considered as CoPs. However, that has more to do with software execution than hardware issues and will not be considered further.

In many cases, CoP control units are combinations of all three methods. One part of the CPU-to-CoP protocol may be master/slave followed by autonomous action by the CoP as it continues to fetch and execute its own instructions until finished. We next consider aspects of data transfer and manipulation in CoPs.

Data Transfer and Memory Interaction

Besides the flow of instructions to the CoP control unit, operands must also be provided. There are at least five ways that CoPs send or receive data:

- the CoP has "built-in" hardwired constant values,
- the CPU executes load/store instructions, but the data are sent or received by the CoP,
- the CPU transfers data to its internal registers, then writes to the CoP,
- the CoP executes its own load/store and transfer operation,
- the CoP has registers that shadow the CPU registers.

At the CPU-level, special function units extend the CPU data path and operate on the same data types. Consequently, they are synchronous with CPU operations. They may have their own registers or

use CPU registers. Caches provide fast access to operands involved in such specialized functions as fixed-point binary arithmetic, array index manipulations, emulation functions, encryption tasks, and so forth.

Memory-level CoPs often manipulate data types that are different from those used in the CPU or special function units. Consequently, they may maintain separate register files. They receive/send data by monitoring the data bus or respond to move operations between it and the CPU registers. The data involved are typically individual words instead of large blocks. As the data types may be different than those used by the CPU, the bandwidth between memory-level CoP and storage may be quite different from the CPU. For example, double-precision floating-point operands may pass directly between floating-point CoP registers and the data cache in a single cycle, while CPU access may require multiple-cycle to transfer the same data.

I/O-level CoPs typically transfer large blocks of data between devices and main memory with little or no interaction with the CPU. The interaction with main memory can either be cycle-stealing or uninterruptible burst transfers. The model for data manipulation matches that of control—very little interaction with the system once initiated.

Having considered many of the issues that distinguish and separate different types of CoPs at the interface level, and ways in which CoPs interact with the rest of the system to send, receive, and manipulate data, we next look in the Section 1.3 how and whether state-of-the-art prototyping platforms fulfill the paradigm that have been evaluated in this section.

1.3 Hardware/Software Co-Verification

Up to 80% of the project development time is spent at the verification step. Moving towards the System-on-Chip (SoC) makes the verification gap even wider than the design gap. In literature we find three different verification methods [1, 18, 19], which are:

1. Formal Verification,
2. Simulation, and

3. Rapid Prototyping.

Each verification method plays its own role in the design process. In this section we briefly explain the individual methods, highlight its strengths and its drawbacks. For the introduction of the prototyping methods we discuss some of the state-of-the-art systems available.

1.3.1 Formal Verification

In Formal Verification the functional correctness of a system is demonstrated with an exact mathematical proof. There are two approaches to proof the correctness. Either we proof the equivalence of a model with a reference model, *Theorem Proving*, or we assure model properties, *Model Checking*.

In theorem proving a process is considered to be theorem proving, if it consists of a traditional proof, starting with axioms and producing new inference steps using rules of inference [20, 21, 22]. This proof is well suited for control flow-oriented applications and is mainly used for regression tests.

The model checking is equivalent to brute-force enumeration of many possible states, although the actual implementation of model checkers requires much cleverness, and does not simply reduce to brute force. It is good for liveliness and security tests of a system. However, automated verification [23] is still subject to research and is currently limited by long computational times for complex circuits.

For the hardware/software co-verification, however, is the formal verification inapplicable as the complexity of such systems, and therefore the possible states, outreaches today's computing power [1].

1.3.2 Simulation

Simulation techniques are used for functional and timing validation of a system and rely on the execution of software models of the Design Under Test (DUT) on powerful execution platforms, e.g. workstations. The simulation time mainly depends on the abstraction level of the models and range from one cycle per second for gate-level simulations up to 10^7 cycles per second for the algorithmic level [24]. Nowadays, *static* timing simulations have to be complemented with *dynamic* simulations in order to cope with deep-submicron effects such as crosstalk

between metal layers because of parasitic capacitances. Moreover, most systems are heterogeneous, i.e., they consists of analog/digital, hardware/software and/or electrical/mechanical components which is tackled by simulator coupling [8], such simulators are often called co-simulators. As the targeted domains are diverse, simulation is aggravated by different abstraction levels of models, stimuli, and time, as well as synchronization problems.

Although domain specific simulators exist, realistic system simulations still fail because of their long runtime. Simulation speedups are achieved by using *cycle-based* instead of *event-based* simulators and dynamic adaptable simulation models [25].

1.3.3 Rapid Prototyping

The term "rapid", in rapid prototyping has two meanings:

- the prototype is obtained rapidly and
- the prototype works rapidly.

For this, a rapid prototyping technique almost automatically converts an input specification into a hardware/software system, which is functional equivalent to the DUT. Working up to 10^5 times faster than simulation [24], the FPGA-based emulation allows to significantly extending the verification coverage. Logic emulation fills a wide verification gap between simulation and actual silicon steps.

In *functional* prototyping approaches the architecture of the final product is of minor interest, and only the functional correctness of a specification has to be checked. In contrast *architectural compliant* prototyping techniques focus on the real target architecture. This emulation technique is well known on *chip-level* where gate-level net lists of digital designs are mapped to acceleration platforms for validation purpose. Almost all modern processor designs rely on chip-level emulation [26]. In *system-level* prototyping the entire system, including any ASICs, is retargeted to a programmable hardware environment, consisting of FPGAs and of-the-shelf components such as microprocessors, memories, ASICs, or other IP-modules, so that the real system architecture can be emulated [27, 28].

Thus, rapid prototyping allows designers to explore the design concepts, verify the hardware design and complete the development, integration, and testing of the system firmware and software before first silicon. This enables fast verification of complex system designs in the real system environment which in turn results in higher-quality products and shorter time to market.

1.3.4 State-of-the-Art in Rapid Prototyping

According to their role and their characteristics, the prototyping platforms can be classified in two major categories:

- Commercial Systems
- Custom Platforms

The first category covers the logic emulation field. The commercial systems' common characteristics are their high capacity. They consists of hundreds of FPGAs or processors placed on boards which are combined into racks. The second category is the more heterogeneous platforms which combine different logic together into one system. The category also covers the single board development platforms.

Following we will give a few representative examples for each of the two categories. The list is incomplete; however, the presented prototyping platforms represent the main hardware concepts used all over. The following literature references provides further general information about rapid prototyping [18, 19, 29, 30].

Commercial Systems

IBM's Simulation Acceleration Hardware was originally built as a custom platform on special purpose hardware accelerators. The early version was called the Yorktown Simulation Engine (YSE) [31, 32]. It consists of a collection of special purpose processor, the so-called *logic processors*, which were interconnected through a large switch. The block diagram in Figure 1.4 illustrates this simplified architecture with a situation where up to 256 logic processors are connected.

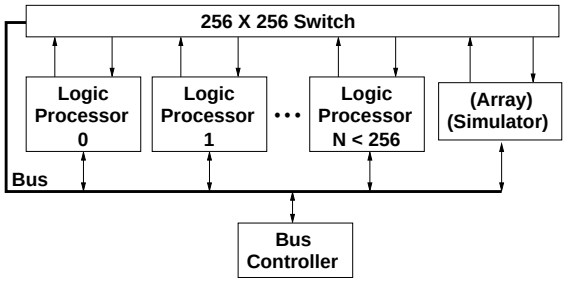

Figure 1.4: Yorktown Simulation Engine Overview

Each logic processor contains an *instruction memory* that stores the interconnection and function type information for a logical network, a *data memory* that holds logic values for signals in the network, and a *function unit* that evaluates logical functions, like shown in Figure 1.5. During simulation, the logic processors simultaneously step through their instruction memories and use the information fetched to access and update values for signals stored in their data memories. In addition, the logic processors transfer signal values between their data memories and the switch to accommodate the communication introduced by the partitioning. From a distant point of view the function unit can be seen as a Configurable Logic Block (CLB) from an FPGA which is evaluated with different configuration and data stimuli in sequence. This way a small mesh of CLBs can be simulated within a logic processor. After developing YSE, a robust production system was developed, the Engineering Verification Engine (EVE) [33]. EVE used a massive network of logic processors. Typically, each run through the sequence of all instructions in all logic processors in parallel constituted one machine cycle, this implementing the cycle-based simulation paradigm. The theoretical speed of EVE was many orders of magnitude faster than any software implementation 2.2 billion gate evaluations per seconds.

In the late 1990s, AWAN, was built as a low-cost system which improved on both the capacity and performance of EVE. AWAN is much like the EVE machine, but it is made with smaller, faster components and has a much improved interconnection strategy. Models

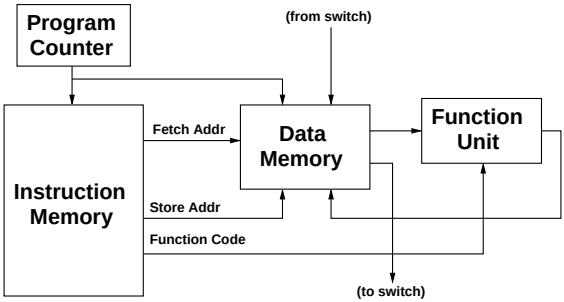

Figure 1.5: Logic Processor Overview

exceeding 31 million gates have been simulated. Speed depends on the configuration, model size, model complexity, and the amount of host interaction. AWAN is marketed by Quickturn under the name Radium. Utilizing the base EVE concepts, a hyper-acceleration and emulation machine called ET3 [34] was developed. ET3 uses logic processors which evaluate three-way input gates. In contrast to AWAN, ET3 has a larger number of processors and a lower depth of sequential three-way-gate instructions/processor. The breakthrough of this technology in the microprocessor and server system space occurred after the latest capacity improvements [34]. ET3 is marketed by Quickturn under the name CoBalt.

These are the only rapid prototyping platforms based on special purpose ASICs. Successor platforms and competitor platforms are hybrid systems combining general purpose processors combined with FPGAs. Therefore, we explained it in greater detail.

Cadence's Palladium Systems are processor-based supporting designs of up to 256 million gates [35]. Hard IP with standard IP blocks are supported through the Palladium IP chassis. Each standard IP block is mounted on a customized board can have up to 1,248 bidirectional signal pins each. Cadence supplies a number of turnkey IP blocks, such as ARM processor models, Xilinx Virtex-II FPGAs, and pin-grid array for user mounting of any silicon.

Mentor Graphics offers three emulation systems: VStationPRO, VStationTBX and iSolve [36]. The emulation is based on a VirtualWires technology [37, 38], which overcomes pin limitations by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA. Wire multiplexing requires scheduling of virtual wires to physical wires, which results in creating a FSM, and also synthesizing multiplexer/demultiplexer logic and registers. Hardware overhead is largely compensated by increasing logic utilization ratio. The emulation clock is broken into a number of micro-cycles determined by a free-running μCLK. The micro-cycles are grouped into sequential phases to support combinational paths that extend across multiple chips. This is performed by the timing resynthesis step, which replaces the user-clock by a single global synchronous clock. The emulation speed is determined as the product of the virtual clock cycle time by the number of virtual cycles in one cycle of the user's clock. The cycle time of the virtual clock is chosen to be the inter-FPGA traversal time. The number of virtual cycles is equal to the number of FPGA crossings in the longest combinatorial path. By solving the pin limit problem VirtualWires approach increases the FPGA utilization ratio from 10–30% in hard wired systems to over 45%. The architecture used in SimExpress and Celaro is based on a full custom chip specifically designed for emulation. The full custom chip comprises an entire emulator, including programmable elements; interconnect matrix, logic analyzer, clock generators, memory interface, and I/O channels. An interconnect matrix guarantees that the propagation delays between any two programmable elements within the chip are fixed and predictable. All programmable element outputs are connected to the on-chip signal probing circuitry, without relying on the compiler to perform the routing, and, consequently, consuming precious interconnection resources otherwise available to the user logic. A bank of I/O channels connects the emulator-on-a-chip to all other emulators-on-a-chip in the system, via a fractal routing network.

Custom Platforms

In the category of the custom platforms we group the systems and hardware of university projects as well as commercial boards which

consists only of a few programmable logic devices. These boards are centered on the recently announced logic devices. There are too many different boards and system build by university and vendors to list them up. We pick out four examples from one vendor and from three universities, and briefly describe them. These systems are developed for the research purposes in hardware-software co-design area, as well as for pure hardware prototyping.

HARDI offers three products: HAPS-10, HAPS-20, and HAPS-34 [39]. The HAPS system is based on two types of boards. The so-called mother board consists of several sockets where daughter boards can be plugged in. These sockets are connected among each other with a configurable network. The flexibility allows the same board to be reused in several projects or configurations by replacing daughter boards containing I/O and custom subsystems. Each socket has 720 pins. Six pins are reserved for power, while the remaining can be used freely. As an option, 708 of these pins can be used for LVDS signaling, including 48 LVDS clocks. The sockets can be used either for attaching daughter boards or for creating wider inter-FPGA buses or for a combination of both.

REPLICA rapid prototyping system [40] is based on a scalable and reconfigurable system architecture. Up to six processing modules execute the tasks of the target system and communicate with each other via communication links. Connectivity is established by SRAM-based bit-oriented switches, which are used to switch one or more signals at compile time. Based on non-blocking switch matrix these devices allow a total flexibility in connecting incoming signals. Some communication links may require additional hardware resources such as memory, glue logic, etc. These resources will be allocated on the interface modules. The set of current processing modules comprise floating point signal-processors, 16- and 32-bit RISC-microcontrollers, and an FPGA-based ASIC emulator.

WEAVER prototyping environment [41, 42] features up to four Xilinx FPGAs on one so-called base module. Base modules can be combined so that bigger designs fit onto the emulation platform. I/O

and processor modules exist as well that can be connected to the WEAVER.

BEE2 [43] is a general purpose processing module based on five Xilinx FPGAs. In addition to the large amount of processing fabric provided by the FPGAs, the BEE2 also provides up to 20 GB of DDR2 DRAM memory. Each of the five FPGAs has four independent channels to DDR2 DIMMs which provides very high memory bandwidth. Finally, the FPGAs on the BEE2 are highly connected with both high-speed, serial and parallel links. The FPGAs are laid out in a star topology with four user FPGAs in a ring and one control FPGA connected to each user. The user FPGAs each have four independent high speed serial channels which are capable of transferring data at 10 Gbps through the connectors. The user FPGA ring consists of parallel connections of 138 high-speed LVCMOS traces between the FPGAs which can run at a maximum of 400 Mbps. The control FPGA has two high-speed serial channels, 64 LVCMOS traces to each user FPGA, and connections to common peripherals such as 10/100 Ethernet, USB, RS232 serial, DVI, and GPIOs.

1.4 Problem Statement

1.4.1 Classification Summary

In Section 1.2 we have reviewed some of the hardware and software issues related to the incorporation of CoPs in SoCs. The main consideration in specifying a prototyping environment for all the issues noted is based on the interaction between the CPU and the CoP.

On the software side, a possible prototyping platform must be able to catch all CoP calls in either instruction form or load/store access. The software tool-chain as well as the software application setup must not be changed because they mainly influence the performance of the application. Thus, the program compilation as well as the application execution must not be different for the target architecture and for the prototype.

A CoP may be coupled in any manner to a processor in hardware. The way the CoP is coupled to the processor limits the resources

the processor and the CoP can share. Whereas a tightly-coupled CoP can access all internal resources of a processor, a loosely-coupled one can only share the external memory. In return a loosely-coupled CoP can run completely asynchronously to the processor, whereas a tightly-coupled CoP runs synchronously to its CPU. A prototyping platform has to interface to any type of CoP and provide it with data. Furthermore, the timing behavior of the CoP must be appropriate to its type.

1.4.2 Co-Verification Summary

Rapid prototyping combines the benefits of simulation with those of hardware prototypes. Moreover, it allows designers to explore the design concepts, verify the hardware design and complete the development, integration and testing of the system software before first silicon. This enables fast verification of complex system designs in real system environment, which in turn results in higher-quality products and shorter time to market.

In Section 1.3.4 we differentiate between commercial prototyping systems and custom platforms. The commercial systems consist of hundreds of FPGAs placed on boards which are combined into racks. The whole system is integrated into the same base hardware and therefore the overall clock can be simply scaled to the longest logic path. This allows a cycle-accurate emulation of software and hardware. However, processors have a highly optimized architecture. To implement them into programmable logic devices requires the clock to be scaled down to a very low speed. Thus, to speed-up the emulation, the processors are simplified and only have the functional behavior. This reduces the accuracy of the hardware/software co-verification as the software does not perform like on the original processor architecture. Furthermore, these prototyping racks are very expensive and the update to newer FPGA technology is dictated by the vendor. To shorten the compilation time of the prototype, one has to buy IPs from the vendor and has to rely on the vendor's selection.

The heterogeneous custom platforms have the advantage that dedicated devices can be combined with programmable logic devices. Thus, the software runs on the original architecture and therefore behaves cycle-accurately. However, the coprocessors that are imple-

mented into programmable logic devices cannot be clocked with the same rate as the real processor. Thus, the clock distribution over the whole platform is not uniform and therefore introduces inaccuracies to the hardware/software co-verification.

In Table 1.1 the main pros and cons of rapid prototyping techniques are summarized.

Table 1.1: Pros and Cons of Rapid Prototyping

Advantages	Disadvantages
• high speed • real system environment • short time to market • full design observability • no sim. model required	• high costs • compilation times • short lifetime • no SW timing

1.4.3 Conclusions

Modern chip integration allows designers to steadily increase the complexity of their architecture. A concern is whether these complex and highly integrated SoCs perform their intended functional behavior as specified in the system specifications. A complete design verification of these SoCs has to cover not only the hardware part of the design, but also the software considering cycle-accurate timing.

Extensive functional simulations of the system enable the detection of errors at the higher abstraction level in acceptable computation times. Starting the verification at high abstraction levels is necessary to increase the designer's confidence in the design and to achieve a functionally correct system. To complete the verification FPGA-based rapid prototyping is used. Mapping the design of the target SoC into an FPGA yields an accurate and fast representation, but following the problems have to be solved to enable a cycle-accurate hardware/software co-verification:

- To reduce the costs and to give the user the freedom of choice of the hardware used, the prototyping platform has to be independent of the underlying hardware.

- The software behaves only accurate for the original processor with the original software design flow. Therefore, the design flow cannot be replaced or changed.
- Also the software itself cannot be changed.
- If the original processor has to be used for the software, how are instruction-set extensions for the processor emulated and prototyped?
- When using any kind of real processor with any programmable logic device, one has to ensure that the data and signals exchanged are synchronized across the clock boundaries.

This thesis presents a design methodology that addresses these problems, and illustrates the development process with different examples.

1.5 Organization

This thesis is organized as follows.

In Chapter 2 we introduce the proposed prototyping platform called CyCoP (Cycle-accurate Coprocessor Prototyping). Our method allows a cycle-accurate coprocessor emulation which solves an important problem in the hardware/software co-verification domain. Moreover, the platform is hardware independent and works with standard tool-chains.

In Chapter 3 we present the implementation details of the platform for tightly-coupled coprocessors. This type of coprocessor extends the instruction set of a processor. We introduce a minimal invasive method to cycle-accurately emulate the instruction-set extension.

In Chapter 4 we describe the implementation details of the platform for loosely-coupled coprocessors. A loosely-coupled coprocessor runs asynchronous to the processor. To be able to cycle-accurately emulate the hardware and software events with the prototype platform, we introduce a process synchronization method. The synchronization method works with the context of the coprocessors and therefore, we present a technique to access the context of the coprocessor.

In Chapter 5 we give performance results of our prototyping platform. Some results are based on real benchmark applications and are compared to real hardware implementation, where it is applicable. Other results are artificial for being able to demonstrate the correctness of the used concept. In Chapter 5 we also give numbers on how accurate the platform is working on the performed prototype. With the Chapter 6 we conclude this thesis.

In the appendix some further implementation details are described which are not required for the understanding of this thesis, but might be of interest for the reader as well.

Chapter 2

CyCoP: Cycle-Accurate Coprocessor Prototyping

In Section 1.2, we classified coprocessors (CoP) and concluded that from all the perspectives there are two relevant categories for a prototyping platform. We mainly have to differentiate between loosely- and tightly-coupled CoPs. The first category is accessed by a CPU over an on-chip interconnect system. The second category extends the data path of the CPU itself and therefore is accessed by the CPU directly over a special-purpose interface.

In Section 1.3, we examined existing prototyping platforms. The main disadvantages of the available systems are that these systems cannot keep up with the progress of programmable logic devices, as these devices are steadily gaining in capacity and speed. Furthermore, there is no accurate timing behavior between the CPU and its CoP, which is relevant for both the hardware/software co-verification as well as the proof of the performance gained.

Because of the knowledge gained, we propose CyCoP, a Cycle-Accurate Coprocessor Prototyping platform, and describe it in this chapter. In Section 2.1 the platform as a whole is introduced to clarify the concept of the platform. Sections 2.2 and 2.3 consider

the hardware and software components of the platform. Section 2.4 presents the method to configure and set up the platform. Section 2.5 introduces the hardware used and software environment for the example implementation. The last section summarizes the findings of this chapter.

2.1 Platform Overview

The CyCoP platform has two main components: the *processor system* and the *coprocessor system*. The processor system consists, in fact, mainly of a standard processor chip with a host board-level interconnect, e.g. a PCI bus. The CoP system consists of off-the-shelf prototyping boards populated with programmable logic devices, e.g. FPGAs. The prototyping components communicate with each other over the board-level interconnect. Figure 2.1 shows an abstract block diagram of how the CoP systems are embedded into the board-level interconnect of the host processor system. In the following, the processor system is simply called the CPU.

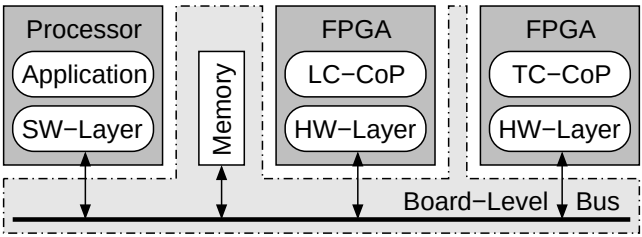

Figure 2.1: Coprocessor Prototyping Platform Topology

Thus, the CyCoP platform is simply assembled depending on the needs of the user. It can be that as many FPGA boards are added to the prototyping system as the board-level interconnect allows. On the other hand, the number of CPUs is currently limited to one, as the platform is designed to emulated CoP behavior. However, under certain conditions more CPUs can be added to the prototyping system.

A special prototyping SW layer, subsequently called the Middleware (MW), runs on the CPU. The MW enables the communication of the software application with its CoPs in the prototyping environment. Therefore, the MW detects and identifies the FPGA boards constituting the CoP system. At run-time the software application accesses the individual CoPs via the MW. For the MW it does not matter whether the application is a simple user application or an entire Operating System (OS). Every CoP call from the application is fetched by the MW and sent to the CoP addressed and the CoP results are provided by the MW to the application. The MW resembles the Hypervisor concept for server virtualization, as it hides the real hardware environment to the application.

Furthermore, the MW hides the delays of the emulation environment from the software application. Thereby, the MW maintains the semblance of "real time" operation for the application. Also, the computation time of the CoP request is taken into account by the MW. Thus, the MW enables a cycle-accurate performance measurement of the software application using the CoP.

In the CoP systems the application request are caught by a HW layer. This HW layer is also integrated into the programmable logic device like the CoP core itself. The HW layer works as a Wrapper for the CoP core. It controls the CoP in- and output, and communicates with MW. Owing to the communication between the Wrapper and the MW, the prototyped SoC is synchronized and guaranties a cycle-accurate emulation.

The communication between a loosely-coupled CoP (LC-CoP) and the application or between a tightly-coupled CoP (TC-CoP) and the application is basically the same. Merely the synchronization behavior is different, as will be further explained in the following sections.

2.2 Middleware

The basic idea of the CyCoP platform is to use the target processor core together with an FPGA, in which the CoP core resides (refer to Figure 2.1). The software application requests the CoP as if the CoP already resided on the same die as the processor core. The MW enables the use of the CoP in the CyCoP platform for the application.

The concept of implementing a specific SW layer to enable the communication between different components in a prototyping framework is known [44, 45]. These SW layers act as a sort of *communication stack* and enable the message passing from one component to the other within the prototype. The SW application explicitly makes use of these communication stacks. Our MW in contrast performs the task of communication enabling without the SW application being aware of this process. Furthermore, as the MW is executed by the CPU in which we measure the performance of the application, we have to take care not to influencing the measurements as well as the application flow.

The block diagram in Figure 2.2 shows in more detail how the software stack is organized in the platform. The MW is between the application and the hardware driver for the board-level interconnects. On the same level as the MW is the OS, because the OS and the MW are called directly by the application and they provide hardware services for the application. On the other hand, the OS may also use the CoP for certain routines or functions. Therefore, in our platform the MW is also called by the OS. This is indicated in the figure by the tilted boxes, with the OS box slightly overlapping the MW box.

