Master Thesis

Design and Implementation of a Multi-Core Power and Performance Emulation Platform

Michael Lackner

Institute for Technical Informatics
Graz University of Technology
Head: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Reviewer: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger

Advisor: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Christian Bachmann

Graz, October 2010

Kurzfassung

Während des Designprozesses eines System-on-Chip (SoC) muss eine Vielzahl an Designparametern beachtet werden. Dies sind zum Beispiel die Leistungsaufnahme, die Verarbeitungsgeschwindigkeit sowie die Zeit für die Entwicklung. Diese Masterarbeit behandelt das Design und die Implementierung einer Leistungs- und Performance-Evaluierungs-Plattform für Multiprozessoren. Mithilfe dieser Plattform ist es für Entwickler während des Designprozesses eines neuen Chips möglich, diesen zu testen und, darauf aufbauend, diesen zu optimieren. Es wurde dabei die Power-Emulations-Technik auf einem FPGA Entwicklungsboard angewandt. Mithilfe dieser Technik ist es möglich, die Leistungsaufnahme eines Chips zur Laufzeit des Systems zu bestimmen. Dies ermöglicht einen Geschwindigkeitsgewinn bis zu einen Faktor von 100 gegenüber einer Software-Simulation. Zusätzlich werden Performance-Statistiken wie Cache-Misses und Register-Zugriffe für jede CPU mithilfe von Sniffer-Einheiten im Chip detektiert und gesammelt. Diese so erzeugten Power- und Performance-Statistiken werden über die Ethernet-Schnittstelle zu einem Host-PC geschickt. Eine Evaluierungssoftware, welche auf diesem läuft, empfängt die Daten und stellt sie grafisch dar. Somit können Schwachstellen im Hardware- oder Softwaredesign schon früh gefunden und verbessert werden.

Stichwörter: Power Estimation, Performance Estimation, Power Emulation, LEON3, Multi-Core, FPGA, Ethernet

Abstract

Exploring the design space for System-on-Chip (SoC) designs is becoming more and more complex. Performance requirements, time to market, power and thermal issues have to be kept in mind. In this work the design and implementation of a power and performance evaluation platform for multi-core systems is presented. With this platform it will be possible to change hardware and software parameters during the design process and to acquire fast feedback on the impact of these changes. We are using the power emulation technique on a field programmable gate array (FPGA) onto which we synthesize a LEON3 multi-core system. With this technique we obtain power data during run-time for every component which is up to 100 times faster than using software simulators. Additionally performance statistics like cache misses and register write/read accesses for every CPU are also collected by hardware sniffer units. The collected power and performance statistics are sent by the Ethernet chip on the FPGA to a host computer. Evaluation software running on the host visualizes the received power and performance data to quickly obtain an overview of the bottlenecks of the current hard- and software design.

Keywords: Power Estimation, Performance Estimation, Power Emulation, LEON3, Multi-Core, FPGA, Ethernet

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Danksagung

Zuerst möchte ich Ass.Prof. Dr.techn. Christian Steger, für die Betreuung und Begutachtung dieser Arbeit, danken. Weiters allen Beteiligten des POWERHOUSE Projekts welche diese Arbeit ermöglicht haben. Speziell meinem Betreuer DI Christian Bachmann welcher immer Zeit für Diskussionen und Hilfestellungen hatte. Besonders seine MatLab Kenntnisse für die Aufbereitung der gewonnen Daten waren sehr hilfreich. Weiters möchte ich DI Andreas Genser für seine Unterstützung beim Adaptieren des Linux Kernels für das Ansprechen meiner Hardware-Komponente danken. Weiters DI Armin Krieg für seine Zeit beim Testen des Quad-Core-Systems. Weiters allen Mitstudenten am Institut dafür, dass man nach einer gemeinsamen Kaffeepause wieder motiviert weiter arbeiten konnte.

Besonders möchte ich mich bei meiner Familie bedanken, welche mich während des Studiums immer unterstützt hat.

Graz, Oktober 2010 Michael Lackner

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The first commercial 4-bit 4004 microprocessor from Intel in the year 1971 consisted of 2300 transistors and operated at a clock rate of 740kHz. Today's microprocessors like the Intel Core 2 DUO contain 100.000 times more transistors than the Intel 4004 and run with a clock speed of three GHz. This dramatic speed increase was possible because every new invented semiconductor manufacturing process, starting from feature sizes of 10μm in 1971 to 32nm today, decreased the power consumption and made higher clock rates possible. When the semiconductor manufacturers reached the 130nm world this correlation was getting weaker. Power consumption and temperature problems emerge when the processing performance of the chip is increased by higher clock speed alone. Chip designers dealt with this problem by increasing the System-on-Chip (SoC) design speed through multi-core designs on a single chip [33, 29, 2].

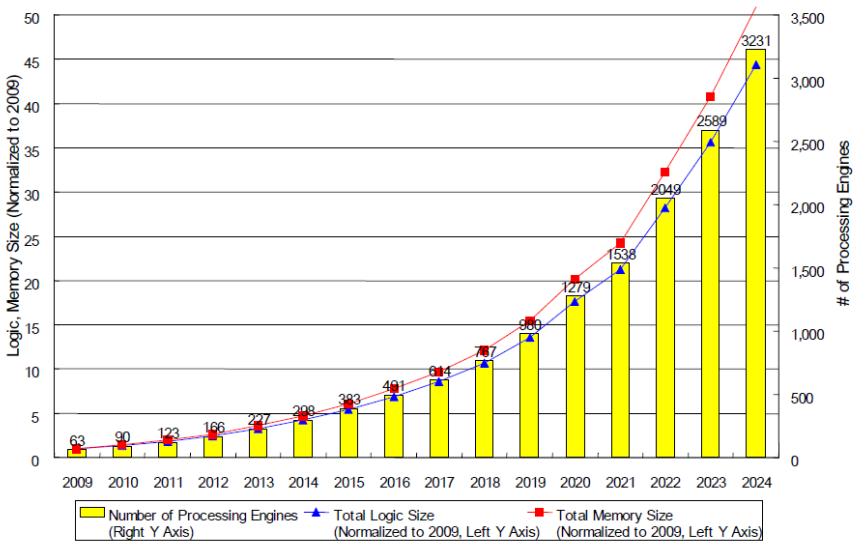


Figure 1.1: Portable Design Complexity Trends [45]

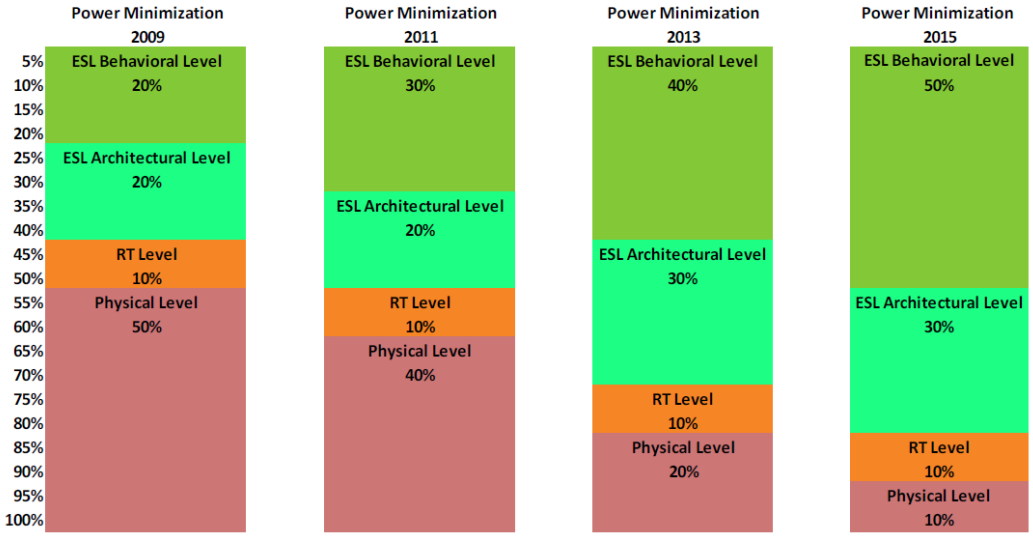


Figure 1.2: Evolving Role of the High-level Design Phases [44]

Exploring the design space for these multi-core SoC designs is becoming more and more complex. This is due to an expected exponential increase of processing engines (PEs) to get the processing performance for future software. This growth from 63 PEs in the year 2009 to an estimated number of 3231 is shown in Figure 1.1. For the system designer it will become increasingly difficult to handle this high number of sub components [45].

These trends will also lead to a growth of the power consumption of the SoC. This fact will lead to more energy saving efforts during the design phase especially for mobile devices because of the slow battery capacity development. To handle these challenges more design effort has to be invested into the high-level design phase of SoC development. Figure 1.2 shows the energy saving potential of the behavioral-level and architectural-level which will increase from 40% to expected 80% in the year 2015. Design optimizations on the physical-level will lead to little power savings reserves of 10% in the future. For this reason an early high level evaluation platform will be necessary to observe required speed and power constraints. Optimizing the software for the used hardware platform will be one of the keypoints during SoC development. A late optimization of failing constraints will lead to significant time to market delays and high costs [44].

Virtual platforms are a widely used tool for early power and performance assessment analysis. They simulate a whole SoC on a host hardware to run software on it. With these tools whole systems can be tested and developers can prove their software to make use of all processors in the system by writing highly parallel applications. With these tools software development and optimization is possible even if the hardware does not exist yet in silicon. Using this technique the hardware dependent software development can start at an earlier design stage which results in time saving as well as better optimized and tested software. However if a physical system has an execution speed of 2000MIPs and is simulated at the gate-level, the virtual hardware speed decreases to 0,002MIPS and at the RT-level to 0,2MIPS. To avoid this speed bottlenecks power and performance emulation on hardware prototyping platforms has been introduced and is used in this thesis [50].

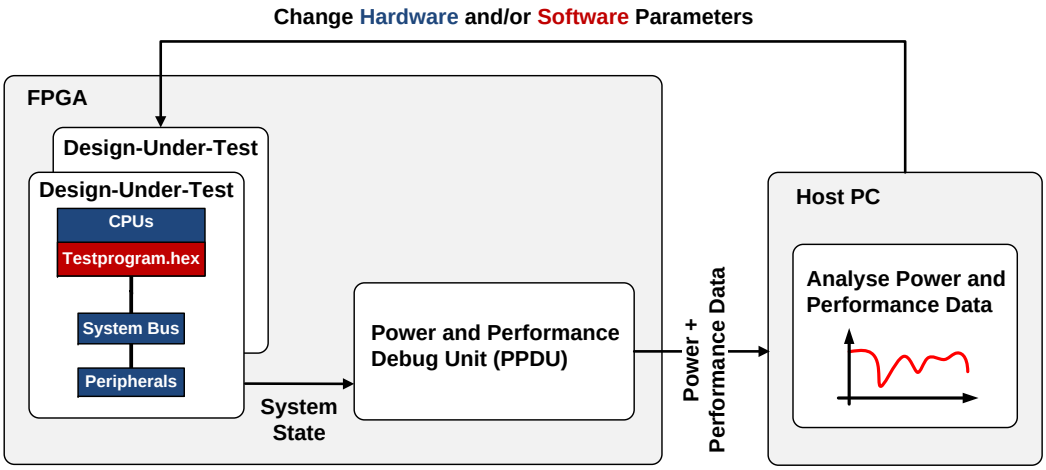


Figure 1.3: Design Space Exploration

1.2 Main Goals of This Thesis

Previously defined requirements have to be kept in mind during the design phase of a SoC. These include performance, time to market, power and thermal issues. In this work we present the design and implementation of a power consumption and performance evaluation platform for multi-core systems. The main goal of this thesis, which is part of the POWERHOUSE [1] project, is creating a generic, noninvasive and transparent power and performance evaluation platform. In detail our general approach and design should be valid for different field programmable gate array (FPGA) boards and different multi-core systems. Furthermore it shall be adaptable for any number of cores and should have no impact on the run-time or estimated power consumption on the FPGA board. The power estimation calculations shall be only dependent on the device-under-test. The last main goal is that power and performance statistics should be observable from outside of the FPGA.

The design space exploration, possible with our platform, is shown in Figure 1.3. With this platform it will be possible to change hardware and software parameters during the design process and acquire fast feedback on the impact of these changes. We are using the power emulation technique on an FPGA onto which we synthesize a multi-core system. The consumed power statistics are estimated during run-time from the switching activity of different signal states of the design. With this technique we obtain power data during run-time for every component which is up to 100 times faster than using software simulators. Performance statistics like cache misses and register write/read accesses for every CPU are also collected with sniffer units in hardware. The collected power and performance statistics are sent by the Ethernet chip on the FPGA board to a host computer. Evaluation software running on the host visualizes the received power and performance data to get a fast overview of the bottlenecks of the current hard- and software design. A big advantage

[1] POWER-aware, Hardware-supported Operating system and Ubiquitous application Software development Environment, funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 815193.

of this platform is that developers at the operating system (OS) level get early feedback about performance or power improvement potential. Before the first tape-out is physically produced, developers can optimize software for the used hardware. Also different strategies for a power-aware OS could be tested when the OS has access to the performance and power statistics during run-time. Testing strategies for power-aware thread scheduling or the compliance to maximum power limits for energy harvesting devices is easily possible even with long test run-times at the OS-level.

1.3 Structure of This Work

In Chapter 2 the power consumption fundamentals of CMOS circuits are explained. Furthermore a general overview of power and performance estimation techniques on different abstraction levels is given. Chapter 3 presents the design overview of the power and performance evaluation platform. Design flows show how the power and performance estimation units are created. The implementation is presented in Chapter 4 based on the case study of a LEON3 multi-core system. Also, an overview of used tools and of the analysis software running on a host PC is given. The usage of the profiling architecture for SW and OS benchmarking as well as optimization is shown in Chapter 5. In Chapter 6 a conclusion of this thesis and ideas for future work are presented.

Chapter 2

Related Work

2.1 Power Consumption in CMOS Circuits

2.1.1 Introduction

In this section the power consumption fundamentals of CMOS circuits and equations for their computation will be presented. The two main sources are [49]:

- Dynamic power consumption
- Static power consumption

2.1.2 Dynamic Power Consumption

2.1.2.1 Charging Capacitance

A predominant part of CMOS power consumption is the dynamic consumption. Dynamic refers here to the charging and discharging process of capacitances. Always when a capacitance is charged over a resistor the same amount of energy that can be stored in the electric field of the capacitance is converted into thermal energy.

$$E_{field} = \frac{1}{2} \cdot C_k \cdot V_{dd}^2 \tag{2.1}$$

Figure 2.1 shows how the current flows during the charge and discharge process of the capacitance, equal to one clock cycle of a CMOS inverter. During the charge process the energy of Equation 2.1 is loaded into the the capacitance and the same amount is converted into heat. If the discharge happens the stored energy in the electric field of the capacitance is also converted to thermal energy. In the end we have an energy consumption of:

$$E_{clock} = 2 \cdot E_{field} = C_k \cdot V_{dd}^2 \tag{2.2}$$

The equation for the dynamic power consumption is now:

$$P_{dyn} = C_k \cdot V_{dd}^2 \cdot \alpha \cdot f_{clock} \tag{2.3}$$

C_k is the sum of different capacitance influences as defined by the characteristics of the used transistors, wires and interconnects. V_{dd} is the voltage of the power source which

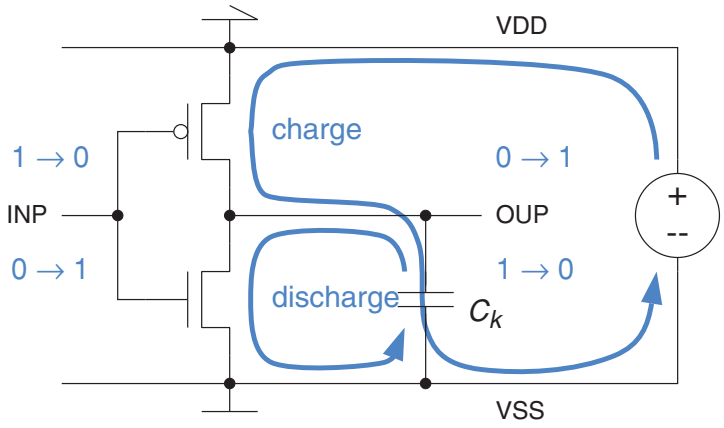


Figure 2.1: Dynamic Power Consumption [32]

has a big influence on the consumed power because of its quadratic influence. During the minimization of the semiconductor manufacturing process this voltage is getting smaller and reduces the influence of the dynamic power consumption. α is the activity factor which lies between 0 and 1. A strategy for reducing this linear factor is for example clock gating which uncouples the clock tree of circuit parts when they are not in use. f is the clock speed and has also a linear influence on the power consumption. If f is increased also V_{dd} must be raised by the designers. If this is not done the circuit could get into an undefined state. This is because a higher V_{dd} voltage can charge capacitances in the circuit faster than a low V_{dd} voltage and reduces thereby the delay time until a steady state is reached [32, 49].

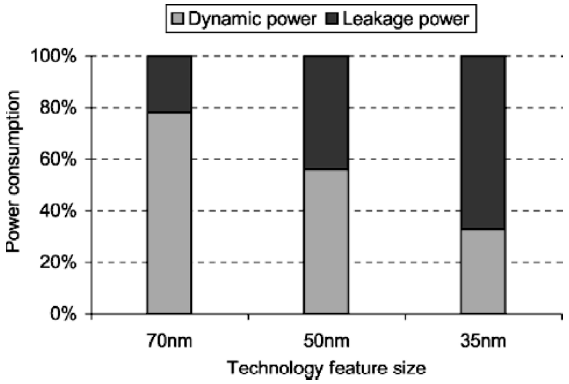
2.1.2.2 Short Circuit

Another main factor for the power consumption is the short circuit current. This occurs when in a CMOS gate the transistor switch to the other state and a conductive connection between V_{dd} and V_{ss} occurs. The Equation 2.4 is a model for this where β depends on the characteristic of the transistor and τ is the delay of the gate [49, 40].

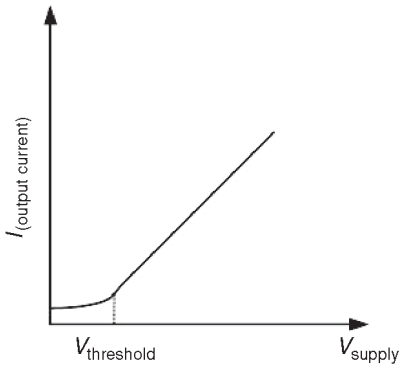
$$P_{sc} = \frac{\beta}{12} \cdot (V_{dd} - 2 \cdot V_{th})^3 \cdot \frac{\tau}{T} \tag{2.4}$$

2.1.2.3 Glitches

If the result of a gate depends on different sub results unnecessary transitional switching of the outputs of the gates could occur during transitions because of different delay times of the logic signals. This additional switching activity consumes more power and must be included into an estimation of the total consumed power [40].



(a) Trends for Power Consumption [49]



(b) Semiconductor Diode as Switch [49]

2.1.3 Static Power Consumption

2.1.3.1 Leakage Power Consumption

In the last decades the dynamic power consumption was one of the main factors for the power consumption in CMOS circuits. Nowadays the leakage consumption is rising and accounts up to 20% - 40% of the total power consumption. In the future this factor becomes even more important for energy-aware circuit design as shown in Figure 2.2(a). A transistor does not resemble an ideal switch, meaning that when it is open it does not exhibit infinite resistance and when it is closed the resistance is not zero. The *leakage power consumption* comes from this non-ideal behavior of CMOS transistors. As shown in Figure 2.2(b) a diode exhibits a leakage current even if the threshold voltage is not reached. The total leakage power is calculated with the following equation:

$$P_{leak} = V_{dd} \cdot I_{leak} \tag{2.5}$$

The total leakage current I_{leak} has different sources. The two most important factors are the subthreshold leakage I_{dsub} which can be exponentially decreased with a rising threshold voltage V_T and the gate leakage current I_{gtun} where electrons can pass through the gate isolator because of quantum mechanic tunneling effects. The leakage power P_{dsub} can be approximated by Equation 2.6. I_s, n, W and L are technology and device parameters, V_t is the thermal voltage and V_{th} is the threshold voltage [49, 54].

$$P_{dsub} = V_{dd} \cdot I_{dsub} = I_s \cdot \frac{W}{L} \cdot V_{dd} \cdot e^{\frac{-V_{th}}{n \cdot V_t}} \tag{2.6}$$

2.2 Introduction to Power and Performance Profiling

2.2.1 Overview

An overview of different techniques for obtaining power and performance statistics from an System-on-Chip (SoC) is shown in Figure 2.2. *Performance statistics* can be measured on real hardware through performance counters implemented on most modern processors described in Section 2.3.4.3 or with the help of software simulations. Performance statistics

correlate with the consumed power of a SoC. Thus synergies between power and performance estimation architectures occur. This can be seen in [16] where the authors present a profiling architecture with self defined performance counters. In their next work [6] they expand the architecture with a power and thermal estimation platform based on these performance statistics. From this point of view performance counters can be also used for power estimation techniques. Based on this power information thermal estimation can be performed.

The *power consumption* of a system can be estimated directly by measuring the power supply. This method is presented in Section 2.3.2. Also software simulations on different abstraction levels such as system-level, gate-level and register-transfer-level are feasible and are discussed in Section 2.3.3. Unfortunately, software simulations exhibit long execution times. Therefore hardware-accelerated power estimation was initially introduced. It can be performed with a co-processor as described in Section 2.3.4.2 or with the help of an FPGA, which is called power emulation, as discussed in Section 2.3.4.4. Also a combination of software and hardware-accelerated power estimation exists which is introduced in Section 2.3.5.

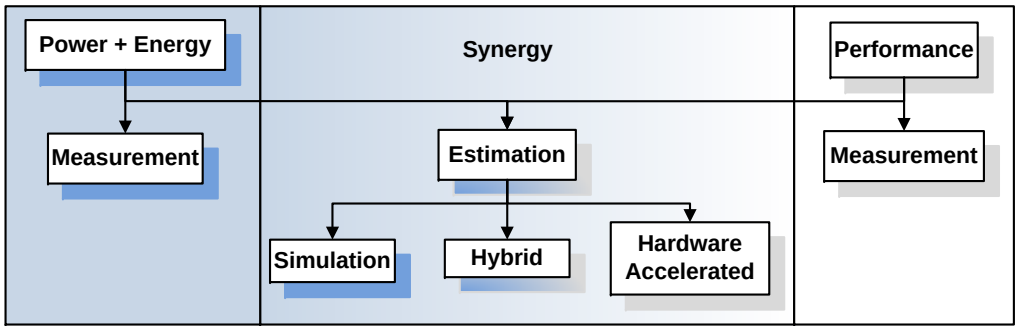


Figure 2.2: Classification of Power/Performance Estimation Techniques

2.2.2 General Power Estimation

2.2.2.1 Analysis Phase

The generic approach for obtaining power information using power estimation techniques at various levels of abstraction is shown in Figure 2.3. The upper part of the sketch is the analysis process that has to be undertaken if not all components and devices are yet known or defined. For the initial power estimation the architecture and the floorplan must be estimated. Furthermore, the signal activity of the components has to be estimated or recorded by executing a benchmark. In general there are two intentions behind power estimation which have different requirements for the estimation phase:

1. Absolute power estimation
2. Relative power estimation

Absolute Power Estimation In the first case we want to obtain a highly accurate power estimate of the final implementation already during the design phase. For example

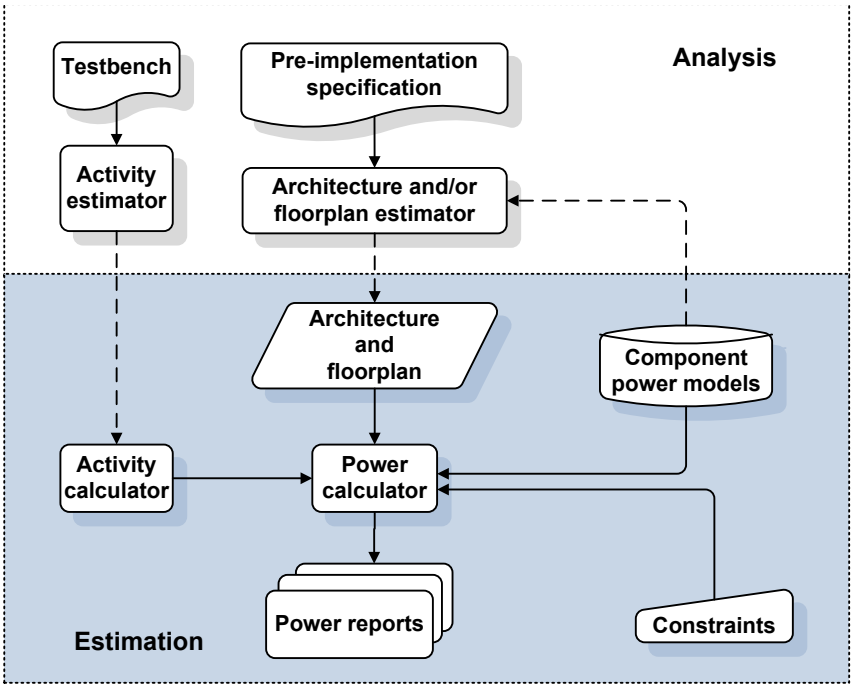


Figure 2.3: Generic Power Estimation Tool Flow [40, with modifications]

when designing a digital circuit with an energy harvesting power source not all energy harvesting devices could serve the current that is needed. Furthermore power spikes during the run-time could lead to supply voltage drops and an undefined behavior of the circuit. In this case an energy buffer must be inserted into the design, leading to extra costs and perhaps leading also to different design decisions. Hence at an early stage of the design phase the design space is narrowed down, leading to a reduction of redesigns and an earlier completion of the final product [40].

Relative Power Estimation The second intention behind power estimation is the comparison of different design solutions in terms of power consumption. This could be for example the implementation of a sorting algorithm. If you look at the potential solutions as black boxes, all algorithms will deliver the same results but their run-time and their memory usage may vary. In this case a comparison of absolute power values is not important. It is much more important for a design decision to know that algorithm A needs 50% more power, time or RAM space than algorithm B [40].

2.2.2.2 Estimation Phase

In the lower part of Figure 2.3 the power estimation process is performed. The component power models include the equations of Section 2.1 on different abstraction levels. For example C_k in Equation 2.3 could be the total capacitance for all gates inside a system-level module such as a CPU. Other inputs for the power calculation are constraints such as the supply voltage V_{dd} or the clock frequency f. Also the activity of the components during the execution of the testbench is estimated. With the help of all this information,

the architecture and floorplan specifications it is possible to calculate power reports for each component or the whole system. The power estimation flow at different abstraction levels is shown in Figure 2.4 [40].

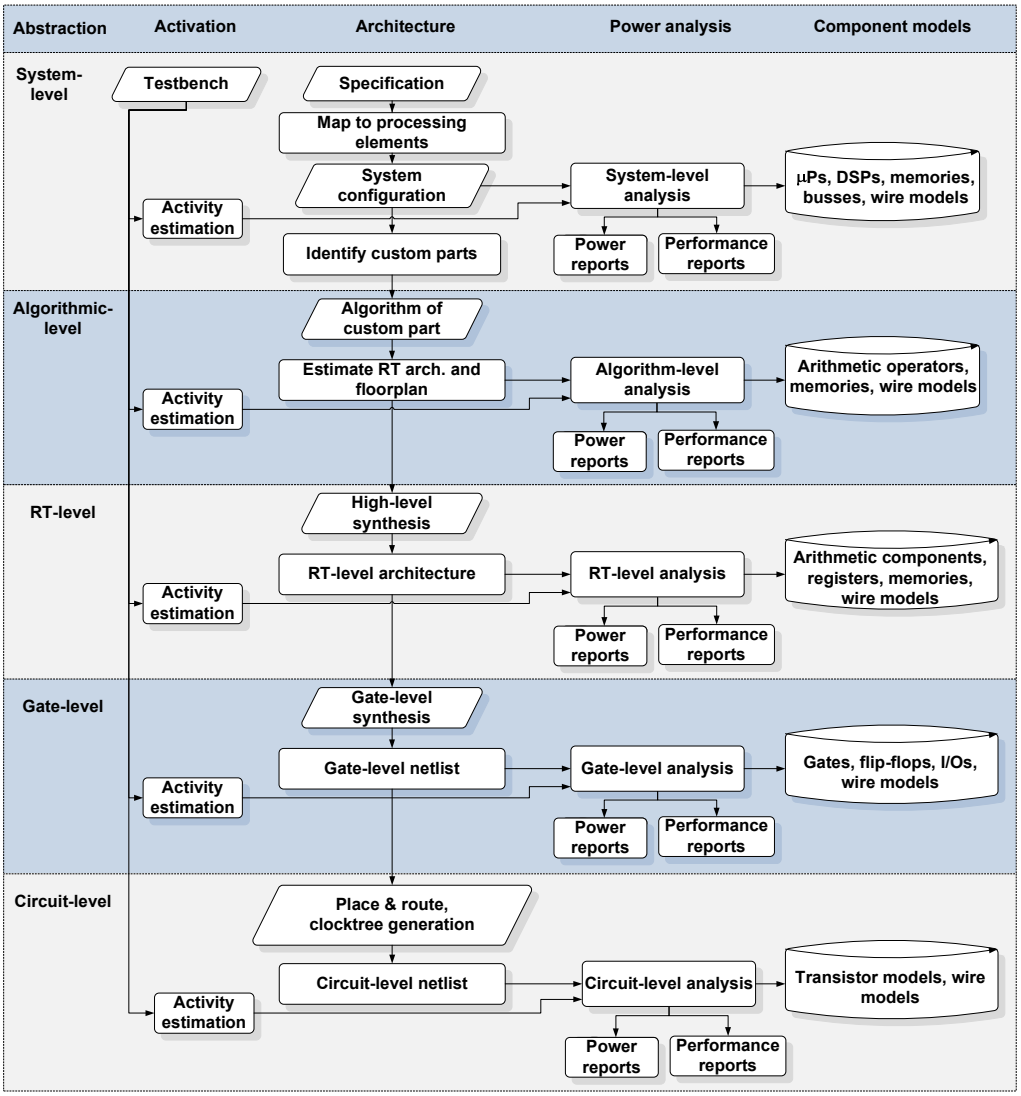
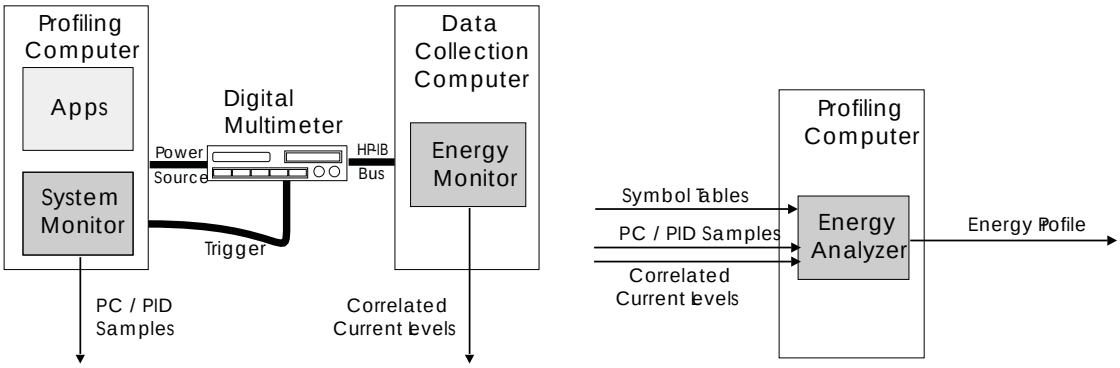


Figure 2.4: Generic Design Flow Including Power Estimation [40, with modifications]

2.3 Power Profiling

2.3.1 Introduction

With power estimation it is possible to explore the power-aware design space during the development of a new hardware platform. For different design problems the best fitting solution can be found during a hardware/software co-design approach. Calculations and logic conditions are partitioned on different modules and decisions can be made on whether



(a) Data Collection (b) Off-Line Analysis

Figure 2.5: PowerScope Architecture [19]

functionality is realized in software or in hardware by individually designed electronic circuits.

2.3.2 Power Measurement

Power consumption measurement rely on an already available silicon implementation of a SoC. In [19] this technique has been applied on a standard personal computer to get information about the total run-time and energy consumption of processes during a benchmark run. The introduced method is so generic that it could be easily applied to SoC designs. In Figure 2.5 an overview of the PowerScope architecture is shown. On a profiling computer an operating system with a modified kernel and the test bench is running. The kernel modifications implement a system monitor writing data about the current program counter (PC), process identifier (PID) and other information into a buffer on the hard disc of the profiling computer when a trigger impulse from the digital multimeter is received. This trigger impulse is always activated when a new power measurement happens and the power data is then transmitted to the energy monitor on the data collection computer. The energy analysis tool combines power information and the correlated program information such as PC and PID as well as run-time statistics for every process during the benchmark. Unfortunately some drawbacks exist when performing power measurements [19]:

- The chip-under-test has to physically exist before a first power measurement is possible. Thus it is an unsuitable method for exploring the power design space during the development phase.
- Automatize power measurement has been developed for optimizing the power consumption of software and not hardware.
- It is not possible to obtain information about subcomponents in the design. Only the power information of the entire chip is available.
- Because of the additional interrupts and their evaluation by the profiling computer during a benchmark the time and power behavior of the original design changes.

- The system-under-test has no run-time access to the power statistics. Therefore run-time power-aware software (e.g., run-time power management) is not possible [19].

2.3.3 Simulation

Power simulations can be performed at different levels of abstraction as shown in Figure 2.4. In this section power simulation techniques at various levels of abstraction will be presented.

First, simulations at the system-level are discussed. Second, the algorithmic-level which consists of source-code-level, instruction-level and functional-bus-model-level is presented. Finally, the RT-level, gate-level and the lowest abstraction level which is the transistor-level are discussed.

The reason for starting the power design space exploration at high abstraction levels is that power saving techniques are very effective on higher levels. The power saving options are less effective closer to the physical-level. Furthermore power optimizations during late design phases will lead to significant time to market delays and high costs. It is expected that the energy saving potential on the behavioral-level and architectural-level will increase from 40% to expected 80% in the year 2015 as shown in Figure 1.2. Design optimizations on the physical-level will lead to little power savings of 10% in the future [44].

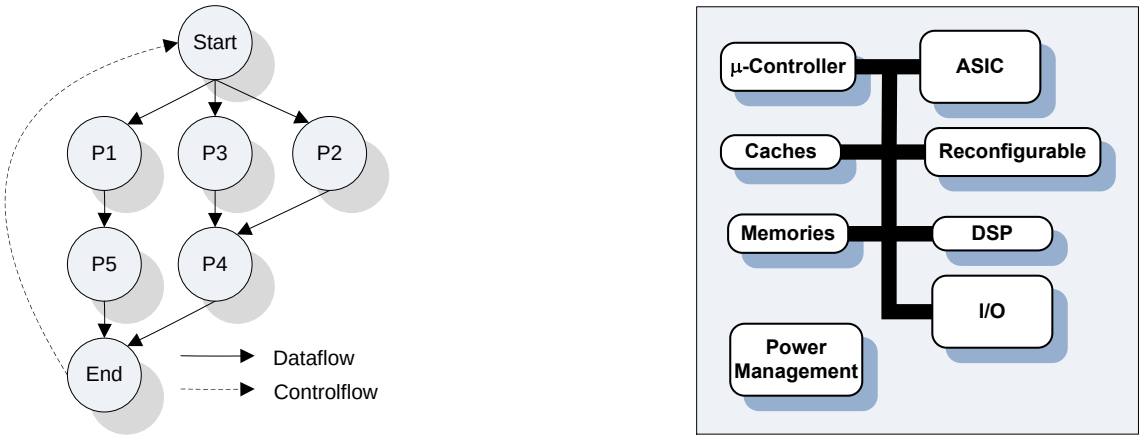
2.3.3.1 System-Level

At the system-level it is important to know the dependencies of different tasks for a given application. These could be shown with the help of a task graph like in Figure 2.6(a) where the data and the control flow is visualized at the system-level. Data dependencies can be easily extracted with this technique. For example process two and three have to be finished before task four can start. Another useful insight is that process one, two and three can be executed in parallel because they do not share resources [40].

The specification can later be mapped to a system-level architecture where main components like controllers or memories are used as shown in Figure 2.6(b). First power management techniques can be developed at this abstraction level. It is important for the overall power consumption that at the system-level a well power optimized architecture is chosen from the available design space. Therefore great care must be taken of the partitioning of tasks into hardware and software components to meet speed, power consumption or space requirements [40].

2.3.3.2 Algorithmic-Level

At the algorithmic-level the tasks during power optimization are the partitioning of algorithms in software or on user specific hardware. The software can be enhanced by optimizing the control flow to gain a speed up in execution time. For example loop unrolling strategies can be employed. For software exploration the algorithmic-level can be split into the following three sub levels [40]:



(a) Task Graph

(b) System-Level Architecture

Figure 2.6: System-Level Architecture and Task Graph [40, with modifications]

- Source-code-level
- Instruction-level
- Functional-bus-model-level

Source-Code-Level Within the source-code-level a fast first power estimation of specific algorithms can be undertaken. The consumed energy E_{total} is calculated by adding the needed execution time or clock cycles for every instruction and multiplying them with the average power consumption E_{proc} of the processor for the given clock frequency. Equation 2.7 shows this fast approach [40].

$$E_{total} = T_{execution} \cdot E_{proc} \cdot f \tag{2.7}$$

Instruction-Level Instruction set simulators (ISS) can calculate the current power consumption on the basis of executed instructions and used operands. The power model maps every instruction to a specific energy value which was created before with characterization benchmarks and measurements of the power supply [37, 53] or by power estimation software tools at the circuit, gate or register-transfer-level [39].
Power Models From Power Supply Measurements: Initial works focused at creating an instruction-level power model to obtain the power costs of any test program [53]. The power model is created by writing characterization programs where the instruction-under-test (IUD) is repeated over and over again. With the current measured by a digital ammeter, the specific power, which is the so called base power cost for the IUD, could be calculated. In real programs the power consumption of different instructions depends on their operands and the changes in the circuit, the so called circuit state overhead costs. The greater the changes of the logical states inside of a processor between the current and the next instruction, more energy will be needed to get to the logical state of the new instruction. Also different cache misses or pipeline stalls have an impact on the real power consumption. The authors of the paper propose to write benchmarking programs

for all these cases. If a program has to be power estimated using this model, first the base costs of all instructions are multiplied with the number of times the instructions are executed. Afterwards the inter-instruction effects like the circuit state overhead costs are added for each pair of consecutive instructions, multiplied with the number of times this pair occurred during the program execution [53].
Power Models From Software Power Simulators: In [39] the power information for creating the power model is derived from a gate-level power simulation. The system-under-test is the LEON3 processor which is also used in our work, but only implemented as a single-core. Research was also conducted on the correlation of register accesses and data switching with power.

To estimate the energy consumed for a specific instruction the same technique as in [53] was used to create two different power models. For the one-instruction-based model the IUT was embedded into a sequence of 10 NOP instructions. This procedure has been repeated 100 times. The reason for including the additional NOP instructions is that the integer pipeline should exhibit no other activity than executing the IUT. A test sequence for each instruction can be created in the following way [39]:

$$100 \cdot (5 \cdot NOP, IUT, 5 \cdot NOP) \tag{2.8}$$

The two-instruction-based power model tests around 6000 possible pairs of instructions to get more accurate estimations. The sequence looks like this:

$$100 \cdot (5 \cdot NOP, IUT_1, IUT_2, 5 \cdot NOP) \tag{2.9}$$

Further research was conducted about the influence of data switching activity on the *add* instruction. Therefore a test bench was written where every cycle more and more switching activity at the operands of the instruction $reg_d = reg_{s1} + reg_{s2}$ occurs. The result was a strong correlation between switching bits and the consumed energy. If all 32 bits change, the power consumption is about three times higher than in the case of no bit toggles shown in Figure 2.7. Also the impact of register correlation on an instruction pair has been tested. Register correlation means that a register occurs as an operand in the two tested instructions. An example where a high increment of the consumed power could be detected is the $ld - add$ pair with the following test operands:

$$ld\ operand \rightarrow \mathbf{reg3}$$
$$add\ \mathbf{reg3} + reg4 \rightarrow reg1$$

The *add* instruction uses the same register $reg3$ as input to which the *ld* instruction writes. Because the add instruction can only be executed after the *ld* operation is done the pipeline advantage can not be taken and no parallelization happens. Two additional instruction unit cycles were added in comparison when no register correlation happens. Therefore the total consumed energy has been doubled.

A power model which takes activity and register correlation into account can improve the power estimation. However this improved model vastly increases the calculation. Hence this improvement was not considered in the final LEON3 power model. It was shown that this instruction-level power estimator design reaches a speedup of 1000 times against commercial gate-level power simulators. The accuracy error was within 6% and 12%. The creation of the highly automated one-instruction-based model takes five hours and for the two-instruction-based model 10 days of calculation time on a Xeon processor with 2.5GHz.

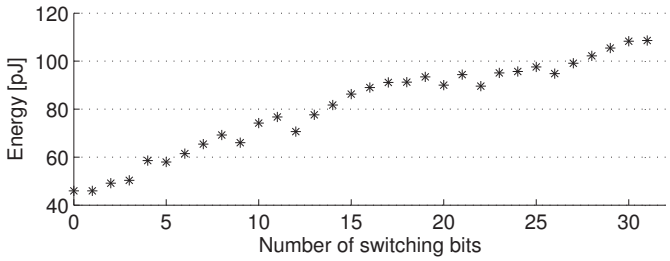


Figure 2.7: Power Consumption vs. Operand Bit Switching [39]

Functional-Bus-Model-Level At this abstraction level the instruction simulator is enhanced with a system bus model. This enables studying the transactions of different data on the bus and finding bottlenecks during program execution. The power model has also been extended to include hierarchical memory architectures to detect cache hit and misses as well as processor specific power models for multi-threading and pipelining. Overall this more accurate power model requires the highest calculation time at the algorithmic-level [40].

2.3.3.3 RT-Level, Gate-Level and Circuit-Level

At the RT-level different commercial tools for power estimation, such as [51], exist. The main parts on this abstraction level are arithmetic components, registers, memories, wire models. The power models for these main parts are created based on macro modeling which is shown in Figure 2.8. First, a set of model parameters $X_i, i = 1, ...n$ are chosen as input for the main part-under-test. Then different input vector pairs are chosen as a training set for the characterization. During the characterization, which could be a gate or circuit-level power simulation, the power values for every training set are calculated. Based on these data the RT-level power model is created. The power model could be built as an equation or as a table based model with $P_{main\_part} = P(X_1, ..., X_n)$ [41].

At the gate-level the power consumption is calculated with power models for logic gates like a NOR-Gate. A commercial power estimation tool at this level of abstraction is Prime Time from Synopsys [52]. The gate-level models can be improved at the circuit-level by calculating for example with transistor and wire models with the Spice simulation program [48]. Power estimation at this level of abstraction has a high estimation accuracy. Every power consumption estimation at a closer level of abstraction to the physical design level requires more calculation time because more complicated power model equations must be calculated.

2.3.4 Hardware-Accelerated

2.3.4.1 Introduction

Software power simulators at low abstraction levels such as the circuit or the gate-level exhibit very long simulation times and are therefore not useful for long benchmark runs. Hence, efficient tests of power saving strategies, like power-aware task switching at the operating-system-level, are not feasible. To derive power information in real-time, hardware-

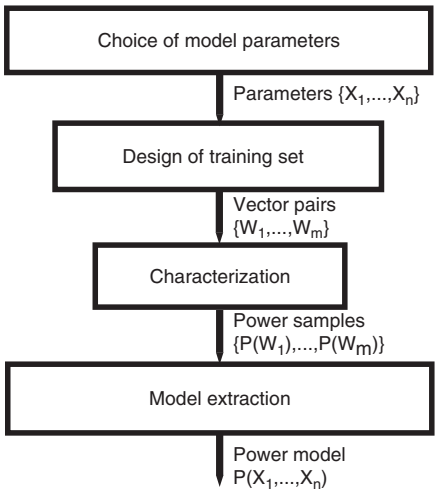


Figure 2.8: Macro Modeling Flow [41]

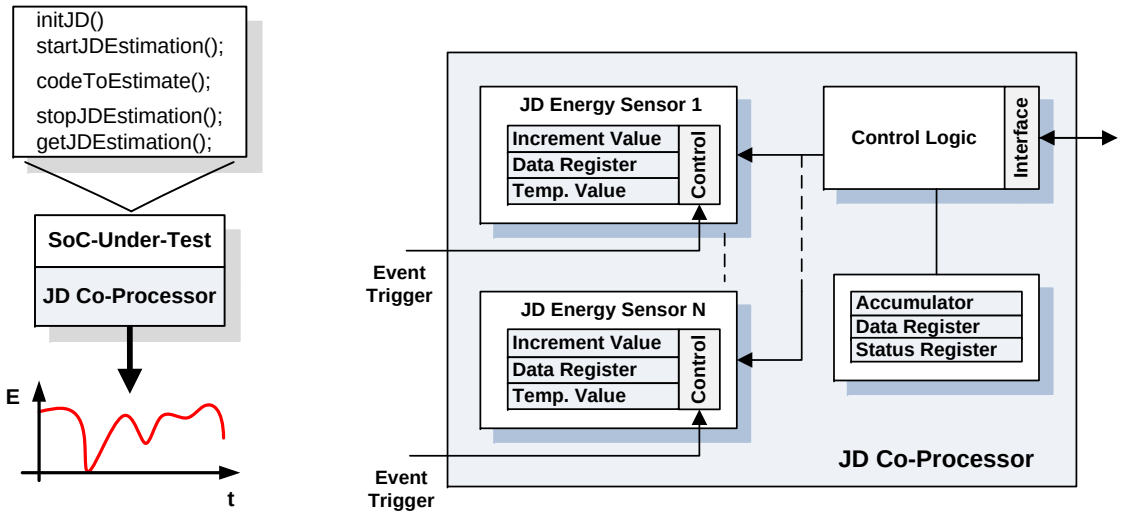
accelerated power estimation was introduced, where the power estimation model, including the device-under-test, is transfered to an ASIC or an FPGA. The estimation model runs at the same clock frequency as the device-under-test. Thus a speedup of up to 1000 times compared to high accuracy simulation techniques can be reached. With this technique software developers can optimize the software at early design stages without needing to wait that the chip is available in silicon.

We will discuss three different types of hardware acceleration in this section. A co-processor can be added to the design, estimating the power consumption based on macro models [25, 26, 38]. The consumed power can be also derived from performance counters [31, 15, 46]. Finally, the power emulation approach is presented [12, 10, 23, 8].

2.3.4.2 Additional Co-Processor

"Joule Doc" In [25, 26] a co-processor is added to the design to enable real-time power analysis and monitoring of the electronic system-under-test. The JouleDoc (JD) co-processor offers configuration and control instructions to obtain data from and to send data to the co-processor. The main idea is shown in Figure 2.9(a) where after the initialization and starting of the co-processor the software code-under-test is being executed. The estimated power sum can then be read out using another instruction [25, 26].

In Figure 2.9(b) the design of the co-processor is shown. The power estimation is based on macro models as introduced in Chapter 2.3.3.3. Power models can be either generated based on data sheets, simulations or test chip measurements. To obtain power values over a long run-time, power values can be summed up during a test run and stored in an accumulator register for later reat out over a controller. If a power model consists of many JD energy sensors, therefore depends on many events, a fine estimation granularity is feasible. If the co-processor should be implemented in silicon a trade-off between estimation accuracy and chip size of the co-processor has to be made.



(a) Power Estimation with JD Co-Processor

(b) Block Diagram

Figure 2.9: JD Co-Processor Overview [25, 26, with modifications].

"Clipper" In [38] the Clipper framework is presented. A processor is enhanced with event counters to enable dynamic power-aware software with the help of macro models. This means that software is able to utilize a trade-off between calculation quality and available energy during run-time. It could be shown that a processor enhanced with the Clipper method consumes only slightly more energy but provides the possibilities for accurate power estimation and power-aware software [38].

An overview of the Clipper method is shown in Figure 2.10. First a gate-level simulation of the processor is performed from which the switching activity is extracted. Based on these results a power analysis is made for the whole processor. To create a power model, events have to be extracted looking at the correlation of control signal changes and power simulations. Control signals with the highest correlation for each component are kept. An optimization step is performed in which control signals with little impact on the power consumption are removed. Afterwards a linear regression is performed to reduce the number of events. Counters, composed of logic gates and flip flops for signal detection, are added to the RT-level design which is synthesized on an FPGA board. The counters are mapped to a memory mapped I/O space to access them during run-time. The circuit area increase is 4.9% and the increased total power consumption is only 3%. The power-aware software can now read out this performance counters and calculate the consumed power, based on the counters and a power model, implemented in software. The authors of this paper decided to make the power model in software because of the overhead of a power model circuit in hardware. The execution of the power calculation in software needs only 232 clock cycles. If this calculation repeats every 100.000 cycles an additional power consumption of only about 0.23% occurs [38].