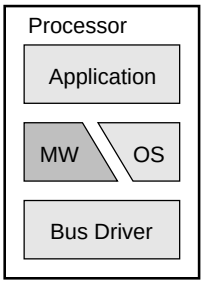

Figure 2.2: Software Layers

To pretend that the application runs in real time, the MW manipulates the *time base* of the CPU. The time base is a register whose content increments once in each period of the CPU clock. This register provides a time reference. To measure the performance of the application emulated using the CoP, the time base is read once at the

start of the application and once at the end. The difference of the two time values gives the computation time the application. The aim of the CyCoP platform is that the measurement shows the performance of the CoP as if it were implemented in the final target technology and design method. The MW manipulates the time base during the operation to achieve the correct result. SW debugger tools often make use of the same mechanism to maintain the semblance of "real time" operation while a system is being debugged [14, 46].

Furthermore, the MW has to execute the CoP requests of the application because the real CoP is not available during the emulation. Given that the real CoP is not available during the emulation, the CPU will encounter an *exception* as soon as the application accesses the CoP. If, for example, the MW is called in place of the *exception handler*, then the emulation of the CoP can happen.

The principle of the emulation is then the following:

1. Each CoP request calls the MW.
2. The MW
   - determines the CoP request,
   - prepares operands,
   - and sends the operands with the operation-code (opcode) to the CoP.
3. The CoP returns the results and execution time to the MW.
4. The MW continues to
   - prepare the results for the application
   - and to manipulate the *time base* of the CPU to hide the emulation time and to calculate the CoP processing time.
5. The application continues.

An application can access a CoP in two different ways. In the first way, the application uses load/store instructions to access the memory or the memory-mapped registers of the CoP over the on-chip interconnect system. In this case the CoP is mostly of a loosely-coupled

type. On the other hand, the application can make use of coprocessor-specific instructions, thus an extension to the original ISA. The task of the MW is to fetch both kinds of CoP request from the application, whether it is a load/store instruction or a coprocessor-specific instruction. This concept is different from the existing methods [44, 45] in which the SW layer is explicitly called by the SW application during the emulation. In the latter case, there are two ways to enable the CoP instruction to call the MW for emulation, all of which belong to the category of the *precise exceptions*. A pipelined processor is said to have precise exceptions, if the pipeline can be stopped so that the instructions immediately preceding the interrupting instruction are completed and those after it can be restarted from scratch [47].

The first method is the illegal exception as explained above. Another method is to replace the original CoP instruction with an instruction that forces a software interrupt, e.g. a *system call* or a *trap*, i.e. a breakpoint. These instructions mainly provide some immediate operand, i.e. a constant within the instruction itself, in which the recoded CoP instruction takes place. The corresponding exception handler then has to be replaced by the MW.

Thus, either the CoP instruction has to be replaced with a specific MW call for the emulation or the MW will be called instead of an *illegal instruction exception* handler. The choice which method is the best suited depends on several aspects. If there is an OS running on the CPU, the OS uses the system call for its own purposes. Therefore, to avoid a modification of the OS for the purpose of emulation, we avoid this method. Furthermore, if the recoded CoP instruction will not fit into the immediate field of the system call or trap, we might decide to use the illegal instruction exception to do the job.

If the CoP has to be accessed with load/store instructions we basically have the same choice as for the CoP instructions. We either keep the original instructions or replace them during compilation with an instruction that forces a software interrupt. If the original instructions are not replaced the CPU has to provide a specific hardware facility to monitor the addresses on the on-chip interconnect. When the CPU accesses the CoP address space, this monitor raises an exception. The PowerPC family [14], e.g., provides the *data address compare* (DAC) debug event with its debugging facilities. In this case the debug event handler would be replaced by the MW.

If no such hardware monitor is provided by the CPU, we propose to replace CoP load/store instructions for our CyCoP platform by a specific MW call at compilation time. One can still decide whether this instruction should be a trap or an illegal instruction, if the CPU provides no debugging functionality at all.

However, these are the only approaches for the new concept of a MW call. If we replace the CoP instruction, either load/store or specific, with an ordinary software function call to jump into the MW, the compiler would generate much more code within the application to save the current register context and to load the argument registers. This would influence the application execution time and flow significantly, and is therefore not a suitable solution for the prototype platform. Even if we used only a branch instruction to jump into the MW, we would not get the correct emulation results because a branch instruction is not *precise*. This fact is explained in greater detail in the following chapters.

Replacing the CoP instructions with function calls will change the size of the application. This has the effect that the original memory organization of the original software has to be adopted. This contradicts the aim of keeping the original design flow. Furthermore, it influences the memory and cache behavior tremendously. If the behavior of the memory and the cache changes for the emulation, the prototype can no longer guarantee cycle-accurate results.

Thus, if the MW call or the execution of the MW pollutes the cache of the CPU, the cycle-accuracy can no longer be guaranteed. Not only the call but also the execution of the MW must happen without affecting the cache of the CPU. To avoid the contamination of the cache, the CPU core has to provide the appropriate memory-managing hardware. If it is possible to exclude a specific memory region from being cached then all load, store, and instruction-fetch operations perform their access in memory, rather than in the respective cache. Though this slows down MW execution, application execution is no longer affected.

If the CPU cannot prevent any memory region from being cached, the MW has to restore the original state of the cache at the end of its execution. Therefore, it has to flush the contaminated cache lines and to touch the original memory region so that the corresponding original cache lines are loaded. This method holds some uncertainties

and should be avoided whenever possible. In the case of an associative cache, the content of the individual cache lines can be constructed, but the state of the replacement scheduler cannot be reconstructed. In an associative cache, the cache has a choice of where to place the cache line requested, and hence of which cache line to replace. Therefore, it uses a replacement scheme like the *least recently used* (LRU). When the MW manipulates the content of the cache, it also changes the state of the replacement scheme, which cannot be reset to the old state.

2.3 Wrapper

The Wrapper is programmed together with the CoP into the programmable logic device, like an FPGA. The Wrapper is the counterpart of the MW for the hardware CoP core. Like the MW, it has to handle the requests from the application coming over the board-level interconnect and has to control the timing behavior of the CoP to guarantee a cycle-accurate emulation.

However, the most important task of the Wrapper is that it stimulates the input signal of the CoP and probes the output signal of the CoP. Wrappers are well known in the world of HDL designers. They have the task of simplifying the I/O signals of a core by merging signals, by converting signals, or even by predefining constant signal levels or parameters for the core. Moreover, the concept of stimulating the I/O signal of a core is used by Wrappers known as testbenches in an HDL simulator [48]. The difference to our proposed Wrapper is the special control mechanism which we implement in our Wrapper for the purpose of synchronization. We will discuss this in greater detail in Chapter 4. Figure 2.3 shows the different modules of the Wrapper architecture and how they are connected with each other.

The module *BusAccessor* is the interface to the board-level interconnect. It manages communication between the Wrapper and the CPU. In the real die implementation, this module can be replaced by another module handling the access to the on-chip interconnect system. Thus, the CoP core logic is independent of its environment. The *BusAccessor* also connects to the *WrapperController* module. This second interface is used to exchange control commands with the MW.

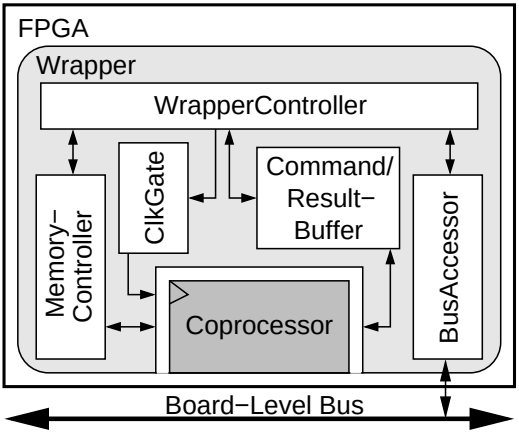

Figure 2.3: Wrapper Architecture

As the name already implies, the *WrapperController* controls the entire Wrapper functionalities and steers the according CoP. The *WrapperController* receives the CoP operations and operands from the MW. These CoP operations are temporarily stored into a specific buffer, the *Command/ResultBuffer*, where they wait to be executed by the CoP core. The MW tells the *WrapperController* how many new operands have to be computed by the CoP and triggers the start of the computation with a specific command. To hold the CoP in a certain state, the Wrapper uses a clock gate to control the CoP clock. The controller releases the clock gate and the CoP starts to compute the new operations. At the same time the controller also starts its own chronometer to measure the computation duration over all operations.

When the CoP has finished all operations, it stores the results back into the *Command/ResultBuffer*. The controller notes the number of results in the buffer and as soon as it reaches the expected number, it stops the CoP again by switching off the clock gate. The controller then notifies the MW that the results are ready to be fetched. Also the computation time is fetched by the MW so that this time can be taken into account for the latency in the application.

The Wrapper consists also of a *MemoryController*, which is an

interface to either internal or external memory. In the memory the controller stores the context of the CoP, which is needed by the synchronization process between application events and CoP events. We will explain this in greater detail in Chapter 4. In that chapter we propose a new process synchronization algorithm which makes use of the context manipulation.

2.4 Design Flow

In the previous sections, we introduced the two modules which enable the cycle-accurate emulation on the CyCoP platform. For each one of the modules we need a specific design flow to integrate the modules into the customer's designs.

2.4.1 Software Design Flow

Figure 2.4 shows the complete design flow for the software part of the CyCoP platform. If we ignore for once the part between the dashed lines, we recognize a standard tool chain for software compilation. For example, this flow represents the GNU Compiler Collection (GCC) [15].

The custom source codes may be available in two forms. The most common form is a high-level programming language like C/C++, but in embedded systems often also assembler is used to program software. The first step therefore, is to bring all the source code into the same form. There the assembler language predominates, as it is already close to the hardware and high-level compilers are able to restore their Intermediate Representation (IR) in the form of the assembler language.

As soon as all source codes are available as assembler code, the design flow starts to deviate from the original one. As we learned in Section 2.2 the original CoP calls have to be replaced with corresponding MW calls. A Perl script [49] parses all custom assembler codes to find CoP instructions, which it replaces with the MW calls, e.g. a *trap*. The parser script has to be configured accordingly by the user so that it is able to recognize the right CoP instructions and encode them in the specific MW call.

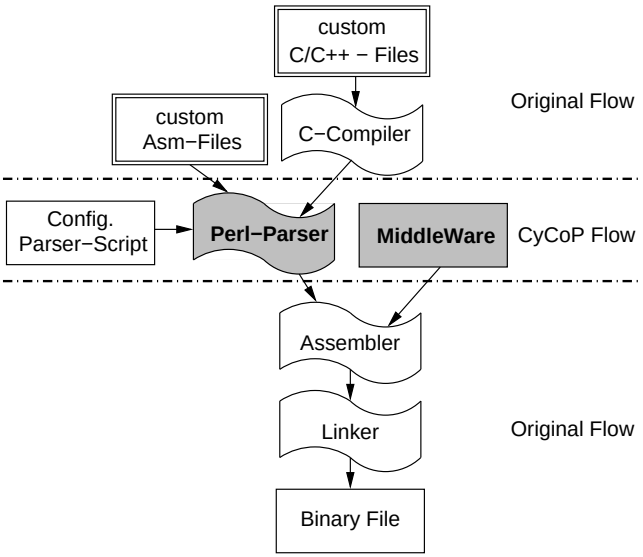

Figure 2.4: SW Compilation Flow

At the same stage of the software design flow, the MW is included into the design. Regarding the MW, the user has to make sure that it is placed at the right position in the memory. The MW call is an exception event which loads the according exception handler. This handler has to be replaced by the MW code. The MW is highly hardware-dependent and is therefore in almost every instance a new design for each CPU type. Only the decoding of the MW call itself can be automated and is taken over from the parser configuration file.

The following steps of the software design flow are retrieved from the original flow. All parsed assembler source codes, including the MW source code, are sent to the assembler, which exports them into the machine code. After the linker, the individual machine codes become a single binary of the complete software. This binary file is then finally executed on the CyCoP platform.

2.4.2 Hardware Design Flow

Figure 2.5 shows the complete design flow for the FPGA design of the CyCoP platform. As in the software flow, we recognize that only a small part of it is different from the standard flow. If we ignore for the time being the part between the dashed lines, we recognize a standard tool chain for FPGA synthesis. For example, this flow represents the Xilinx ISE design flow [50].

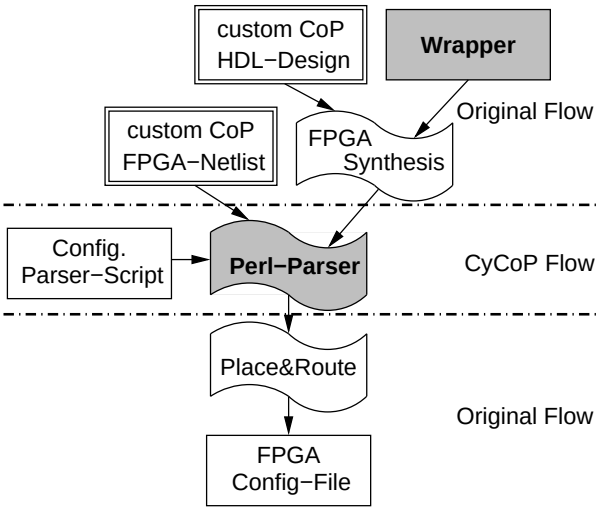

Figure 2.5: HW Synthesis Flow

The CoP can be available in different forms. A core designed "in house" mainly exists in form of a Hardware Description Language (HDL), such as VHDL or Verilog. However, licensed IP cores mostly are not available in HDL for copyright reasons. These IPs are distributed only in form of a net list. Thus, like in the software flow, we have to choose an IR layer to make sure that we can handle all the design elements needed for the prototype simultaneously. As the synthesis tool exports its output into a net list form we choose the net list as the IR to manipulate them for the CyCoP platform. The drawback of this approach is that almost every synthesis tool vendor has

its own net list format. However, the Electronic Design Interchange Format (EDIF) is predominantly used as a neutral net list format. Only the EDIF format makes the licensing model of IP cores possible. Therefore, EDIF is the favorite format as the IR layer in the CyCoP synthesis flow. However, in the current implementation only Xilinx's net list format NGC is supported [50].

The Wrapper is available as VHDL code. This simplifies the configuration of the in- and output ports to connect the CoP. Also the size and depth of the command and result buffers must be specified by the user, as well as the encoding of the Wrapper requests from the MW. After the configured Wrapper has been synthesized, all prototyping modules are available as a net list.

The CyCoP design flow now enters into its specific stage, where it diverges from the original one. To achieve full control over the CoP core by the Wrapper, some modifications are done by a Perl script. The modifications affect only the sequential logic of the CoP core. As we learned in Section 2.3, the Wrapper controller steers the clock of the CoP core. To implement real clock gates in an FPGA design is a difficult task and requires numerous manual interventions. Therefore, the Perl script extends a sequential component with a combinatorial enable signal, which computationally has the same effect as the clock gate.

Furthermore, the Perl script lines up all memory elements of the CoP core in several scan-chains. This is done to control the context of the CoP core. In Chapter 4 we will discuss this in greater detail.

After the manipulation of the net list of the CoP core by the Perl script, the synthesis flow reenters the original flow. The net lists are placed on and routed to a specific FPGA device by the PAR tool. The output of the tool is a Bitfile, which can be downloaded into the FPGA itself. This is the final configuration file for the FPGA, and we are now ready to run the CoP with the CyCoP platform.

2.5 Hardware Setup

To show the feasibility of the CyCoP concept, the platform is implemented in real hardware. Figure 2.6 shows the setup used for the platform. The CyCoP platform is designed to make use of real hard-

ware components, like a processor chip, combined with programmable logic devices, like an FPGA. On the right half of the figure we find a motherboard-like PCB, on which a processor chip is mounted. The PCB consists further of PCI slots. In one of these slots a PCI card is plugged in on which an FPGA is mounted. Thus, these are the components needed by the CyCoP platform. On the left hand of the figure we find some additional equipment which is not part of the CyCoP platform, but they are used to monitor and measure the test runs.

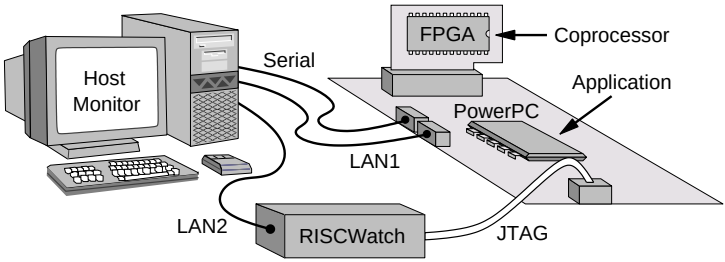

Figure 2.6: Hardware Setup

2.5.1 The Processor Board

One of the main components is the processor board. This board consists of the processor which has the real CPU core implemented on a real die. All the evaluated applications are running on this chip. Also the MW is therefore programmed on this processor.

We used the IBM PowerPC 440GP Evaluation Board [51] for the purpose to evaluate the feasibility of the CyCoP concept. The IBM evaluation board is similar to a normal PC motherboard. The board comprises

- a PowerPC 440GP,
- 128 MB of DDR 266 MHz memory,
- two Ethernet 10/100 Mbps ports,
- two RS-232 ports,

- and four PCI-X slots.

The board-level interconnect is the PCI-X bus which is compatible to the normal PCI bus. Thus, either PCI-X cards or PCI cards can be used in the slots.

The main component on the board is the PowerPC 440GP. It is a 32-bit RISC processor. This processor is a SoC design with a PowerPC440 core as the main processing unit, which is connected over an on-chip bus with many peripheral cores. The PowerPC440 core runs at 400 MHz and its ISA is a 32-bit implementation of the Book E Enhanced PowerPC architecture. To enhance the performance of this embedded processor, its architecture comprises pipelined and superscalar operation and large L1 caches.

An important advantage of the PowerPC 440GP is that there are 8 KB of on-chip SRAM connected to the on-chip bus. The Middleware (MW) has to be implemented into a cache inhibited memory region which reduces the execution performance of the MW drastically. As on-chip memory can be accessed faster than off-chip memory, the performance penalty can be reduced by storing the MW in this on-chip memory.

The Ethernet and serial port of the evaluation board are used to monitor the outputs of the prototype system. The board has no possibility to connect neither a display nor a keyboard, therefore, all in- and outputs are made over these ports.

2.5.2 The FPGA PCI-Board

The CoP system of the CyCoP platform is implemented in programmable logic devices which are connected over the board-level interconnect to the CPU. As the IBM evaluation board provides a PCI-X bus, the CoP system consists of a PCI card mounting an FPGA. At the beginning of the project we only used the Spyder-Virtex-X2E FPGA prototyping board [52] (see Figure 2.7) to implement the CoP system.

The FPGA on the Spyder board can be arbitrarily configured, enabling the implementation of any application-specific extension for the PowerPC core. The board comprises

- a Xilinx Virtex-Series FPGA device XCV800,

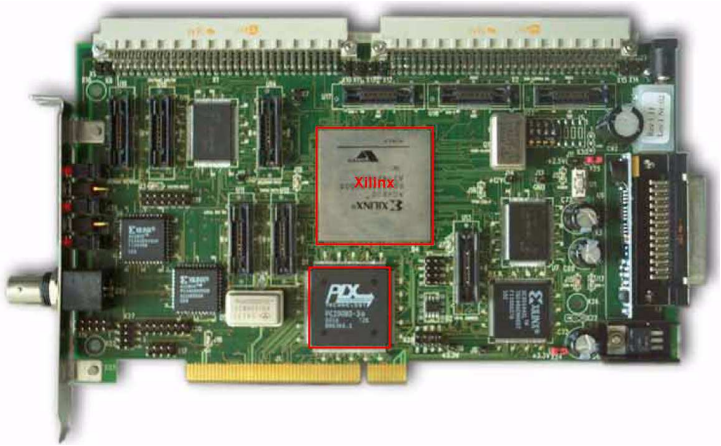

Figure 2.7: Spyder FPGA Board

- 8 Mb of SRAM memory,
- and a dedicated PCI bus arbiter PLX9080.

The FPGA consists of 800 K of equivalent gate logic and runs at 40 MHz. Thus, an implemented CoP core on this FPGA runs ten times slower than the CPU core. The FPGA is indirectly connected to the PCI bus. Thus, the *BusAccessor* in the Wrapper of the CoP system does not consist of the PCI protocol which is rather complex and therefore space consuming in an FPGA. The actual PCI protocol is executed by the PLX9080 which is a PCI bus arbiter. This arbiter acts as a bridge between the PCI bus and a local board bus. The local bus is much simpler in its architecture and fits therefore well into the FPGA in place of the *BusAccessor* of the Wrapper. A drawback by using the arbiter is that it introduces additional delay on the PCI bus access, which reduces the overall performance of the CyCoP platform.

The Spyder board only consists of volatile memory, therefore, on every power-up of the system the board has to be reconfigured with the CoP core and the Wrapper. The configuration is done over the PCI bus by the MW. Thus, the software driver of the MW must be able to configure the FPGA over the PCI bus. The introduced latency

can be ignored as the configuration happens only once at the startup of the platform.

A bigger problem of the Spyder board is that it comprises an outdated FPGA device. For most modern CoP cores the FPGA is too small. Therefore, a second PCI board is added to the project. The second board is from Amirix and is called AP130 platform FPGA development board [53] (see Figure 2.8). The board comprises

- a Xilinx Virtex-II Pro FPGA device XC2VP30,
- 64 MB of DDR memory,
- 4 MB of Flash memory,
- a dedicated PCI bus arbiter PLX9056,
- and a non-transparent Intel PCI-PCI bridge 21555.

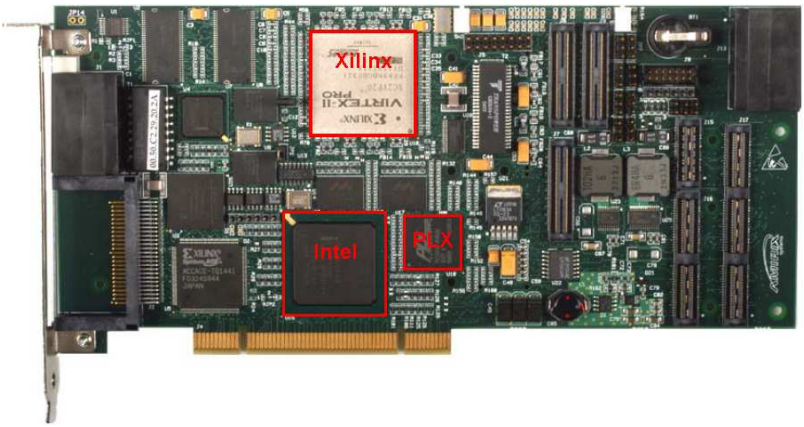

Figure 2.8: Amirix FPGA Board

This board has the advantage that it comprises non-volatile memory where the FPGA configuration can be stored. The FPGA configuration is programmed only once and is available afterwards after every power-up. Also the FPGA has higher capacity than the one on the Spyder board. The Virtex-II Pro series also consist of embedded

PowerPC405 cores. However, they are unused for this project as the PowerPC405 is significantly different from the PowerPC440. Due to the more modern process technology the FPGA can also be faster clocked. The Virtex-II Pro is running at 100 MHz.

The drawback of this board, however, is that the board comprises two bridges to access the host PCI bus from the FPGA. Like on the Spyder board, these bridges introduce additional delay on the PCI bus access. The impact of this delay is discussed in detail in Chapter 5.

2.5.3 The Monitor Equipment

The IBM evaluation board provides no ports to attach a display or a keyboard. In order to provide the user a terminal to interact with the prototype, an additional PC is used. On this PC a terminal program is configured to communicate with the PowerPC 440GP over a serial port. At runtime the application's terminal output, like a `printf`, can be monitored at the PC.

At the power-up of the PowerPC 440GP boots with a program called U-Boot [54]. This boot-loader provides the user with basic functions to control the PowerPC 440GP. For example, binary files can be downloaded into the evaluation board memory per TFTP. Afterwards the boot-loader jumps to the address of the downloaded binary and user application is executed. Thus, the Ethernet port of the evaluation board is used to download binary files to the memory.

In Figure 2.6 we find additional equipment which is connected to the IBM evaluation board. The RISCWatch [55] is a hardware and software development tool for the PowerPC family. The source-level debugger and processor-control features provide the user with the tools needed to develop and debug hardware and software quickly and efficiently. With the RISCWatch the complete internal state of the PowerPC 440GP can be monitored and manipulated. This part of the equipment actually is a feature, but the CyCoP platform also works without it.

2.6 Summary

In this chapter we introduced the CyCoP platform, which enables cycle-accurate emulation of a coprocessor with its processor. The concept of the platform is hardware-independent and can be adapted to any hardware environment. Only two modules are needed to enable the emulation. The Middleware resists on the same CPU as the software application. The Middleware offers the application to access the prototyped coprocessor as if it already resided on the same die. Furthermore, the Middleware hides any latency introduced by the prototype from the application and simulates a real-time environment for the application.

The Wrapper is the counterpart of the Middleware for the coprocessor core. All coprocessor requests that the Middleware fetches from the application are sent from the Middleware through the Wrapper to the coprocessor core. The Wrapper controls the coprocessor core so that the core only runs when requests are present. The Wrapper also counts the computation cycles of each request to provide the Middleware with the time. The Middleware is then able to take this time into account for the application run time.

Subsequent chapters describe the implementation of the platform for the two coprocessor classes, namely, the loosely- and tightly-coupled coprocessors. The implementation details are kept as general as possible and independent of the underlying hardware setup. However, certain implementation details are specific to the hardware used and emphasized as such.

Chapter 3

Tightly-Coupled Coprocessors

In the previous chapter we introduced the concept of the CyCoP platform and the details of its architecture. We learned in Section 1.2 that the coprocessors (CoP) can be divided into two main categories; tightly- and loosely-coupled once. This chapter handles the implementation details of the CyCoP platform for the tightly-coupled CoPs (TC-CoP).

In principle the task of verifying a TC-CoP demands only to simulate or emulate the new instruction-set architecture (ISA) of the processor. Many solutions to this task are already available and in Section 3.1.1 we briefly describe the advantages and the drawbacks of these solutions. In Section 3.1.2 we face the challenges of the CyCoP approach, which originate from the modern processor architecture.

With the solution introduced by the CyCoP platform we overcome the drawbacks of the other solutions. Section 3.2 introduces the concept of replacing original CoP instructions with MW calls applicable for the CyCoP platform.

To analyze the capabilities of the MW call concept we sum up in Section 3.3.1 the emulation latencies and overhead. With the findings of Section 3.1.2 we examine in Section 3.3.2 the emulation accuracy of the CyCoP platform emulating TC-CoPs. This leads us to the

limitations of the platform which we discuss in Section 3.4.

In the previous chapter we consciously omitted certain details about the Middleware (MW) and the Wrapper. In Section 3.5 we describe the details of the MW and Wrapper relevant for the TC-CoP emulation, which were omitted in the previous chapter. The last section of this chapter summarizes the findings and concludes this chapter.

3.1 State of the Art

3.1.1 Related Work

For the verification of a new ISA many tools and techniques already exists. In this section we list only the ones which have the capability to cycle-accurately emulate or simulate the target system. We briefly describe the method with their strength and their drawbacks regarding the co-verification.

Simulators

Instruction-Set Simulators (ISS) mimic the behavior of software application on a target processor, running on a host computer. These simulators should be fast to handle the increasing complexity of processors, flexible to handle all features of applications and processors, e.g. runtime self-modifying codes, multi-mode processors; and retargetable to support a wide spectrum of architectures. Although in the past years, performance has been the most important quality measure for the ISA simulators, retargetability is now an important concern, particularly in the area of embedded systems and SoC designs.

The ISSs perform up to a few tens of MIPS on a desktop PC [56]. As the processor models for these simulators are simplified to improve simulation performance, such the simulators do not indicate exactly the performance of future processor. Furthermore, the software application has to be adapted to the simplified processor model and the simulator environment. Some ISS even only emulate the behavior of the instruction without any details of the actual processor architecture [57]. Even so the simulators allow decent software verification; the verification of the hardware core is ignored with this method.

Co-Simulators

Co-simulators combine an ISS with an HDL simulator [58]. This provides the possibility to emulate the CoP at the Register-Transfer Level (RTL). Such a co-simulator is significantly slower than a single ISS, as the level of simulation details is high in an HDL model. Because of the many events necessary to simulate the software operations, this level of simulation needs to be fast. Thus, the models are required to be high-level [1]. The co-simulator has therefore, only one advantage over the ISS, that it also simulates the hardware.

FPGA Emulators

Many designers choose programmable logic devices, e.g. FPGAs, as an alternative prototype platform to implement the entire processor [59, 60, 61]. This has the advantage that software and hardware can be verified at the same time. Furthermore, modern FPGA allow clock frequencies of several 100 MHz, which provide a good emulation performance. Configurable and extensible processors like the Xtensa from Tensilica [62] are soft-core processors and therefore they can be compiled into an FPGA.

Hard-core extensible processors are highly optimized on the gate level as well as on the architecture level. As the use of the FPGA is limited by the number of Configurable Logic Blocks (CLBs) and the routing capability, it is difficult to implement the processor with all its functionalities in the FPGA without simplification of the core. Furthermore, the vendors of hard-core extensible processors do not grant any access to their processor net lists. Therefore, the use of an FPGA emulator is limited to the available processor net lists.

3.1.2 Processor Characteristics

Modern processor architectures are highly optimized. These architecture optimizations affect the instruction flow of the code sequence. The following subsections introduce the main optimizations concepts and describe how they affect the emulation accuracy of the CyCoP platform.

Pipelining

Today's processors are all pipelined in order to enhance their instruction throughput and clock frequency. For instance the processor in Figure 3.1 has a five-stage pipeline. Thus, up to five instructions are being executed simultaneously. Also the data path extension, caused by the TC-CoP, can be pipelined.

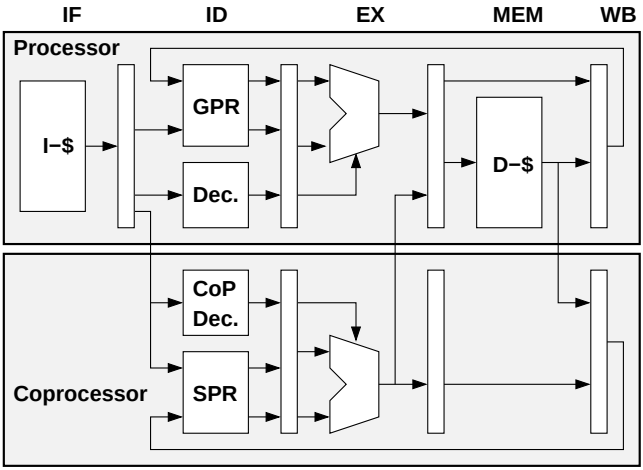

Figure 3.1: Pipelined Processor with a CoP

The CyCoP concept denotes that it makes use of the original CPU core for the emulation. Thus, the instructions of the original ISA are perfectly pipelined on the platform. A sequence of original instructions is perfectly "emulated". To achieve the same perfect emulation for a CoP instruction sequence the MW has to parse after the call at the address of the caller instruction whether other CoP instructions are following. If so, the MW has to send them all to the CoP together. The wrapper stores them in the sequence of their arrival and issues them into the CoP itself. The CoP executes the instructions in a pipelined manner. Thus, the emulation is also perfect.

However, the situation changes if either of the sequences is interrupted by the other type of instructions. Let us assume we have the example five-stage processor of Figure 3.1 with an integer multiply-

accumulate (MAC) CoP and the following code is executed:

```
add  r7, r3, r4; /* r7 = r3 + r4 */
sub  r5, r3, r4; /* r5 = r4 - r3 */
madd r8, r5, r7; /* r8 = r8 + (r5 · r7) */
```

In every cycle one instruction is issued into the pipeline, starting with the addition. After the subtraction is issued, the MAC instruction (`madd`) will follow in the next cycle. As both operands for the MAC operation have not been yet computed at this point in time this instruction cannot be issued. The instruction will not issue until both operands are available. In such a case with a data hazard [47] between a CoP instruction and a processor instruction, the time to wait for the operands of the CoP instruction is not constant.

To emulate the issue time of a CoP instruction after a processor instruction correctly, the MW needs to know the exact state of the processor. As the MW is running on the processor itself, it is not able to determine the exact state of the processor. The MW call must ensure that all instructions that are issued into the CoP have no data hazard any more. All processor instructions have to *commit*, i.e. that all results are safe. Therefore, the MW call has to be a precise exception.

Thus, the data hazards and their influence on the instruction throughput are perfectly emulated among instructions sequences of the same type—original or CoP. For accurate emulation between the two types further measures have to be taken.

Instruction-Level Parallelism

Instruction-level parallelism (ILP) is a further method to enhance the performance of a processor [47]. It exploits possible parallelism among instructions. A processor which consists of several execution pipelines has the ability to execute multiple instructions in the same pipeline stage simultaneously. For example the processor in Figure 3.1 actually has two execution pipelines in parallel. Therefore, it is theoretically possible for the processor to issue one original and one CoP instruction concurrently. The processor core could, of course, have even more execution pipelines than only one. This requires that at the Instruction Fetch (IF) stage of the processor more the one instruction can

be fetched at once. The Instruction Decode (ID) stage decides how many and where the instructions are issued. Such architectures are called a superscalar processor [47].

The parallelism between an original instruction-flow is perfectly explored as the application runs on the real processor. If the MW explores after a call the subsequent instructions as well, it can provide a CoP with multiple execution pipelines also with more than only one CoP instruction at once. Therefore, also the parallelism between a CoP instruction-flow is explored perfectly. However, the problem is again up to the instruction type transition. To emulate the issue time of a CoP instruction after an original instruction correctly, the MW needs to know the exact state of the processor. As the MW is running on the processor itself, it is not able to determine the exact state of the processor.

For the CyCoP platform, the question is whether the processor core provides a separate issue port for the CoP. As these issue ports are expensive area-wise, the core usually only has issue ports for the original execution pipelines—the CoP has to share one with a processor-specific execution pipeline.

Flow Control Instructions

Flow control instructions are instructions that influence the execution flow of an application. In a C-program, e.g., the `if`-statement or the `goto`-statement are flow control instructions. When the execution reaches these points in the source code, the next instruction to be executed is out of the old instruction sequence.

The conditional branches are a challenge to improve the performance of pipelined processors. The execution of the condition, e.g. a compare operation, might be still in execution when the branch instruction is issued. The problem is that the processor cannot calculate the address for the next instruction until it knows whether the branch has to be taken or not. Therefore, the processor tries to *predict* the outcome of the condition computation [47]. These processors start to execute the following instructions even though it might be wrong. If it turns out that the processor did predict wrongly, the current results have to be deleted, i.e. the pipeline has to be *flushed*.

3.2 The Middleware Call

In Chapter 2 we discussed the principals of the CyCoP platform. The MW enables the use of the TC-CoP in the CyCoP platform for the application. The MW has to execute the CoP requests of the application because the real TC-CoP is not available during the emulation. For example, we assume an application that performs floating-point calculations on a TC-CoP. If the application is written in C, typical statements performing floating-point and integer calculation are

```
float fx, fy;
int   ix, iy;
ix = iy + ix;
fx = fy * fx;
```

This compiles as the following sequence of PowerPC assembly instructions:

```
add  r4, r3, r4; /* integer addition */
fmul f7, f8, f7; /* float multiplication */
```

Given that the real TC-CoP is not available during the emulation, the CPU will encounter an *illegal instruction* at the `fmul` instruction. Thus, the CPU generates an *illegal instruction exception* because it will not recognize the opcode of the CoP instruction. If, e.g., the MW is called in place of the illegal instruction exception handler, then the emulation of the CoP can happen. Thus, the task of the MW is to fetch coprocessor-specific instructions from the application and to execute them for the emulation environment.

There are two ways to enable the CoP instruction to call the MW for emulation, all of which belong to the category of the *precise exceptions*. A pipelined processor is said to have precise exceptions, if the pipeline can be stopped so that the instructions just before the interrupting instruction are completed and those after it can be restarted from scratch [47].

The first method is the illegal exception as explained above. Another method is to replace the original CoP instruction with an instruction that forces a software interrupt, e.g. a *system call* or a *trap*, i.e. a breakpoint. These instructions mainly provide some immediate operand, i.e. a constant within the instruction itself, in which the

recoded CoP instruction takes place. The corresponding exception handler then has to be replaced by the MW.

Thus, either the CoP instruction has to be replaced with a specific MW call for the emulation or the MW will be called instead of an *illegal instruction exception* handler. The choice which method is the best suited depends on several aspects. If there is an OS running on the CPU, the OS uses the system call for its own purposes. Therefore, to avoid a modification of the OS for the purpose of emulation, we avoid this method. Furthermore, if the re-encoded CoP instruction will not fit into the immediate field of the system call or trap, we might come to the decision to use the illegal instruction exception to do the job.

If we would replace the CoP instruction with an ordinary software function call to jump into the MW, the compiler would generate much more code within the application to save the current register context and to load the argument registers. This would influence the application execution time and flow significantly, and is therefore not a suitable solution for the prototype platform. Even if we used only a branch instruction to jump into the MW, we would not get the correct emulation results because a branch instruction is not *precise*. This fact is explained in greater details in the following chapters.

By replacing the CoP instructions with function calls the size of the application will be changed. This has the effect that the original memory organization of the original software has to be adopted. This contradicts with the claim to keep the original design flow. Furthermore, this influences the memory and cache behavior tremendously. If the behavior of the memory and the cache changes in for the emulation, the prototype cannot guarantee cycle-accurate results anymore.

3.3 Analysis

3.3.1 Emulation Performance

To be able to compare the CyCoP approach of emulating instruction-set extensions with the methods described in Section 3.1.1, we have to analyze the possible emulation performance of the CyCoP. During emulation, the latencies introduced by the prototyping environment

is hidden from the application. However, to estimate the performance of the CyCoP platform itself, these latencies must be known.

The principle of the emulation in the point of view of latencies is the following:

1. Emulation is setup at power up.
2. Each CoP instruction rises a precise exception.
3. The MW decodes the CoP instruction and sends the operands with the opcode to the wrapper over the PCI bus.
4. The Wrapper runs the CoP and returns the results and execution time to the MW.
5. The MW prepares the results and returns to the application.

In Figure 3.2 the sequence of the listing is shown graphically and tagged with a variable which will be discussed later in this section. As we can see, the application and the coprocessor runtime of the future architecture is delayed in the emulation by the MW and Wrapper execution time.

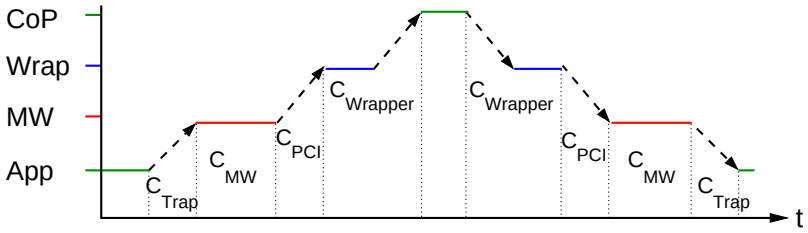

Figure 3.2: Emulation Timing Sequence

The total runtime of the emulation T_{Em} can be expressed with the following equation:

$$T_{\mathrm{Em}} = C_{\mathrm{Setup}} + T_{\mathrm{App}} + C_{\mathrm{Em}} \tag{3.1}$$

The sum includes the constant setup time C_{Setup} of the emulation, the runtime of the application on the target system, consisting of

the CPU and the CoP, T_{App} and the emulation latency C_{Em}. In the list above the points 2–5 represent in detail the emulation latency. For the further analysis the terms *time* and *latency* are defined as a *number of clock cycles of the CPU.* Thus, all the measurements are made relatively to the clock cycles of the CPU even when performed elsewhere in the system.

Point two of the listing represents the trap latency C_{Trap} which consists of two parts. The first part comes again from the fact that the trap is a precise exception, thus the trap "waits" for all older instructions to commit. The number cycles the MW has to wait is expressed with the variable E_{MWCall}. We will discuss this in greater detail in Section 3.3.2. The second part is the time the processor needs to actually execute an exception, T_{IRQ}. The total trap latency is the sum of these two parts multiplied by the number of MW calls n_{MWCall}:

$$C_{\mathrm{Trap}} = n_{\mathrm{MWCall}} \cdot (E_{\mathrm{MWCall}} + T_{\mathrm{IRQ}}) \tag{3.2}$$

Point three in the list describes the actual execution of the MW and the data transfer over the PCI bus. It is similar to the point four where the execution time of the Wrapper has to be taken instead of the MW. The execution time of the MW T_{MW} is constant for all CoP instructions n_{CoP} and as the MW parses the source for subsequent CoP instructions this time counts for all CoP instructions executed by the application. The fact that the MW is executed of a cache inhibited memory region only slows down T_{MW}.

$$C_{\mathrm{MW}} = n_{\mathrm{CoP}} \cdot T_{\mathrm{MW}} \tag{3.3}$$

The PCI transfer time C_{PCI} depends on the bandwidth of the bus and the amount of data to be transferred. For every CoP instruction (n_{CoP}) two words has to be transferred; one for the request and one for the result. For every MW call (n_{MWCall}) two additional control words are needed for the Wrapper controller.

$$C_{\mathrm{PCI}} = \frac{(n_{\mathrm{MWCall}} + n_{\mathrm{CoP}}) \cdot 2 \cdot \mathrm{words}}{\mathrm{PCIBandwidth}} \tag{3.4}$$

The Wrapper has only a small fixed amount of FPGA cycles k of overhead every time it is call. Since the k cycles are executed within the FPGA we have to take the frequency ratio between the FPGA

f_{FPGA} and the CPU f_{CPU} into account. The number of Wrapper calls is equal to the number of MW call n_{MWCall}. For the total Wrapper overhead C_{Wrapper} the following equation can be written:

$$C_{\mathrm{Wrapper}} = n_{\mathrm{MWCall}} \cdot k \cdot \frac{f_{\mathrm{CPU}}}{f_{\mathrm{FPGA}}} \tag{3.5}$$

Summing all up the following equation results from all emulation latency:

$$C_{\mathrm{Em}} = C_{\mathrm{Trap}} + C_{\mathrm{MW}} + C_{\mathrm{PCI}} + C_{\mathrm{Wrapper}} \tag{3.6}$$

In Chapter 5 we will get real numbers for the individual latencies and can quantify them.

3.3.2 Emulation Accuracy

In Section 3.1.2 we pointed out some architectural specialties of modern processors. These characteristics highly affect the execution behavior of the source code. On CyCoP this behavior is exactly modeled for instruction sequences of the same type; either original or CoP instructions. However, at the transition from one type to the other CyCoP we need some additional information about the internal state of all pipeline registers of the processor, to extract the real issue time of the CoP instruction or vice versa. In this section we express the introduced emulation inaccuracy of the CyCoP platform by using only precise exception as the MW call without further arrangements.

Instruction-Level Parallelism and Pipelining

In case of a superscalar architecture the processor may issue CoP instructions in parallel to the original instruction set. We take the following pseudo-code of intermixed assembler instructions of original instruction set and CoP instruction set as a simple example and map it onto a architecture model:

```
cpu_0;    /* processor's original instruction */
cpu_1;
cop_0;    /* coprocessor instruction */
cop_1;
cop_2;
```

```
cpu_2;
cpu_3;
```

This could be mapped on an abstract example architecture like the one shown in Figure 3.3. The figure only shows the issue unit with two ports for the original instructions ($n_{\mathrm{CPUport}} = 2$) and two dedicated for the CoP instructions ($n_{\mathrm{CoPport}} = 2$) and the according execution pipelines.

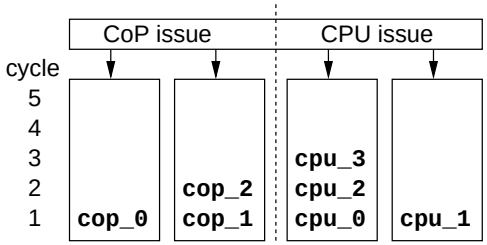

Figure 3.3: Exploiting the full ILP

We see that this instruction sequence could be executed back to back and would take three cycles, not comprising the execution latency of the pipeline. If we map the same instruction sequence on the CyCoP platform, the execution would virtually look as pictured in Figure 3.4.

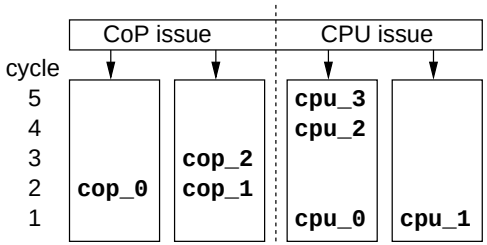

Figure 3.4: Emulated Coprocessor ILP

Additional cycles are introduced in the emulation. The emulator hides the latencies of the emulator platform and accurately performs the execution of the instruction sequences within the CPU and the

CoP. In this figure we can quantify the number of cycles (E_{ILP}) that the emulated run time differs from the actual one, depending on the number of dedicated CoP issue ports n_{CoPport}. E_{ILP} is again proportional to the number of MW calls n_{MWCall}:

$$E_{\mathrm{ILP}} = \begin{cases} 2 \cdot n_{\mathrm{MWCall}} & \text{if } n_{\mathrm{CoPport}} > 0 \\ 0 & \text{else} \end{cases} \tag{3.7}$$

However the picture in Figure 3.3 could show also a slightly different situation, in case there would be a data dependency between that CPU instruction `cpu_1` and the CoP instruction `cop_0`, so-called data hazard. The execution of the CoP instruction would be delayed until the data is ready from the previous instruction. Then the Figure 3.4 would show a too little number of delayed cycles. If the CPU instruction is of an arithmetic type the delay between the two instructions can be quantified with the length of the execution pipeline of the CPU instruction n_{CPUpipe}.

In case the CPU instruction is of a load type, causing a data hazard with the following CoP instruction, the delay depends on the latency of the data to be transferred from its source (e.g. external memory) to the internal register. This number is very system specific but can be quantified as well, since the source and its latency is known. For memory accesses we have to distinguish between a cache miss (n_{miss}) or hit (n_{hit}) to model the latency and for other external sources a fixed bus latency (n_{bus}) can be modeled.

However, the MW call "waits" until all instructions, which are issued previously into the individual execution pipelines, are finished with execution. Only after all instructions have committed, the MW is called and the application time is stopped by the MW. If any of these instructions trigger a data hazard on the following CoP instruction, this CoP instruction would have to wait for the previous instruction to commit as well. Therefore, the emulation behaves exactly the same as in the future architecture. No inaccuracy in the cycle-behavior of the emulation appears.

For an instruction sequence without any data hazards between the CPU instruction and the CoP instruction, the stopping of the time happens too late, as the CoP instructions in the final architecture do not wait for all instructions in the pipelines to commit. To quantify the

number of inaccurate emulated cycles (E_{MWCall}) the same numbers are taken into account as already estimated for the data hazard case:

$$E_{\mathrm{MWCall}} = \begin{cases} n_{\mathrm{CPUpipe}} - 1 & \text{if arithmetic CPU instruction} \\ n_{\mathrm{hit}} - 1 & \text{if memory cache hit} \\ n_{\mathrm{miss}} - 1 & \text{if memory cache miss} \\ n_{\mathrm{bus}} - 1 & \text{else} \end{cases} \tag{3.8}$$

We need the same discussion on the transition from the CoP instructions to the CPU instructions. From a theoretical standpoint all evaluated architectural details about the opposite transition are here true as well. The Wrapper waits until all instructions, which are issued into the CoP, are finished with execution before it sends it back to the MW. In case of a data hazard between the CoP and the CPU instruction the behavior would be expected to be correctly reflected. This is true for the arithmetic types, however, the load latency has to be performed and measured on the CPU side and sent to the CoP together with the actual request and the loaded value. The Wrapper takes this number into account for the emulation of the load instruction.

For the non-hazard case we quantify the number of inaccurate emulated cycles (E_{Wrapper}) the same way as already done for CPU. The only change is the number of pipeline depth, which now is equal to the CoP execution pipeline depth:

$$E_{\mathrm{Wrapper}} = \begin{cases} n_{\mathrm{CoPpipe}} - 1 & \text{if arithmetic CoP instruction} \\ n_{\mathrm{hit}} - 1 & \text{if memory cache hit} \\ n_{\mathrm{miss}} - 1 & \text{if memory cache miss} \\ n_{\mathrm{bus}} - 1 & \text{else} \end{cases} \tag{3.9}$$

The MW can detect any kind of data hazards between the CPU and CoP instructions and vice-versa. At a data hazard only the inaccuracy E_{ILP} is taken into account for the final time base manipulation, the rest of the measurement already show the correct cycle times. If there is no data hazard, either way, the MW has to trace at least $\lceil IPC \rceil$ (Instructions Per Cycle) —since the time base for the analysis is the CPU clock, the IPC turns into a number without a unit— instructions from either the CPU or the CoP instruction trace to

find the most lasting instruction and pick the according E_{MWCall} or E_{Wrapper}. With that a very high accuracy also in the emulated run time can be achieved and the error can be quantified at the end of the run time.

Flow Control Instructions

As the MW call is also issued in the processor core, it will be predicted in the same way as any other instruction and there will be therefore no difference between emulation and future CoP instructions behavior. However, as we use a precise exception for the MW call, the CoP never has to be flushed: The prediction of a processor instruction will never affect the emulated CoP, even though the MW call behaves in an accurate way within the processor itself.

3.4 Limitations

In the section above we discussed the sources of inaccuracy for the CyCoP method of emulating instruction-set extensions. These inaccuracies can only be eliminated if the MW knows the entire internal state of the processor core. With this information it would be possible to calculate the exact issue time of the CoP instruction and manipulate the processor time base accordingly.

There are processors, such as the PowerPC 440GP [14], which are able to trace the running code in real time. This tracer would provide the kind of information CyCoP needs to emulate the CoP 100% cycle-accurate. However, the emulation time would then increase significantly because at every MW call, the entire trace would have to be read and analyzed. Hence, in our first implementation we omitted the tracer method.

A more general solution to this problem is the following. In Chapter 2 we learned that in the software design flow the original source code is parsed to replace CoP instructions with MW calls. If this parser checks also how many instructions are executed simultaneously in the processor before the transition happens, the parser can estimate the “waiting” time of the MW call for all instructions to commit. This estimation can be encoded together with the rest into the MW call

and the MW takes this time into account for the emulation. With this method the introduced emulation inaccuracy is reduced. However, effects like a cache miss for instructions or for data are ignored. Nevertheless, this solution clearly enhances the emulation accuracy.