The Clipper method has been tested with a self written power-aware JPEG encoder software. Dependent on the current power budget the encoding quality of the pictures is adapted between 13 levels [38].

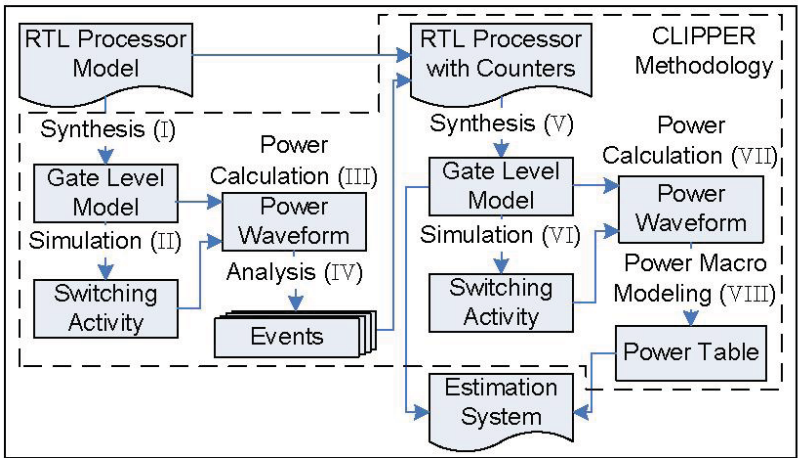


Figure 2.10: Clipper Method Overview [38]

2.3.4.3 Using Existing Performance Counters

Almost all modern microprocessors contain hardware performance counters to count special events during the execution of a program, like cache misses, branch predictions or bus transactions. Research has been conducted on how to obtain overall and per component power information with the help of these counters. A problem in all papers is that the simultaneous access to all counters is not possible during run-time. In [31] a single-core CPU is used to test run-time power estimation. In [15] an off-line calculation is necessary to obtain the power consumption estimate. A power model on a multi-core CPU and with real-time power-aware task scheduling is implemented in [46].

Run-Time Single-Core Power Estimation: In [31] existing performance counters are used to obtain run-time overall and per component power statistics of a CPU. However not for every power relevant event a correlation to existing performance counters exists, therefore heuristic methods are used to estimate the missing performance counters. Depending on the inner structure of the CPU the heuristics combine measurable performance counter data to generate a more accurate power profile [31].

The general approach is presented in Figure 2.11 and was tested on a Pentium Pro running Linux 2.2.16. The scheduler inside of the Linux kernel was modified to log the performance counters of interest during the run-time of the program-under-test. Thus not all relevant performance counters could be read out at the same time, a multiplexing algorithm was added to the scheduler to derive all relevant data. It could be shown that smaller structures of the CPU, which are 24% of the total power, can not be estimated with performance counters or heuristic methods at the Pentium Pro. Generally this work shows the potential of power estimation at existing CPUs without hardware modifications and little power and performance overhead [31].

Off-Line Single-Core Power Estimation: In [15] a linear power model was used to estimate the consumed power of an Intel XScale processor with the help of hardware performance counters. Five performance events with high power correlation were selected and used as

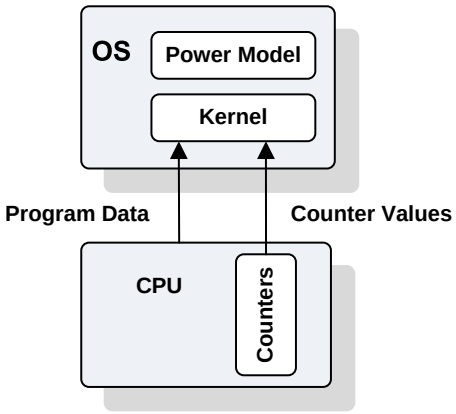


Figure 2.11: General Approach for Performance Counter Based Power Estimation [31, with modifications]

expressed in Equation 2.10 to model the total CPU power $Power_{cpu}$ [15].

$$Power_{cpu} = \alpha_1(E_1) + \alpha_2(E_2) + \alpha_3(E_3) + \alpha_4(E_4) + \alpha_5(E_5) + K_{cpu} \tag{2.10}$$

E_1 counts the number of instruction fetch misses, E_2 counts data dependencies, E_3 represents transaction lookaside buffer (TLB) misses, E_4 counts instruction TLB misses, E_5 represents the number of executed instructions and K_{cpu} represents the constant power value for the idle CPU. The linear parameters α_1, α_2, α_3, α_4 and α_5 are determined by various benchmarks and the use of linear algebra to minimize the estimation error. Multiple runs are necessary to calculate the so called power weights α_1 to α_5 because during the run-time of one benchmark only two counters can be read out. With this linear model an easy adaption to different voltages or frequency configurations is possible by modifying the power weights of the power model. The authors of this paper could show that the average error is within 4% compared to real hardware measurements. To access the performance counters during run-time the timer interrupt routines of the OS were modified to store the counter values into a buffer every 10ms. The overall performance overhead with the added code is only 2% and the difference to the power consumption of an unmodified OS is less than 1% [15].

Research was also made for external memory power estimation. Due to the fact that no performance counter directly shows memory transitions, a linear memory power model has been developed. This model depends on instruction cache misses, data dependencies and a constant idle factor to estimate the consumed SDRAM power. The results for this memory estimation shows an average error of 70% and can be explained by the lack of performance counters which indicate memory accesses directly.

In this work, compared to [31], no power-aware OS techniques are feasible during run-time but the accuracy of the power estimation is much higher. Furthermore no hardware details must be known because of the general usability of the linear power model for different CPUs [15].

Real-Time Multi-Core Power Estimation: In [46] a power estimation on a chip multi-processor (CMP) AMD Phenom CPU was performed. The main task was to perform run-time power measurements and calculate the chip power consumption of different running

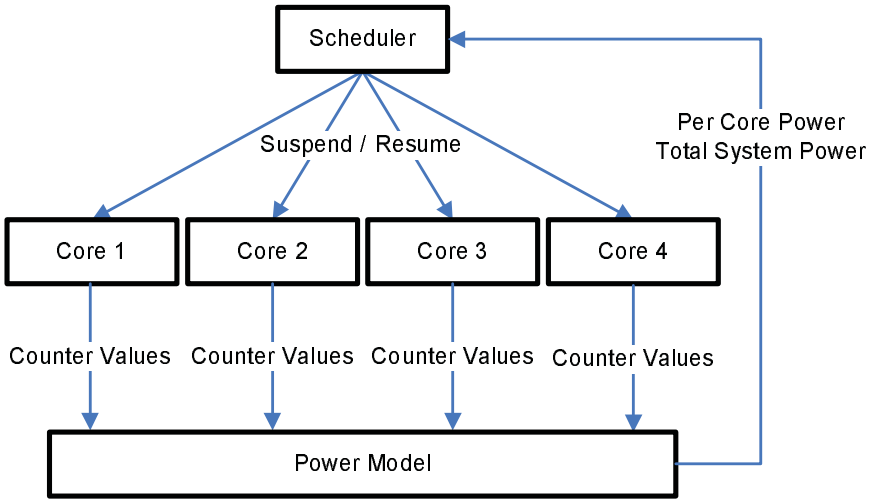


Figure 2.12: Power-Aware Scheduler [46]

software threads based on power models. The thread power information is used to perform thread scheduling. Compared to [15] no off-line power computation is necessary. For creating the power models four power counters were selected. The amount of usable counters is limited because the AMD Phenom is not able to read out more than four of them at the same time. Micro benchmarks were written to obtain data on their activity and the correlation to power. After this a piece-wise linear power model was created with the help of linear regression. Also the impact of temperature on the power consumption was researched. The result shows that run-time temperature knowledge would lead to more accurate models because of the increasing leakage current with temperature rises. The impact was estimated to be an up to 10% higher power consumption. Due to the lack of temperature per core sensors this feature was not built into the power model. To test their per core power estimation a power-aware thread scheduler was written in C. On the basis of a given power budget the scheduler suspends and resumes different threads on the CPU with the help of the run-time power estimation shown in Figure 2.12. The overall result was that the scheduler worked well for different thread programs and could meet the given power envelope. The power estimation average error was reported to be below 10% [46].

2.3.4.4 Power Emulation

Power emulation was introduced to speed up power estimation by using hardware prototyping platforms. The main idea is to implement power estimation in hardware and benefit from the amenities of parallel calculation and high clock rates in hardware. Compared to software simulations a high power estimation speedup during the design of a new electronic circuit is possible with power emulation. Tests of power-aware software or the estimation of power consumption during the boot sequence of an OS are possible. Research was also done to decrease the additional hardware overhead of power emulation on the circuit-under-test like in [12]. In contrast to performance counters the chip does not have to exist in silicon and therefore power emulation is a very good instrument for

fast and accurate power estimation during the design exploration.

RTL Power Emulation Power emulation at the RT-level was first introduced in [13]. This work has been advanced by the authors in [12] where some techniques were introduced to decrease the high hardware overhead of power emulation at this relative low abstraction level. In Figure 2.13 the additional components needed for power emulation are shown. Every component of the original design is extended with a power model by connecting the input signals of the RT-level components to the input of the power model. To synchronize the power estimate creation of all power models the power strobe generator has been inserted. Now the power aggregator sums up all estimated power values to a total power estimate [12, 13].

The additional emulation circuits increase the original chip area by 18.2 times. Without area optimization techniques, power emulation at the RT-level would be not applicable to large designs like MPSoCs on a single FPGA board. To reduce this area consumption, the authors of the paper advocated to combine several RT-level components to one power model. Thus many power models are replaced by one simplified power model which leads to less accuracy and a change of component granularity. Generally, every area reduction method decreases the level of accuracy. Therefore a tradeoff has to be found. Another area saving potential is the power correlation of components which means that the power of component P_X can be transformed by the function f to another power model P_Y with the equation $P_Y = f(P_X)$. If the function f needs less area space than the power model P_Y, a reduction of chip area occurs. Also the reuse of a power model for several components is possible. A multiplexer at the input of the power model could select every clock another component for power estimation. Other resource sharing techniques are feasible like using a power estimate over more clock cycles leading to a lower sampling rate of the power model. All this area saving techniques lead to a reduction of the emulation chip size from 18.2 times to an average of 3.1 times [12].

The general power emulation design flow can be seen in Figure 2.14. The RT-level design-under-test is enhanced with the power model library. After an area and power estimation optimization phase the design-under-test is synthesized to a hardware prototyping platform. During the execution of the testbench a power profile is created by summing up all power model results. In the end it can be shown that power emulation can achieve an average speedup of over 500 times and an estimation error of 3.4% with an area increase of 3.1 times [12].

High Level Power Emulation In [10] power emulation was used by adding self-defined activity counters into a multi-core LEON3 system-under-test. These counters, read out during a timer interrupt of the OS, feed a power model implemented in software. The power information is used for power profiling and a power-aware scheduling algorithm to prevent hot spots. In Figure 2.15 an overview of the architecture is shown. To avoid large area consumption by the power emulation technique like [12] the authors of this paper use the system-level. Power models for the integer pipeline, integer register file, caches and AHB bus controller were created and 36 counters which identify events in those units were added. This approach used only 3% of the available look up tables (LUT) of the used FPGA. The counters were mapped into the memory space to calculate the consumed power with Equation 2.11 for each component by means of software.

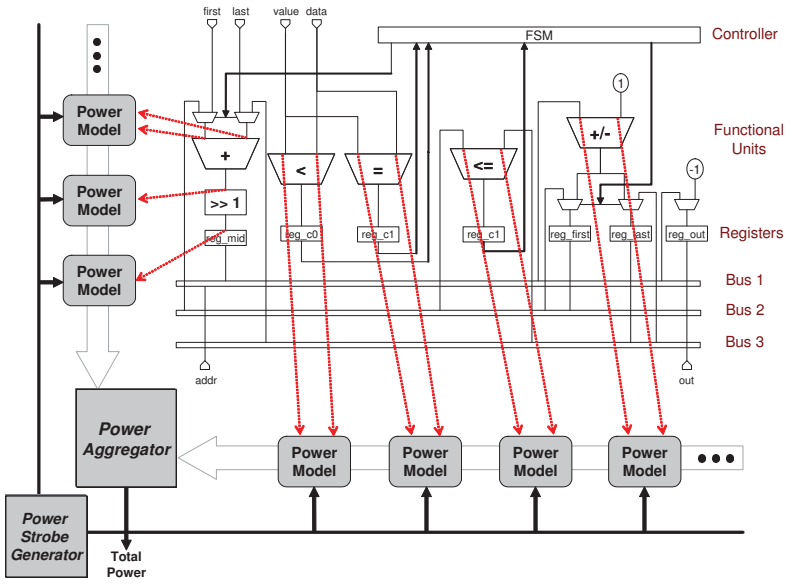


Figure 2.13: RTL Design Extended With Power Emulation [12]

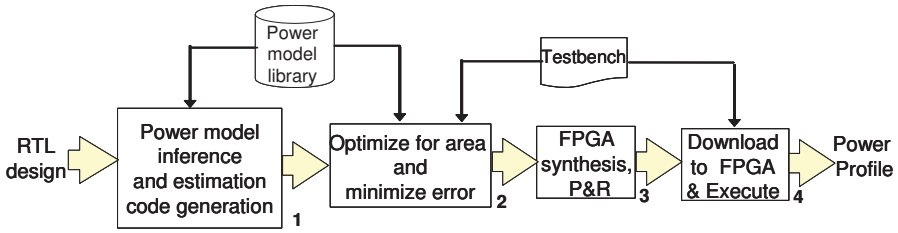


Figure 2.14: Power Emulation Design Flow [12]

$$P_{component} = P_{idle} + \sum_{i} \frac{(E_i \cdot n_i \cdot f)}{cycles} \tag{2.11}$$

The total power $P_{component}$ is the idle power P_{idle} plus the sum of the energy per events E_i multiplied with the event counter n_i and the clock frequency f divided by the clock *cycles*. The energy per events E_i were extracted by running micro-benchmarks in a gate-level power estimation and calculating the correlation factor between counter activity and component power consumption. To obtain a run-time power profile the Linux kernel has been modified so that during the timer interrupt, i.e., every 10ms, the read out of the counters and the power calculation based on the power models takes place. The power values are stored in a buffer and can be transmitted over a RS-232 interface to a host PC. To reset or start the counters during the program execution special commands were added to the instruction set of the LEON3 core. The result of the power emulation profiling compared with the gate-level simulation exhibits an estimation error of 10%. To test power-aware task scheduling on a multi-core system the kernel was modified so that new power values are only calculated every 100ms. It could be shown that task migration

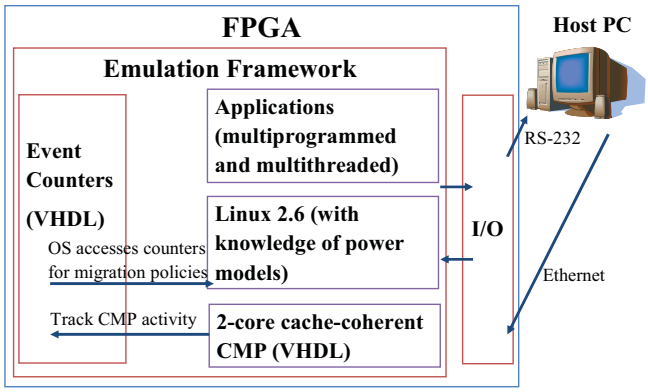


Figure 2.15: System-Level Power Emulation Architecture [10]

based on the consumed power works well with this power emulation technique. The speedup against architecture-level software simulators is up to 35 times. A disadvantage of this work is that it is not cycle-accurate because a new power value is calulated only every 10ms. In general this paper is a further development of performance counter power estimation discussed in Section 2.3.4.3. Self defined performance counters were added into the design-under-test which makes this work usable for power estimation at early design stages [10].

In [23] a power emulation architecture is introduced with its design flow, which makes real-time power profiling possible. Also the power estimation unit is explained which is based on a power model implemented in hardware on an FPGA prototyping platform. In a case study power-critical Smart Card applications have been profiled and an relative error less than 10% could be shown. In [8] research has been done how this power estimation unit can be characterized to get an accurate power model. Also a software development flow is introduced which different tools have to be used to get a power and functional trace for designers at early design stages [23, 8].

2.3.5 Hybrid Power Estimation

In [24] hybrid power estimation is introduced which combines software simulation and hardware-accelerated power emulation to get a fast and accurate power consumption feedback. The reason to combine these two techniques is that hardware components which can be synthesized onto an FPGA are not always available at early design stages. On the other hand power emulation for big designs needs large and therefore expensive FPGAs to map the hardware description code onto them. For example at the RT-level power emulation increases the chip area up to 3 times compared to the original design. Power estimation using software simulators lacks the necessary simulation speed for fast design exploration. This process can be accelerated by means of power emulation [24, 12].

An overview of hybrid power estimation is shown in Figure 2.16. Functional models and their corresponding power models are fragmented onto an FPGA board and a software simulation environment on a host PC. For example, the component A executes its functional model and its power model at the host PC. In contrast the whole component C is emulated on the FPGA board and component B has the power model on the host PC

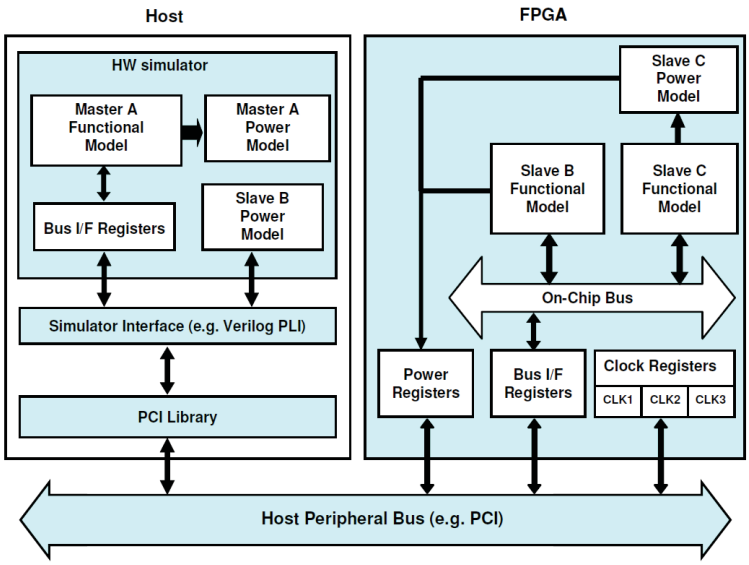


Figure 2.16: Hybrid Power Estimation Architecture [24]

and the functional model on the FPGA board. For estimating power data all functional models have to commit their current states to the power models. For this reason the communication between host PC and FPGA board is established over the host peripheral interface bus (e.g., PCI). Also the exchange of the calculated power values per component to the software aggregator, which sums up all estimated power values, is made over this bus. On the FPGA board, clock registers control the emulation clock frequency of different components and can be set by the host PC. In the cycle based mode communication between the host PC and the FPGA board is performed every simulated and emulated clock cycle. This leads to a big communication overhead which is the main problem in hybrid power estimation. This overhead can be reduced by adding data compression components to the PCI interfaces. Another reduction possibility is that the communication is only undertaken when an important event occurs in the design, like a transaction to a component, which results in a state change of the respective component. Only when such a state change happens the power model has to be updated instead of every cycle. In an implementation of the frame-work a speedup between 56 and 332 times and no loss of accuracy could be shown in comparison to system-level power simulations [24].

2.4 Performance Profiling

2.4.1 Introduction

The reasons for system designers to require performance statistics during the design process are manifold. First, software architects want to find speed and other bottle necks in software programs. This could be for example extensive I-cache or D-cache misses or too heavy usage of internal bus with too little bandwidth. With the help of performance statistics the software designer can optimize the software for the given platform. If

these tests are used early during design exploration of a new architecture, the hardware designer has opportunities to optimize the hardware for the given software application. At high abstraction levels these statistics can be extracted through C or C++ models. Transaction-level models in SystemC are also available with more accuracy but less speed. Furthermore, cycle-accurate frameworks in SystemC, VHDL or Verilog can be used. All these simulation techniques exhibit the general drawback that they do not offer real-time simulation speed which would be necessary for testing programs at the OS-level. Real-time performance estimation is possible by reading out the hardware performance counters of a given processor. This technique has been used in Section 2.3.4.3 where the state of the system-under-test is derived from performance counters to generate a power profile. Also hardware-accelerated performance estimation with an FPGA board is possible like in Section 2.3.4.4 where performance counters are added to the design. Hardware-accelerated power and performance extraction systems can take advantage of strong synergies because power models can use the performance statistics as input to generate power estimation statistics [16, 11].

2.4.2 Hardware-Accelerated

Off-Line Performance Statistic Access In [16] a design framework for architectural cycle-accurate MPSoC design exploration on an FPGA board is presented. The performance statistic evaluation is done on a host PC running a graphical user interface. With this framework it is possible to evaluate different hardware configurations with varying processor cores, memory architectures and interconnection mechanisms [16].

An overview of this framework is shown in Figure 2.17. Every subsystem is composed of the three main modules which are the processing core, memory subsystem and interconnection mechanism. During the design exploration any processor can be used because only the public instruction set processing part is required. One of the main advantages of this architecture is the virtual platform clock manager (VPCM) which generates the clocks for the cores and the memories of different sub systems. With the help of this feature it is possible to virtually evaluate a system-under-test with virtual 200 MHz even if the maximum emulation speed on the FPGA is 100 MHz for the core and the connected memory has a maximum clock of 50 MHz. The VPCM always suppresses clock signals for the core when the memory controller is not ready to return the request in the given virtual clock time. When this occurs the internal states of the core are preserved until the core clock resumes. The core clock signal starts again when the memory is ready. For the core internally it looks as if itself and the memory would run with 200 MHz as well because no additional wait cycles are needed [16].

For an easy extraction of statistic information from the design-under-test, sniffer units with a predefined skeleton are available. With the help of this skeleton the designer only has to indicate the ports or registers to monitor. The obtained performance statistics are then sent with the help of a finite state machine (FSM) in the sniffer skeleton over the dedicated statistics bus to the statistics manager. There the data is stored in a buffer until the network dispatcher is ready to generate a MAC packet containing the statistic data. The MAC packet is sent through the Ethernet port to a host PC where a graphical evaluation software runs. With the help of the VPCM it is possible to stop the clocks of all sub systems when the performance statistic extraction over the Ethernet occurs. Thus

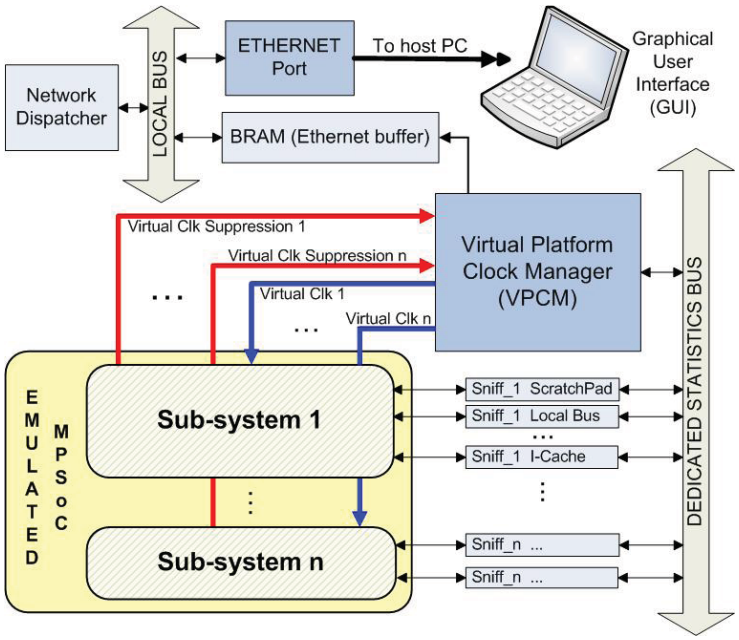


Figure 2.17: Emulation Based MPSoC Performance Extraction Framework [16]

the behavior of the system-under-test does not change [16].

As a result it could be shown that software, running on the system-under-test, needs no modifications to extract performance statistics during run-time. An evaluation speedup could be reached at up to 1140 times compared to cycle-accurate software simulations [16].