3.5 Implementation

In Chapter 2 we gave an overview of the concept of the CyCoP platform. This section describes in greater details how the MW and the Wrapper are adapted to our hardware setup, to emulate tightly-coupled CoPs.

3.5.1 Middleware

As MW call we used the *trap* instruction, thus the MW replaces the debug interrupt handler. Using trap instructions within the PowerPC has a great advantage: to maintain the semblance of "real time" operation while an application is being debugged, the PowerPC trap instruction automatically stops incrementing the *time base* for as long the debug event bit is set. This greatly facilitates the cycle measurement.

The time-base value at the end of a MW call is calculated with the equation

$$t_{\text{Base}} = t_{\text{Base}} + CoP_{\text{Cycles}} \cdot \frac{f_{\text{CPU}}}{f_{\text{CoP}}} - t_{\text{Trap}} \tag{3.10}$$

The CoP execution time CoP_{Cycles} is multiplied with the ratio of the frequencies. For the CoP frequency the target frequency of the future architecture is taken. The product is added to the current time-base value. Furthermore, the exception execution of the trap call takes a constant time, which we have to subtract from the new time. The trap instruction needs a constant time to stop the time-base register.

The PowerPC trap instruction `twi` provides a 23-bit immediate operand. This provides sufficient space for our implementation to encode the CoP instructions, e.g. we used eight bits for the CoP opcode and five bits for three operand register. Thus, we can encode $2^8 = 256$ instructions and address $2^5 = 32$ registers.

Normally, application code is stored in cached external SDRAM. Calling the MW from there causes its code to be loaded into the instruction cache. Therefore, when the application continues, the cache content has changed, which influences the further execution time of the application. Owing to the cache architecture and the necessary flexibility of the MW, it is difficult to predict how the instruction cache will be modified and what the impact on the application will be.

An easy solution is to disable caching of the MW's memory page. However, this increases the emulation time, because every instruction is fetched from uncached external SDRAM, which has much higher access latency than the cache. The PowerPC 440GP provides a fast, on-chip 8 kB SRAM. Thus, the MW code is placed into this on-chip memory and the caching is inhibited for this address space.

Also the MW stack is placed within the on-chip memory region. Accordingly, also data cache is inhibited during the MW execution and avoids "contamination" of the data cache of the application. However, to provide a realistic emulation, there might be CoP load and store instructions that actually have to contaminate the data cache of the application.

A big part from all instructions of an application is load/store instructions, as the operations must be provided with operands which are mainly located in the external SDRAM. Thus, a major part of the processor performance depends on the memory access delay. By the introduction of data caches the memory delay has been reduced significantly. A data located within the cache is accessed in a single cycle. However, if the data is not located within the cache, the bigger the penalty is.

To provide a realistic emulation, the CoP load/store instructions have to contaminate the data cache of the application. On the one hand the CoP data are faster accessed, but on the other hand valuable cache space is occupied. To properly emulate the external memory access time, the CoP load/store instructions are not directly replaced by a trap instruction, but by a load/store instruction of the processor core. Thus, the memory access is done with a General-Purpose Register (GPR). This is performed in real time.

However, if the compiler does not schedule already at compile time a GPR to store the CoP data, the content of the GPR has to be tem-

porarily stored somewhere else. If the original GPR content is store temporarily into the MW stack, this would create significant artificial latency within the application execution time. Thus, the original content has to be moved into another register within the processor. The *link* register could be one possible target, because its content is automatically stored in the procedure stack by the compiler. Some processors like the PowerPC provides some additional registers in the processor core that are never allocated by a compiler. Therefore, these registers are predestinated to temporarily store the content of the GPR. For example, the PowerPC single-precision floating-point load instruction

```
lfs FRT, D(RA); /*load content at D+RA into FRT*/
```

is replaced by

```
mtsprg SPRX, RX;      /* move RX to SPRX */
lwz    RX, D(RA);     /* load into RX */
twi    <lfs FRT, RX>; /* trap call */
```

The PowerPC instruction `mtsprg` moves the content of the GPR `RX` into the special-purpose register `SPRX`. This special-purpose register is specific for the PowerPC architecture. The integer load instruction `lwz` then gets the memory content at the address `D+RA` and stores it into the GPR `RX`. After performing the load in real time the MW is called with the trap instruction `twi`. The immediate field of this instruction tells the MW that a CoP load has been performed and that the value is stored into the GPR `RX`. The MW has to restore the original content of the GPR `RX` and sends the loaded CoP value into the designated CoP register `FRT`.

The move instruction introduces for every CoP load/store instruction within the application, n_{CoPLS}, an additional latency which appears only in the emulation. Due to this additional latency the overall emulation latency would increase by

$$E_{\mathrm{CoPLS}} = n_{\mathrm{CoPLS}} \cdot t_{\mathrm{RegMV}} \tag{3.11}$$

Where t_{RegMV} represents the number of cycles the move instruction takes. Fortunately, the move instruction is performed in constant time and can be taken into account by an appropriately manipulating the time base of the processor.

3.5.2 Wrapper

The Wrapper provides the specific processor interface for the CoP. To emulate the CoP pipelining or superscalar capabilities, a series of instructions and operands are processed in consecutive cycles. As we cannot reliably transfer instructions from the processor to the FPGA in each clock cycle, the Wrapper buffers them first. When the CoP has computed the results, these are stored back into that buffer (refer to Figure 2.3).

The Figure 3.5 shows a simplified block diagram of the Wrapper. It only shows in detail the wiring between the CoP core and the command and result buffer of Figure 2.3. This simplification assists to better understand the mechanism between the CoP and the command and result buffer.

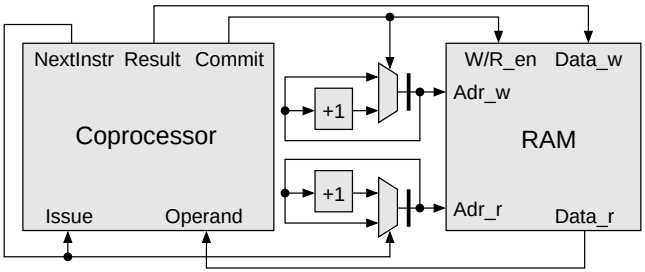

Figure 3.5: Coprocessor Access to the Command and Result Buffer

The main core of the command and result buffer is a dual-port RAM, with one write and one read port. The address pointers of the individual ports (*Adr_w* and *Adr_r*) can be incremented automatically with some steer signals. The *WrapperControl* (refer to Figure 2.3) is responsible that these address pointers are set at the right position every time the controller restarts the CoP core.

The CoP consists of five port signals. The *Result* and *Operand* ports are the main data in- and output of the CoP. The other three signals *Issue*, *NextInstr*, and *Commit* are handshake signals for the data ports. If data is valid at the *Result* port, the CoP rises the *Commit* signal. This signal automatically enables the write port of the RAM and the result data is stored at a predefined position. Further-

more, the *Commit* signal triggers the address pointer of the write port to increment. Thus, the next result data will be stored in the next higher address position within the RAM.

The *Issue* signal notifies the CoP that a new instruction is ready for execution at the *Operand* port. The *Issue* signal increments also the address pointer of read port of the RAM. Thus, in the next read cycle the next operand is read from the RAM. As soon as the CoP is ready for a new operand, it raises the *NextInstr* signal. This signal is actually the same signal as the *Issue* signal. Thus, if the CoP core is ready for the next operand it issues itself a new operand from the RAM. It is up to the *WrapperControl* to ensure that as soon as all operands are read from the buffer, that the *Issue* signal is disabled and as soon as the last result is committed it switches off the clock for the whole CoP core (refer to Figure 2.3).

3.6 Conclusions

In this chapter we presented how the CyCoP platform emulates a tightly-coupled CoP. Compared with the conventional tools, like the simulator or the co-simulator, CyCoP enables more emulation performance on the highest possible hardware accuracy which allows the designer to efficiently verify hardware/software co-design. Furthermore, the software runs already on real hardware. With CyCoP the designer is independent of processor vendors to provide him the net list, which he can integrate into an FPGA emulator.

A significant advantage of the CyCoP platform is that the application runs on the real processor and can therefore stay unmodified. Instruction sequences of the same type, either original or CoP instructions, are perfectly emulated and result in real run time behavior. The emulation of the transition in-between these two instructions types is affected by the processor specific architectural optimizations. Pipelined and superscalar processors introduce an inaccuracy in to the CyCoP platform. However, we can quantify the error, in order that the designer can take this error of measurement into account.

This chapter also introduces solutions to minimize the error of measurement in the CyCoP platform. The general solution requires only a small extension in the software design flow, but it does not

neglect the whole error of measurement. The second solution requires additional hardware equipment and special processor support, which is available for only a selection of processor types. With this hardware support a 100% cycle-accurate emulation is possible, but it slows down the emulation performance.

In Section 3.3.1 we identified the emulation overhead. The individual elements, which are responsible for the overhead, are hardware specific and we quantify them in Chapter 5.

Chapter 4

Loosely-Coupled Coprocessors

In the previous Chapter 3 we handled the tightly-coupled coprocessors (CoP) and how they are realized in CyCoP. This chapter addresses the category of the loosely-coupled CoPs.

As we learned in Section 1.2, loosely-coupled CoPs can run fully independent to the CPU. As the processes are running either on the CPU or on the CoP causal event ordering is required to provide cycle-accurate emulation. Section 4.1 introduces the process synchronization and describes the problem that has to be solved to successfully emulate loosely-coupled CoPs. Known methods for the synchronization problem are evaluated in Section 4.2. This section points out the limits of the methods to use them in a hardware prototyping environment.

In Section 4.3 we come up with a new algorithm which overcomes the limitations of the known methods. The emulation behavior of this new algorithm is analytical analyzed in Section 4.4. Due to the implementation of the algorithm in a real application we learned also the limitations of it. In Section 4.5 we list them up. In Section 4.6 we handle the real implementation of this theoretical algorithm into CyCoP. Finally in Section 4.7 we summarize and conclude our findings of this chapter.

4.1 Process Synchronization

Loosely-coupled CoPs can be roughly categorized in two types; stateless and state-dependent CoPs. The result of a stateless CoP depends only on the current input. The space-time diagram in Figure 4.1 shows how a CPU process communicates with a process of a stateless CoP. The vertical direction represents space, and the horizontal direction represents time. The dashed arrows denote messages. As soon as the *request* message of the CPU process arrives at the CoP, the latter starts to compute the *result*, which is sent back to the caller.

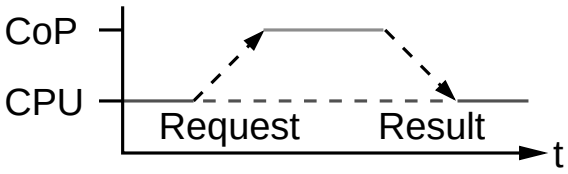

Figure 4.1: Stateless Coprocessor

As we can see in Figure 4.1 the CoP process is automatically synchronized by the CPU request. With the aim to emulate such a system cycle-accurately, the CPU sends with the request a time-stamp. When the CoP receives the request, it can update its internal timer with the received time-stamp. The CoP then computes the result of the request and sends back a message to the CPU consisting of a new time-stamp. The conveyed time-stamp is equal to the old time-stamp plus the computation time of the request.

In the meantime, when the CoP is computing the result, the CPU process is waiting for the CoP messages and holds its own internal timer. As soon as the result arrives at the CPU, the process checks the new time-stamp obtained with the message. In case where the CPU in reality would interrupt its work until it receives the result from the CoP, the CPU process then updates its internal timer with the new time and resumes work. If the CPU in reality wouldn't interrupt its work, denoted with the dashed line in Figure 4.1, the CPU process resumes work only for the time difference between the two time-stamps without considering the CoP result. After the time difference the CPU "officially" takes over the result for its further progressing.

For the second category of loosely-coupled CoPs the introduced simple synchronization method cannot be applied anymore. Figure 4.2 shows the same situation as Figure 4.1, but this time the CoP is state-dependent. The example shows that the arrival time of the *request* message is important, as the state of the CoP changes any time. The arrival time may also influence the *result*.

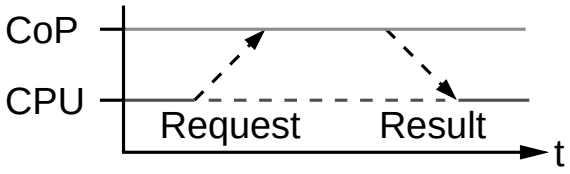

Figure 4.2: State-Dependent Coprocessor

An example for this type of CoP is a timer CoP. A timer CoP consists of a large array of timers [63, 64]. CPU processes can arbitrarily start and stop every individual timer within this array. Thus, the request to stop a timer has to be accurate in time. Otherwise the timer may expire and generate an expiration event; even so the CPU process has already requested to stop this timer.

If we use the same *blocking-read* synchronization method as in the stateless example, we lose accuracy in the emulation. Both processes in Figure 4.2 are able to rise any time a message event for the other process, as their internal state can change any time. With the blocking-read method one process becomes the "slave" of the other process. The slave can only progress, when the master has issued a message. On return the slave only can rise a request event, after the master has done so. Thus, the accuracy of the slave events for the master depends on the frequency of master events for the slave.

The next section gives an overview on related work. It briefly introduces other synchronization methods and prototyping platforms which use the method to synchronize the processes with the method.

4.2 Related Work

4.2.1 Kahn Process Network

The synchronization model we presented in the previous section was introduced by Kahn [65] and is called a Kahn Process Network (KPN). The KPN model of computation assumes a network of concurrent processes that communicate in a point-to-point fashion over unbounded FIFO channels, using *blocking-read* synchronization primitives. In Figure 4.3 three concurrent processes P1–P3 are represents. The arrows denote how messages are exchanged between the processes, thus they represent the unbounded FIFO channels between the processes. For example, process P3 reads the content of the FIFO coming from P1. If the FIFO is empty, P3 is blocked. As soon as P1 puts a message into the FIFO channel, P3 restarts to process.

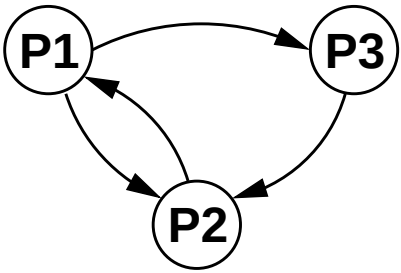

Figure 4.3: Simple Kahn Process Network (KPN)

The model is primary used for signal processing applications. In the streaming data domain data chunks are sent from one process to the next. A process is waiting for the next data chunk to appear at its input FIFO. To achieve cycle-accuracy emulation the data chunks are provided with a time-stamp when it leaves the previous process. The next process updates its own time-base according the incoming time-stamp of the received data chunk.

There are many rapid prototyping platforms which are specialized for the signal processing domain. The underling synchronization method of these platforms is the KPN model [44, 45, 66]. The KPN model cannot be applied to a system where processes cannot be

blocked, like state-dependent processes which again runs on real hardware. Incoming events can change the internal state of such a process. However, as the process cannot be blocked, it can have progress further in time as the time-stamp of the event, thus it would have missed this event already in an earlier state.

4.2.2 Communicating Sequential Processes

In the Communicating Sequential Processes model (CSP) [67], every node or block is a thread-like process that continuously executes unless suspended by a communication. These processes execute concurrently, and the rendez-vous communication protocol dictates that transfer of data only occur when both communicating processes are ready to communicate. Thus, the CSP model is similar to the KPN model with the difference that its processes stop at communication unlike the KPN process, which starts only after communication.

The CSP model is also a data driven model, like the KPN, and therefore, it has the same drawbacks when it is used to emulate state-dependent processes in a hardware prototyping platform. The cycle-accuracy cannot be ensured.

4.2.3 The Logical Clock

Lamport [68] introduced a *logical clock* that can be used to totally order the events in distributed systems. Implementing his method into a hardware prototyping platform to emulate time-dependent data flows would assume that all processes run on physical clocks, which are scaled the same and are synchronized, at least with only a small error.

The scaling of the logical clock to the slowest process is an obvious approach. The RPM [69] prototyping platform makes use of this model. The goal of the platform is to emulate multiprocessor systems with nine identical boards. However, if the platform consists of different components such as any real processor combined with any FPGA boards, it becomes difficult, if not impossible, to scale the logical clock of the system to the slowest process. A CoP which is realized in an FPGA runs only at very slow clock rates, but a real processor cannot

be scaled down to any possible clock rate. Due to the complex clocking architecture of a processor consisting of PLLs or equivalent, the scaling range of the clock is limited [70].

4.3 The Roll-Back Algorithm

In this section we present a new process synchronization method, which is used in the CyCoP platform to cycle-accurately emulate state-dependent CoPs [71]. This method not only manipulates process internal time-bases like in the KPN or CSP model, but it also handles the state of a process at any time. Consequently the new method is able to handle also state-dependent CoPs running different clock domains.

Figure 4.4 shows the fundamental problem of message event synchronization between two processes on CyCoP where the clock domains are scaled differently. Events are either a request or a result message from or to any process. In the example the two time lines represent the progress of two different processes on CyCoP. One process runs on an FPGA and the other on a CPU. The time segments of the two time lines represent the logical clock, i.e., the virtual clock for the entire system. As the process on the FPGA runs slower than it will run on the future architecture, the ticks on the FPGA time line are delayed compared with the ticks on the CPU time line.

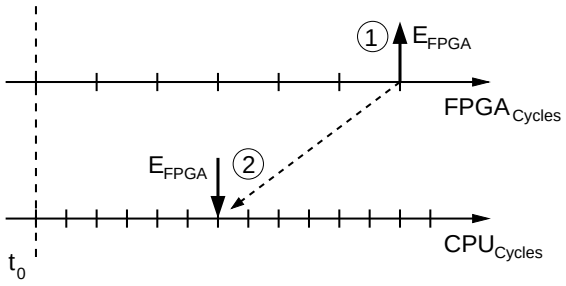

Figure 4.4: Space-time diagram of a CPU and an FPGA process running on differently scaled clock domains

Let us assume that at a specific time t_0 both processes are syn-

chronized, as shown in Figure 4.4. If the FPGA process generates an FPGA event E_{FPGA} ① after n_{FPGA} ticks, this event would have appeared much earlier in time at the CPU process ②.

Therefore a solution to emulate the appearance of the events correctly is shown in Figure 4.5. The processes are started at time t_0 separately and in sequence. There the FPGA process is started first and will run until it generates an event E_{FPGA} ① and then it is held. The CPU process, which has been held at time t_0 up to now, will be started to run exactly n_{FPGA} ticks ②. The idea is that the CPU process will progress after n_{FPGA} ticks using the new input from the FPGA process. A problem occurs if during the n_{FPGA} ticks, the CPU process also generates a new event E_{CPU} ③. However, the FPGA process has progressed already for n_{FPGA} ticks, but without the new input from the CPU process, which may have changed the result of the event E_{FPGA}. And thus a conflict arises.

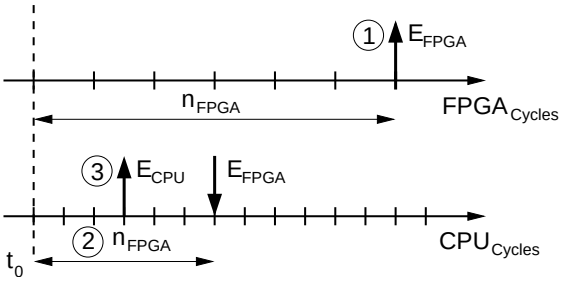

Figure 4.5: Order conflict when processes progress sequentially

The problem can be solved if we can reset the FPGA process to the state at time t_0. As SRAM-based FPGA have the capability to be reconfigured, we are able to reset the FPGA process to the original state if we have stored the context of the FPGA process at time t_0. In Figure 4.6 the FPGA process has been reset, and now runs for n_{CPU} ticks ④. At this new synchronization time, $t_0 + t_{\mathrm{EventCPU}}$, the context of the FPGA process is stored again and the process progresses with the new input of the CPU until it generates a new event E_{FPGA} ⑤.

The Roll-Back Algorithm (RBA) to synchronize processes on CyCoP with differently scaled clock domains is rather simple. At ini-

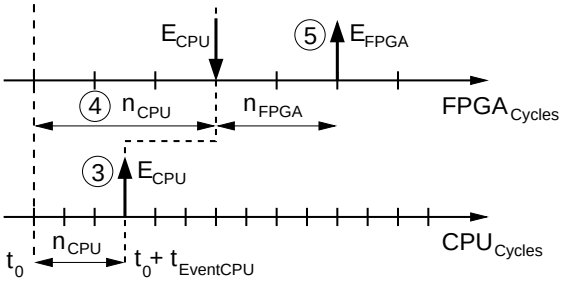

Figure 4.6: Event-ordering algorithm by resetting of the initial state of a process

tialization both processes are synchronized and held at a specific time t_0.

1. Store the context of the FPGA process at a specific synchronization time t_i.
2. Progress the FPGA process until either it generates an event E_{FPGA} or the number of ticks n_{FPGA} reaches a maximum number $n_{\max}$.
3. Hold the FPGA process at this state.
4. Progress the CPU process for n_{FPGA} ticks. If the CPU process generates in the meantime an event E_{CPU} itself, the CPU process is held at this time $t_i + t_{\mathrm{EventCPU}}$. Then
   (a) Restore the context of the FPGA process to reset it to the state at time t_i.
   (b) Progress the FPGA process for n_{CPU} ticks.
   (c) Use the new input from the CPU process for the further progress.
   (d) Repeat the algorithm starting at step number 1.
5. Take the new input from the FPGA process for the further progress of the CPU process.

6. Repeat the algorithm at step number 1.

This algorithm can be adapted to more than only one FPGA process. In this case the context of each FPGA process is stored at time t_i. Then all FPGA processes run in parallel until they generate an event or reach the maximum number of ticks $n_{\max}$. All of the numbers of ticks used are then compared with each other, to find the minimum number $n_{\min}$ of ticks. The FPGA processes that have a higher number of ticks than the minimum are reset to the state at time t_i and restarted for $n_{\min}$ ticks. Also, the CPU process is restarted for $n_{\min}$ ticks. The remainder of the algorithm is the same as listed above.

4.4 Analysis

4.4.1 Emulation Performance

During emulation, the latency produced by the RBA is hidden from the CPU application. However, to estimate the performance of the CyCoP platform itself, this latency must be known. The term *time* or *latency* within this analysis determines a *number of clock cycles of the CPU* if not otherwise specified.

The total runtime of the emulation T_{Em} can be expressed with the following equation:

$$T_{\mathrm{Em}} = C_{\mathrm{Setup}} + T_{\mathrm{App}} + C_{\mathrm{Alg}} \tag{4.1}$$

The sum includes the constant setup time C_{Setup} of the emulation, the runtime of the application on the target system T_{App} and the algorithm latency C_{Alg}. In the remainder of this section we analyze the algorithm latency and show that there is an optimum of the maximum number of exploration cycles $n_{\max}$.

Figure 4.7 illustrates again in a timing diagram how the algorithm latency is embedded in the actual application runtime. The diagram shows the latency of a CoP event (Ev_{CoP}) and an application event (Ev_{App}) during the emulation.

In the following subsections we separately analyze the latencies of the application event and the CoP event for the emulation.

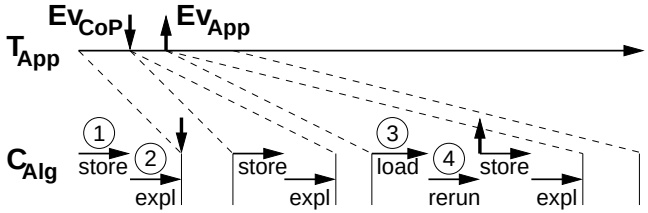

Figure 4.7: Timing Diagram of the Roll-Back Algorithm

Coprocessor Event

The RBA starts with the progress of the FPGA to explore CoP for any event. First, the context is stored to be able to roll-back from the exploration phase (refer to ① in Figure 4.7). The time needed for this process, C_{Store}, depends on the data volume N_{Data} to be saved and the bandwidth B_{SRAM} of the SRAM:

$$C_{\mathrm{Store}} = f_{\mathrm{CPU}} \cdot \left\lceil \frac{N_{\mathrm{Data}}}{B_{\mathrm{SRAM}}} \right\rceil \tag{4.2}$$

The remaining latency of the exploration phase (refer to ② in Figure 4.7) depend on whether a CoP event appears. This probability is described by the function p_{CoP}. At an appearing CoP event, the exploration phase is stopped and the CPU is run up $\mathrm{E}[T_{\mathrm{EvCoP}}]$ cycles to the CoP event time and the MW generates a software interrupt, which costs C_{emInt} cycles.

If new CoP event appears in the exploration phase ②, the maximum cycles $n_{\max}$ are progressed in the algorithm and the MW initiates another step with latency C_{Int}. The total latency for a CoP event can be expressed as follows:

$$\begin{aligned} C_{\mathrm{Step}} \quad = \quad & C_{\mathrm{Store}} + p_{\mathrm{CoP}} \left(\frac{f_{\mathrm{CPU}}}{f_{\mathrm{FPGA}}} \mathrm{E}[T_{\mathrm{EvCoP}}] + C_{\mathrm{emInt}} \right) + \\ & (1 - p_{\mathrm{CoP}}) \left(\frac{f_{\mathrm{CPU}}}{f_{\mathrm{FPGA}}} n_{\max} + C_{\mathrm{Int}} \right) \end{aligned} \tag{4.3}$$

The frequency fraction of the CPU and the FPGA is multiplied to the cycle numbers to normalize the equation to a single clock domain.

The Equation (4.3) contains the expectation $\mathrm{E}[T_{\mathrm{EvCoP}}]$. It specifies where in the exploration phase the event is expected. This value has also an influence on the expected duration of a step, which can be computed the following way:

$$\mathrm{E}[T_{\mathrm{Step}}] = p_{\mathrm{CoP}}\mathrm{E}[T_{\mathrm{EvCoP}}] + (1 - p_{\mathrm{CoP}})n_{\max} \tag{4.4}$$

Replacing the probability p_{CoP} with its probability density function $p(t)$ following can be expressed:

$$p_{\mathrm{CoP}} = \int_0^{n_{\max}} p(t)\,dt \tag{4.5}$$

$$\mathrm{E}[T_{\mathrm{EvCoP}}] = \frac{1}{p_{\mathrm{CoP}}}\int_0^{n_{\max}} t \cdot p(t)\,dt \tag{4.6}$$

Using a constant probability p for the CoP event and assuming that $\frac{1}{p} > n_{\max}$ sets the following parameters for Equation (4.13):

$$p(t) = \begin{cases} p & \text{for} \quad 0 \leq t < \frac{1}{p} \\ 0 & \text{else} \end{cases} \tag{4.7}$$

$$p_{\mathrm{CoP}} = p \cdot n_{\max} \tag{4.8}$$

$$\mathrm{E}[T_{\mathrm{EvCoP}}] = \frac{1}{2}n_{\max} \tag{4.9}$$

Application Event

An application event appears in the phase when the algorithm progress the application for exploration time dictated by the CoP exploration phase. The FPGA is therefore farther advanced in time as the CPU. If now an application event appears, the CoP has to be reset to the last saved context before the application event and run up to it. The latency for the application events can be expressed as follows:

$$C_{\mathrm{App}} = C_{\mathrm{Int}} + C_{\mathrm{Load}} + \frac{f_{\mathrm{CPU}}}{f_{\mathrm{FPGA}}}\mathrm{E}[T_{\mathrm{EvApp}}] \tag{4.10}$$

C_{Int} introduces the latency of the MW interrupting the application process and executing the event instruction. The context of the CoP

must be loaded by the Wrapper into the CoP (refer to ① in Figure 4.7). The time needed for this process is expressed by C_{Load}:

$$C_{\text{Load}} = C_{\text{Store}} \tag{4.11}$$

The last term in Equation (4.10) is the time needed to run the CoP from the last saved context time to the current CPU time (refer to ② in Figure 4.7). $\text{E}[T_{\text{EvApp}}]$ represents the expectation of n_{CPU}. It is multiplied with the frequency fraction of the CPU and the FPGA to normalize the equation to a single clock domain.

The expected offset $\text{E}[T_{\text{EvApp}}]$ depends on the event distribution over the application. For an equal distribution of the application events, this will result in:

$$\text{E}[T_{\text{EvApp}}] = \frac{1}{2} n_{\max} \tag{4.12}$$

Total Emulation Latency

Now as we have analyzed the individual latencies for an application event and a CoP event, we are able to put all together in a single equation to express the total algorithm latency:

$$C_{\text{Alg}} = T_{\text{App}} \left(p_{\text{App}} C_{\text{App}} + \frac{C_{\text{Step}}}{\text{E}[T_{\text{Step}}]} \right) \tag{4.13}$$

It consists of the application event latency C_{App} and the per iteration step latency C_{Step}. These latencies are multiplied with the probability that events appear during the emulation. The processor events are modeled with the probability p_{App}. $\text{E}[T_{\text{Step}}]$ represents the expected number of cycles each iteration step will take.

Replacing all parameters in Equation (4.13) results in:

$$\begin{aligned} \frac{C_{\text{Alg}}}{T_{\text{App}}} \quad = \quad & p_{\text{App}} \left(C_{\text{Int}} + C_{\text{Load}} + \frac{f_{\text{CPU}}}{2 f_{\text{FPGA}}} n_{\max} \right) + \\ & \frac{C_{\text{Store}} + p \cdot n_{\max} \left(\frac{f_{\text{CPU}}}{2 f_{\text{FPGA}}} n_{\max} + C_{\text{emInt}} \right)}{n_{\max} - \frac{p}{2} n_{\max}^2} + \\ & \frac{1 - p \cdot n_{\max}}{n_{\max} - \frac{p}{2} n_{\max}^2} \left(\frac{f_{\text{CPU}}}{f_{\text{FPGA}}} n_{\max} + C_{\text{Int}} \right) \end{aligned} \tag{4.14}$$

The resulted fraction in the Equation (4.14) represents the *emulation overhead factor* which is the ration between original application run time and the algorithm latency. Choosing the right exploration time $n_{\max}$ for the RBA is an optimization problem of Equation (4.14) in order to minimize the emulation overhead. The exploration time determines how long the CoP can run in advance until it is stopped and the CPU allowed catching up. Choosing a long exploration time generates fewer exploration phases and reduces the latency for them. On the other side, when a CPU accesses the CoP it must be set back to its last saved context. If the exploration time is too long, the latency associated to setting back and running it up to the CPU time base will grow much larger. Additionally, more unnecessary exploration phases are conducted.

An optimal exploration time can be calculated analytically from Equation (4.14). The solution of the derivation of the following equation gives the optimum:

$$\frac{\partial C_{\text{Alg}}}{\partial n_{\max}} = 0 \tag{4.15}$$

The derivation of Equation (4.14) results in a quadratic formula and has therefore an analytical solution. However, the solution of the quadratic formula does not reduce the complexity and we are not interested in exact solutions but rather in good estimations. Therefore, a plot of Equation (4.14) should be sufficient enough.

Figure 4.8 shows the plot of the emulation overhead factor (Equation 4.14) for different probabilities p with varying maximum exploration time $n_{\max}$. It clearly shows that the length of the exploration time has an optimum. There is a trade-off between the exploration time and the number of context handling. If the probability of an event is high, it is worth choosing a higher exploration time to increase the probability that an event happens during the exploration phase. Thus, the number of context store should be reduced to a minimum. However, if the exploration time is too long, the number of context loads increases. And this increases again the total emulation latency.

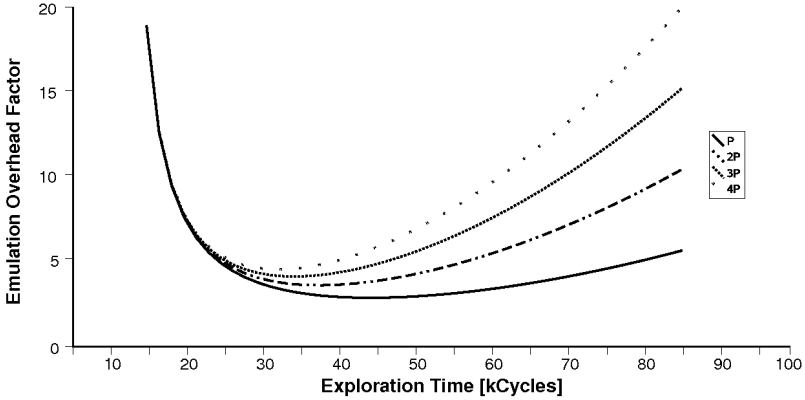

Figure 4.8: Plots of Equation (4.14) for constant probabilities p – $4p$ with varying $n_{\max}$.

4.4.2 Emulation Overhead

The key function of the RBA is the capability to load and store the CoP context. It allows the emulation system to set the CoP state back to an earlier time as needed by the algorithm.

The context of the CoP could be retrieved using a configuration interface of the FPGA, as used in many projects [72, 73]. These interfaces allow one to write and read the configuration of parts of or the entire FPGA. The *configuration stream* obtained from such a configuration interface includes information about the memory element contents as well as the configuration of the combinatorial logic of the FPGA. From this information, the memory element contents can be retrieved and a new configuration stream can be formed to restore their values. However, the bandwidth of these interfaces is limited and considerable effort is needed to filter the interesting data out of the stream.

Another possibility for storing and loading the CoP's context is to use a scan path [74]. With a scan path all memory elements are interconnected at a specific mode, such that they form one large shift register, or *scan chain*. Figure 4.9 shows three such chains, each interconnecting one row of memory elements. The little clouds symbolize

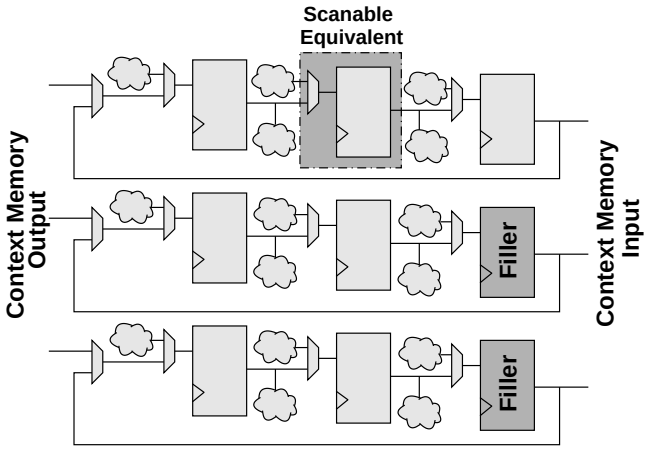

Figure 4.9: Scan Chains with Loop Backs

the connections to the logic during normal operation. Data presented on the input of this chain can then be shifted consecutively from one memory element to the other. The data stream appearing at the output includes the contents of all memory elements in the circuit. As the state of sequential logic is represented by the contents of its memory elements, the data stream through the scan chain contains the complete CoP context.

The context of a CoP may not only consist of the content of its registers, but also of the content of potential RAM modules. Therefore, these RAM modules have to be included into the scan path. Figure 4.10 shows an example scan-wrapper for a dual-port RAM module which is inserted by the script. The scan-wrapper consists of a counter (CNT) which addresses every memory address line sequentially. The output of the RAM is then latched into a Register of the length of the RAM data width. These output register again are part of the scan path. After every memory read access the content of the output registers are scanned out.

By using multiple scan chains in the CoP logic, the data can be accessed in parallel and the context can be exchanged very efficiently. In Equation 4.2 we already consider to use the full bandwidth B_{SRAM} of

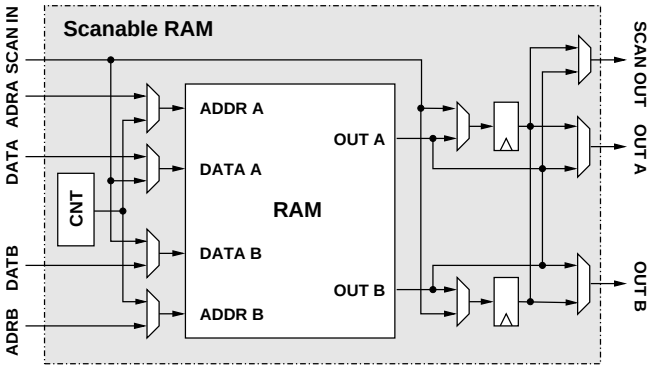

Figure 4.10: Scanable Memory

the SRAM. Thus, the whole port width to the SRAM is used. If only a single scan chain is implemented, then only a fraction of the SRAM bandwidth utilized. Considering that one bit at the clock frequency of the FPGA can be stored, the equation becomes the following:

$$C_{\text{Store}} = f_{\text{CPU}} \cdot N_{\text{Data}} \tag{4.16}$$

Thus, by using the full bandwidth the store penalty can be reduced by almost a factor of the number of the SRAM port width. Therefore, not only a single scan chain is implemented into a CoP core, but a number of scan chains, that is equivalent to the SRAM port width (see Figure 4.9). As the number of scan elements is normally not equal to the multiple of the memory data width, additional registers are inserted in the chains, to create equal chain length. The darker boxes in Figure 4.9 are the registers inserted to generate scan chains of equal length.

The ends of the scan chains are connected to the chain inputs such that the chains form loops. During context storage, the chain end data is written to the memory and at the same time loops back to the scan chain inputs. After a complete context save, all memory elements contain again their initial value, and the CoP can resume its operation without further interference. On a context load, the scan chain inputs are fed from the memory output, whereas the scan chain outputs are not used.

Some experiments show [75] that a D flip-flop instrumented for scan in FPGAs is around 80% larger in area.

4.4.3 Emulation Accuracy

The RBA theoretically allows a 100% accurate timing behavior in the emulation of a CPU with its according CoP. With the possibility to set the state of a CoP to any time the accuracy is assured. The CPU application is interrupted either when the application accesses the CoP or when the CoP notifies the application in any manner—either with an interrupt or a status flag.

For the application interruption we use two methods. One is the MW call, which represents the access of the application on the CoP, and the other method is the use of a timer interrupt. The accuracy of the application emulation behavior depends therefore on these two interrupts and how accurate they are.

The MW call belongs to the category of the *precise exceptions*. A pipelined processor is said to have a precise exceptions, if the pipeline can be stopped so that the instructions just before the interrupting instruction are completed and those after it can be restarted from scratch [47]. The exception is precise in the aspect of the instruction flow; however, the cycle-accuracy of the exception depends on the number of instructions which are on the fly to be executed and their latency.

For the other application interrupt we analyze the behavior of the timer interrupt. This interrupt is in contrast to the MW call cycle-accurate in the appearance of the interrupt. The error of this interrupt comes from the uncertainty when it is started. The MW starts this timer, but the MW has to take into account that the application has to be relaunched again. Thus, the timer must start exactly at the relaunch of the application. This relaunch, however, does not take a constant time. Therefore, a small error is introduced in the start of the timer interrupt. Furthermore, certain timer may not offer the accuracy of single cycles, but of higher granularity.

Putting all our findings together into a single equation, the following can be written:

$$E_{\text{total}} = n_{\text{MWcall}} \cdot IPC \cdot n_{\text{Pipe}} + n_{\text{Timer}} \left(e_{\text{Start}} + e_{\text{Gran}} \right) \qquad (4.17)$$

The error introduced by the instruction precise behavior of this exception is expressed by the product of the number of MW calls (n_{MWcall}), of the typical instructions per cycle (IPC) of the CPU, and of the pipeline depth (n_{Pipe}). The second product in the equation expresses the uncertainty of the start of a timer (e_{Start}) and the granularity of the timer (e_{Gran}) multiplied by the number of timer interrupts (n_{Timer}).

The total cycle-error E_{total} finds its minimum in the case of a minimal n_{MWcall}. This is the case of a maximum exploration time n_{max} of the RBA. If $n_{\mathrm{max}} \rightarrow \infty$, then no intermediate steps are made by the RBA and then n_{MWcall} becomes minimum. This conflicts therefore with the performance of the emulation.

4.5 Limitations

The RBA relies on the fact that the context of a CoP is the sum of all its memory and register content. This implies that the CoP does not get any external data inputs others than the one from the CPU. Of course such an external input could also be buffered together with a timestamp of the arrival time. The buffer content would be cleared in the next exploration phase of the algorithm.

If the CoP issues data to a process external from the CyCoP environment, the output of the CoP must be kept in a buffer, like the one for the external input data, and can be released only at the next exploration phase of the RBA. The external output thus is not as cycle-accurate any more. Its cycle-accuracy depends on the expected exploration time of the emulation.

Theoretically also the context of the CPU-like CoP could be stored and restored so that the RBA can synchronize several different processors. The problem here is that this usually is very memory- and time-intensive, and therefore not very efficient.

4.6 RBA Implementation

The goal of the implementation is to provide CyCoP the capability to cycle-accurately emulate any type of loosely-coupled CoPs. A CoP,

which will reside on the same die as the processor on the target chip, will communicate over an on-chip interconnect system with the processor. During the emulation, messages cannot simply be exchanged between the processor and CoP over this on-chip interconnect system, but must instead pass a series of additional devices of the prototyping platform. This emulation latency must be hidden for an accurate performance measurement.

The execution of the algorithm is mainly controlled by a Finite-State Machine (FSM) shown in Figure 4.11. The synchronization protocol used in the emulation follows from the explanation of how the Wrapper FSM works.

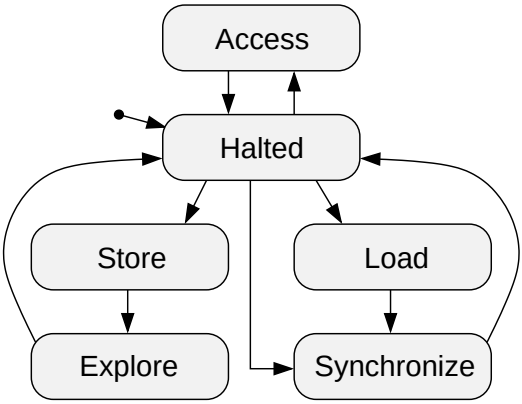

Figure 4.11: Finite-State Machine

Initially, the FSM is in its *halted* state, where it waits for events from the CPU process. While in this state, the clock for the CoP is turned off. As soon as a CPU process wants to access the CoP, the MW stops the CPU process. The MW then first sends a timestamp with the current processor time base, n_{CPU}, to the Wrapper. The Wrapper compares its local time, n_{FPGA}, with the received timestamp. If the local time is larger than the processor time, meaning that the CoP has progressed further than the CPU process, the last saved CoP context is restored in the *load* state. Then the CoP runs in the *synchronize* state until it reaches the CPU time n_{CPU}

If the local time n_{FPGA} is smaller than the received timestamp, the *load* state is omitted, and the CoP is directly progressed until it reaches the CPU time n_{CPU}. While the FSM is in the *synchronize* state, any events generated by the CoP are ignored. After the FSM is synchronized with the processor time base, it enters again the *halted* state.

The MW now passes the actual CPU message to the CoP, and the FSM enters into the *access* state and enables the CoP clock. It returns to *halted* upon completion of the data exchange. The processes are then still considered to be synchronized.

The MW then orders the FSM to enter an exploration phase, i.e., the Wrapper runs the CoP until the later generates an event or it reaches the maximum number of ticks $n_{\max}$. As the Wrapper must be able to reset the CoP to the current state, it needs to save the current context before the FSM is moving to the *explore* state. When the CoP has either reached $n_{\max}$ or generates an event, the FSM returns to the *halted* state. After this exploration phase, the MW starts the CPU process to progress until the CPU reaches the CoP's local time. If the CPU process does not generate an event, i.e., if it wants to access the CoP during this time, a new exploration phase is initiated. Otherwise the procedure as described above is repeated.

4.6.1 Wrapper

The Wrapper state machine described in above is implemented in the *WrapperControl* module (see Figure 2.3). It controls all other modules and manages the time base for the CoP. This time base is incremented at every clock cycle of the *access*, *explore*, and *synchronize* states. A copy of the time base is saved together with the context at the *store* state and written back when a context load occurs. In addition, the *WrapperControl* module reacts on events generated by the CoP.

4.6.2 Middleware

There are two possibilities for the communication between processor and CoP. A software application uses load/store instructions to access the memory or the address-mapped registers of the CoP over the on-chip interconnect system. The CoP notifies the application of some

event by raising either an interrupt signal or a status flag. For the CyCoP this communication scheme has to be emulated by the MW. The MW ensures that the behavior of these communication methods is identical to that of the target system during the emulation. This also implies control of the time base of the processor, which must be synchronized at data exchange. The time base is used for performance measurements and should therefore always reflect the execution time as it would appear on the target system.

The resulting tasks for the MW are the following:

1. All load and store instructions that access the CoP memory or registers must be intercepted and synchronized with the recipient.

2. Interrupts or status flags set by the CoP must appear in the correct moment and may also interrupt the application flow.

3. The time base of the processor should always reflect the execution time on the target system when the application is running.

For the first task all load/store access of the software application to the CoP region must force the application to branch into the MW part. One possibility to intercept load/store instructions that access the CoP is to actually replace them with software interrupts equivalent to the tightly-coupled scheme. On encountering such an instruction, the processor enters into a specific interrupt handler which will be replaced by the MW. For example, the PowerPC ISA contains the *twi* instruction, i.e., a breakpoint, which provides a 23-bit immediate field. These bits can be used to encapsulate the original instruction information. The MW extracts this information and it then executes the access to the CoP in the FPGA, updates the time base, and returns to the application, as the RBA dictates.

The drawback of this approach is that it requires the application code to be manipulated. Furthermore, it uses the same 23-bit immediate to encode the load/store instructions as well as the new instructions of the tightly-coupled CoP.

The debug facilities of the PowerPC 440GP [70] offers the feature to monitor all memory accesses and to interrupt on specific address ranges, i.e., a so-called *Data Address Compare* (DAC) event. Using

this feature, the application process execution is interrupted before the actual instruction and the MW is entered. The MW then executes the original instruction in the same manner as when using a *twi* instruction.

The DAC event is a very valuable feature for the implementation of the RBA in the PowerPC. The drawback of the use of this feature is, that the debug facility introduces on every load/store instruction an additional one extra cycle of latency. For a typical application up to 36% [47] of all instructions are load/store instructions. Thus, the introduced latency of the emulation is quite striking. To get rid of the extra latency and therefore, to achieve a higher cycle-accuracy, the only solution is to not make use of the DAC event. Instead one should replace all CoP related load/store instructions with a trap instruction (refer to Section 2.2).

By using the DAC event as a MW call the Equation 4.17 has to be corrected. $n_{\mathrm{l/s}}$ represents the additional cycle the MW call introduces for every load/store instruction within the application. The error introduced by the instruction precise behavior of this exception is expressed by the product of the number of DAC events (n_{DAC}), of the typical IPC of the CPU, and of the pipeline depth (n_{Pipe}). Putting all our new findings together into a single equation, the following can be written:

$$E_{\mathrm{total}} = n_{\mathrm{l/s}} + n_{\mathrm{DAC}} \cdot IPC \cdot n_{\mathrm{Pipe}} + n_{\mathrm{Timer}} \left(e_{\mathrm{Start}} + e_{\mathrm{Gran}} \right) \quad (4.18)$$

The second task of the MW is to generate interrupts for the software application at the same moments as the CoP would do in the target system. To determine the time of such an event, the MW lets the CoP explore the next n_{max} cycles to see whether it generates an event during this time as dictated by the RBA. It then lets the processor synchronize with the CoP time. If the exploration phase stopped on an event generated by the CoP, the MW performs the appropriate actions such that the application receives the CoP event. One possibility is that it starts the interrupt handler that the application provided for what it assumed to be the CoP interrupt. In this way the MW fulfills its second task. If the CoP did not generate an event during its last exploration phase, the MW continues with the next CoP exploration phase.

The processor can be forced to enter the MW after a certain number of application cycles using a *General Purpose Timer* (GPT) timer of the processor. The GPT generates an interrupt after a previously configured number of cycles have elapsed. The MW intercepts this interrupt and performs the actions as described above. If the CoP did not generate an event, the GPT interrupt is hidden, and the application process does not notice that it has been interrupted.

The third task of the MW is to emulate the correct advancing of the time base on the processor. Therefore it must subtract the time used for the execution of the MW code and the communication overhead of the emulation system. The MW then adds communication delays of the target system to the time base as needed.

On the PowerPC 440GP this task is simplified as it allows stopping the time base on the interrupts used for the emulation. The MW then only has to add the appropriate delays to the time base before continuing the application process.

4.6.3 Design Flow

The emulation of loosely-coupled CoPs on CyCoP impacts the design flow as described in Section 2.4. Due to the powerful debug facility of the PowerPC 440GP, however, only the hardware flow is influenced.

FPGA Synthesis

Figure 2.5 of Section 2.4 shows that the goal for the CyCoP FPGA synthesis is to keep it simple as possible. The scan path through the CoP, which is needed by the RBA, is inserted directly into the net list of the CoP. This gives us the opportunity to not only emulate HDL-designs but also FPGA net lists of custom CoPs.

A Perl script parses the CoP net list and replaces all registers found with a *scanable equivalent* (shaded in Figure 4.9). Such a scanable equivalent contains a copy of the replaced module and a multiplexer which allows the selection between a normal input and the input during scan mode. Furthermore, the scanable equivalent contains two more input ports, namely, one for the scan-enable signal, and one for the preceding register element of the scan chain. All memory modules are replaced with *scanable RAMs* (Figure 4.10).

All scan inputs and memory element outputs of the CoP are then connected to the *ScanHandler* module of the Wrapper, and a complete synthesis including the scanable CoP and all Wrapper modules is performed.

4.7 Conclusions

In this chapter we showed how the CyCoP platform has to be extended with the RBA to permit a cycle-accurate emulation of state-dependent CoPs. In contrast to existing solutions, the RBA offers the possibility to emulate state-dependent CoPs even on differently scaled clock domains within the platform. This avoids having to scale all system elements accordingly and furthermore provides the possibility to replace the interconnect system with a board-level system.

The emulation performance and the cycle-accuracy of the implementation in the real hardware environment highly depend on the number of message events between the CPU and its CoP. The higher the number of exchanged message events the lower the performance and the accuracy of the emulation. To optimize the numbers within an emulation run one has to decide either for a better performance or for a higher accuracy. A formula to make an estimation for both numbers is given.

The RBA relies on the fact the complete context of a CoP can be stored and reproduced in any point in time. Therefore, for the emulation of external input and output special actions have to be taken. The size of the context theoretically is not limited for the algorithm, however, in real environment it is limited to the available memory and the performance of the emulation.