Run-time Performance Statistic Access In [11] a profiling architecture for multi-core SoCs is introduced. An implementation on a dual-core LEON3 system was shown to proof the concept. During run-time the processors have access to the generated performance statistics [11].

The authors of the paper split their MPPA architecture into the main parts event sensing, where event sensors detect specific hardware events, and the event collecting part where the accumulation of the detected events occur. This is shown in Figure 2.18. The event sensors are integrated into the specific components where the events occur. It is important that these sensors have no impact on the original design so that its behavior does not change. The event counters are combined to a monitor module which gives the processors the ability to start or stop the profiling and also to access all event counters. With this approach it would be possible for processor A to access the performance statistics of processor B. Using this knowledge A could schedule tasks, to execute on B, if B is in idle mode [11].

This architecture has been implemented on a dual-core LEON3 system shown in Figure 2.19. The monitor module has been attached to the AMBA bus. Event sensors have been integrated into the AMBA bus, I-Cache and D-Cache modules to extract statistics from there. To enable access from the processors to the monitor module, a device driver has been written for Linux. This enables the memory mapping of the event counters to the

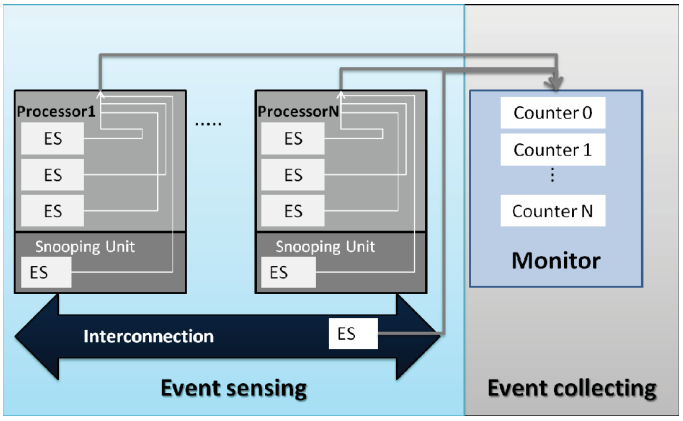


Figure 2.18: Event Sensing and Event Collecting Design [11]

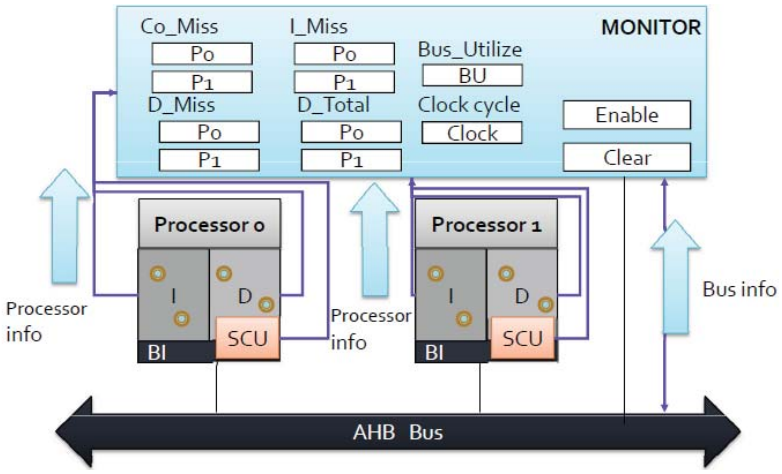


Figure 2.19: MPPA Implementation Overview [11]

user address space [11].

The MPPA implementation increases the total gate count about 0.66%. With the help of this architecture the speed of a software application has been optimized by a factor of 1.228 times due to D-cache miss rate reduction techniques. This example shows the potential of this work for fast software optimizations through performance statistic evaluation [11].

Chapter 3

Design of the Multi-Core Emulation Platform

3.1 Introduction

In this chapter we present the design of our power and performance profiling architecture which consists of the multi-core system-under-test and an additional power and performance debug unit (PPDU) on an FPGA as shown in Figure 3.1. Based on the switching activity of the multi-core system the PPDU generates power and performance profiles and sends them to a host PC where these statistics can be evaluated.

In the following sections we will explain the main requirements for our PPDU design. An overview of the PPDU with its main parts is given in Section 3.3 and the communication interfaces to and from the host PC are introduced in Section 3.7.1. The software which will be used for receiving and visualizing the statistics is introduced in Section 3.9. In Section 3.8 a tool flow is illustrating how this profiling architecture can be used during the design process of software or hardware to produce power and performance statistics.

3.2 Requirements

- The PPDU should profile power and performance statistics during the run-time of a multi-core SoC.

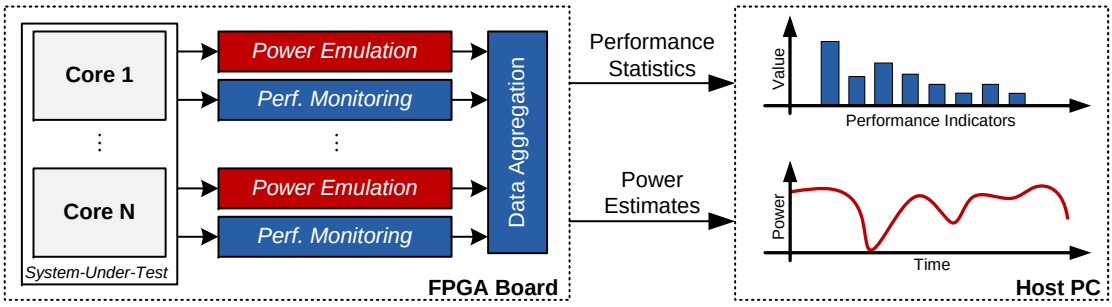


Figure 3.1: Power and Performance Profiling Architecture [9, with modifications]

- The emulation technique should be used which accelerates power and performance estimation with the help of an FPGA board.
- The presence of the PPDU should neither modify nor disturb the functional behavior and the execution of the multi-core design-under-test. Furthermore, the PPDU may not introduce timing delays leading to changes in the execution time of a given software application. Finally, the PPDU must not influence the power consumption estimation for the system-under-test. For example if power and performance data must be transfered, the normal bus should not be used because this would disturb the normal data transfer by generating additional traffic on the bus.
- The framework is intended for the design space exploration, therefore chip area or power consumption of the PPDU itself are only constrained by the used emulation platform.
- Our design considerations should be platform and software independent. The changing of the FPGA board or using another operating system on the cores should be easily feasible.
- The collected power and performance data and statistics should not be constricted to the system or to a core respectively. The whole statistic data should be observable by a host PC for data analyzing purposes.
- The whole design should be generic and general enough so that parts of the system architecture such as the numbers of cores, can be easily adapted.

3.3 PPDU Overview

In Figure 3.2 an overview of the multi-core design-under-test with an additional event sensing and power estimation unit is presented. Event sensors detect specific events on the basis of the states of different components. The power estimation unit consists of a power model which estimates the current power consumption based on the component states. The power and performance data is then sent to an event and power collection unit where counters store the number of occurred events and the consumed power. The collected profiles are sent over the output interface to the host PC on which an evaluating software runs. This software captures the data and visualizes the statistics. The host PC communicates with the PPDU unit over the input communication interface and can send control commands such as start or stop capturing instructions. All these parts of the design are discussed in detail in the next sections.

3.4 Performance Event Estimator

As described in Section 2.4 the event estimator should detect different events such as cache misses and processor stalls based on the component states of the system-under-test. For this reason event sensors are integrated into the performance event estimator with a specific detection circuit for every event. The types of events and the basic circuits required to detect these events are described in the next sections. First, the design flow for creating a performance estimator unit is explained.

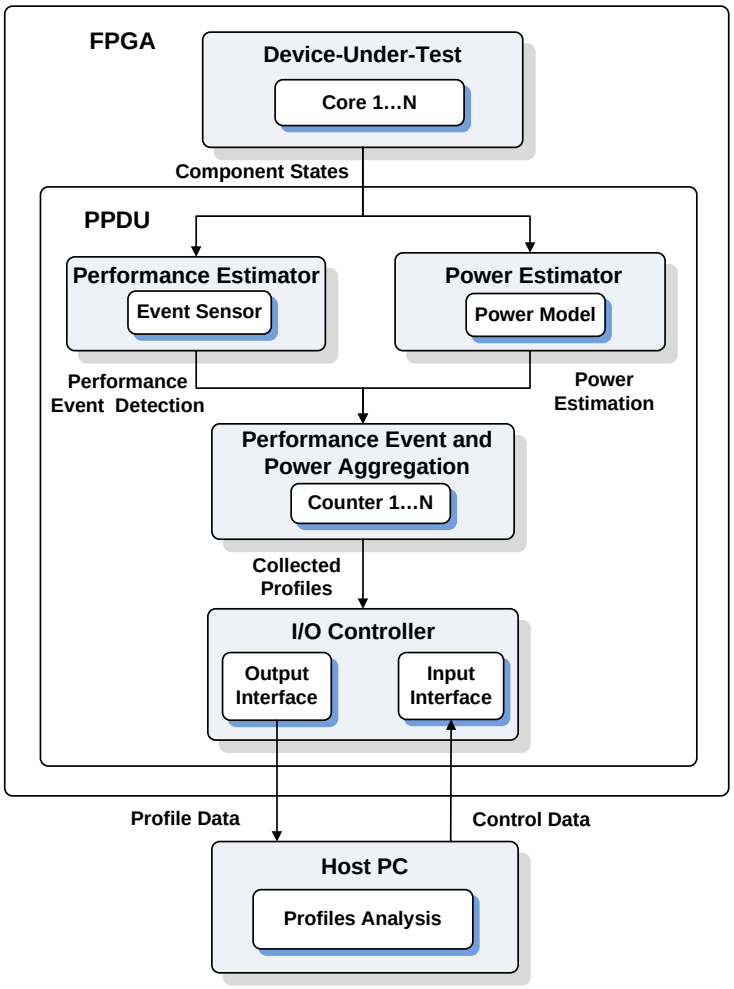


Figure 3.2: PPDU Design Overview

3.4.1 Design Flow

An overview of the design flow for creating a performance sensor and the intermediate steps are shown in Figure 3.3. First a test program has to be constructed which causes the events which should be detected. In general any program language could be used for which a compiler is available for the given platform. Programs written in C or assembler offer the benefit of a hardware-oriented programming style. During the execution of the programs the expected occurrence of an event like a D-Cache miss must be known. This is easier without additional abstraction layers like, e.g., in Java. A good architectural knowledge of the design-under-test is important to write benchmarks which cause the desired event.

Further on, the whole device-under-test and the binary test programs are fed to a RTL simulator where signals and events can be traced during the execution of the program. The correlation between signals and events is being found by searching the input and output signals of components for correlating signal transitions or logical states inside of a state machine of the observed components. If a state is only a logical condition inside of

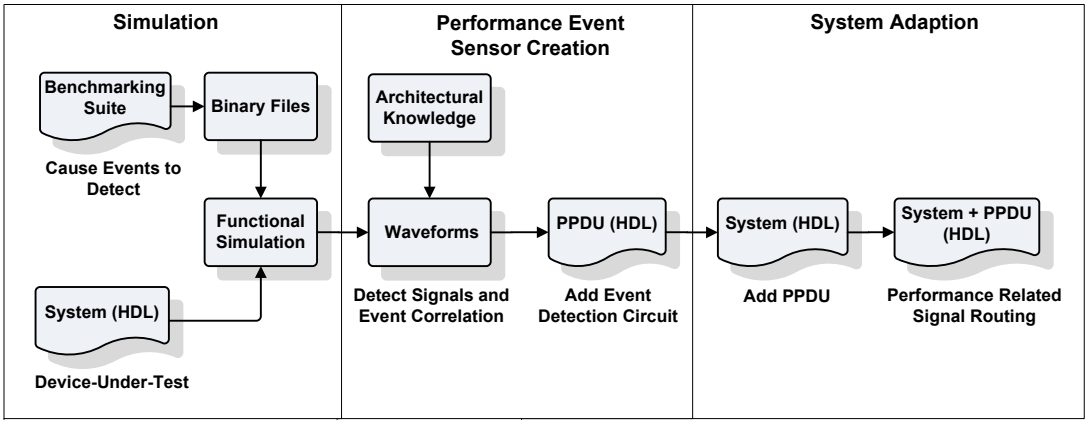


Figure 3.3: Performance Estimator Creation Design Flow

a component but has no connection to outside devices, a digital circuit must be added to the hardware description source file of the component. Knowledge of the design and the source code must be present to identify the signals indicating the occurrence of an event.

If all performance relevant signals are detected inside the multi-core device-under-test these signals must be routed through every component layer of the design to the event sensor unit. There an event detection circuit is added which combines the information of one or more relevant signals and indicates the occurrence of an event.

3.4.2 Event Types

There are different possibilities how signals or component states can correlate with performance events as shown in Figure 3.4. The first example is an event correlation to the positive edge of a signal. The next example is an enable signal where an event occurs every clock cycle when the selected signal is high. The last example is the state machine of a component where every time this state is reached a specific event occurs. Also a combination of these examples is possible. For example, the event could only occur when signal A exhibits a positive edge and the signal B is high.

3.4.3 Event Detection Circuits

The event detection can be achieved using different circuits. In Figure 3.5 three examples are shown. First, an example for a positive edge detection circuit is shown. Two D flip-flops A and B and a logical gate, which compares the output of the flip-flops, are needed to build this event detection. Every clock cycle the flip-flop A stores the current input signal and flip-flop B holds the one clock delayed state. When B holds logical zero '0' and A has logical one '1' the edge detection circuit indicates '1' because this means that at the input a positive edge occurred. In the next example three specific signal states must occur to detect an event. To realize this function logic gates are wired together. The event is detected in this example when all three input signals are '1'. In the next example an event correlates with a state inside of a FSM which is not visible from the outside. For this reason a detection circuit must be inserted into the component with an additional

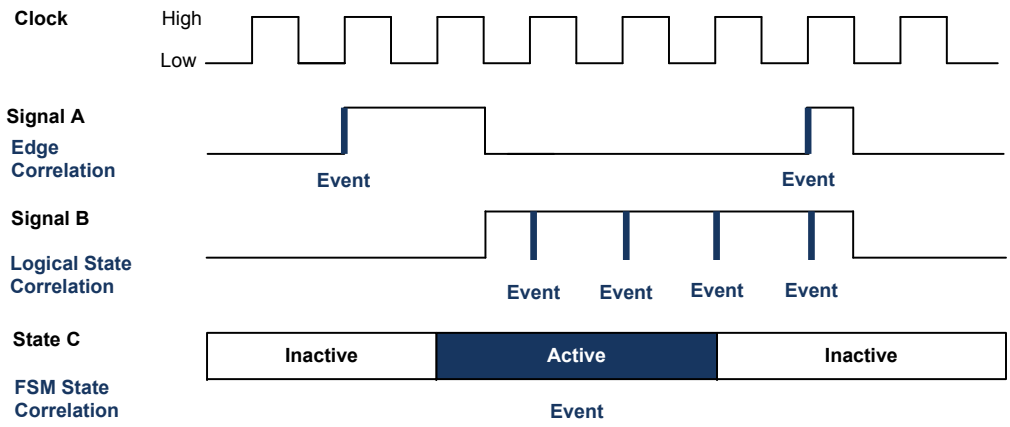


Figure 3.4: Correlation Examples Between Events and Signals/Component States

output signal. In this example, written in HDL pseudocode, the event and therefore the output signal is '1' when the FSM state "idle" is detected.

These circuit and code examples represent skeletons. For the implementation the event detection circuits have to be implemented synchronously with the rest of the design. It is also important that an event is detected only one time when it occurs and not multiple times. For this purpose, combinations of these techniques can be used.

3.5 Power Estimator

To obtain power values during the run-time of the system the power estimation technique is used as described in Section 2.2.2. The estimated power is calculated with the help of a power model which uses the state of the multi-core system-under-test as input. In this work the power estimator and the power model have not been created from scratch but existing power estimator hardware was adapted and integrated. The component is given as VHDL code created during the POWERHOUSE [42] research project, for a single-core LEON3 system [23, 8, 7].

During this work the power estimator code was adapted for multi-core usage and its integration into the newly designed PPDU was undertaken. The given power estimator creates cycle-accurate power estimates for the entire system as well as for sub-components during run-time.

3.5.1 Design Flow

An automatic design flow for creating the power estimation unit and integrating it into the design-under-test is described in [7]. During the functional and the power simulation different benchmarks are executed on the system-under-test. Based on the created trace files the power model parameters are selected by different algorithms. First, a pattern matching is carried out which selects specific signals based on user-supplied name patterns. After this, signals which do not change their state during the benchmark or are redundant in the design are removed. Now a cross-correlation based on the switching activity and the power profiles is conducted. Signals with low correlation to power changes are removed.

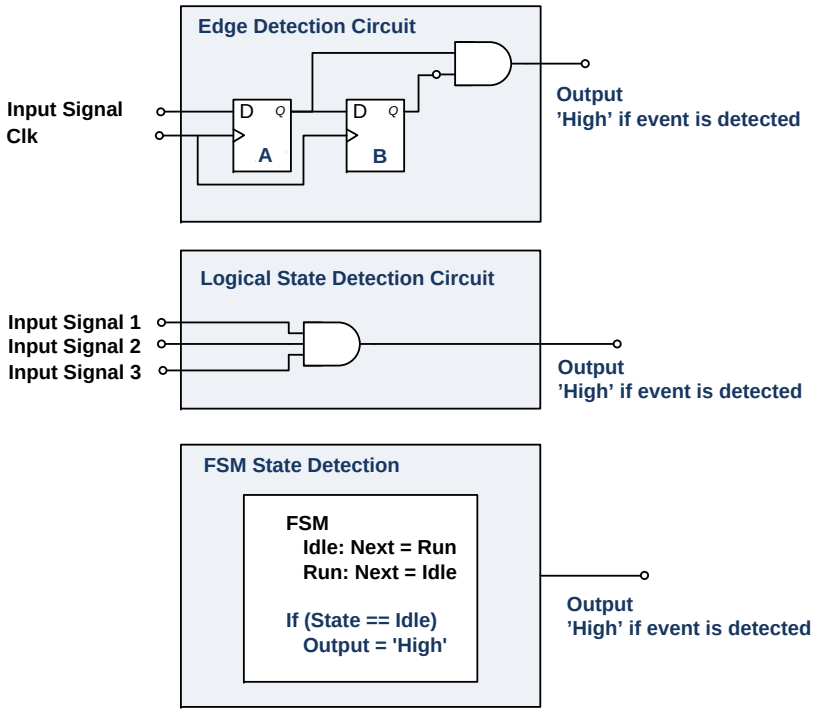


Figure 3.5: Event Detection Circuit Examples

The power model, which is based on a non-negative linear regression, is then created. The power estimation component with the power model can be integrated into the PPDU which will be integrated into the system-under-test. In the end the power-related signals, based on which the power model estimates power values, are routed from different hierarchical levels to the PPDU. This design flow is presented in Figure 3.6 [7].

3.6 Performance Event and Power Collector

The storage space for power and performance statistics is limited on an FPGA board and the storage space requirement is linearly increasing with the number of cores used in the design-under-test. If a benchmark runs for 1s on a single-core system at a clock rate of 50MHz and generates every clock cycle an 8bit estimate of the total power consumption, the data from the benchmark run has a size of 47,7Mbyte which can be seen in Equation 3.1. For achieving a cycle-accurate evaluation a connection to the host PC of around 381,5Mbit/s would be required.

$$DataSize = ClockCycles \cdot PowerBits = 50 \cdot 10^6 \cdot 8bit \approx 48828kbyte \approx 47,7Mbyte. \quad (3.1)$$

Note that 8 bit of data per clock cycle is a very low assumption for the amount of generated statistic data. If the FPGA contains enough memory it would be possible to store the power and performance statistics during run-time in the memory and send it to the host PC after the benchmark has finished. However long benchmarks will also exceed

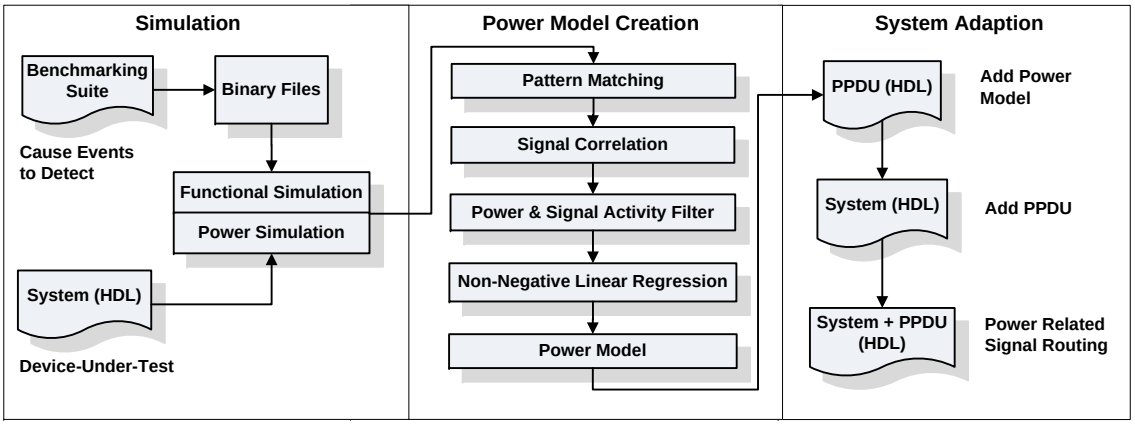


Figure 3.6: Power Estimator Creation Design Flow

the memory capacity of the largest FPGA device. For also being usable on FPGAs with no storage extension, our design solution must send the statistics during run-time to the host PC. Therefore an event and power collection unit has been added to the design which stores the data for the transmission until the output device is ready to send the power information to the host PC. The power and performance data between the transactions are added up and stored in counters. This means that the design is able to handle different output devices with varying baud rates. For every power or performance event which should be counted, an own counter will be realized. For every additional bit we assign to a counter we can double the number of samples that can be stored. The required counter width for the last example with a 1s benchmark, 50MHz and 8bit/Hz would account to 34bit to sum up all the 8bit samples which is shown in Equation 3.2. The needed communication speed decreases from 384Mbit/s to 34bit/s. However this large data summation represents a large quality degradation because all the dynamic information of the power consumption during the benchmark execution gets lost. For this reason a tradeoff must be made between transmission bandwidth and loss of dynamic information.

$$CounterWidth = \lceil ld(ClockCycles \cdot 2^{PowerBits}) \rceil = \lceil ld(ClockCycles) + PowerBits \rceil = ld(50 \cdot 10^6) + 8 \approx 26 + 8 = 34bit \tag{3.2}$$

3.7 PPDU I/O Communication

In this section we discuss the communication between the host PC and the FPGA board. Our PPDU on the FPGA board should be configurable from a host computer or a user program. This communication needs little bandwidth because control commands only require some bytes and these instructions have to be sent only at the beginning and at the end of a benchmark. On the other hand, the communication between the PPDU and the host computer requires high-speed communication because a large amount of power and performance statistics are generated and must be sent to the host PC.

Connection Type	Connection Speed
JTAG	5Mbit/s
RS232	1Mbit/s
Ethernet	10/100Mbit/s
USB 2.0	480Mbit/s

Table 3.1: Connection Types on the Xilinx GR-XC3S-2000 FPGA

3.7.1 Standard Communication Interfaces on FPGA Boards

At the beginning of the design we know that we will implement our power and performance emulation on a Xilinx GR-XC3S-2000 development FPGA board containing different communication interfaces. It offers an on-board JTAG, RS232, Ethernet and USB 2.0 communication interfaces. The corresponding PHY devices which handle the physical layer of the OSI model are already integrated onto the board for Ethernet and USB 2.0. The different types with their connection speeds are listed in Table 3.1. We have to choose from these interfaces to create our I/O communication from the FPGA to the host PC. This is illustrated in Section 3.7.2 and Section 3.7.3 [17].

3.7.2 Host to PPDU Communication

After the system is synthesized on the FPGA a configuration must still be possible. The PPDU should be configurable by the user for every benchmark which runs on the FPGA board. A selection should be possible which power component or performance counter information is sent to the host PC. For this reason a control logic with control registers has to be designed. To control these registers we need read and write access to registers and memories of the LEON3 design running at the FPGA board. The bandwidth of the communication interface can be very low because control commands only contain some bytes and are only sent from time to time. Thus, the most important requirement for the communication interface is that it can be easily integrated into our power and performance evaluation platform.