Chapter 5

Experimental Results

To demonstrate the feasibility of the CyCoP concept, the platform is implemented in real hardware. Two case studies, one for a tightly-coupled coprocessor (CoP) and one for a loosely-coupled CoP, illustrate our concept and the quality of the results.

In this chapter, we first examine the performance and the accuracy of an example implementation of a tightly-coupled CoP in Section 5.1. We extend the ISA of the PowerPC 440GP with single-precision floating-point instructions. We assess the quality of the result by comparing it with a PowerPC 440EP chip which consists already of a hardware floating-point CoP.

In Section 5.2 we examine the performance of CyCoP on loosely-coupled CoPs. As an example implementation we choose a large timer manager CoP which perfectly fulfills the requirement of a state-dependent CoP. The last Section 5.3 summarizes our findings.

5.1 Tightly-Coupled Coprocessor Results

We extend the basic PowerPC 440GP [14] core with a single-precision Floating-Point Unit (FPU). This comprises the advantage that all standard compilers do support floating-point source code, so we can build our prototype upon standard software design flows. Furthermore, there are many test applications and benchmarks freely avail-

able which make use of the floating-point operation. Thus, we are able to operate with a real tool-chain and real applications.

We first introduce the implemented FPU with all the available instructions. Then we describe the work load with which we tested and measured the FPU. The results are presented at the end of this section.

5.1.1 Floating Point Unit

To demonstrate the operativeness of the CyCoP platform, the publicly available single-precision FPU from the OpenCores.org project [76] is used. This FPU is configured for the Virtex-II Pro FPGA of the Amirix AP130 platform FPGA development board [53]. We modified and extended the FPU to fit our needs as CoP core. Figure 5.1 shows the block diagram of the CoP core.

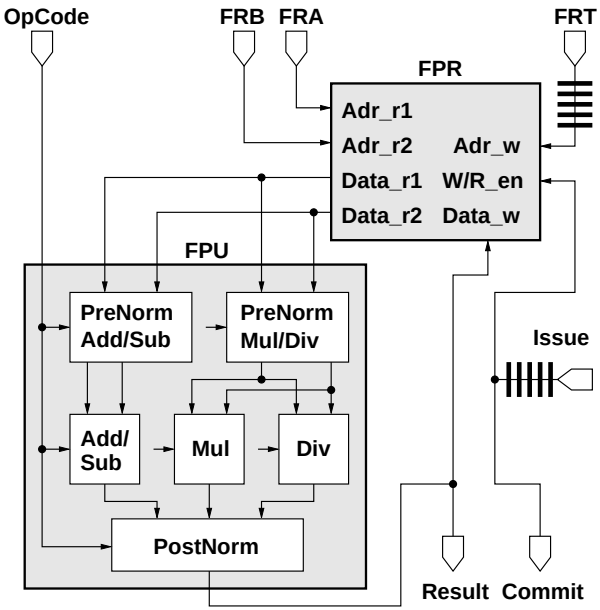

Figure 5.1: Single-Precision Floating-Point Unit

We added 32 Floating-Point Registers (FPR) to the FPU to provide a load-store architecture, i.e., a register-register architecture. The FPR file is realized as a three port FPGA RAM. Two ports are read ports and one port is the write port. The CoP core consists of five-stage pipeline. Thus, the address and write enable signal for the write port are simply the pipelined input signals *issue* and *FRT*, with *FRT* being the address of the result register of the CoP instruction. The input signals *FRA* and *FRB* are the addresses for the operand registers coming from the CoP instruction.

The operands coming from the FPR are fed together with the *OpCode* from the instruction into the FPU where the result of the operation is computed. The CoP core is able to execute the following floating-point instructions:

`fadd`	`FRT,FRA,FRB`	$FRT = FRA + FRB$
`fsub`	`FRT,FRA,FRB`	$FRT = FRA - FRB$
`fmul`	`FRT,FRA,FRB`	$FRT = FRA \cdot FRB$
`fdiv`	`FRT,FRA,FRB`	$FRT = FRA \div FRB$
`fcfi`	`FRT,FRB`	Convert from Integer
`fcti`	`FRT,FRB`	Convert to Integer
`fcmp`	`FRA,FRB`	Compare
`fabs`	`FRT,FRB`	Absolute Value
`fneg`	`FRT,FRB`	Negate
`lfs`	`FRT,IM`	Load Immediate
`stfs`	`RT,FRB`	Store to GPR
`fmr`	`FRT,FRB`	Move Register

The conversion functions of the FPU are special and will play an important role when we run the test applications. Detailed information will follow in this section.

The CoP core has a five-stage pipeline and every instruction except the division takes equally long for the execution. The division takes nine clock cycles to compute. As the CoP is pipelined data hazards are detected and hold the instruction from being executed until the result is committed. The result is by-passed one stage before the final stage to shorten the stall time in the CoP.

5.1.2 Work Load

For the test application software, the benchmark suite MiBench [77] is used as it targets the embedded-processor market, for which the PowerPC 440GP is designed. It provides numerous test programs from areas such as,

- consumer,
- office,
- automotive/industrial,
- network,
- security, and
- telecommunication.

Given that not all of these applications make use floating-point operations, only a selection of applications are prepared for CyCoP:

> *basicmath*: The basic math test performs simple mathematical calculations that often don't have dedicated hardware support in embedded processors. For example, cubic function solving, integer square root and angle conversions from degrees to radians are all necessary calculations for calculating road speed or other vector values. The input data is a fixed set of constants.
>
> *FFT/IFFT*: This benchmark performs a Fast Fourier Transform (FFT) and its inverse transform (IFFT) on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.
>
> *lame*: Lame is a GPL'ed MP3 encoder that supports constant, average and variable bit-rate encoding. It uses small and large wave files for its data inputs which are converted into the MP3 format.

susan: Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Image (MRI) of the brain. It is typical of a real world program that would be employed for a vision based quality assurance application. It can smooth an image and has adjustments for thresholds, brightness, and spatial control. Two pictures, with different size and complexity, are processed with either the corner or the edge detection function from Susan. The small input data is a black and white image of a rectangle while the large input data is a complex picture.

At this point it shall be mentioned that most of the original benchmark applications make use of Operating System (OS) library functions such as memory allocation and file-system access. However, for the measurements of the experiments on the CyCoP no OS is installed. The benchmark applications run in a standalone mode and all the OS library functions are statically compiled into the applications' binary and therefore, some modification in the original code were taken.

Furthermore, all applications are compiled with the GNU Compiler Collection (GCC) [15] using different compiler optimization levels (O0–O3). This allows us to get different instruction-stream behavior out of a single application. The optimization levels contain the following strategies:

-O0 : At this optimization level GCC does not perform any optimization and compiles the source code in the most straightforward way possible. Each command in the source code is converted directly to the corresponding instructions in the executable file, without rearrangement.

-O1 : This level turns on the most common forms of optimization that do not require any speed-space tradeoffs. With this option the resulting executables should be smaller and faster than with -O0.

-O2 : This option turns on further optimizations, in addition to those used by -O1. These additional optimizations include instruction scheduling. Only optimizations that do not require any speed-space tradeoffs are used, so the executable should not increase in

size. This option is generally the best choice for deployment of a program, because it provides maximum optimization without increasing the executable size.

-O3 : This option turns on more expensive optimizations, such as function in-lining, in addition to all the optimizations of the lower levels -O2 and -O1. The -O3 optimization level may increase the speed of the resulting executable, but can also increase its size. Under some circumstances where these optimizations are not favorable, this option might actually make a program slower.

Due to the redistribution of the instructions because of the different optimization strategies the number of MW calls of the CyCoP vary, thus we expect also a change in the emulation time. However, with the optimizations we still expect the platform to provide us an increase of the application performance due to the compiler optimizations. To get a reference on how the compiler optimizations effect the application performance we run the same application again on a PowerPC440 with an integrated FPU.

5.1.3 Emulation Performance

With the term emulation performance we address the performance of the CyCoP platform itself. The performance metric is the ratio between the benchmark program execution time and the number of executed instructions. This provides us the instruction rate (MIPS) for the individual benchmark applications of the CyCoP.

Figure 5.2 shows the numbers of the instruction rates achieved for the individual MiBench applications. For this diagram the application are compiled with the optimization level -O0 of the GCC compiler. Thus, each command in the source code is converted directly to the corresponding instructions in the executable file, without rearrangement.

The performance reaches from 0.6 MIPS for the FFT application up to 21.2 MIPS for the Susan corner calculation of the large picture. This big difference in performance between the two test applications can be explained with the ratio between the number of MW calls and the number of integer instructions. The right ordinate gives the number of this ratio for the individual test applications. The number

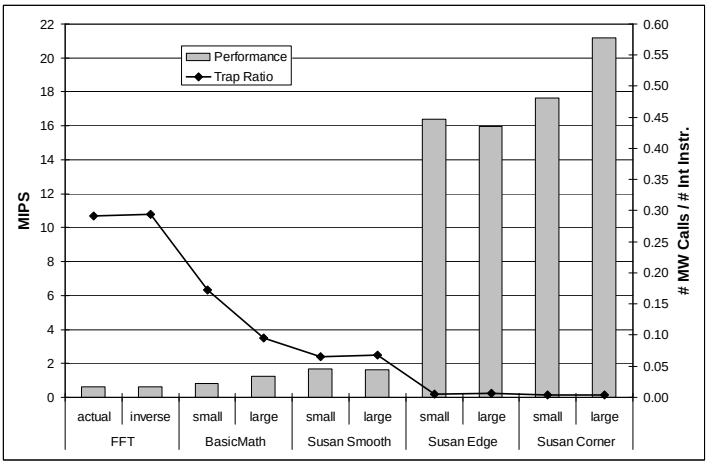

Figure 5.2: Performance achieved for MiBench with Optimization -O0

for the FFT function is at 0.30, thus almost every third executed instruction is a MW call which reduces the performance. On the other hand, for the Susan corner calculation the ratio is at 0.004 which results in a high performance.

At this point one must say that the Susan corner application runs 52 times longer and it executes 74 times more instructions than the FFT function. In point of view of a meaningful result the Susan corner application is to favor. The test application with most executed instructions is the Susan smooth calculation for the large picture. It executes 1632 times more instructions than the FFT function and performs with 1.7 MIPS at a MW call ratio of 0.07.

Figure 5.3 and Figure 5.4 show the same diagram as Figure 5.2, but in these diagrams the test application are compiled with the optimization level -O2 and -O3 of the GCC compiler. These optimization levels of the GCC compiler tries to mix floating point sequences instruction with integer instructions which reduces the data hazards in the pipelined architecture and makes better use of the superscalar ar-

chitecture, as the FPU is a separate execution pipe. Thus, on a real processor this optimization results in a better performance.

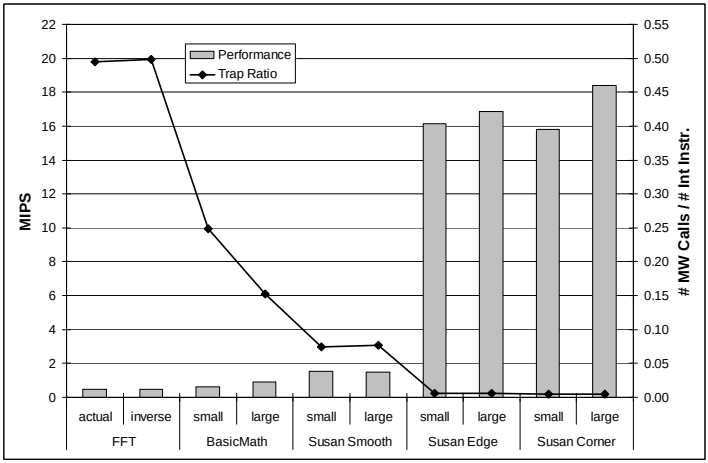

Figure 5.3: Performance achieved for MiBench with Optimization -O2

However, for the CyCoP platform this sort of optimization results into a higher MW call ratio. For the FFT function the ratio reaches up to 0.5, which basically means that every second executed instruction by the processor itself is a MW call. The performance decreases to 0.5 MIPS. The same effect can be observed at the Susan corner application. Its performance reaches with -O2 18.4 MIPS and the MW call ratio increases to 0.005. An increase of 33% in the MW call ratio results in a 20% performance loss. We can conclude that the emulation overhead is significant for our example hardware setup. In the following we investigate the emulation overhead in more detail.

5.1.4 Emulation Overhead

In Chapter 3 we theoretically analyzed the total emulation overhead depending on the number of MW calls and CoP instructions. Ta-

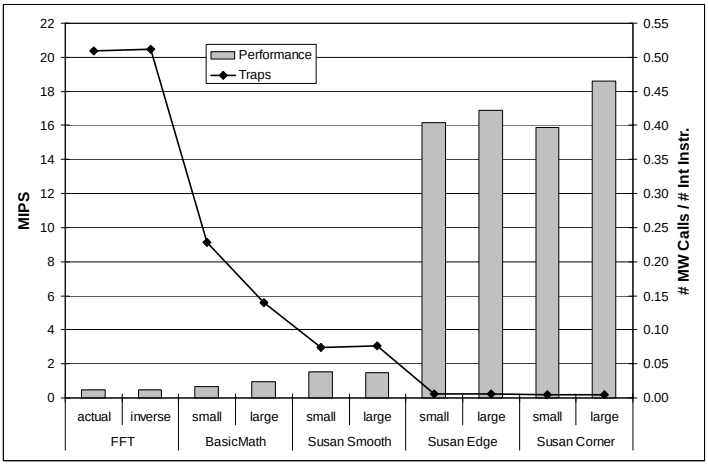

Figure 5.4: Achieved Performance for MiBench with Optimization -O3

ble 5.1 lists the latencies for the individual emulated instructions. The latency is measured in CPU cycles, with the PowerPC 440GP running at 400 MHz.

We observe that the most emulated instructions have about the same latency except the *store* and the *compare* instruction. The reason for this is that, in contrast to the other instructions, these two instructions provide results to be read by the MW. Thus, the results of the other operations are kept within the CoP registers and the results are not sent back to the MW over the PCI bus. This saves about additional 700 cycles of latency, which are a significant portion of the overall latency. In the theoretical estimation we estimated that all CoP instructions provide a result to be read back by the MW. As we can see by doing this only selectively we avoid unneeded overhead.

Form the instruction latencies we can conclude that the PCI bus access makes a big difference. Table 5.2 lists the latencies of read and write operations for a single word on the PCI bus. The numbers

Table 5.1: Emulator Floating-Point Operation Latencies

Instruction	Cycles
Arithmetic	3116.23
Load	3199.94
Store	3849.56
Move	3152.28
Compare	3846.21

are given in CPU cycles (400 MHz). The measurements prove the observation for the instruction latency, that the PCI bus access takes a significant part of the emulation.

Table 5.2: Hardware Latencies

Operation	Cycles
PCI Read	1134.46
PCI Write	6.57
$C_{\text{Trap}} + C_{\text{MW}}$	1242
Measuring Fault	6

In Chapter 2 the Amirix board used was introduced. This board consists of two dedicated PCI bridges. Thus, the FPGA cannot access the PCI bus directly. This matter of fact affects significantly the overall performance of the example implementation of the CyCoP platform. A more careful selection of the PCI board can improve the performance.

Table 5.2 lists also the $C_{\text{Trap}} + C_{\text{MW}}$ time. This is the accumulated time that it takes the processor to load the according exception handler, i.e. the MW, and the basic MW operations without the emulation of an actual CoP instruction. Together with the PCI bus access the sum of these latencies make up two third of an arithmetic CoP instruction latency. However, in contrast to the PCI access the $C_{\text{Trap}} + C_{\text{MW}}$ time cannot be improved anyhow.

The internal processor behavior as well as the internal timing of the individual instructions, was analyzed by using the PowerPC's trace capabilities. An external trace tool, such as RISCWatch, allows non-

invasively tracing of the code running in real time. The trace not only provide the actual execution flow but also timing information of every individual instruction. This tool allowed us to measure the $C_{\text{Trap}} + C_{\text{MW}}$ time of the system cycle accurate.

In case a tracing capability was absent the interrupt latency was estimated by using the decrementer exception. The decrementer continues to tick after passing through a zero count, a free-running counter, it is well suited for latency measurements. The decrementer exception can be used to measure interrupt latency by the following method:

- Enable the decrementer exception and initialize the decrementer count by writing the desired value to the decrementer.

- Ensure the decrementer exception handler code reads the decrementer count value at the earliest possible point.

Because the decrementer continues to tick after passing through zero and generating the interrupt request, the ones complement of the decrementer count is the elapsed count corresponding to the interrupt latency. The error bound of our calculated result, ranges from zero to three. This error margin will always be positive. In order to gather a statistically significant data sample, we ran a total of 1024 decrementer exception loops. The exception handler code logs the decrementer count. The decrementer exception handler generates and writes a random number to the decrementer, resetting the decrementer countdown value. Utilizing this method, the decrementer will generate interrupts at various points in the code so that we can get a random sampling.

Figure 5.5 shows the overall emulation overhead factors for the individual MiBench applications. The diagram presents the numbers for the different GCC optimization levels of every application at once.

The diagram shows the inverse behavior as the performance diagram in Figure 5.2 and Figure 5.3 or Figure 5.4. This becomes clear if we take into account that the performance of the emulation decreases indirect proportionally to the emulation overhead. We observe that the overhead within an individual benchmark application increases, as the MW call ratio increases. Thus, the $C_{\text{Trap}} + C_{\text{MW}}$ time takes a significant part of the overhead. For example, for the large BasicMath

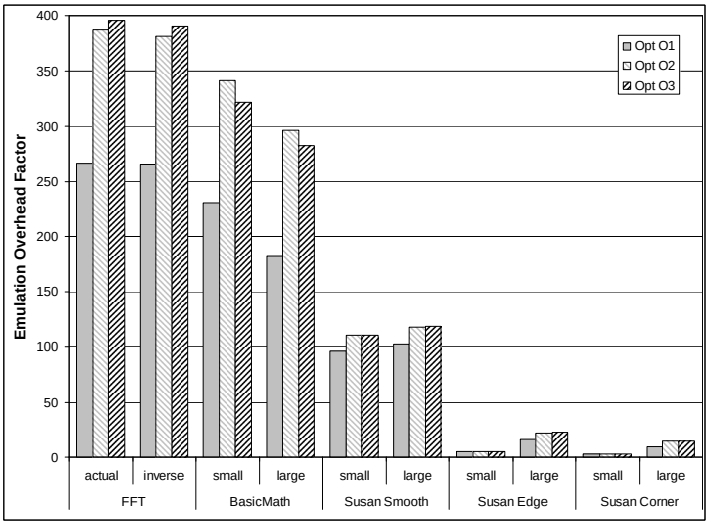

Figure 5.5: Emulation Overhead for MiBench with different Optimizations

application the call ratio increases by 60% and also the overhead increases by 62.6%. If the MW call ratio is over 50% like in the FFT application at the optimization level -O3, then the emulation overhead almost takes 400 times longer than the original computation. On the other hand if the call ratio is about 0.4% then the overhead only takes 3 times as long as the original computation.

5.1.5 Emulation Accuracy

Up to now we only looked at the emulation performance of the CyCoP platform for the IBM evaluation board. Even though the results are of interest, equally interesting is the question of accuracy. We want to quantify the inaccuracy of the introduced method. In the following the measurements refer to the same work load and prototype as for the performance measurements. The shown numbers refer to the emulated run time of the application, thus the time CyCoP pretends for the

application. The emulated time is measured without the optimization of a parser estimation during compilation time (refer to Section 3.3.2).

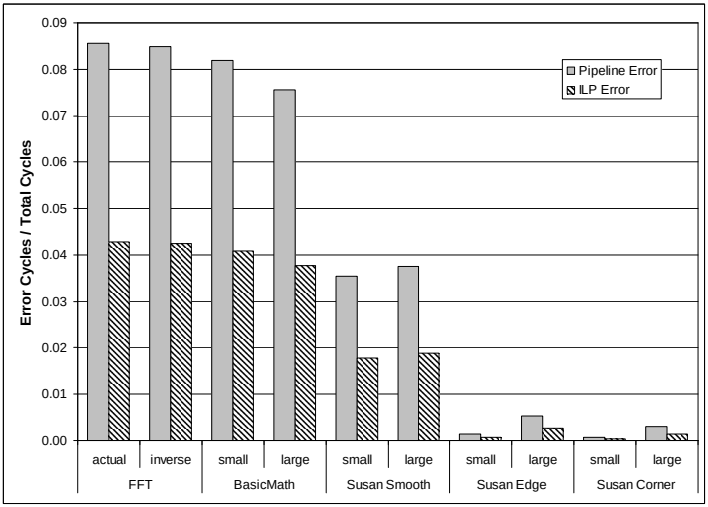

Figure 5.6: Measurement Error Ratios (Optimization -O0)

The PowerPC440 core [14] is a seven-stage pipelined and dual-issue superscalar architecture. The PowerPC440 core contains three execution pipelines. Each of these execution pipelines consists of four stages. With the analysis of Chapter 3 we are able to estimate the emulation inaccuracy. In Figure 5.6 the relative emulation error for the individual test application is shown. For the PowerPC440 core the pipeline error and the superscalar error apply. The diagram shows the two error types individually.

As both emulation errors depend on the number of MW calls, the situation is the same as in the performance measurements. The more MW calls are done per original instruction, the worse becomes the result. Therefore, we observe again a big difference between the FFT function and the Susan corner calculation. But even though in the FFT emulation almost after every second instruction a MW call follows, the measurement error for the pipeline reaches not more than

8.6%. The error for the superscalar reaches only 4%. For the biggest test application among the MiBench, the Susan smooth calculation, the pipeline error reaches 3.8% and the superscalar error only 1.9%. This application is more representative due to its long run and the fact that 7% of all instructions are floating-point instructions.

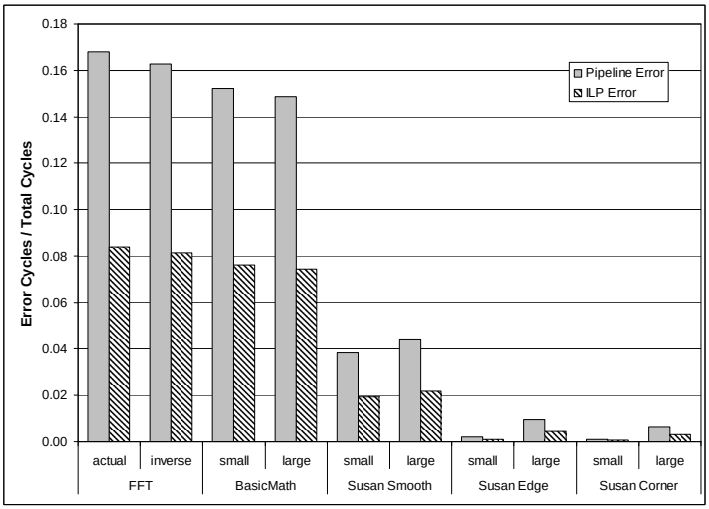

Figure 5.7: Measurement Error Ratios (Optimization -O2)

In Figure 5.7 and Figure 5.8 the MW call ratio is increased again by using the compiler optimization level -O2 and -O3. The ratio number for the FFT function is over 51% and the number for the pipeline error is 13.2%. But more interesting are the numbers of the BasicMath application at the -O3 optimization level. Even though the MW ratio is less for the BasicMath application than for the FFT function it is performing worse. Its pipeline error reaches 14.2% and its superscalar error 7.1%. The explanation of this phenomenon is found in the IPC_{av} number of the BasicMath application. The compiler optimization improves the IPC_{av} number for the FFT function, but it stays almost constant for the BasicMath application. With the -O3 optimization option the compiler starts to in-line certain functions,

which is not very effective for the BasicMath application, but for the FFT it clearly improves its performance.

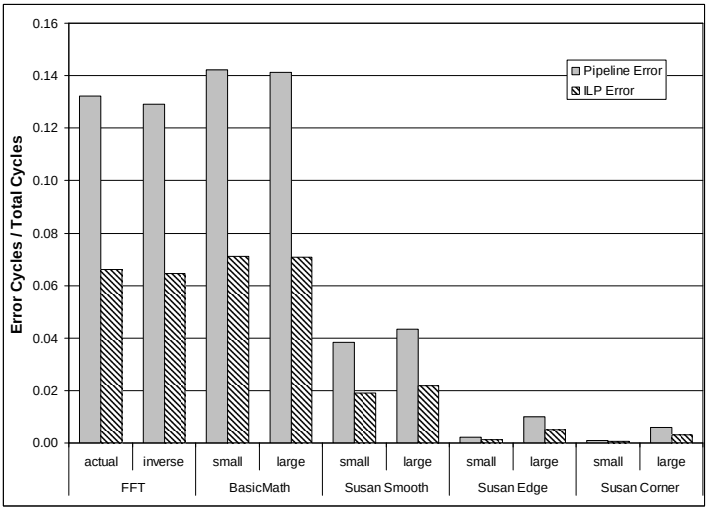

Figure 5.8: Measurement Error Ratios (Optimization -O3)

Thus, it is time to also have a look at the processor's performance numbers. To monitor the emulated performance the emulated run time is divided by the total number of executed instructions, without the MW instructions. This results in Cycles Per Instruction (CPI) which specifies the actual processor architecture performance very well.

Furthermore, we compare the CPI numbers of the prototype with the numbers of a real processor. The PowerPC 440EP comes with a PowerPC440 core and it does support floating-point instructions. Its FPU does single- and double-precision operations, thereof we only use the single-precision operation. The Yosemite evaluation board [78] used consists of a PowerPC 440EP.

However, we have to point out that the architecture of our prototype FPU and the architecture of the PowerPC 440EP FPU fundamentally differ. The PowerPC440 FPU incorporates a five-stage

arithmetic pipeline working in parallel with a four-stage load/store pipeline. The pipelines enable two instructions (one load/store and one arithmetic) to be issued during each cycle. Floating-point instructions execute with three- to five-cycle latency, except for division. Our FPU allows only single-issue and has a fixed latency for all instructions.

Most important difference between both FPUs is the integer conversion capability. The PowerPC440 FPU makes use of the fact that its internal architecture is 64-bit width and therefore it incorporates only the `fcti` instruction (converts floating-point to integer) as dedicated function. As our prototype is a true single-precision architecture, also the `fcfi` instruction (converts floating-point from integer) are implemented into the FPU. Readers, who are interested in more details about the conversion function in the PowerPC, should refer to the Appendix A in this book.

Figure 5.10 show the CPI numbers of the individual test application for each processor. We can see that the numbers for the first three applications are close together between the two processors. The difference can be explained with the architectural difference between the two FPUs. The prototype demonstrates the same behavior like the real FPU within an application. The CPI number decreases if the work load is increased in the same application.

The applications Susan edge and corner highly depend on the conversion function. On the right ordinate of the Figure 5.10 we indicate the difference between the instruction counts. As explained above, the two architectures convert integers in different ways. By the numbers provided by the Figure 5.10 one could conclude that it would be worth to integrate a dedicated conversion function into the PowerPC440 FPU. However, the impression is deceptive, as the rounding of the conversion results also plays a significant rule in the observed instruction count difference.

Beside the fact that the two FPU architecture are different and therefore the direct comparison between them provides no quantitative results, we can check the relative behavior of the prototype against the real implementation according to the compiler optimizations. Figure 5.9 and Figure 5.11 show again the graphs with the compiler optimizations options enabled.

Comparing the individual CPI numbers of the applications once

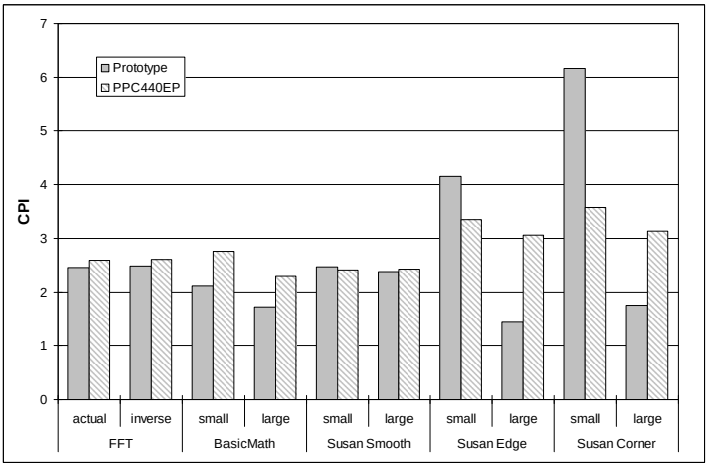

Figure 5.9: CPI Comparison between the Prototype and the PowerPC 440EP (Optimization -O0)

run on the prototype and once run on the real hardware we observe that it shows the same behavior comparing -O0 with -O2. Also the same is true for the real applications for the number of -O3. Thus, running the prototype with real applications and performing longer tests, which is exactly the target tasks of a having a rapid prototyping environment, the application behavior is accurately reflected.

5.2 Loosely-Coupled Coprocessor Results

To demonstrate the feasibility of a cycle-accurate emulation of a state-dependent, loosely-coupled CoP on the CyCoP platform, such a CoP must be found. The test CoP has to generate events which can be measured upon their timing correctness. This is important for being able to verify the process synchronization between software and hardware.

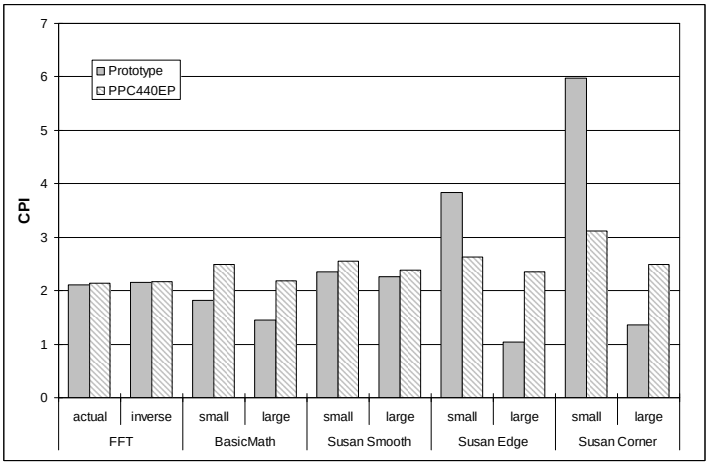

Figure 5.10: CPI Comparison between the Prototype and the PowerPC 440EP (Optimization -O2)

A perfect example CoP for the test is the so-called timer manager [63, 64]. Such a timer manager consists of several thousands of timers, which can be individually started and stopped. At the expiration of a timer the timer manager notifies the owner of the individual timer about the expiration. Network protocols, such as the Transmission Control Protocol (TCP) or the Stream Control Transmission Protocol (SCTP), make use of timers to guarantee a reliable data transfer. For every set data packet a timer is started. If this timer expires without having received the acknowledgement from the destination, the source has to retransmit the same data packet again. Therefore, these timers are called retransmission timers.

First we briefly introduce the test application together with the timer CoP used. Then we present the performance results of this timer CoP, is configured for the Virtex FPGA of the Spyder development board [52].

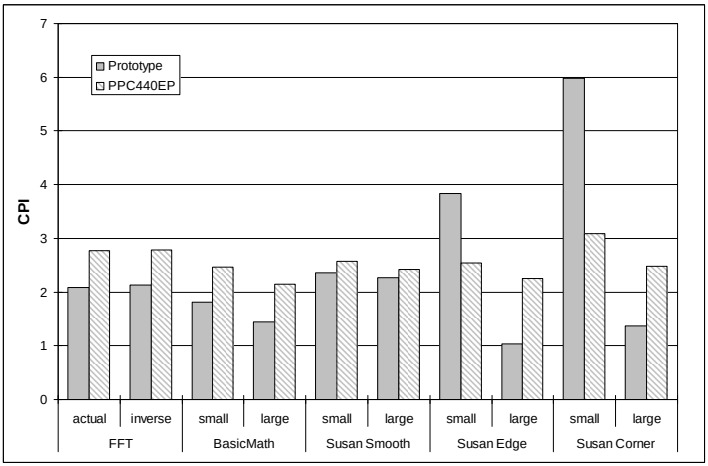

Figure 5.11: CPI Comparison between the Prototype and the PowerPC 440EP (Optimization -O3)

5.2.1 Timer Coprocessor

Although used informally in the literature, the notions for the timer facilities are terms that are not generally used in protocol specifications. Therefore, we will use the notation as introduced in [79].

Timeout denotes an event. Such an event is produced by a *timer*, and can be consumed by a network protocol machine. A timer is a device that generates timeouts. The point in the time when a timer has to produce the timeout is specified in terms of an absolute or relative time quantity. In conjunction with a timer, this quantity is called *time value*.

A *timer manager* is a device which contains multiple timers. Every timer can be accessed over a single interface and the timer manager autonomously serves all the timers.

The definition of the timer in terms of operation primitives exchanged over the entity boundary of the protocol machine [79] is as

follows:

Table 5.3: Timer Primitives

Primitive	Parameter
`start`	*time value, timer identity*
`stop`	*timer identity*
`expire`	*timer identity*

Table 5.4: Parameters of Timer Primitives

Parameter	Description
time value	the time duration of the timer specified
timer identity	the name or label of a related timer

The first two primitives are commands to initiate and terminate the operation of a single timer. In the network protocol code, they appear as normal instruction calls. The third primitive is the notification of the actual timeout, and communicated to the protocol entity as an interface event. From the protocol entity's point of view, the timer manager can be characterized as an abstract data type "timer". The first two operations create and delete instances of timers. The third operation is the "expired" message produced by a timer upon a timeout.

By implementing a timer manager in a hardware CoP, the TimerCoP, the individual timers managed by this CoP cannot be implemented as dedicated hardware decrementers. This is due to the fact, that there are too many of them and this would take too much die size. Therefore, they are stored in some kind of RAM. The TimerCoP presented in [63] is able to manage ten-thousands of timeout values. However, it is over designed for our needs to test the CyCoP platform. Figure 5.12 shows the block diagram of the simpler TimerCoP architecture used.

The central element of the TimerCoP is the RAM, where all timeout values are stored. This RAM is continuously addressed by a counter to retrieve all stored timeout values in sequence. Then the value is compared again the *time base* of the TimerCoP. Thus, the

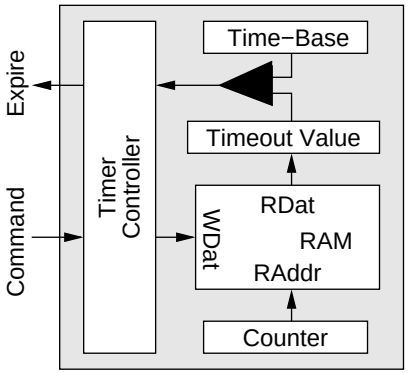

Figure 5.12: Timer Coprocessor Block Diagram

timeout values are stored in absolute values. This allows a simple comparison without further arithmetic.

The specialty of this TimerCoP is that the minimum timer granularity for this kind of timer manager is

$$T_{\min} = \frac{n_{\text{Timers}}}{f_{\text{Counter}}} \tag{5.1}$$

Thus, it cannot be smaller than the counter takes to check all stored timeout values. In the example implementation of the TimerCoP we used 2048 timers running with a Frequency of 3 MHz.

The controller in Figure 5.12 receives the *start* and *stop* commands from the network protocol over the on-chip interconnect. The controller manages the RAM according to the command from the protocol. If the controller detects a timeout event, it then notifies the protocol.

A detailed report for this TimerCoP architecture and its functionality can be found in [80].

5.2.2 Work Load

The perfect work load to stimulate the TimerCoP would therefore be a network protocol like the TCP. The protocol stacks of such network protocols is rather complex. To modify an existing protocol stacks,

to adopt it for our needs, is a big task and the resulting testbench does not fulfill our needs completely. To verify the correctness of our Roll-Back Algorithm (RBA) implemented in the CyCoP platform we have to prove that all CoP events are issued at the right time for the software application.

Therefore, we decided to produce an artificial stimuli stream for the TimerCoP, executed by the CPU. A Perl script generates a sequence of commands that *starts*, *stops* or checks on timers. The commands generate a Poisson-distributed sequence of *start* commands and use a uniform distribution for the timeout value of the timers. The timers are then *stopped* with a certain probability or allowed to *expire*.

The script intends to generate a usage pattern that resembles the one of a TCP stack. The idle time in between two commands is filled with a wait loop, which sums up a counter to a certain value. This simulates the processing of the TCP stack. The MW is called when the application accesses the TimerCoP during the `StartTimer` and `StopTimer` commands. If the MW receives a timeout event from the TimerCoP, it generates an interrupt to notify the software application. The application provides an interrupt handler for this case, which simply reads the *timer identity*, and records the current processor time as expiration time. The script also generates function calls `CheckExp`, which checks whether the timer that should have expired actually did so.

```
...
StartTimer( 43, 1396 );
WaitTicks( 452 );
StartTimer( 44, 2456 );
WaitTicks( 245 );
StopTimer( 43 );
WaitTicks( 3497 );
CheckExp( 44 );
...
```

The code sequence above shows an extract of a sample code generated by the Perl script. The script has three parameters to customize the stimuli; the number of timers used, the mean value for the Poisson-distribution, and the percentage of expiring timers can be set.

5.2.3 Emulation Performance

In this section we address the RBA performance of the CyCoP platform. As we use an artificial work load to stimulate the TimerCoP, we have to define first a meaningful value to be able to measure the emulation overhead. In the following we use an emulation overhead factor, which is calculated by dividing the real time by the emulated time. The reader has to keep in mind, that the work load is artificial. Thus, this factor should not be taken as a quantity but rather as a quality factor.

In Chapter 4 we discussed the theoretical overhead behavior by varying the maximum exploration time T_{Exp} of the RBA. The graph of the equation shows that there is an optimum for the exploration time which leads to the lowest possible emulation overhead. In Figure 5.13 the graph of the equation is shown again.

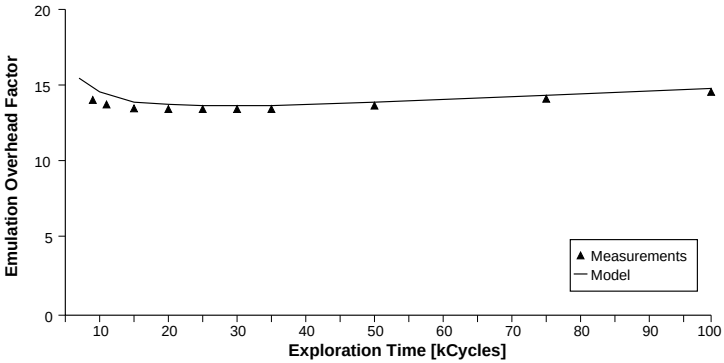

Figure 5.13: Comparison of Model and actual Measurements

The graph of the equation has been computed with the measured parameters of the hardware setup used. Table 5.5 lists the measured values of the needed parameters for the equation. The numbers are given in CPU cycles (400 MHz). The variable probability values are computed by dividing the number of CoP accesses and interrupts by the emulated time.

Measuring points are added to the graph in Figure 5.13. These measuring points are gained from the TimerCoP with a work load

Table 5.5: Hardware Costs for the RBA

Operation	Parameter		Cycles
Interrupt costs	C_{Int}	=	1427
Emulated CoP interrupt	C_{emInt}	=	273
Clock frequency ratio	q_{F}	=	12
Store/Load CoP context	C_{Store}	=	786,480

that matches the probability value used by the equation. The graph shows that the model overestimates the costs by about 2% at the computed minimum. The slope of the asymptote is less steep than the measurement. This indicates that the costs for the steps C_{Step} are lower than measured because of the measurement error of the individual analysis. The computed optimum for the exploration time T_{Exp} is also higher than actually required by the measurement.

The numbers in the graph of Figure 5.13 show that the overhead factor is about 14 for the optimal exploration time T_{Exp}. This is mainly due to the fact that the FPGA runs 12 times slower than the CPU. As the RBA requires that the CPU and the FPGA run in parallel, the theoretical minimum for the real runtime would be the emulated time on the FPGA plus the emulated time on the CPU:

$$T_{\text{MinEmulation}} = T_{\text{FPGA}} + T_{\text{CPU}} = T_{\text{App}} \cdot (12 + 1) = 13 \cdot T_{\text{App}} \quad (5.2)$$

The fact that the emulation reaches a minimum factor of 14 compared to the theoretical minimum of 13 shows that the RBA performs well.

The minimum for the measurements depends on how many events have to be synchronized during the emulation. Figure 5.14 shows the graphs for the same TimerCoP under different loads. Note that the ordinate only shows the range of interest. This is done to highlight the difference among them.

The graphs in Figure 5.14 show that the optimal T_{Exp} becomes smaller the higher the number of events. The probability increases that a CPU event happens during an exploration phase, if it is too long. At a CPU event the explored cycles are lost and run again. Thus, if there are many CPU events, the exploration time should be shorter. The figure also shows that the costs increase in general, and also the slope of the asymptote becomes steeper. This is the consequence of

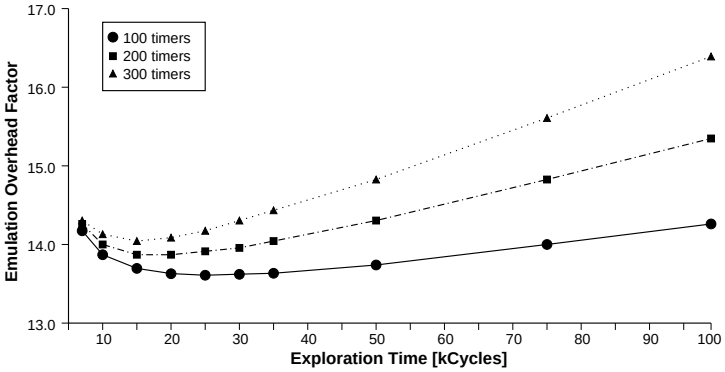

Figure 5.14: TimerCoP Behavior at increased Usage

the higher penalty of restoring the CoP context on a CPU event.

5.2.4 Emulation Overhead

The RBA not only introduces an overhead in time but also in space. The algorithm requires to have access to the whole context of the CoP. This access is granted to the Wrapper by inserting scan-chains into the design during the synthesis. These scan-chains cause the original design to increase by the size, as for every register additional multiplexers are inserted.

Table 5.6 lists the utilization of FPGA resources by the original CoP design. The most important number is the slice utilization. Xilinx groups several CLBs together into a slice. Thus, the slices are an indication of how many configurable resources are occupied by the design.

The numbers in the third column of the Table 5.6 shows the resource utilization of the TimerCoP after adding the scan-chain. The design uses now 66% more of the FPGA resources. This number matches with the results of other analyses [75].

The Wrapper resources are listed in the fourth column of the Table 5.6. Its size is manly constant over all projects in which it is used. Only the number of filler cells used to balance the scan-chain lengths and the size of the command buffer vary from project to project. In

Table 5.6: FPGA Resource Allocation by the TimerCoP

	CoP	Scanned	Wrapper	Total
Flip Flops	473	579	279	858
4 input LUTs	1353	2151	762	2913
Occupied Slices	746	1241	376	1617
Size Increase		66%	50%	117%

our example implementation the Wrapper increases the CoP design by 50%. Thus, the most resource overhead results from the insertion of the scan-chain.

By adding a multiplexer to every register in the CoP design for the scan-chain, we extend also the critical path in the design with an additional delay. Therefore, the maximum frequency achieved of the original design reduces. The original maximum frequency for our TimerCoP design is 500 MHz. With the scan-chain the design achieves 90.9 MHz. The numbers are gained from the synthesis tool, but not evaluated with the hardware.

5.2.5 Emulation Accuracy

Because of the artificial work load used to measure the performance of the RBA the analysis, the measurement of the real accuracy complicates. The Equation 4.18 requires the work load IPC and the number of total load/store instructions, $n_{\mathrm{l/s}}$, to quantify the accuracy. As explained in Section 5.2.2, not a real protocol stack has been implemented for the TimerCoP, but an artificial one. Therefore, the directly measured number of load/store instructions and IPC are also artificial. To overcome this problem we decided to take the numbers of the MiBench benchmark used for the FPU, as they correspond to numbers of a real application. Furthermore, we present the numbers with relative values, to facilitate to weigh the individual factors of the equation.

Table 5.7 lists the parameters taken from the MiBench application Susan with GCC optimization O2. The number of load/store instructions, $n_{\mathrm{l/s}}$, are already given as relative values. Means the numbers are divided by the total runtime cycles of the according application.

Table 5.7: Parameters for Accuracy Estimation

	Smooth		Edge		Corner	
	small	large	small	large	small	large
IPC	0.392	0.418	0.381	0.425	0.320	0.403
$n_{\mathrm{l/s}}$	0.185	0.197	0.179	0.199	0.151	0.189
$IPC \cdot n_{\mathrm{Pipe}}$	1.570	1.673	1.522	1.699	1.281	1.610
$e_{\mathrm{Start}} + e_{\mathrm{Gran}}$	4	4	4	4	4	4

The DAC debug facility of the PowerPC is a very powerful functionality. However, the drawback of using this functionality is that it introduces an extra latency cycle to every load/store instruction of the application. We experience that up to 45% of all instructions at runtime are load/store instructions. Thus, its portion to the total inaccuracy is not negligible. In Table 5.7 we learn that the DAC event makes up 20% of a constant inaccuracy for the emulation. One might consider not to use the DAC event as MW call because of its strong effect on the emulation accuracy.

Figure 5.15 shows the graphs for the different parameter sets coming from the MiBench applications. The graphs represent the ratio of total accuracy error over the total runtime with different numbers of DAC and timer events. The offset of the graphs comes from the above mentioned extra latency of the load/store instructions. A work load for which 1% of all instructions are CoP instructions, the inaccuracy is about 22% of which already 20% are from the debug facility. Thus, by replacing the MW call with trap instructions we can reach an inaccuracy of about 2%.

A work load with 10% CoP instructions of total instructions, thus a very heavily used CoP, the inaccuracy reaches about 38%. With a trap instruction as MW call, therefore, we would reach an inaccuracy of about 18%.

Thus, the biggest portion of the emulation inaccuracy we can avoid by using a different MW call type. The other factors are hardware-dependent and therefore, cannot be reduced. However, with realistic numbers of total CoP instructions, the inaccuracy comes to acceptable few percent.

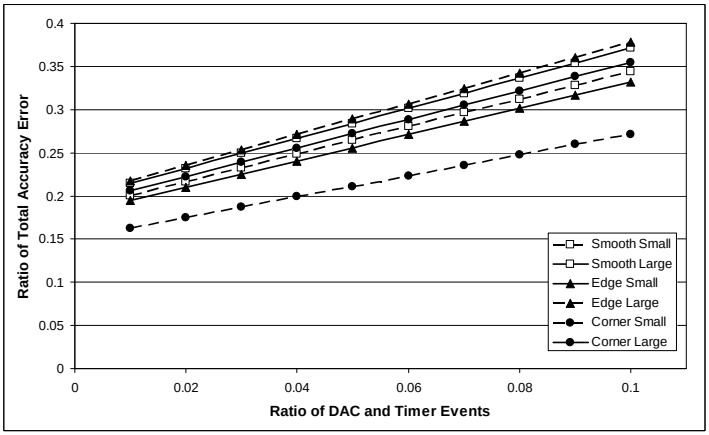

Figure 5.15: RBA Accuracy Estimation

5.3 Summary of Experimental Results

In the first part of this chapter we presented the measurement results for a tightly-coupled CoP implementation in the CyCoP platform. We extended the PowerPC 440GP with a single-precision FPU and run the test application of the MiBench benchmark with it. The results demonstrate that our CyCoP platform outperforms any simulator which reaches the same detail accuracy. Depending on the ratio between floating-point and integer instructions of an application, the prototype emulation achieves a performance in the range of 0.6–21.1 MIPS and accuracy in the range of 85.8% – 99.9% without further optimization. We compared the emulated performance of the prototype with the performance of a real hardware FPU integrated in the PowerPC 440EP. Despite of the architectural differences between the two FPUs qualitatively the prototype showed the same behavior as the real FPU. We conclude that using the CyCoP approach to prototype a tightly-coupled CoP delivers high emulation performance

at detailed architecture combined with high accuracy.

In the second part of the chapter we presented the measurement results for a loosely-coupled CoP in the CyCoP platform. As a case study we took a timer manager as CoP and stimulated it by randomly starting and stopping individual timers in the CoP. To verify the correctness of the RBA we measured the expiration event of a timer in the application and compared them with our expectations. For the exploration variable of the RBA we tried different values for the maximum and measured the emulation overhead. The measurement fit very well with our theoretical mathematical analysis. Thus, the maximum for the exploration variable can be estimated up front the emulation to achieve a minimal emulation overhead. CyCoP enables the event synchronization between a software process and a hardware CoP which allow a cycle-accurate hardware/software co-verification.

Chapter 6

Conclusions

In this thesis, we investigated problems in the hardware/software co-verification methods. We observed that simulators suffer from the tradeoff between accuracy and performance. The higher the level of detail for a model is, the slower the simulator performs. To overcome the performance issue, developers take a higher abstraction level for their models but this causes inaccuracies in the simulation results. On the other hand, hardware emulators run orders of magnitude faster than every simulator at the highest level of detail. However, hardware emulators are inflexible, expensive, and have a short lifetime. Furthermore, it is a rather complex task to implement a complete processor in programmable logic devices.

For state-dependent coprocessors (CoP), we argued that existing approaches for process synchronization cannot handle the events between hardware and a software process strictly cycle-accurately. The solutions described in the literature have been designed for pure software process synchronization and can therefore be used only to a limited extent for hardware-prototyping platforms.

In the work presented here, we addressed these problems by means of a new rapid prototyping method. Our solutions are summarized in Section 6.1. We conclude this thesis in Section 6.2 with an outlook on future work, for which this research laid the foundation.

6.1 Contributions of this Thesis

The research presented in this thesis has contributed a new rapid prototyping method to the field of hardware/software co-verification:

- We proposed a novel rapid prototyping method we call Cycle-accurate Coprocessor Prototyping (CyCoP). The CyCoP platform consists of several contributions, which are designed to fit together. The two main components of the platform are the *processor system* and the *coprocessor system*. The processor system consists, in fact, mainly of a standard processor chip with a host board-level interconnect, e.g. a PCI bus. The CoP system consists of off-the-shelf prototyping boards populated with programmable logic devices, e.g. FPGAs. The coprocessor (CoP) accesses of the software applications, running on the real processor, are enabled with a software layer called Middleware (MW). The CoP cores, running on programmable logic devices, are stimulated and controlled by a hardware layer called Wrapper. The MW hides the hardware architecture of the prototype and its introduced delays from the software application, to pretend real time behavior for the application. Thus, the MW enables a cycle-accurate co-verification of the hardware/software co-design.