The GRMON monitor from Gaisler Resarch makes debug access to the design-under-test possible. This software is available for Windows and Linux and can establish a connection to the board over the JTAG interface. It is possible to insert breakpoints for debug reasons. Furthermore write and read access to the memory and registers of the system on the FPGA is provided. It is not necessary to change any HDL code to enable GRMON access over JTAG. For this reason the JTAG connection fulfills all requirements for the host to FPGA communication [43].

Another possibility for controlling the PPDU is by software running at the emulated system. This is achieved by mapping the control register into the memory space of the test program. Thus every test program can for example start and stop the PPDU itself. This technique was introduced in [38, 10] and will also be implemented in our PPDU. Note that also test program transparent operation is available in our approach, allowing the profiling of an application without modifications.

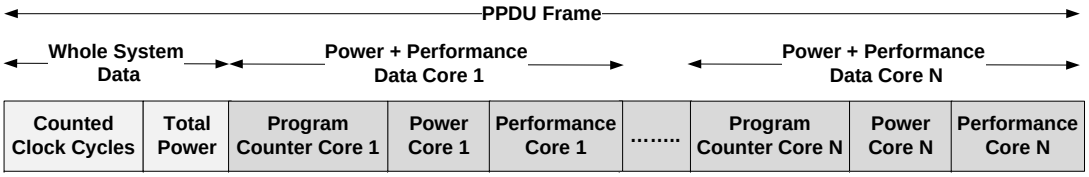


Figure 3.7: General Structure of a PPDU Frame

3.7.3 PPDU to Host Communication

For the communication from the PPDU to the host PC different communication interfaces can be chosen as listed in Table 3.1. The requirements for the interface are twofold. First, it should be as fast as possible so that we do not loose too much dynamics in the power and performance profiles. Second, the receiving and statistic extraction module at the host PC should be operating system independent and easy to implement. Thus, only the Ethernet interface with 100Mbit/s and USB 2.0 with 480Mbit/s, with the highest connection speeds at the FPGA board, remain as choices. To obtain an operating system indepent software we chose to write our receiving host program in Java. Because of approved Ethernet APIs and to avoid USB driver problems, we decided to create our FPGA to host communication based on the Ethernet interface.

A PPDU frame, which consists of power and performance statistics data for every core in the system-under-test, consists of the following parts as shown in Figure 3.7. This frame has to be sent over the Ethernet connection to the host PC including the power and performance statistics which are represented by counter values summed up over some clock cycles. A general PPDU frame consists of the whole system data statistics like the system clock cycle counter and the total power. Also specific core statistics are included in a PPDU frame like the power per component, performance statistics and the current program counter. With the program counter value an allocation of the power and performance profiles to the source code is possible by the performance analyzing software.

To get a fast overview of how many clock cycles of the LEON3 system will be averaged by adding up all power and performance values the following calculation was made with an assumed number of statistics data for a hypothetical PPDU frame which is shown in Table 3.2. This PPDU frame consists of the system clock counter and statistics for a single-core system which are the current program counter value, the total core power and two performance statistics. Altogether this hypothetical PPDU frame consists of 112 bit which have to be transmitted over the Ethernet connection before the next PPDU frame can be sent. Considering a single-core system operating at 50MHz and a 100Mbit/s Ethernet connection, ignoring data overhead, we can transmit every clock two data bits. This means that the host PC receives every 56 clock cycles power and performance statistics of this time interval. For a multi-core system every core statistic is sent individually which leads to a PPDU frame size of 208 bit for dual-core and 400 bit for a quad-core system. The number of clock cycles which are averaged during multi-core usage are listed in Table 3.3.

When we use the 100Mbit/s Ethernet interface for our host to PC communication power and performance statistics can be sent with a satisfying dynamic accuracy. Summed up statistics can be sent for a time interval of 56 clock cycles for a single-core system and

Counter Name	Counter Width
Counted System Clock Cycles	16 Bit
Core Program Counter	32 Bit
Core Total Power	32 Bit
Core Performance Statistic 1	16 Bit
Core Performance Statistic 2	16 Bit

Table 3.2: Counter Sizes For a Hypothetical PPDU Frame

Number of Cores	PPDU Frame Length	Averaged LEON3 Clock Cycles
1	112 Bit	56
2	208 Bit	104
4	400 Bit	200

Table 3.3: Number of Averaged Clock Cycles for Multi-Core Systems

200 clocks for a quad-core system which is short enough to make power and performance optimization decisions based on the them.

3.7.4 Data Aggregation and Ethernet Frame Generation

To make the profiling architecture ready for multi-core usage a core multiplexer has been added into the design which is shown in Figure 3.8. Now the processor statistic transmission rotates through all processors in the design during the sending process over the Ethernet interface. With the help of this multiplexer and a reset logic it is possible to send the power and performance statistics for a variable number of cores via the Ethernet interface. Hence we obtain a generic design which is able to profile an arbitrary number of processors in the multi-core system-under-test. A larger number of cores in the design result in more generated statistics data that must be sent over the Ethernet interface. Thus every additional core results in a worse timely resolution because the accumulator for every core has to sum up the statistics data over a longer time period until the data can be sent.

Every core is sending power and performance statistics to its own aggregation unit. In this unit one counter is implemented for every generated performance or power statistic. The concatenation of all power and performance counter values of a processor forms a PPDU frame. The output of each aggregation unit is connected to the input of the core multiplexer which selects one of them for the output, connected to the Ethernet controller unit. This frame is buffered by the Ethernet controller until the transmission over the Ethernet interface can be carried out. After transmission the accumulator for the buffered PPDU frame is reset and the core multiplexer selects the next PPDU frame to transmit.

An example where two processors send their power and performance statistics over the Ethernet interface and the PPDU frame interleaving functionality is shown in Figure 3.8. All power and performance estimation units start generating statistic data at the same time. When the Ethernet interface is ready to send data the statistic data of core one is buffered and the accumulator unit of core one is reset. The Ethernet interface now starts transmitting the buffered data to the host PC. During the transmission time the

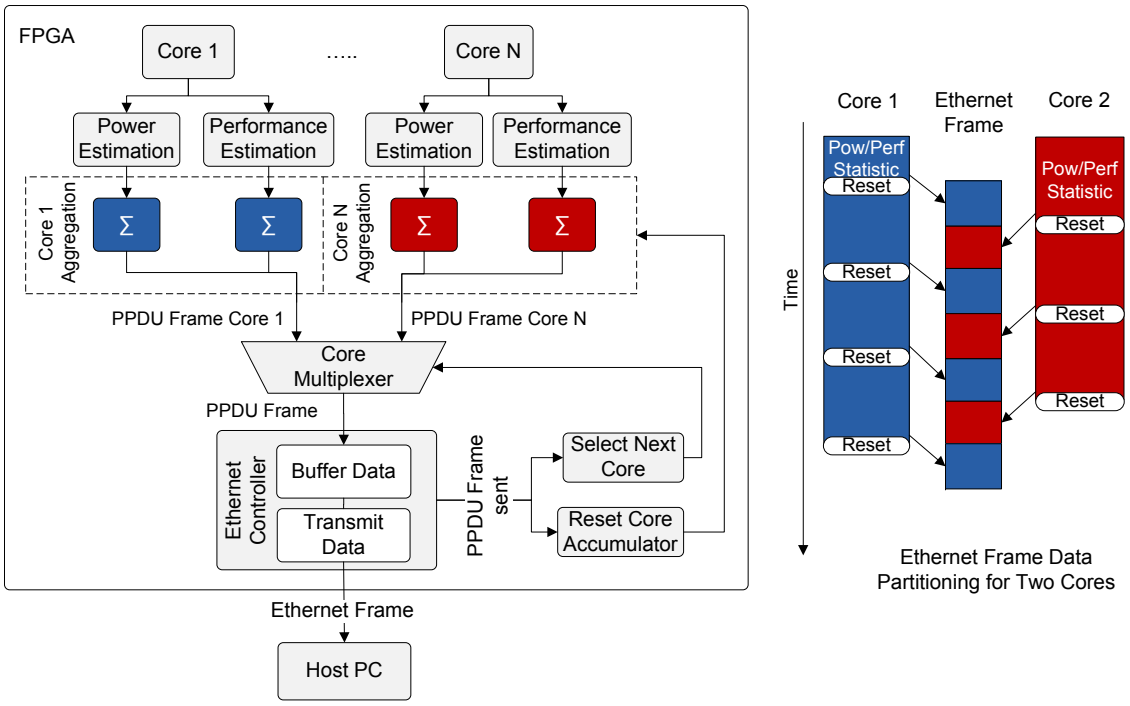


Figure 3.8: Data Aggregation and Ethernet Frame Generation

accumulator units of both cores sum up the estimation data. When the PPDU frame of core one is transmitted the statistic data of core two is buffered and the accumulator unit of core two is reset. A larger number of processors in the design does not impact the overall power or performance estimation during a benchmark run. Only the timely resolution will get worse, resulting in a more coarse grained statistics data sampling.

3.8 Software Analysis Flow for a System with PPDU

This section illustrates how our profiling architecture can be used during the design process to obtain power and performance statistics of software and hardware-under-test. In Figure 3.9 an overview of a software development flow including our PPDU design is shown. First the software program which should be analyzed is compiled for the multi-core system and machine code is generated. If the PPDU control registers are mapped into the memory space of the test program an easy control of the PPDU tracing is possibly inside of the test program. Also the netlist of the multi-core system with the PPDU is loaded onto the FPGA board with the machine code of the software program to test. Now the power and performance traces are transmitted over the Ethernet connection to the host PC where a Java program receives the data and displays it.

3.9 Power and Performance Analyzing Software

In this section we will discuss the design of our analysis software running at the host PC. Based on our previous design decisions the power and performance statistics from the

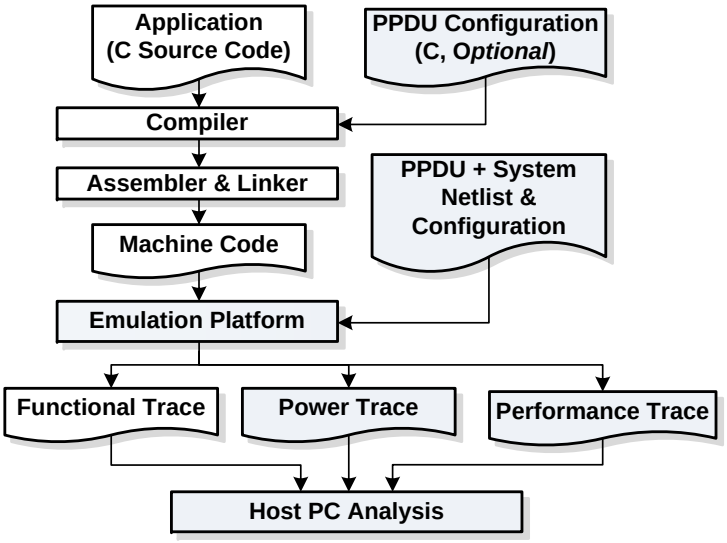


Figure 3.9: Software Development Flow Utilizing Power and Performance Emulation [8, with modifications]

PPDU on the FPGA will be sent over the Ethernet interface to the host PC.

3.9.1 Requirements

The main requirements for the host software are:

- Platform independent, runnable on Windows and Linux.
- Run-time Ethernet data capturing of all profile data with no data loss.
- Visualization of the power and performance profiles.
- Profiles should be exportable into a format for storage and post-processing.
- Software should be configurable for different PPDU settings with a configuration file.

3.9.2 Analyzing Software Parts

An overview of our software and its main parts are shown in Figure 3.10. For achieving a platform independent software design we decided to program the host software in Java [28]. We use the free software library JPCAP [34] to receive the Ethernet packets which are sent from our PPDU. The data is preprocessed by a data parser which can be configured by a XML file. This configuration is necessary because the PPDU can be configured to send different profile statistics to the host PC. The configuration file instructs the software how to interpret the received data. The data parser component has to know how much processors are in the design and which bits represent the content of a power or performance counters. Generally the data parser component can be realized in Java and uses no special libraries. The parsed data is then pre-processed which can be for example an averaging

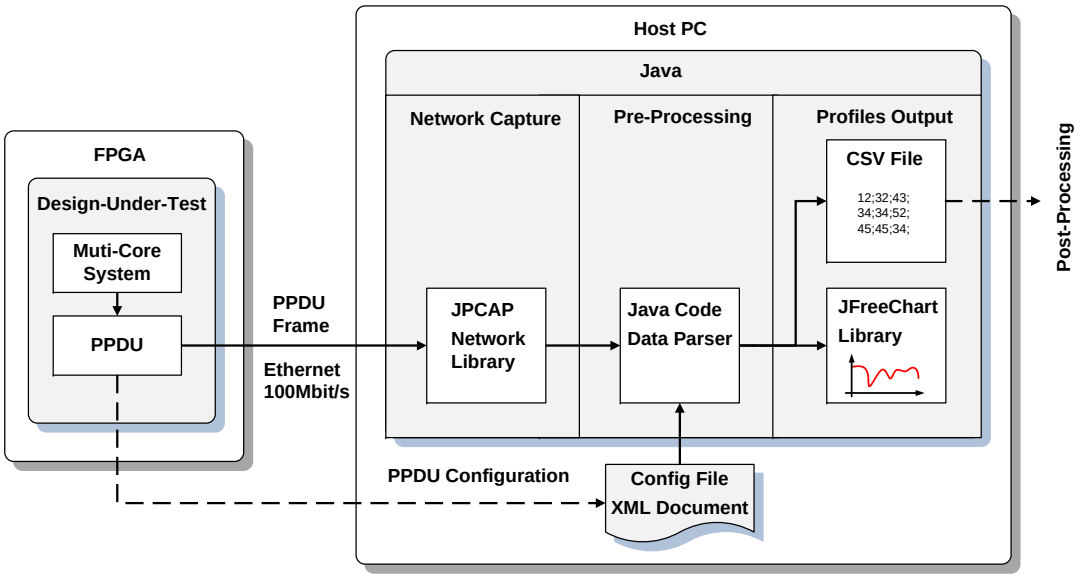


Figure 3.10: Overview of the Host Evaluation Software

step to reduce the amount of data. The pre-processed data can then be stored in a CSV file. We decided to choose the CSV file format because this data format can be easily imported by post-processing programs like MATLAB [35] which provides mighty filtering and visualization methods. To obtain a fast overview of the received statistics a visualization module with the help of the free software library JFreeChart [30] will be implemented.

Chapter 4

Implementation of the Multi-Core Emulation Platform

4.1 Overview

In this chapter a case study of our power and performance evaluation platform using a LEON3 multi-core system is presented. The main parts of the implementation are:

- LEON3 IP library from Aeroflex Gaisler [21]
- SnapGear 2.6.21.1 embedded Linux distribution [20]
- Xilinx GR-XC3S-2000 development board [18]

An overview of the PPDU integration into the LEON3 design is shown in Figure 4.1.

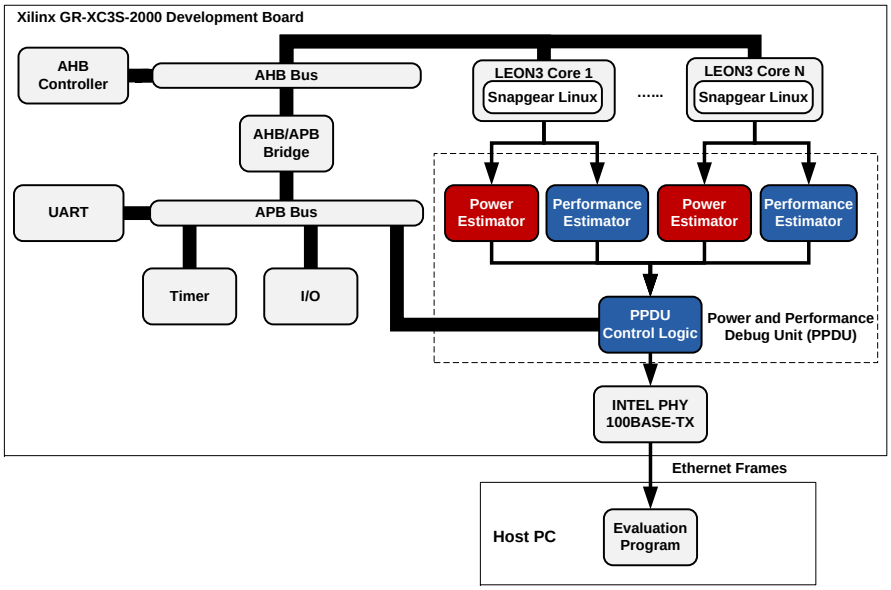


Figure 4.1: Integration of the PPDU Into the LEON3 Design

4.2 Used Tools

To prove the design a case study with the LEON3 IP library, provided from Aeroflex Gaisler [21] under the GPL license, has been made. The components of the library are implemented in the hardware description language VHDL. All sub-components and the processors are connected over an AMBA bus inside the multi-core design. The LEON3 design supports symmetric multi processing (SMP) up to 16 cores. The library is fully generic and synthesizable for an FPGA or a silicon implementation. An overview of the software and hardware used during the implementation of the profiling architecture is shown in Figure 4.2. The profiling architecture was added to the LEON3 design to generate power and performance profiles, which are sent over the Ethernet interface to a host PC. Therefore the PPDU is connected to the AMBA bus and different signals which indicate the current state of the processor are routed from every core to the PPDU. To create the *performance estimation* circuits inside the PPDU, first characterization benchmarks

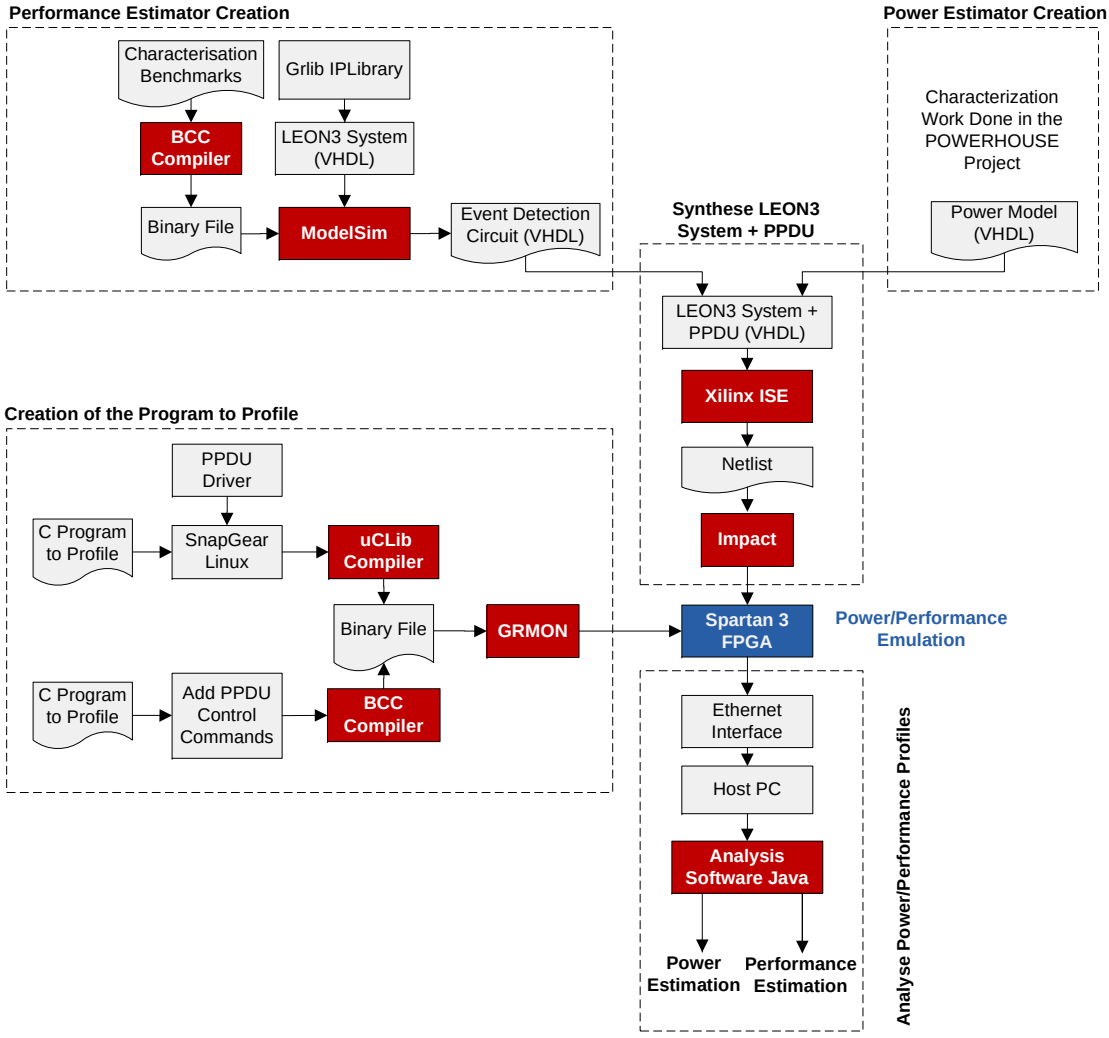


Figure 4.2: Used Tools During the Implementation

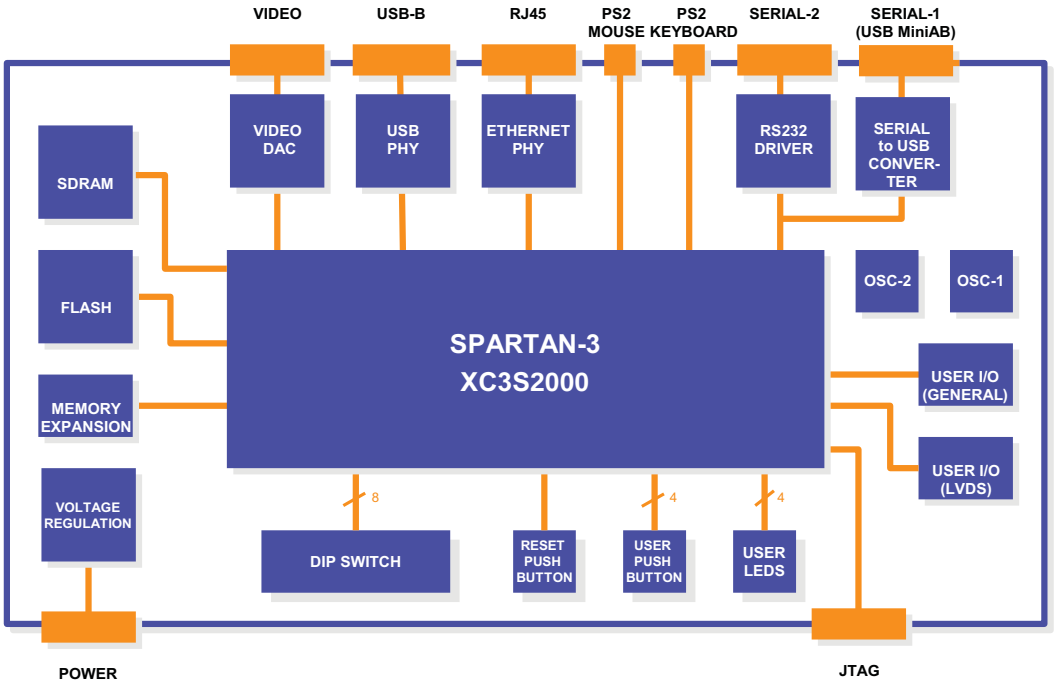


Figure 4.3: Development FPGA Board GR-XC3S-2000 Block Diagram [22, with modifications]

are written in C causing the events to be detected. These events are for example cache misses or register write/read accesses. These benchmarks are compiled with the Bare-C Cross Compilation (BCC) system. The binary benchmark files and the LEON3 multi-core design are simulated in ModelSim SE 6.6b [3]. After the simulation process signals in the design which correlate with the occurring events in the benchmark are manually identified. These signals are routed from different hierarchical layers to the performance estimation unit of the PPDU. For every event to be detected an own detection circuit is implemented in VHDL.

The *power estimation* unit of the PPDU was created in the POWERHOUSE [42] project for a single-core system. To obtain a generic design every processor is connected to its own power and performance estimation instance. After this the whole design is converted into a netlist for the FPGA by the Xilinx ISE 12.3 [1] tool which is a design environment for FPGAs. The generated netlist is sent by the Xilinx Impact tool [1] over the JTAG USB device onto the FPGA device.