- We introduced a new process synchronization method that enables the event synchronization between hardware and software processes. The events of a state-dependent CoP cannot be synchronized by using a *blocking read* primitive as used in other synchronization methods, such as the Kahn Process Network (KPN). The Roll-Back Algorithm (RBA) introduced exploits the fact that the CoP core is implemented into a programmable logic device. Therefore, we can grant the Wrapper full access to the CoP context during the synthesis of the design. The manipulation of the CoP's context is key for the RBA. It allows the Wrapper to control the CoP's state and, if needed, to set it back to an older state. This principle allows us to cycle-accurately emulate events between a state-dependent CoP and a software process.

- We proposed a new method of autonomously fetching any CoP request by the SW application without the invocation of the SW programmer or the application. The MW call introduced either keeps the original CoP instructions of the application or replaces them by trap instructions. Because of the nature of the MW call introduced, the software binaries for the CyCoP exhibit the same behavior as the original once. The size of the binary stays the same in any case. Therefore, the entire addressing scheme of the emulation is equal to the original one.

- The modular concept introduced for the CyCoP platform with the two main elements MW and Wrapper makes the platform hardware-independent. Therefore, the CyCoP concept can be adapted to any hardware environment providing a real processor connected to any programmable logic device. This has the advantage that the prototyping hardware can be compiled with modules from different technologies and vendors. For example, if during the prototyping phase a more powerful FPGA is issued, the user can simply replace the old hardware with the new one. If several equivalent hardware modules are available from different vendors, the user can take also the prices into account for the evaluation. Being able to combine the prototype hardware with any vendor modules also has the advantage that the user is independent of any vendor and therefore does not have to wait for the vendor to switch to newer technology, for example.

- The CyCoP design flow can be seamlessly integrated in user-defined design flows using conventional tools. Both flows, the software and the hardware flow, make use of a characteristic Intermediate Representation (IR) to integrate the CyCoP design layers into a standard design flow. For both design flows we chose an individual IR that is currently supported by all tools needed. Thus, CyCoP neither depends on a specific support of a design tool, nor does the design flow differ from the original flow, which is crucial for the software flow. This allows the design to use the original design tools for the software of the future architecture.

- The CyCoP introduces an efficient and accurate way to emulate

tightly-coupled CoPs. An Instruction-Set Simulator (ISS) has to deal with the tradeoff of being either accurate or achieving high performance. Our emulation performance measurements with benchmark applications on the sample implementation yielded the same numbers as the fastest IISs, but with the advantage of high accuracy and that the test application ran in the real hardware environment.

6.2 Future Work

In this thesis we presented the concept of the CyCoP platform with its software and hardware modules, enabling cycle-accurate emulation. During the implementation and testing of the CyCoP platform some interesting problems arose which are worth further investigations. In this section we outline the most interesting once.

6.2.1 Design Automation

The design flow of the current implementation of the CyCoP platform is simple, but nevertheless efficient. However, it still involves considerable manual configuration by the user. A goal of the CyCoP design flow should be to have a "single button press" solution. Thus, many configuration steps should be automated in the design flow.

In the software flow, the replacement and re-encoding of the CoP instructions into trap instructions can be performed by the compiler itself. A retargetable compiler can be extended with a switch that forces it to give the output either with the real CoP instructions or with the re-encoded trap instructions. The new encoding is then automatically provided to the MW source code during compilation.

The automation of the hardware design flow is almost achieved by the automatic scan-chain insertion into the CoP design. However, the Wrapper's direct ports to the CoP core have to be configured manually. The in- and output signals are wired by the user with the command and result buffer of the Wrapper. An automation of this task should be possible by parsing the appropriate HDL testbench of the CoP core. Thus, the testbench is recompiled and the outcome is the Wrapper for the prototype.

6.2.2 Roll-Back Algorithm

For the RBA we see one potential improvement. In the current implementation only one process is active while the other is idle. If the Wrapper could store multiple contexts and keep track of them, it would be possible for the CoP to run in parallel to the software process. At a software event the Wrapper must know which context it has to restore. The MW can also already set the timers for possible CoP events, but it must be able to clear them at a restore of the CoP context.

It is also desirable to extend the algorithm to the Bus Functional Models (BFM) on an FPGA. This BFM could be used by several CoP concurrently on the FPGA. The Wrapper would sit as a master port at the BFM and monitor only those events that apply to the CPU. In this way a complete SoC design could be synchronized with the software process.

6.2.3 Design Flow Parser

In Chapter 3 we investigated the limitations for the accuracy due to architectural characteristics of processors. In the software design flow the original source code is parsed to replace CoP instructions with MW calls. This parser can be enhanced to check also how many instructions are executed simultaneously in the processor before the transition happens, the parser can estimate the "waiting" time of the MW call for all instructions to commit. This estimation can be encoded together with the original instruction opcode into the MW call, and the MW takes this time into account for the emulation. With this method the emulation inaccuracy introduced is reduced. However, effects such as a cache miss for instructions or for data are ignored. This solution would, however, clearly enhance the emulation accuracy.

6.2.4 Hardware Debugging Capabilities

A software debugging tool like the GNU Debugger (GDB) [46] allows a user to see what is going on "inside" a software application while it executes. It provides the user with information on the state of the

application at the moment it crashed.

In Chapter 4, where we introduced the possibility to store and load the context of any coprocessor we focused only on the implementation of the RBA. In fact, this capability can be used for efficient debugging of the CoP core.

In [81] the authors propose a Source Level Emulator (SLE) as a method to close this gap by combining behavioral simulation with hardware emulation. The idea of SLE is to run the hardware application on an emulator hardware and to keep the correlation between hardware elements and the behavioral VHDL source such that it is possible to stop the hardware by interrupting the clock and to extract values of variables in the source code by reading registers of the circuit. This correlation is mainly obtained through logging the synthesis steps of the high-level synthesis.

In contrast to the SLE in our CyCoP platform, the breakpoints are not set in the behavioral HDL source but in the software application running on the CPU.

Appendix A

Data Format Conversion

High-level languages define rules specifying various implicit conversions or coercions, in addition to the explicit conversion requests in the source code. Compilers may execute these conversions between different types using calls to functions in the run-time library. For simple cases, the compiler may emit the code directly.

In this chapter we investigate how the GCC compiler specifically handles the integer to single-precision floating-point conversion for the PowerPC target. This is for interest for us as the conversion algorithm involves double-precision operations which are not supported by our prototype implementation. To provide still an accurate comparison between the real FPU in the PowerPC 440EP processor and our prototype, we need to find a solution to overcome the double-precision operation. The following sections we discuss the conversion from a single-precision floating-point value to an integer value and vice versa respectively.

A.1 Floating-Point to Integer

The PowerPC ISA provides an instruction `fctiw` which converts a single-precision floating-point value to an integer value and stores the result into a FPR. Thus, there is a hardware supported operation which does the job.

In the example given in Table A.1 a floating-point value in `FR1` is converted to an integer in `R3`. The processor always transfers values between the FPR and the GPR through memory. However, even though the value in `FR2` contains only a 32-bit value after the conversion the compiler uses a double-precision store instruction (`stfd`) to store the content of `FR2` into the program stack. The load instruction (`lwz`) makes sure it gets the lower word from the stack, where the actual integer is stored, into the `R3`.

During the emulation all double-word load/store are replaced with single-word once, as our prototype FPU only contains 32-bit FPR. To ensure that the `lwz` instruction gets the right value from the stack, the displace value (`disp`) from the original `stfd` instruction has to be adopted, i.e. increased by four (see Table A.1).

Table A.1: Floating-Point to Signed Integer

GCC Compiler			Prototype	
`fctiw`	`FR2,FR1`		`fctiw`	`FR2,FR1`
`stfd`	`FR2,disp(R1)`	$\Longrightarrow$	`stfs`	`FR2,disp+4(R1)`
`lwz`	`R3,disp+4(R1)`		`lwz`	`R3,disp+4(R1)`

Before the emulation we parse the original source code for the `fctiw` instructions. If the parser finds a conversion instruction it replaces the according `stfd` instruction with a `stfs` instruction and increases the displace value by four.

A.2 Integer to Floating-Point

In contrast to the previous conversion the PowerPC 440 ISA does not provide a specific instruction for the conversion of an integer value into single-precision floating-point value. The GCC compiler inserts therefore a software routine. The routine first flips the integer sign bit and places the result in the low-order part of a double-word in the program stack. Afterwards the routine creates the high-order part with sign and exponent fields such that the resulting double-word value interpreted as a hexadecimal floating-point value is $0\mathtt{x}1.00000ddddddd \cdot 2^{52}$. $0\mathtt{x}ddddddd$ is the hexadecimal sign-flipped integer value. Then,

the double-word is loaded as a floating-point value. Finally, the hexadecimal floating-point value $0x1.0000080000000 \cdot 2^{52}$ is subtracted from the previous value to generate the result.

The code sequence in Table A.2 converts an integer in `R3` to a floating-point value in FR1.

Table A.2: Signed Integer to Floating-Point

GCC Compiler			⟹	Prototype	
`# FR2 = 0x4330000080000000`					
`lis`	`R0,0x4330`			`lis`	`R0,0x4330`
`stw`	`R0,disp(R1)`			`stw`	`R0,disp(R1)`
`xoris`	`R3,R3,0x8000`		⟹	`mr`	`R3,R3`
`stw`	`R3,disp+4(R1)`			`stw`	`R3,disp+4(R1)`
`lfd`	`FR1,disp(R1)`		⟹	`lfs`	`FR1,disp+4(R1)`
`fsub`	`FR1,FR1,FR2`		⟹	`fcfiw`	`FR1,FR1`
`frsp`	`FR1,FR1`		⟹	`fmr`	`FR1,FR1`

In contrast to the real FPU our prototype coprocessor consists of a special instruction to convert an integer to a floating-point (`fcfiw`). Thus, for the emulation we have to make sure that the integer value is loaded unchanged into a FPR. The `xoris` operation has to be replaced by a dummy operation like a move operation `mr` (refer to Table A.2). Before the emulation a parser checks for three things. It observes whether the value 0x4330 is stored into the stack. Furthermore, whether there is an `fsub` instruction followed by an `frsp` instruction, which operates on the result of the subtraction. If these conditions are fulfilled the parser replaces the `xoris`, `fsub` and `frsp` with the according instructions like described in the example.

Also care has to be taken with the stack manipulation. The double-word load instruction (`lfd`) is replaced with a single-word, thus the displace value (`disp`) has to be increased by four as well.

The code sequence in Table A.3 converts an unsigned 32-bit integer to a floating-point value. This code example is similar to the example given for the signed case in Table A.2.

The floating-point value is constructed in the stack, as before, but the sign bit is not flipped. For the subtraction the hexadecimal floating-point value $0x1.0000000000000 \cdot 2^{52}$ is used to produce the

result.

Table A.3: Unsigned Integer to Floating-Point

GCC Compiler				Prototype
`# FR2 = 0x4330000000000000`				
`lis R0,0x4330`			`lis`	`R0,0x4330`
`stw R0,disp(R1)`			`stw`	`R0,disp(R1)`
`stw R3,disp+4(R1)`			`stw`	`R3,disp+4(R1)`
`lfd FR1,disp(R1)`		$\Longrightarrow$	`lfs`	`FR1,disp+4(R1)`
`fsub FR1,FR1,FR2`		$\Longrightarrow$	`fcfiw`	`FR1,FR1`
`frsp FR1,FR1`		$\Longrightarrow$	`fmr`	`FR1,FR1`

The parser works the same as in the signed case except for the `xoris` operation which is not used in this conversion.

Bibliography

[1] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SoC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, 1999.

[2] C. Fields. Design reuse for fpgas. In *Xcell Journal*, volume 37, pages 40–42. Xilinx, 2000.

[3] SIA: Semiconductor Industry Association. http://www.sia-online.org.

[4] ITRS: International Technology Roadmap for Semiconductors. Semiconductor Industry Association (SIA). http://public.itrs.net, 2004.

[5] A. Anant. *Presentation to EDAC Meeting*. Sun Microsystems, April 2000.

[6] A. Bechtolsheim and A. Raza. *Addressing Complex System and IC Verification Bottlenecks*. Cisco Systems, September 2001.

[7] S. K. Roy, S. Ramesh, S. Chakraborty, T. Nakata, and S. P. Rajan. Functional Verification of System on Chips—Practices, Issues and Challenges. In *Proc. of the Asia South Pacific Design Automation/VLSI Design Conference (ASP-DAC)*, pages 11–13, 2002.

[8] J. A. Rowson. Hardware/Software Co-Simulation. In *Proc. of the Design Automation Conference (DAC)*, pages 439–440, June 1994.

[9] P. M. Hansen. *Coprocessor Architectures for VLSI.* PhD thesis, University of California at Berkeley, 1988.

[10] M. Hohenauer, H. Scharwächter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1276–1283, 2004.

[11] ARM Limited. *ARM Architecture Reference Manual*, 1999.

[12] MIPS Technologies. *Core Coprocessor Interface Specification*, 1.16 edition, August 2004.

[13] Tensilica Inc. Xtensa. http://www.tensilica.com.

[14] IBM Corporation. *The PowerPC 440 Core*, 1999.

[15] Free Software Foundation Inc. GCC: The GNU Compiler Collection. http://gcc.gnu.org.

[16] ACE: Associated Compiler Experts. CoSy: Compiler development System. http://www.ace.nl/compiler/cosy.html.

[17] IBM Corporation. *The Cell Broadband Engine Architecture*, 2005.

[18] M. Glesner and A. Kirschbaum. State-of-the-Art in Rapid Prototyping. In *Proc. XI Brazilian Symposium on Integrated Circuit Design*, pages 60–65, September 1998.

[19] H. Krupnova and G. Saucier. FPGA-Based Emulation: Industrial and Custom Prototyping Solutions. In *Proc. of Field Programmable Logic and Applications (FPL)*, pages 68–77, August 2000.

[20] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1992.

[21] R.S. Boyer, M. Kaufmann, and J.S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, pages 27–62, 1995.

[22] S. Owre, J.M. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Int. Conf. on Automated Deduction*. Springer-Verlag, 1992.

[23] J. P. Bowen and M. G. Hinchey. Ten Commandments of Formal Methods ... Ten Years Later. *IEEE Computer*, 39(1):40–48, January 2006.

[24] N. Ohba and K. Takano. An SoC Design Methodology Using FPGAs and Embedded Microprocessors. In *Proc. of the Design and Automation Conference (DAC)*, pages 747–752, June 2004.

[25] K. Hines and G. Borriello. Dynamic communication models in embedded system co-simulation. In *Proc. of the 34th Design Automation Conference (DAC)*, pages 395–400, 1997.

[26] R. Helaihel and K. Olukotun. *Emulation and Prototyping of Digital Systems*. Kluwer, 1996.

[27] M. Courtoy. Rapid system prototyping for real-time design validation. In *Proc. of the 9th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pages 108–112, June 1998.

[28] A. Kirschbaum and M. Glesner. Rapid Prototyping of Communication Architecture. In *Proc. of the 8$^{\text{th}}$ International Workshop on Rapid System Prototyping (RSP)*, pages 136–141, 1997.

[29] W. Rosenstiel. *Hardware/Software Co-Design: Prototyping and Emulation*, chapter 3. Kluwer Academic Publisher, 1997.

[30] J. Darringer, E. Davidson, D. J. Hathaway, B. Koenemann, M. Lavin, J. K. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. EDA in IBM: Past, Present, and Future. In *Proc. of IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, volume 19, pages 1476–1497, December 2000.

[31] T. Burggraff, A. Love, R. Malm, and A. Rudy. The ibm los gatos logic simulation machine hardware. In *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers*, October 1983.

[32] M. Denneau. The Yorktown simulation engine. In *Proc. of the 19th Design Automation Conference (DAC)*, 1982.

[33] D. K. Beece, G. Deibert, G. Papp, and F. Villante. The IBM engineering verification engine. In *Proc. of the 25th Design Automation Conference (DAC)*, pages 218–224, 1988.

[34] R. Goering. Two machines seen as paving the way for next-generation microprocessors—Quickturn boosts emulation to 20M gates. Electronic Engineering Times (EETimes), November 1998.

[35] Cadence Design Systems Inc. http://www.cadence.com.

[36] Mentor Graphics Corporation. http://www.mentor.com.

[37] R. Tessier, J. Babb, M. Dahl, S.Z. Hanono, and A. Agarwal. The Virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environment. In *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 1994.

[38] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *IEEE Workshop on FPGA-based Custom Computing Machines (FCCM)*, April 1997.

[39] HARDI Electronics Inc. http://www.hardi.com.

[40] A. Kirschbaum, S. Ortmann, and M. Glesner. Rapid Prototyping of a Co-Processor based Engine Knock Detection System. In *Proc. of the 9$^{\text{th}}$ International Workshop on Rapid System Prototyping (RSP)*, pages 124–129, 1998.

[41] T. Buchholz, G. Haug, U. Kebschull, G. Koch, and W. Rosenstiel. Behavioral Emulation of Synthesized RT-level Descriptions Using VLIW Architectures. In *Proc. of the 9th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pages 70–75, 1998.

[42] U. Kebschull, G. Koch, and W. Rosenstiel. The WEAVER Prototyping Environment for Hardware/Software Co-Design and Co-Debugging. In *DATE, Designer Track*, pages 237–242, 1998.

[43] C. Chang, J. Wawrzynek, and R.W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design and Test of Computers*, pages 114–125, 2005.

[44] N. Busá, M. Verberne, R. Peset Llopis, and S. Ramanathan. Rapido: A modular, multi-board, heterogeneous multi-processor, pci bus based prototyping framework for the validation of soc vlsi designs. In *IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 159–164, July 2002.

[45] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[46] Free Software Foundation Inc. Gdb: The gnu project debugger. http://www.gnu.org/software/gdb/gdb.html.

[47] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[48] S. Sjoholm and L. Lindh. *VHDL for Designers*. Prentice Hall, 1997.

[49] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, Inc., 3rd edition, 2000.

[50] Inc. Xilinx. Integrated Software Environment (ISE) Foundation. http://www.xilinx.com/ise.

[51] IBM Corp. *PPC440GP Evaluation Board: User's Manual*, 2002.

[52] X2E. *Spyder-Virtex-X2E: User's Manual*, 1.11.1 edition, 2001.

[53] Amirix Systems Inc. *PCI FPGA Development Board: User Guide*, 06 edition, 2004.

[54] W. Denx. Das u-boot: Universal bootloader. http://sourceforge.net/projects/u-boot.

[55] IBM Corp. *RISCWatch Debugger: User's Manual*, 2002.

[56] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proc. of the 40th conference on Design automation*, pages 758–763, 2003.

[57] J. Schnerr, G. Haug, and W. Rosenstiel. Instruction set emulation for rapid prototyping of socs. In *Proc. of the Design Automation and Test in Europe (DATE) Conference*, pages 562–567, March 2003.

[58] M. Puig-Medina, G. Ezer, and P. Konas. Verification of configurable processor cores. In *Proc. of the 37th Conference on Design Automation*, pages 426–431, 2000.

[59] D. Langen, J. Niemann, M. Porrmann, H. Kalte, and U. Rückert. Implementation of a risc processor core for soc designs: Fpga prototype vs. asic implementation. In *Proc. of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC)*, April 2002.

[60] M. Gschwind, V. Salapura, and D. Maurer. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(2):241–250, April 2001.

[61] F. Pogodalla, R. Hersemeule, and P. Coulomb. Fast prototyping: A system design flow for fast design, prototyping and efficient ip reuse. In *Proc. of the Seventh International Workshop on Hardware/Software Codesign*, pages 69–73, May 1999.

[62] R. E. Gonzalez. Xtensa — a configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[63] S. Dragone, A. Döring, and R. Haugenau. A Large-Scale Hardware Timer Manager. In *Proc. of the ANCHOR Workshop*, pages 24–29, June 2004.

[64] K. Septinus, S. Dragone, M. Langner, and P. Pirsch. A Scalable Hardware Algorithm for Demanding Time-Out Management in Network System. In *Proc. of the PARS Workshop*, May 2011.

[65] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of IFIP Congress 74*, pages 471–475. North-Holland, Amsterdam, August 1974.

[66] O. P. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *Proc. of the 14th International Symposium on System Synthesis (ISSS)*, pages 1–6, September 2001.

[67] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[68] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[69] L. A. Barroso, S. Iman, J. Jeong, K. Oner, K. Ramamurthy, and M. Dubois. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer Magazine*, February 1995.

[70] IBM Corp. *PPC440GP Embedded Processor: User's Manual*, 2002.

[71] S. Dragone and C. Lombriser. The Ordering of Events in a Prototyping Platform. In *Proc. of the IEEE International Workshop on Rapid System Prototyping*, pages 211–217, June 2005.

[72] J. Mignolet, V. Nollet, P. Coene, D. Verkest, and V. Lauwreins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. of the DATE*, pages 986–991, March 2003.

[73] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA coprocessors. In *Field-Programmable Logic and Applications*, pages 121–130, August 2000.

[74] T. W. Williams and K. P. Parker. Design for testability — a survey. *IEEE Transactions on Computers*, C-31(1):2–15, January 1982.

[75] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. In *Field-Programmable Logic and Applications*, pages 483–492, August 2001.

[76] Rudolf Usselmann. *Open Floating Point Unit.* The Free IP Cores Projects: www.opencores.org, 2000.

[77] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4$^{\text{th}}$ Annual Workshop on Workload Characterization*, pages 83–94, December 2001.

[78] Embedded Planet. *PPC440EP Evaluation Board: User's Manual*, 2005.

[79] E. Mumprecht, D. Gantenbein, and R. F. Hauser. Timers in osi protocols: Specification versus implementation. In *Proc. of Int'l Zurich Seminar on Digital Communications*, pages 93–98, March 1988.

[80] C. Lombriser. Process synchronization methods for prototyping platforms. Master's thesis, Swiss Federal Institute of Technology (ETH) Zurich, 2005.

[81] G. Koch, U. Kebschull, and W. Rosenstiel. Debugging of behavioral vhdl specifications by source level emulation. In *Proc. of the Conference on European Design Automation*, pages 256–261, 1995.

Patents

§1. A. Döring, S. Dragone, A. Herkersdorf, R. Hofmann, and C. Kuhlmann, *Method and Apparatus for Using FPGA Technology with a Microprocessor for Reconfigurable, Instruction Level Hardware Acceleration*, JP3900499, 2007-01-12

§2. S. Dragone and A. Döring, *Coupling a General Purpose Processor to an Application Specific Instruction Set Processor*, US7293159, 2007-11-06

§3. S. Dragone and A. Döring, *Detecting a Timeout of Elements in an Element Processing System*, US7552226, 2009-06-23

§4. A. Döring, S. Dragone, A. Herkersdorf, R. Hofmann, and C. Kuhlmann, *Method and Apparatus for Using FPGA Technology with a Microprocessor for Reconfigurable, Instruction Level Hardware Acceleration*, US7584345, 2009-09-01

§5. A. Döring, S. Dragone, A. Herkersdorf, R. Hofmann, and C. Kuhlmann, *Method for Using FPGA Technology with a Microprocessor for Reconfigurable, Instruction Level Hardware Acceleration*, US7603540, 2009-10-13

§6. A. Döring, P. Sagmeister, J. Rohrer, S. Dragone, R. Glauberg, F. Auernhammer, and M. Gabrani, *System and Method Utilizing Programmable Ordering Relation for Direct Memory Access*, US7613850, 2009-11-03

§7. S. Dragone and A. Döring, *Detecting a Timeout of Elements in an Element Processing System*, US7725591, 2010-05-25

§8. S. Dragone and A. Döring, *Coupling a General Purpose Processor to an Application Specific Instruction Set Processor*, US7831805, 2010-11-09

§9. S. Dragone, T. Visegrady, and V. Condorelli, *Indirectly-Accessed, Hardware-Affine Channel Storage in Transaction-Oriented DMA-Intensive Environments*, US8140792, 2012-03-20

Biography

Silvio Dragone was born in Switzerland. During his studies at the Swiss Federal Institute of Technology Zurich (ETH Zurich), Switzerland, he worked as an intern for Oerlikon Contraves, Switzerland, in 1999, and for IBM Research GmbH, Zurich Research Laboratory, Switzerland, in 2000. Also in 2000, he wrote a research thesis entitled "IC Design: A FireWire GPS Reference Time Chip" at the Integrated Systems Laboratory of ETH Zurich and in 2001, another research thesis entitled "Porting of an OS on the Intel IXP1200 Network Processor" at the Computer Engineering and Networks Laboratory of ETH Zurich. He completed his graduate thesis entitled "IOF — SPI-5 Receiver" in 2002 at IBM Research GmbH and received a *dipl. El.-Ing. ETH* degree in Electrical Engineering from the ETH Zurich. In the same year, he joined the IBM Research GmbH as a PreDoc, first in the Network Processor Hardware group, and then in the I/O Networking Architecture group.