The development board for the implementation of the profiling architecture is the GR-XC3S-2000 board manufactured by Pender Electronic Design [18]. Important components for the implementation on this board are the Xilinx Spartan3-2000 field programmable gate array (FPGA), the 10/100Mbit/s Ethernet PHY chip from Intel [27] and a JTAG FPGA programming interface. An overview of the development board is shown in Figure 4.3.

If a program shall be profiled without the OS, control commands which write to the memory mapped registers of the PPDU are inserted into the code to start and stop the

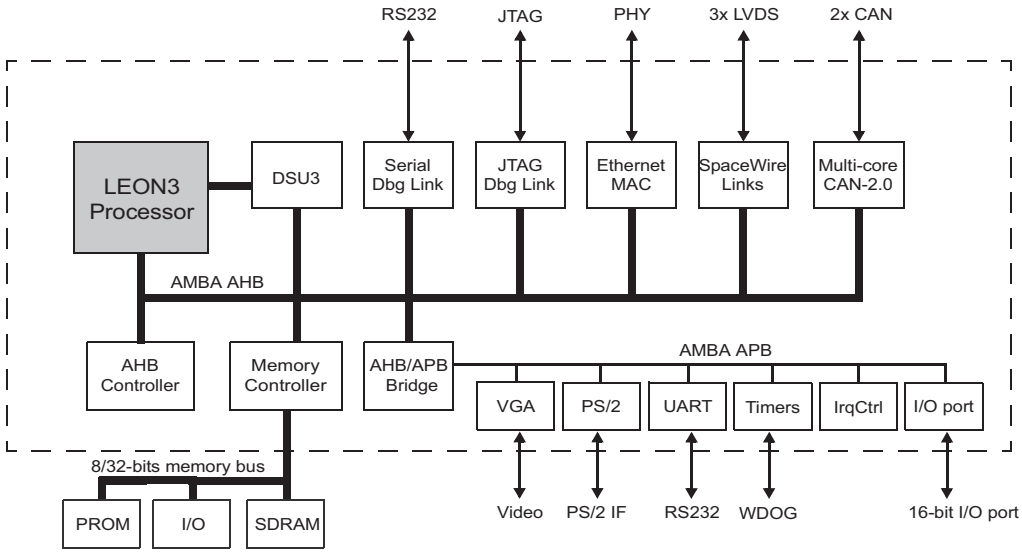


Figure 4.4: LEON3 Template Design by the GRLIB IP Library [4, with modifications]

profiling process. The program is then compiled with the BCC compiler which generates a binary file. If the written program is a process in the Linux SnapGear OS the access to the PPDU is performed through a Linux driver. With this driver a user program has access to the PPDU registers which are otherwise not accessible. The Linux OS including the user program was then compiled by the uCLib compiler. The binary file can now be written by the GRMON tool onto the FPGA where it is executed by the emulated LEON3 system. The power and performance data of the cores are sent over the Ethernet interface onto the host PC where a Java program receives, parses and displays these statistics.

4.3 GRLIB LEON3 IP Library

4.3.1 IP Library Overview

The GRLIB IP library is provided from Aeroflex Gaisler [21] under the GPL license. This library consists of different hardware modules written in VHDL which can be connected over an AMBA interface. This library is fully synthesizable and can therefore be used for FPGA development or for silicon implementation. An overview of this template system is shown in Figure 4.4. In this example a LEON3 processor is connected over the high-speed AMBA AHB bus to peripheral components like the memory controller or the AHB/APB bridge for the AMBA bus. All components which do not require high communication bandwidth such as the RS232 interface, timers or the interrupt controller are connected to the low speed AMBA APB bus. The control registers of the components are memory mapped and can therefore be accessed over software, running on the LEON3 processor.

With the help of the JTAG interface and a debug module, designers have access to the system during run-time. The extraction of different information like the integer registers or instruction cache entries is possible over the GRMON software debug tool [43]. This tool also enables the manipulation of registers inside the processor and can write

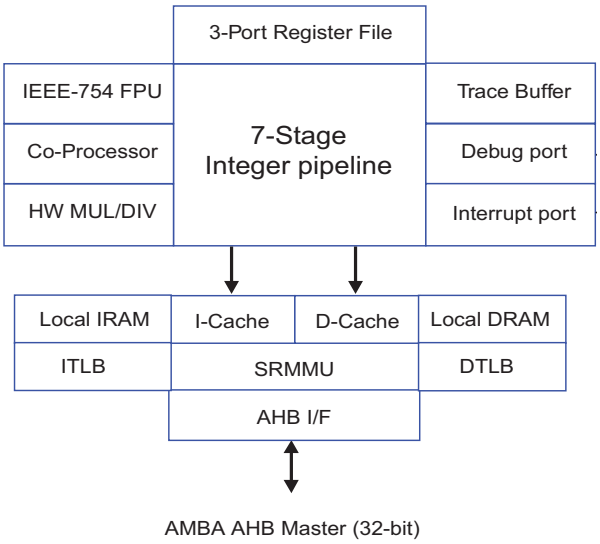


Figure 4.5: LEON3 Core Components [5]

and read data to specific memory addresses over the debug support unit. The GRLIB enables symmetric multi processing which means that up to 16 LEON3 processors can be instantiated in the design, communicating over the AMBA bus. The processor cores execute instructions independent from each other and do not share caches. To avoid inconsistent data in the caches of the processors, when working with data from the main memory, cache snooping should be enabled. The LEON3 implementation also supports a power down mode. The template configuration implements an Ethernet MAC controller which has been deactivated. This was necessary because only the PPDU should have access to the Ethernet interface to avoid complex sharing logic. Parts of the design and the processor can be modified over a configuration file or a configuration GUI, e.g., the number of LEON3 cores or the cache sizes [5, 4].

4.3.2 LEON3 Processor

An overview of the LEON3 processor is shown in Figure 4.5. The LEON3 is a 32-bit Sparc V8 processor and is therefore a reduced instruction set computer (RISC). Due to the fact that the processor is a Harvard architecture the instruction and data memory is split, two independent caches are implemented. One cache is responsible for caching instructions, the other for storing the data words. The integer unit (IU) consists of 7 stages:

- **Fetch:** New instruction is fetched from the cache or loaded from the AMBA bus.
- **Decode:** Instruction is decoded and addresses are generated.
- **Register access:** The operands to execute the instruction are loaded from the register file.
- **Execute:** The arithmetic logic unit (ALU) performs the instruction with the loaded operands.

- **Memory:** Data is read/written from the main memory.
- **Exception:** If traps or interrupts occurred they are resolved in this stage.
- **Write-back:** The results of the ALU calculation are stored back into the register file.

The IU can be stalled during execution by the I-cache and D-cache if operands or instructions cannot be provided. Therefore the IU is halted, no other operations are performed and the missing data are loaded from the memory. This leads to extra wait clock cycles during the execution of an instruction when a cache miss occurs. A program has only access to specific registers which are defined in a register window. Generally there are in, out, local and global registers for a register window. Due to the fact that the LEON3 is part of the RISC processor family, all data manipulation like arithmetic/logical/shift are done in registers and not in the main memory. To transfer data from the registers to/from the memory the load/store instructions must be used [47, 4, 5].

4.4 PPDU Hardware Implementation

4.4.1 Introduction

An overview of the hardware implementation of the PPDU in the LEON3 design is shown in Figure 4.6. Every LEON3 core is connected to its own power and performance estimation unit.

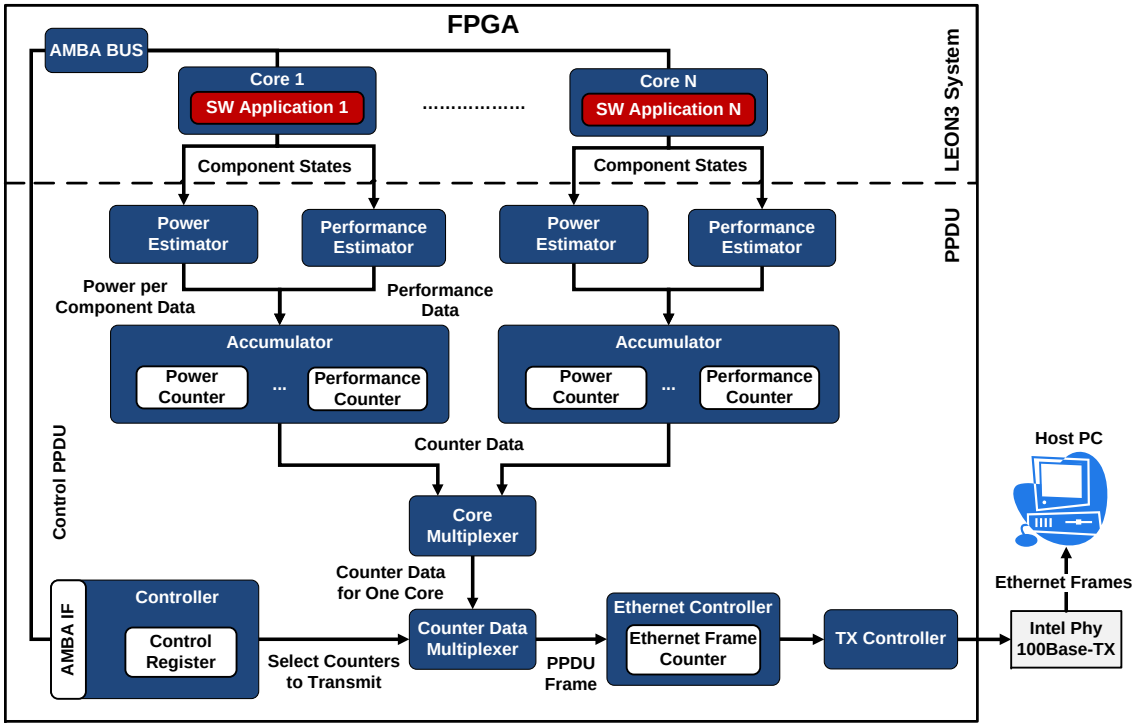


Figure 4.6: Overview of the PPDU Implementation on the FPGA

- **Power/Performance Estimation Units:** These units create cycle accurate estimations which are summed up in the accumulator unit by different counters. The size of these counters are fully generic to make them easily adaptable to the data they should monitor. For example a counter which sums up the occurrence of logical instructions could be implemented using a smaller counter than a counter which counts the total number of clock cycles.

- **Multiplexer:** A core multiplexer selects the core whose power and performance counters are transmitted over the Ethernet interface. Due to the fact that the transmission of all available counters would lead to a loss of dynamic information, a second multiplexer selects only specific counters to transmit. With this multiplexer it is possible to choose only the information to transmit that are important for the analysis of the system. The reduced amount of data leads to a finer time resolution. The selected power and performance counters form a PPDU frame which is sent to the MAC controller unit.

- **Controller Unit:** The counter data multiplexer is controlled by the controller unit which has an AMBA bus interface and three control registers. These memory mapped 32-bit registers can be accessed by programs running at the cores and by the GRMON tool. The control registers start and stop the profiling process and select the counters for the PPDU frame.

- **Ethernet Controller:** The Ethernet controller is responsible for creating an Ethernet frame and filling the Ethernet data field with PPDU frames. After every transmitted PPDU frame the MAC controller resets the counters of the transmitted core and the core multiplexer selects another core to transmit its power and performance information. The Ethernet controller communicates with the TX controller, which is part of the GRLIB library, over a 32-bit data interface and several control ports.

- **TX Controller:** The TX controller is responsible for calculating the CRC checksum and the preamble for every Ethernet frame. The TX controller communicates with the Ethernet PHY chip on the development board over the standardized MII/RMII interface. This chip represents the physical layer of the OSI network model. The power and performance profiles of the PPDU are then transmitted over an Ethernet cable with 100Mbit/s to a host PC.

4.4.2 Controller Unit

The controller unit holds the three control registers for the PPDU and is connected over the AMBA bus to the whole LEON3 system. To enable the AMBA access the controller unit is connected with the `apbi : in apb_slv_in_type` and `apbo : out apb_slv_out_type` ports. Also the following VHDL code, shown in Listing 4.1, has been added to the controller unit to register the PPDU component at the AMBA bus.

```
constant pconfig : apb_config_type := (
      0 => ahb_device_reg (VENDOR_PPDU, PPDU_AMBA, 0, 0, 0),
      1 => apb_iobar (paddr, pmask));
```

Listing 4.1: Add Component to AMBA Bus

The constants PPDU_AMBA and VENDOR_PPDU have been defined in the file grlib\amba\devices.vhd and implement a unique identifier for the PPDU component connected to the AMBA bus. The control registers are shown with their memory mapped AMBA addresses in Figure 4.7. The ENABLE bit (31) enables starting and stopping the transmission of power and performance profiles over the Ethernet interface. The 2-bit field MODE (30:29) selects the power and performance counters for a PPDU frame. Three different modes are currently implemented and shown in Figure 4.8. If mode 11 is selected all power and performance counter are used to form a PPDU frame. Mode 01 selects the performance statistics D-cache stalls, I-cache stalls and register writes. Mode 10 selects one power consumption counter and one performance counter. These power and performance counters are selected by the POWER SELECT and PERFORMANCE SELECT registers. Every bit of the two selection registers represents a power or performance counter, starting at the most significant bit of the registers. The mode 00 is currently not implemented and can be used for creating a new mode in the the controller unit. A code example in Listing 4.2 shows how the configuration of the PPDU in a C-file can be easily achieved by using methods, configuration structure and defines implemented in the course of this thesis. In this example the PPDU is configured to mode 10 and additional to the main statistics in this mode, the I-cache hits and the power estimation

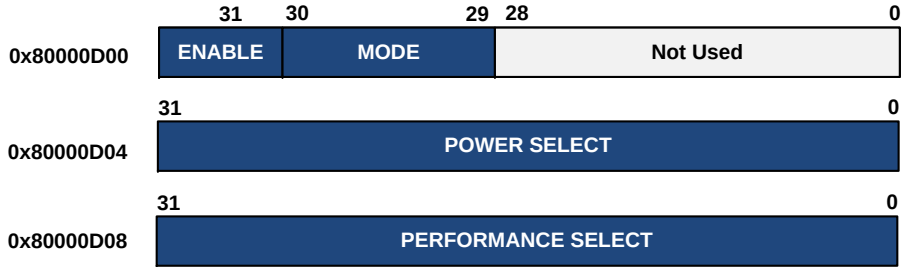


Figure 4.7: PPDU Control Registers

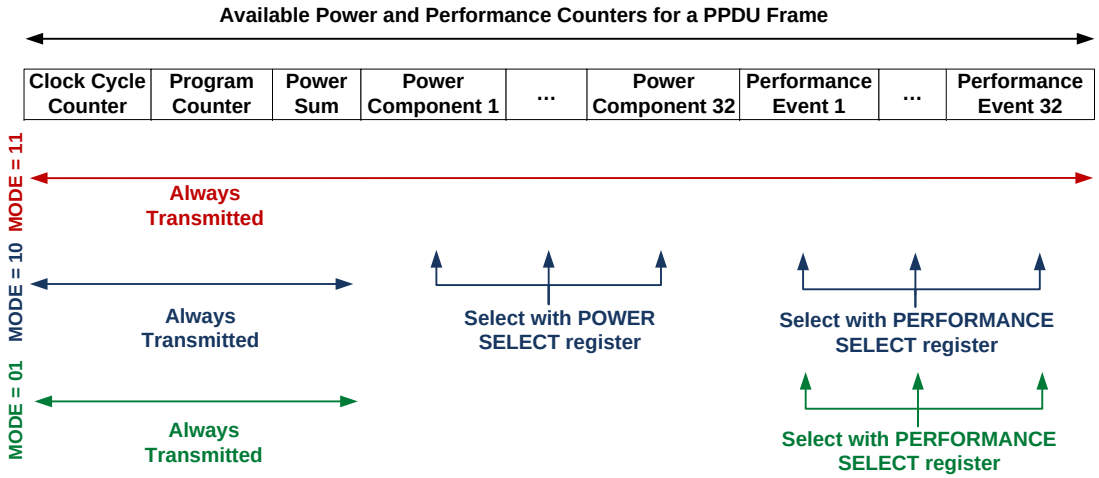


Figure 4.8: PPDU Modes

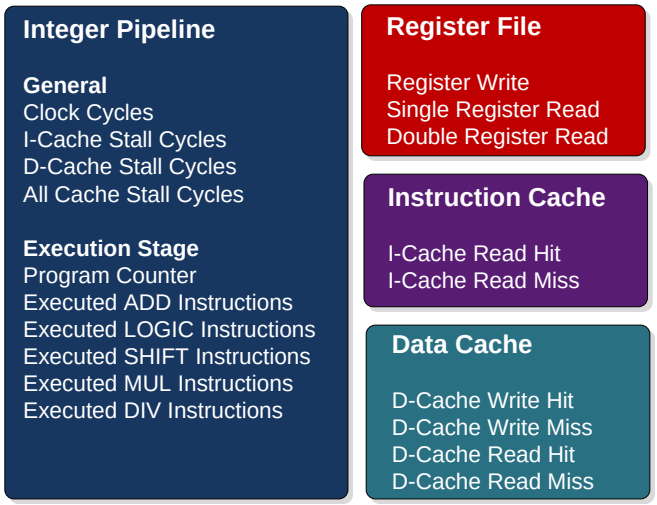


Figure 4.9: Implemented Performance Counters

of the first subcomponent are profiled. The profiling process is started by calling our start sub-routine `start_ppdu(&ppdu_config)` with the configuration structure as a parameter. Afterwards the code which should be tested is called by the method `code_to_test()`. The profiling process is stopped by calling the `stop_ppdu()` function after the code to test finishes execution.

```
struct ppdu_config_t ppdu_config;
ppdu_config.mode        = MODE10;
ppdu_config.perf_select = I_CACHE_HIT;
ppdu_config.pow_select  = SUB_POWER_1;
start_ppdu(&ppdu_config);
code_to_test();
stop_ppdu();
```

Listing 4.2: Configuring PPDU for Profiling

4.4.3 Performance Estimator Unit

4.4.3.1 Performance Counter Types

For every performance indicator to be profiled a performance detection circuit has been implemented which feeds a counter in the accumulator unit. All implemented performance event counters are shown in Figure 4.9 and are extracted from four main components [5]:

- The *integer unit* consists of a 7 stage integer pipeline. Performance statistics collected from this unit include the number of clock cycles of the last collection period and the number of I-cache as well as D-cache stall cycles. Stall cycles always occur when a pipeline stage in the integer unit needs data which is not available in the cache and must be loaded from the instruction or data memory. Thus the integer unit stops the execution and waits until the instructions or operands are loaded.

This hold event is signaled by the low active `holdn : std_ulogic signal` input port of the integer unit and can be triggered by the I-cache and D-cache unit. Also the number of executed instructions summarized in the five main groups ADD, LOGIC, SHIFT, MUL and DIV are counted. These events are extracted from the execution pipeline stage of the integer unit by adding additional output ports to the component which show the current executed instruction. The MUL and DIV instructions can only be used when the processor is configured to use an additional hardware-accelerated multiplication/division unit.

- The *register file unit* in the LEON3 processor possesses two read interfaces and one write interface, consisting of address, data and enable ports. Thus two different register addresses can be read out at the same time. The write and read operations to the registers are performed in one clock cycle after the enable signal is detected. The ports which indicate access to the register files are for read operations the high active `re1,re2 : std_ulogic` and for write operations the `we : std_ulogic` enable ports.

- The *cache system* of the processor consists of an I-cache and D-cache unit. They are fully configurable in cache size and replacement policy, e.g., strategy least-recently used, least-recently-replaced or (pseudo-) random. The I-cache is used during the instruction fetch procedure of the IU. The D-cache is accessed from the memory stage. By means of an AMBA bus connection the caches can load missing operands or instructions from the memory.

4.4.3.2 Benchmarks to Detect Cache Performance Events

For the purpose of causing performance events for the caches, six benchmarks have been written which perform write/read hits/misses for every cache type in a loop as shown in Appendix B.2. After creating the benchmarks in C and compiling them with the Bare-C Cross Compilation (BCC) the binary files were simulated on a single-core LEON3 system. The simulation software is ModelSim SE 6.5d [3] which operates at the RT-level and can display the signal changes of the whole system in a graphical user interface.

Signal changes were identified that correlate with the events in the loop of the benchmark. An I-cache hit occurs always when the data of the address which should be loaded is already buffered in the cache. This I-cache hit event can be caused repeatedly in a loop program because in the loop the instructions are loaded from the same addresses. To cause I-cache misses, the function `sparc_leon23_icache_flush()` was inserted into the loop which clears the content of the cache and therefore produces cache misses during the execution of the loop. D-cache read hits can be created by a software benchmark by reading a variable, which is stored in the memory. After a first read miss, because the data of the accessed address is not buffered in the cache at the first execution in the loop, all other accesses should generate D-cache read hits. To avoid compiler optimizations, which would for example optimize and therefore remove repetitive assignments to a variable with the same data, the keyword `volatile` should be used for creating the variables which are accessed. Read misses can be implemented by using the method `leonbare_leon3_loadnocache16(address)` from the BCC. This method loads data from an address and ignores the caches. The D-cache is implemented as a write-trough cache

which means that if a write hit occurs, the data changes are written into the cache and the main memory. If a D-cache write miss occurs only the data in the main memory changes and the data is not inserted into the D-cache. This cache characteristic is called no-allocate. This means that a write hit can only occur when first a read operation has been performed on the same address on which the write operation is performed. Therefore there are no execution speed improvements of a D-cache write hit compared to a D-cache write miss because the write process to the main memory needs more clock cycles than writing to the cache [5].

4.4.4 Ethernet Communication Functionality

The Ethernet controller is implemented as a finite state machine and responsible for sending as many PPDU frames as possible over a 100Mbit/s Ethernet interface to the host PC. The Ethernet controller is connected with the TX controller which handles the Ethernet frame header generation, CRC calculation and the communication with the PHY chip on the development board. The software on the host PC itself has to process the received Ethernet packets fast enough so that no packets are lost. This is important because we send as many Ethernet packets as possible over the Ethernet interface and do not wait for an acknowledge from the host PC.

The different states of the Ethernet controller unit are presented in Figure 4.10. In general this unit is responsible for filling the data field of the Ethernet frame with PPDU frames, transmitting these data to the TX controller and resetting the power and performance counters at the accumulator unit. The TX controller is adopted from the GRLIB library where it is used in the Gaisler Ethernet MAC component. The TX controller receives its input clock from the PHY Ethernet chip which generates a 25MHz signal for the 100Mbit/s Ethernet communication. By using the TX controller we do not have to communicate directly with the Intel PHY chip [27] on the development board. Starting a new Ethernet frame is done by setting the input port `tx_en : std_ulogic` of the TX controller to high. The TX controller then starts generating a new Ethernet frame by sending seven bytes 0xAA preamble data and the SFD byte 0xD5, generated for Ethernet synchronization reasons. After sending these data the TX controller changes its output port `read` which signals the Ethernet controller that it can send data to the TX controller. These data are forwarded to the PHY chip and sent over the Ethernet interface. Before the PPDU frames can be included into the Ethernet frame, the header data to obtain a valid Ethernet frame must be transmitted. This header data are first the MAC destination and source addresses, the Ethernet type field and after this, in the Ethernet data field, the PPDU frames. The Ethernet type field defines how the data inside of the Ethernet data field is interpreted by the next OSI layer. For example an Internet Protocol version 4 (IPv4) field is identified by the type field content 0x0800. The receiver of this Ethernet packet parses the type field and knows that the data field has to be interpreted as an IPv4 packet which consists of a header with IP source and destination addresses. Our PPDU type field content is 0x88AA which is unused by other protocols. The analysis software at the host PC interprets the Ethernet data field now as a two byte Ethernet frame identifier field and additional PPDU frames containing the power and performance statistics.

To avoid the data overhead which occurs by every new created Ethernet packet we have decided to place as many PPDU frames into a single Ethernet frame as possible and

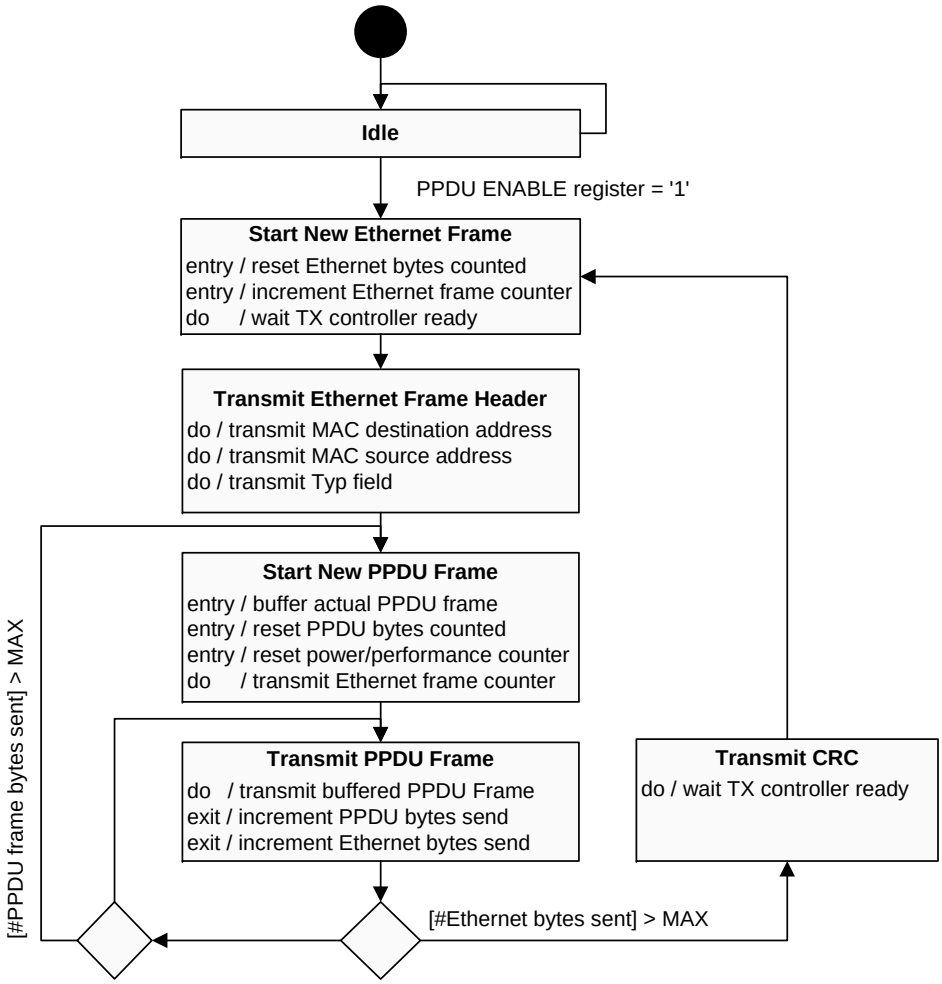


Figure 4.10: State Machine of the Ethernet Controller

start sending a new Ethernet frame directly when the old frame has been sent. Figure 4.11 shows how Ethernet frames look like if they are composed and filled with PPDU frames and the Ethernet frame identifier by the Ethernet controller. The overall byte count for the data field of a PPDU frame has to be below 1500 bytes which is the maximum specified data length for a valid Ethernet frame. For the case that the host PC could not process the received Ethernet packets quickly enough and has to drop packets, an Ethernet frame counter has been implemented into the Ethernet controller which is incremented every time a packet has been sent.

The power and performance data should be transmitted as fast as possible so that the averaging effect of undersampling is decreased. With the help of the Ethernet frame identifier the analysis software, running at the host PC, can detect missing packets from the PPDU by comparing the Ethernet frame identifier value for every frame. If a missing packet is detected the user can be informed. This information is important because in a multi-core system, the PPDU frames do not contain a processor identifier and can only be identified by their sequence in which they are received. If packets are missing, this

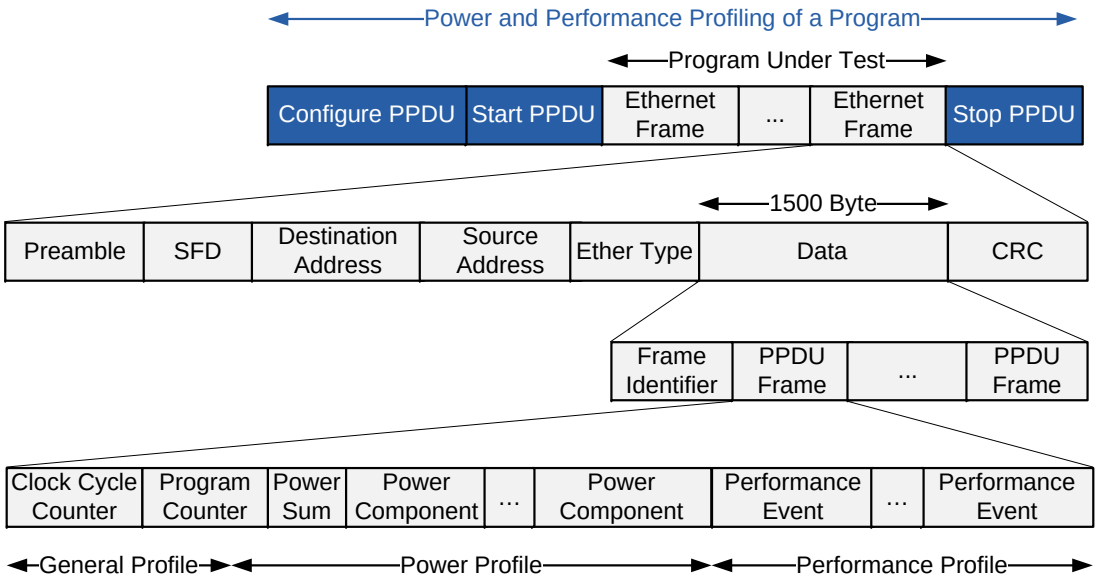


Figure 4.11: PPDU Frames Embedded in an Ethernet Frame

sequence could get mixed up and the PPDU frames are assigned to wrong processors during the parsing process of the software.

The communication between the Ethernet controller and the TX controller is implemented using a handshake protocol over a 32-bit data interface. Always when the TX controller has buffered the 32-bit input data the TX controller inverts the state of its `read` port. The Ethernet controller detects this change and assigns the next data which should be transmitted to the data port. To signal the TX controller that new data is ready for buffering the Ethernet controller inverts the state of its output port `read_ack`.

After sending the Ethernet data field the TX controller automatically calculates the CRC checksum over the transmitted data and places this checksum at the end of the Ethernet frame. After the Ethernet frame has been completely sent, the TX controller changes the state of its output port `done` to report the Ethernet controller that it is in idle state and ready to start a new Ethernet packet. The TX controller communicates with the PHY chip on the development board over a 4-bit data MII interface. The Ethernet PHY chip on the FPGA represents the physical layer of the OSI network model and can detect data collisions on the Ethernet wire. Due to the fact that we use a full-duplex Ethernet configuration and a crossover cable we do not consider collisions during the sending of data or that other network devices slow down the transmission speed. The only network component which sends data over the Ethernet cable to the host PC is the development board with the LEON3 system and our PPDU. The connection speed to the host PC is 100Mbit/s which is the highest network speed which is supported by the PHY chip.

4.4.5 PPDU System Integration

Our PPDU implementation sends and receives data from different units in the implementation as shown in Figure 4.12. In general, these units can be divided into two different

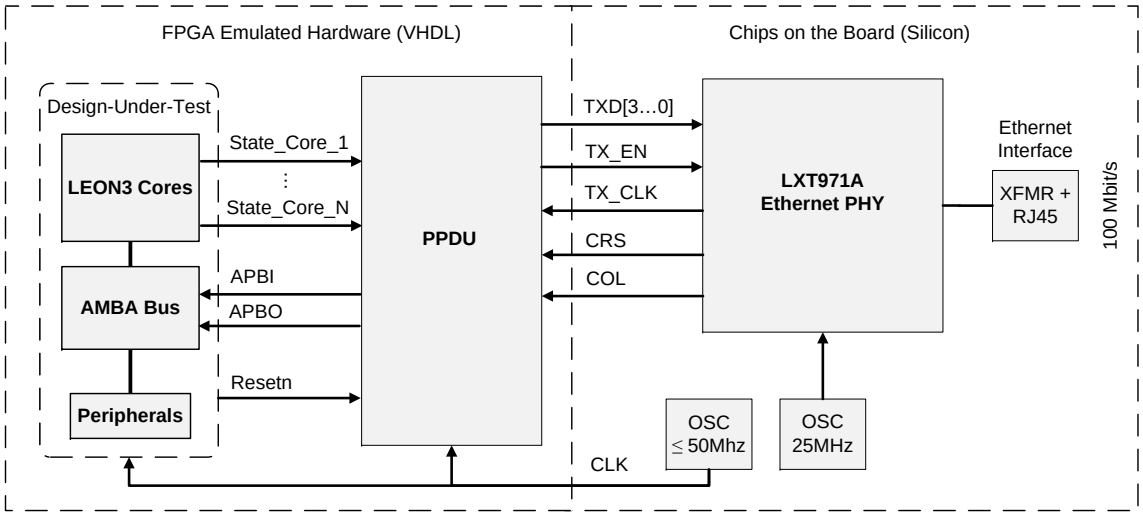


Figure 4.12: PPDU System Integration

types. First, emulated hardware units described in the hardware description language VHDL and emulated on the FPGA. These are the GRLIB LEON3 system components and our PPDU. Second, real chips implemented in silicon on the development board like the Intel Ethernet PHY chip.

The PPDU is integrated into the given LEON3 system by routing the power and performance relevant signals for every processor in our design-under-test to the PPDU. These signals are used as input for the power models and the performance detection circuits to create cycle accurate estimations. For the power estimation unit a tradeoff has to be found between the number of power relevant signals to route from different levels of the design to the PPDU and the estimation accuracy of the power model. A power model which depends on more input signals has a higher accuracy but also needs more chip space on the FPGA. The power relevant signals represent the current state of the processor and are summarized by the PPDU input signals `State_Core_1` to `State_Core_N` shown in Figure 4.12. The synchronous implementation of the PPDU uses the same clock `CLK` as the system-under-test and the negative reset input `Resetn`. The connection to the AMBA bus, which enables the access to memory mapped control registers of the PPDU, is made by the AMBA interface which consists of the input port `APBI` and the output port `APBO`. In this implementation we do not use the ability to receive data over the Ethernet interface. Therefore only these signals are used which are necessary for a communication with the PHY chip to send Ethernet packets.

4.5 Profiling Control

In our implementation two control mechanisms for starting and stopping the profiling process of an application of interest running at the emulated LEON3 system are available. These possibilities are shown in Figure 4.13 and listed below:

- Control of the PPDU by the host PC analysis software.
- Control of the PPDU by software.

4.5.1 Control of the PPDU by the Host PC Software

By starting and stopping the analysis program on the host PC every code can be benchmarked without the need of modifications. For this the ENABLE field of the control register must be set from the default value '0' to '1' before the benchmark is executed. This enables the profiling process on the PPDU and starts the sending of power and performance statistics over the Ethernet connection. The register setting can be carried out with the GRMON tool by writing to the AMBA bus address of the PPDU control registers. If the PPDU should be enabled with mode 01, the following code must be executed:

```
Command: Address:    Value:
wmem     0x80000D00 0xA0000000
```

This technique is beneficial if code, which has access to the control registers, can not be directly inserted into the benchmark or if the source code is only available as a binary file. Using this approach also the profiling of any section inside of an operating system such as the boot process can be achieved. A disadvantage of this technique is that we can not synchronize the start and stop of the profiling program on the host PC and the execution of the code of interest in a benchmark. This could be a problem if only a short piece of code of the benchmark should be profiled. However, our PPDU frame contains the program counter so a correlation to code is generally possible if we know the instruction memory address where the code to test is placed, e.g., by inspecting the memory map file.

4.5.2 Control of the PPDU by Software

When the benchmarking is started and stopped by software running on the emulated system it is possible to profile only the specific benchmark running at the LEON3 system. The control commands which start and stop the PPDU are inserted before and after the code which should be profiled in the benchmark.

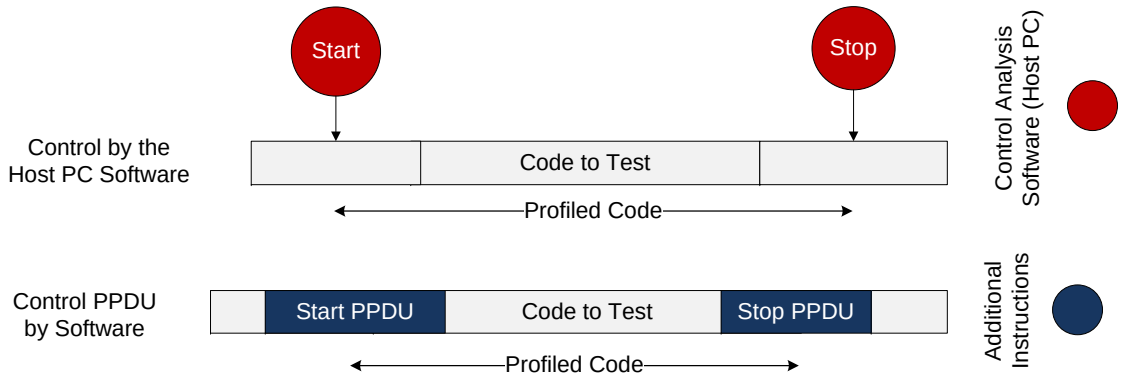


Figure 4.13: Possibilities to Control the Profiling Process

```
start_ppdu();
code_to_test();
stop_ppdu();
```

With this technique we can synchronize the execution of the code to test and the resulting power and performance profiles generated by the software running at the host PC. A disadvantage is that write access to the registers must be possible and that we slightly change the run-time behavior of the benchmark because of the additional control code. However, the additional control instructions lead to very few run-time characteristic changes which can be generally ignored.

If the code of interest is running on an OS we need to get access to the memory space of the PPDU control registers which are generally not available from a user program running on the OS. In our implementation we use the SnapGear Linux for LEON3 [20] and provide a PPDU Linux driver to access these registers over IO-control system calls by a user program. An overview how to start the profiling by an user space program is shown in Figure 4.14. The arguments, which are stored into the registers of the PPDU, are committed from the user space over a system call interface to the kernel space which holds the PPDU device driver. These driver has write and read access to the AMBA bus to which the PPDU is connected.

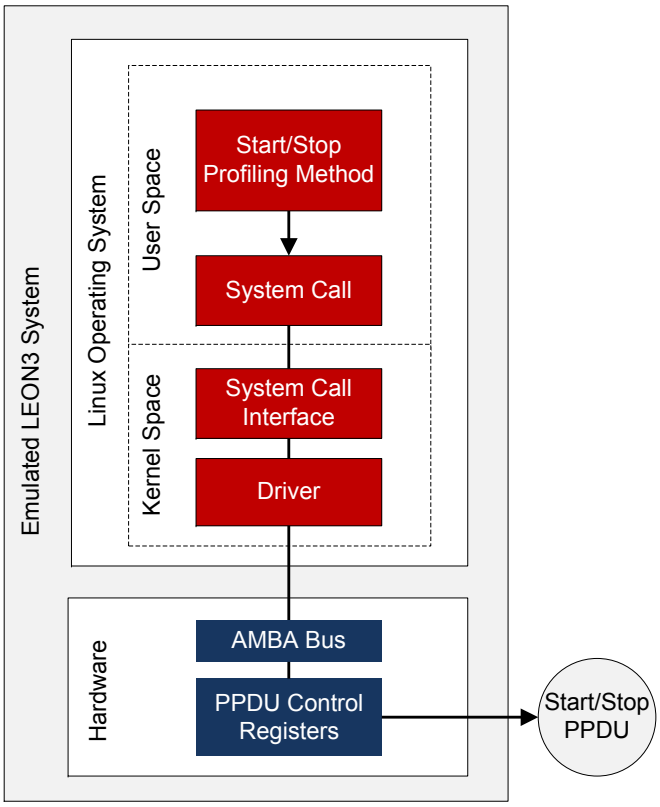


Figure 4.14: Controlling the PPDU by User Space Methods

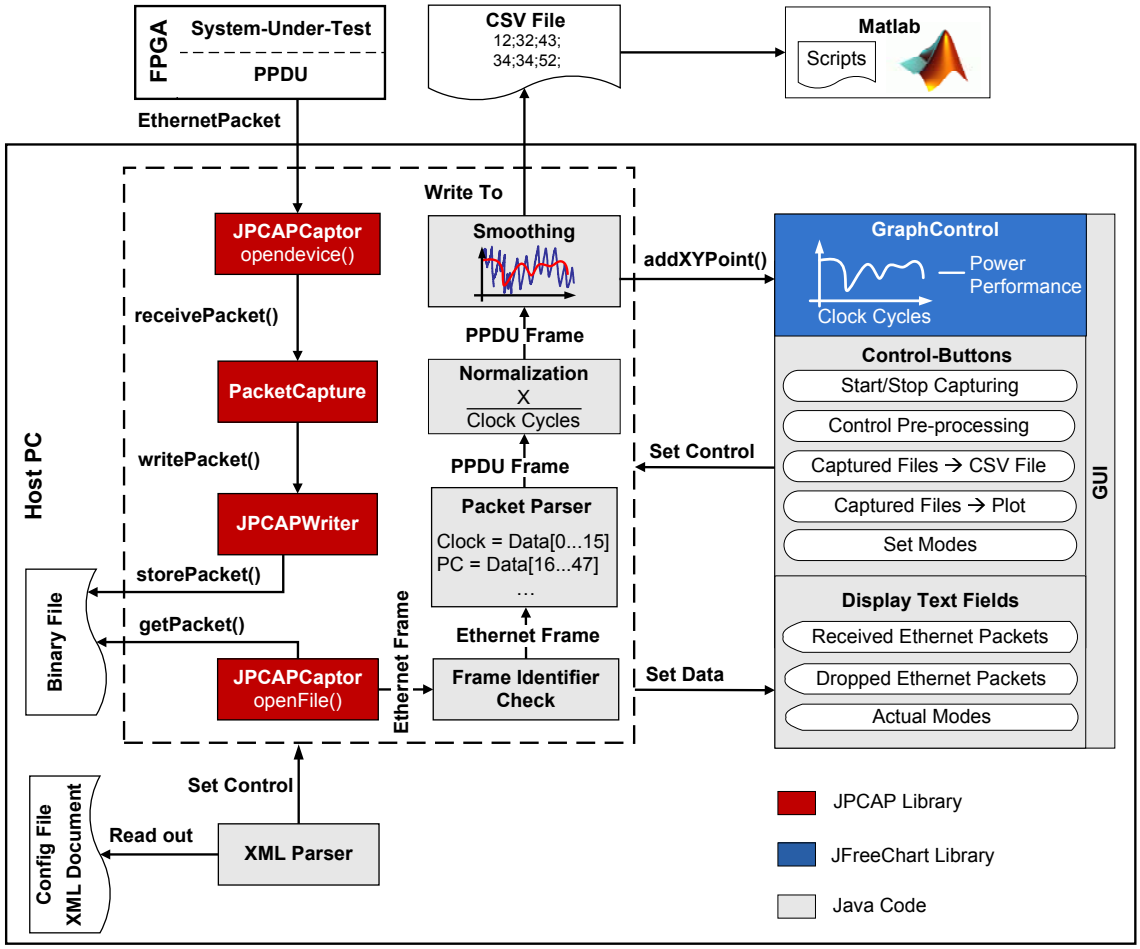


Figure 4.15: Implementation Overview of the Analysis Software

4.6 Analysis Software

The power and performance statistics generated by the PPDU must be received, parsed, pre-processed, displayed and stored for post-processing. The software providing this has been implemented in Java to enable a platform-independent and portable profiling computer program. The receiving process must be fast enough so that no data get lost because no acknowledge mechanism is implemented in the PPDU. Thus the PPDU can not retransmit an Ethernet packet to the host PC when a packet got lost or could not be processed fast enough.

To obtain an overview of the received data a visualization of the power and performance statistics has been implemented. Is is also possible to export the statistics to a comma separated values file for post-processing. An overview of the implemented analysis software components is shown in Figure 4.15.

To capture the Ethernet packets in Java the JPCAP library [34] has been used. This library generally enables the sending and receiving of Ethernet packets. To receive packets a new object of the JPCAPCaptor class was created and the Ethernet interface was opened

for packet capture using the openDevice() method. To receive only packets which have been sent to the host PC from the PPDU, a packet filter was registered. The parameters for the setFilter() method were set to `ether proto 0x88aa`. Hence only incoming Ethernet frames with the type field 0x88aa, identifying the PPDU, are considered for capturing. The receivePacket() method is implemented by the PacketCapture class. This method is called every time when a new Ethernet packet, which passes the packet filter restrictions, is captured. Our PPDU utilizes the maximum size of the data field of an Ethernet frame and starts immediately sending a new frame when the last packet has been sent. Thus the Java program has to process around 100Mbit/s of raw data. When this data would be stored to the main memory of the PC long benchmark runs would not be possible because the whole memory would be filled after short time. The profiling of a one minute benchmark generates around $60s \cdot 100Mbit \approx 750Mbyte$ of data. To solve this storage problem the Ethernet packets are first stored onto the hard disc (HDD) of the PC during a profiling process. After the profiling is stopped, the captured packets from the HDD can be read out and processed without real-time constraint. The read-out data is then parsed, pre-processed and stored into a CSV file on the HDD or displayed in a graphical user interface (GUI).

In the current implementation the Ethernet packets are first stored by the JPCAP-Writer class to a binary file on the HDD. The getPacket() method, running in a loop, extracts the Ethernet packets from the binary file after the capture is complete and passes them to the frame identifier check method. Missing packets can be detected by the profiling software because of an inconsistent frame identifier. The number of dropped packets is displayed by the GUI. The verified packets are committed to the the packet parser where the PPDU frames are extracted from the data field based on the PPDU made settings of the XML configuration file. In the configuration file all information necessary to extract the PPDU frames from the Ethernet data field is stored. Therefore the configuration of the PPDU has to be the same as in the configuration file. Otherwise the packet parser of the profiling software is not able to interpret the received data in the Ethernet frame data field correctly. An example of a XML configuration for the PPDU mode 01 is shown in Listing 4.3.

```
<mode>
<name>Mode 01</name>
<cores number="2"></cores>
<field length="2">Clock Cycles</field>
<field length="4">Program Counter</field>
<field length="4">Power Sum</field>
<field length="2">I-Cache Stalls</field>
<field length="2">D-Cache Stalls</field>
<field length="2">Register Writes</field>
</mode>
```

Listing 4.3: XML Analysis Software Configuration File Example

In this example the packet parser gets the information that two processors are sending their power and performance statistics during the profiling process. Thus the PPDU frames, which represent the statistics during a number of clock cycles are generated alternative

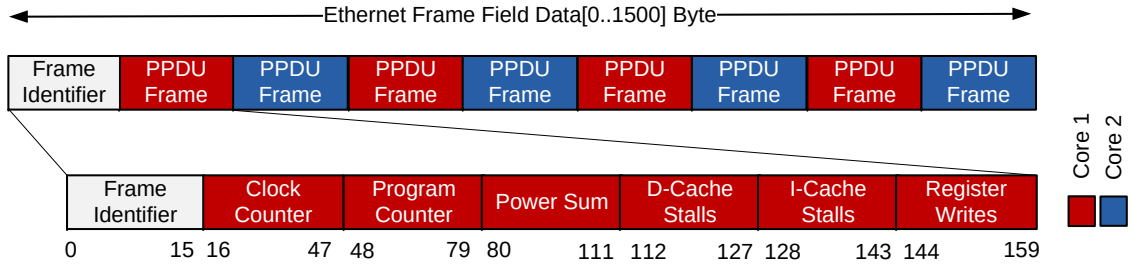


Figure 4.16: Interpreting the Data of a Two Core LEON3 System and Selected PPDU Mode 01

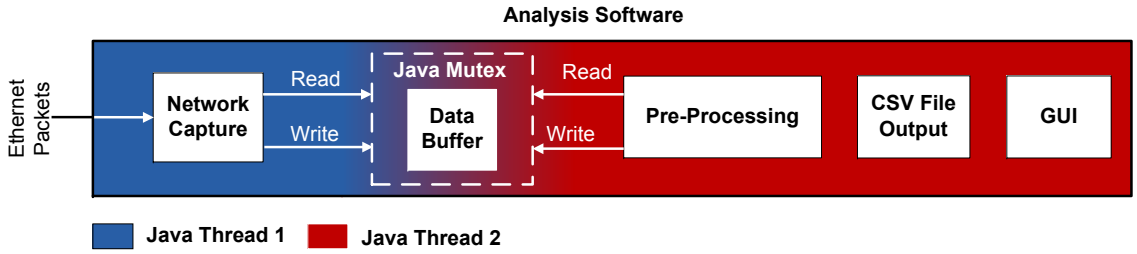


Figure 4.17: Analysis Software Thread Synchronization

by each core. This behavior is shown in Figure 4.16. In the XML file the name of the mode, the interpretation order and the number of bytes for every statistic counter are stored. For example the I-cache stall field is 16-bit wide and is at the position of bits 128 to 143. As described before, the number of averaged clock cycles varies between PPDU frames. Different time intervals occur for example during the end and the start phase of a new Ethernet frame due to the Ethernet header and trailer data. The normalization by the number of clock cycles enables the comparison of two PPDU frames generated over different time intervals.

After the normalization process the PPDU frames can be further averaged over a user defined number of frames to decrease the amount of data. The processing of a big number of PPDU frames and therefore the graph creation needs a lot of memory space and processor time of the analysis PC. The averaging reduces the data points which must be processed and therefore increases the analysis speed.

The smoothed statistics are then passed over to the GraphControl class which uses the JFreeChart library [30] to create a visualization of the power and performance profiles. This is done by calling the method AddXYPoint() which adds a new data point to the graph. It is also possible to write the PPDU frames to a CSV-file on the HDD. Every row in the CSV file represents one PPDU frame, starting with the first received frame. The columns of the file, split by the semicolon, represent the PPDU power and performance counter values in the order defined in the XML configuration file. With the help of the CSV files data exchange with other programs is possible. Later post-processing of the power and performance statistics stored in the CSV file can be performed , e.g., with Matlab. It is now also possible to archive the power and performance statistics for every benchmark easily in a CSV file for later use. For interacting with the analysis software a

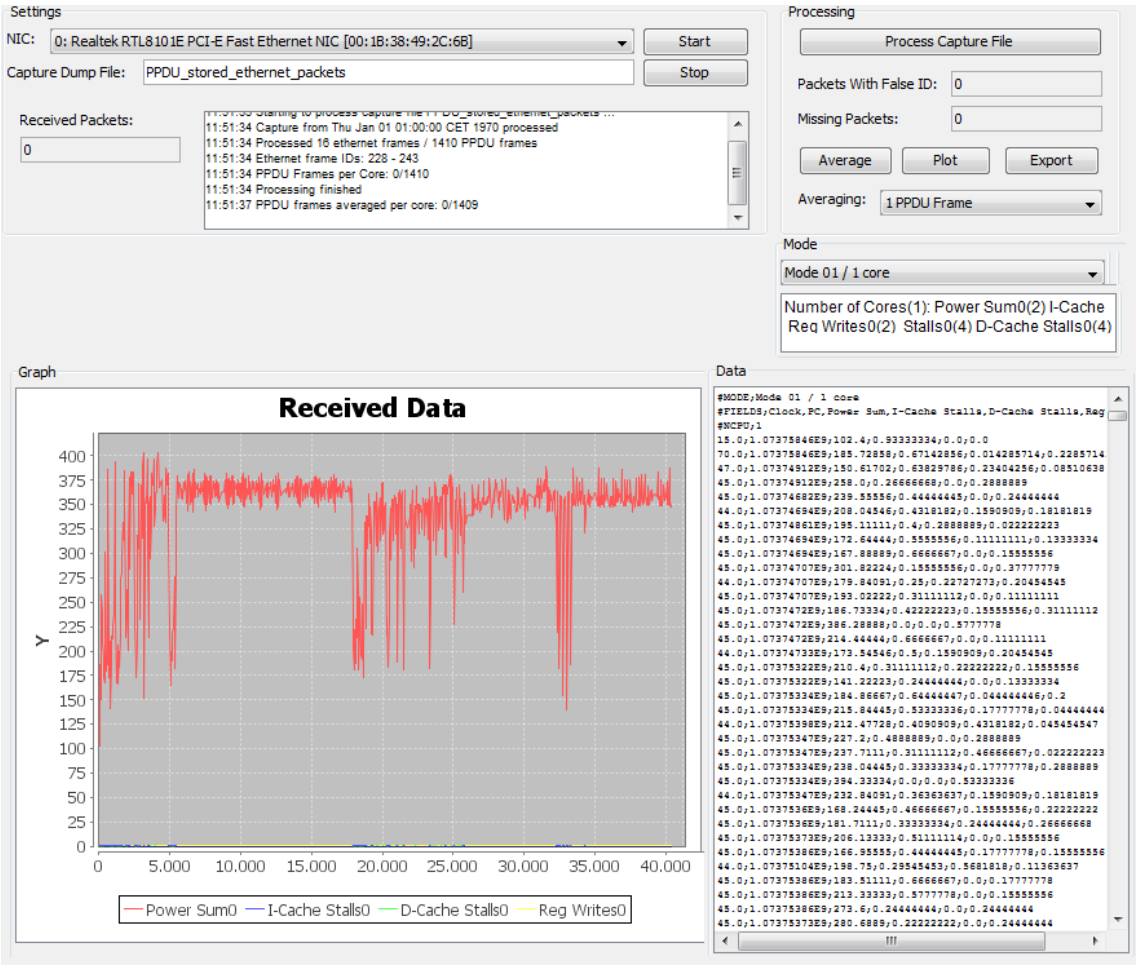


Figure 4.18: Analysis Software GUI

graphical user interface (GUI) has been implemented which is shown in Figure 4.18.

The Java program was split into two independent threads for performance reasons which is shown in Figure 4.17. Thread one processes the network capturing and the read and write accesses to the binary Ethernet file on the HDD. In thread two runs the frame ID check, parsing process, pre-processing and the GUI. The two threads communicate over a data buffer variable with the Java synchronized statement. Only one thread can have read or write access to the buffer variable by the Java mutex construct. Thus the problem that two threads are working with different buffer data copies can not occur.

Chapter 5

Results

5.1 Introduction

To validate and test the usefulness of the PPDU different benchmarks were performed. First the received data from the PPDU has been validated against results from a software simulator. After this the PPDU has been tested for being employed during the hardware and software design process to optimize a system for different constraints. For example different automatic software optimization settings of a compiler and their influence on power, execution time or binary file size have been tested. Software has also been manually optimized, based on the received PPDU statistics. The great power consumption difference between setting the processor into power down mode against busy waiting is also shown. Furthermore the usefulness of the profiling architecture for long running benchmarks is shown, e.g., the profiling of a boot or task migration process of an OS.

5.2 Comparison of PPDU Emulation vs. RTL Simulation

For validating the received profiles over the Ethernet link from the PPDU, we compared benchmarks with the results of a RT-level simulation generated with ModelSim. These power traces have been created by logging the power results from the power estimation unit to disk during the simulation process. These cycle-accurate traces were then moving-average filtered to simulate the averaging of the data over several clock cycles performed in the PPDU. The comparison between RT-level simulation and PPDU emulation is shown in Figure 5.1. The small differences between the two power profiles are due to the fact that the moving average filter applied to the simulation data employs a fixed averaging interval, while the PPDU sums up values over a varying number of clock cycles based on the availability of the Ethernet interface for data transmission.

5.3 Profiling of SW Optimizations

5.3.1 Compiler Optimizations

Compilers typically offer different options to optimize the source code that should be transformed into an executable binary file. We have tested different optimization settings with

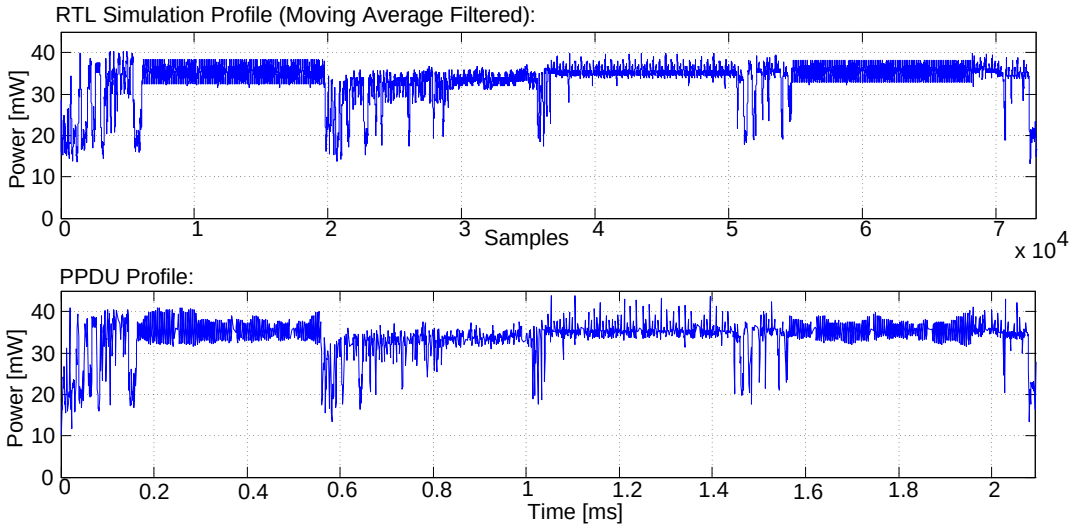


Figure 5.1: Power Profile Results Comparison Between the RT-level Simulation and the PPDU Emulation of the Coremark Benchmark

the LEON3 Bare-C Cross Compilation (BCC) system on our power and performance profiling architecture running at 35MHz clock speed. The BCC offers the same optimization options as the GNU Compiler Collection (GCC) [14] and therefore enables the compilation with a trade-off between compiler-speed, execution-speed and binary file size. The Coremark benchmark has been tested with five different compiler flags. The profiling results for all available compiler settings are presented in Table 5.1.

Compiler Setting:	O0	O1	O2	Os	O3
Consumed Energy:	190,46μJ	73,14μJ	66,54μJ	71,83μJ	65,82μJ
Energy Saved:	0%	61,61%	65,1%	62,29%	65,44%
Execution Time:	5,23ms	2,16ms	1,97ms	2,11ms	1,91ms
Binary File Size:	168kB	160kB	157kB	152kB	167kB
I-Cache Stall Cycles:	6394	2887	2794	2593	3189
D-Cache Stall Cycles :	10367	1436	1393	1473	1141
Number Register Writes:	61655	28533	24399	26046	24396

Table 5.1: Comparison of Different Compiler Optimization Settings

- **-O0:** With this setting no optimization techniques are applied during the compilation. The resulting binary file has the highest power consumption, execution time and binary file size. However the compilation time is reduced to a minimum. The resulting power and performance profile with this setting is shown in Figure 5.2.
- **-O1:** If a program should be optimized without large compile time overhead this setting can be used. All optimization techniques are enabled which only require little additional compilation time. Compared to an unoptimized version the execution time is halved and the energy saving potential is 61,61%.

- **-O2:** This option increases the compile time but also the execution speed. Special functions like loop unrolling or inlining are not activated. The saved energy increases to 65,1%.
- **-Os:** When the memory available on the target device is limited, this compiler option can be used to reduce the binary file size to a minimum as compared to other settings. All code optimization techniques are disabled that would increase the code size, e.g., inlining. Compared to the lowest and the highest optimization setting the binary file size can be reduced by approximately 10%.

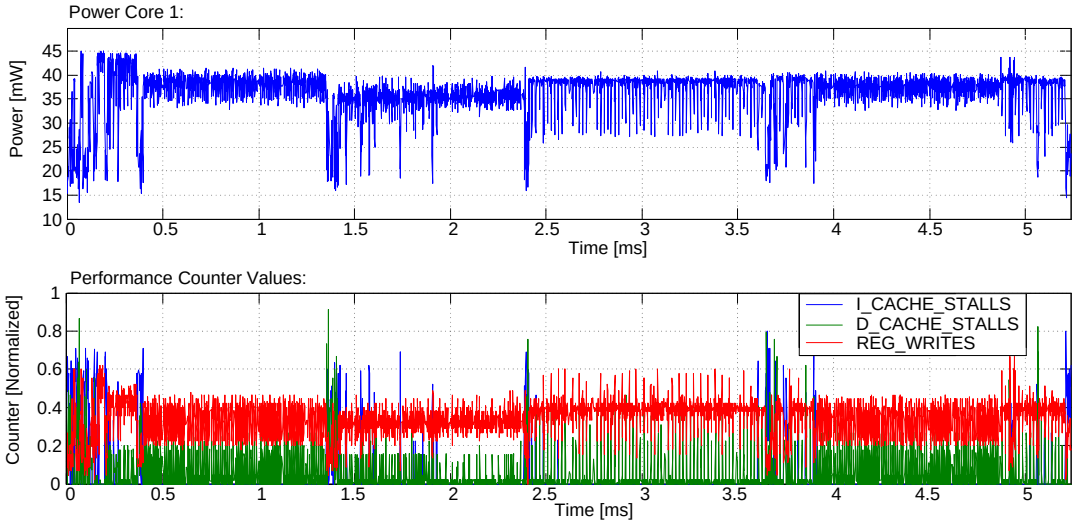


Figure 5.2: Coremark Profile without Compiler Optimization -O0

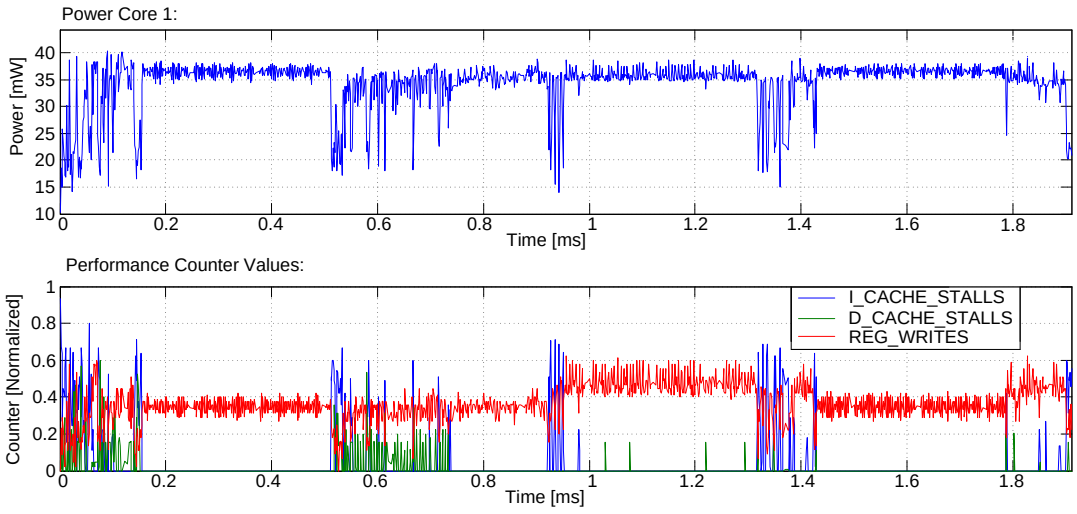


Figure 5.3: Coremark Profile with the Highest Compiler Optimization Level -O3

- **-O3:** A profile of the benchmark compiled using this setting is shown in Figure 5.3. Compared to the unoptimized version less D-cache misses occurred. Thus, the IU

does not have to wait until missing operands are loaded from the main memory. Therefore the processor can execute the code much faster without stall cycles. This compiler flag enables all optimization options resulting in the highest execution speed and power savings but also in the longest compilation time. Due to the fact that inlining is used, the code size, compared to O1, O2 and Os, increases. As a result an energy saving of 65,44% and an execution speedup of 63,48% are obtained.

This example shows how our power and performance profiling architecture can be used by a designer to quickly evaluate different compiler settings. The software can be optimized for different requirements such as compilation speed, binary size, execution time or energy consumption.

5.3.2 Manual Optimizations

In this example we demonstrate the benefits of using our profiling architecture during the software optimization process. A self-written program which manipulates every element of an array twice has been created, representing a manipulation from the field of digital signal processing. The power and performance profile of this test program has been recorded with the PPDU. Based on the collected statistics manual code optimizations have been implemented. Pseudocode for the un-optimized and optimized code version are given in Table 5.4 and illustrated in Figure 5.5. The PPDU generated statistics for all optimizations are given in Table 5.2.

- **No Optimization:** Figure 5.6 shows the profile of this version. The program generates many D-cache misses on our LEON3 system because the D-cache is limited in size. After some time, cached addresses are overwritten by new entries based on the implemented replacement policy. This leads to the problem that all buffered data memory entries from loop 1.1 are not available for the second loop 1.2. This leads to a time-consuming reloading of data from the main memory. Thus, more clock cycles are needed during the execution, leading to a higher overall energy consumption. In the profile it is possible to identify the starting point of the different loops by looking at the I-Cache misses. During the execution of the loops no I-Cache misses are occurring because the same instructions are executed. During the change from loop 1.1 to 1.2 new code must be loaded from the instruction memory, generating I-Cache misses, as shown in the performance profile. If this code profiling is done at early design stages the designer of the system can explore the design space by

Self-Made Optimization:	None	Loop Splitting	Loop Fusion
Consumed Energy:	68,21μJ	62,11μJ	50,78μJ
Energy Saved:	0%	8,95%	25,56%
Execution Time:	2,51ms	2,03ms	1,65ms
I-Cache Stall Cycles:	25	62	13
D-Cache Stall Cycles:	30288	15977	15314
Number Register Writes:	17163	17109	12862

Table 5.2: Comparison of Manual Code Optimization

varying hardware and software parameters. For example the D-cache size could be incremented to make all array contents used in loop 1.1 storable. Thus loop 1.2 can quickly access all its data from the D-cache. The optimal D-cache size could be found by trying different cache size settings with the help of our profiling system. Of course this hardware optimization leads to higher hardware costs but enables a lower energy consumption and an execution speed-up. Another improvement possibility is optimizing the software by loop splitting or loop fusion.

No Optimization	Loop Splitting	Loop Fusion

`Loop1.1(i=0;i<SIZE;i++)`
`  array[i]=array[i]+i;`
`Loop1.2(i=0;i<SIZE; i++)`
`  array[i]=array[i]*2;`
	`Loop2.1(i=0;i<SIZE/2;i++)`
	`  array[i]=array[i]+i;`
	`Loop2.2(i=0;i<SIZE/2;i++)`
	`  array[i]=array[i]*2;`
	`Loop2.3(i=SIZE/2;i<SIZE;i++)`
	`  array[i]=array[i]+i;`
	`Loop2.4(i=SIZE/2;i<SIZE;i++)`
	`  array[i]=array[i]*2;`
		`Loop3.1(i=0;i<SIZE;i++)`
		`{`
		`  array[i]=array[i]+i;`
		`  array[i]=array[i]*2;`
		`}`

Figure 5.4: Pseudocode of Manual Optimization

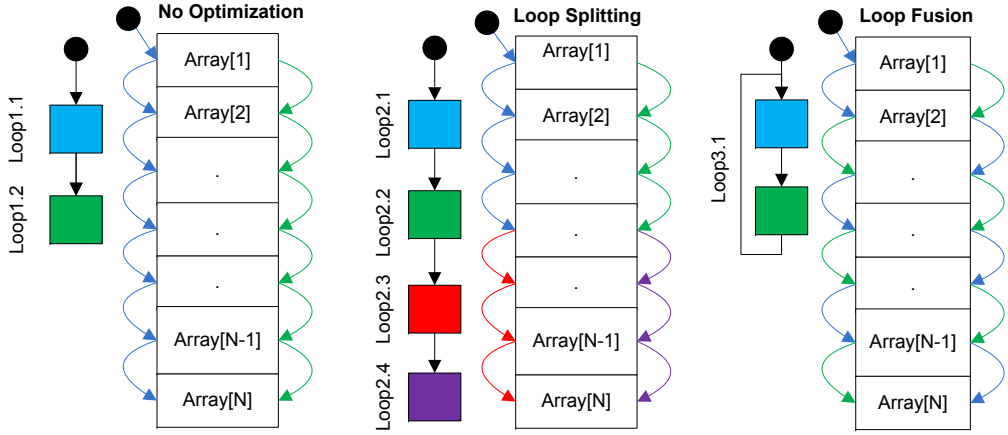


Figure 5.5: Array Manipulation Explanation

- **Loop Splitting:** In a first optimization step the processing of the array has been split into two parts. First the upper and then the lower array indices are processed twice. The resulting power and performance profiles are shown in Figure 5.7. All operands which have been cached during the loops 2.1 and 2.3 are now available for the loops 2.2 and 2.4. Therefore the register write activity is much higher because the IU must not be halted in loop 2.2 and 2.4 until the data from the memory are loaded. This leads to an increased power consumption of the chip during the phase when the operands for the instruction can be loaded directly from the caches.

Due to the fact that the array manipulation instructions can be processed faster the incremented power does not lead to a higher overall energy consumption. Instead we obtain a speed-up of 19,12% and an energy consumption reduction of 8,95%.

- **Loop Fusion:** In a next optimization step we have tested loop fusion where all instructions of the algorithm are executed in loop 3.1. This avoids the additional instructions which have to be included in the code during the loop splitting optimization to control four different loops. Hence the number of I-cache misses due to controlling instructions for different loops is diminished. Also register accesses can be reduced because of less loop control overhead. In the loop every cached operand is used by the next instruction which results in no multiple loads of data from the

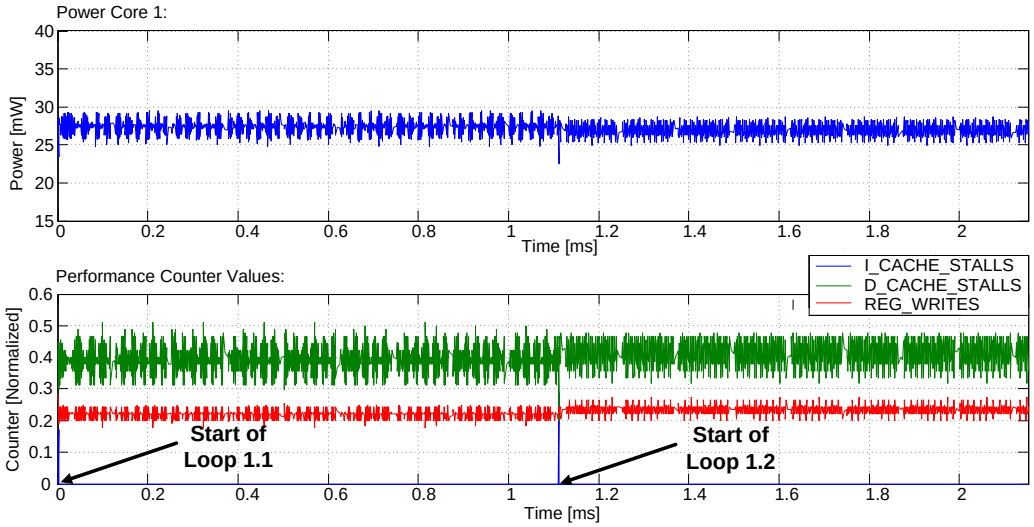


Figure 5.6: Unoptimized Array Manipulation

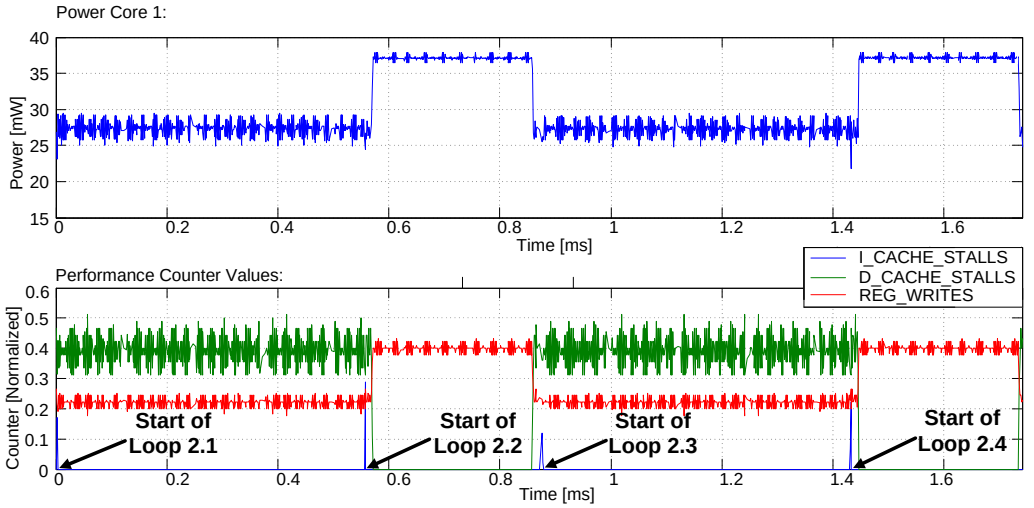


Figure 5.7: Optimized Array Manipulation with Loop Splitting

Busy Waiting	Power Down Waiting
`for(i=0;i<1200;i++)` `  z=z+1;`	`startTimer(1000);` `asm volatile("wr %g0, %asr19");`

Table 5.3: Code for Busy and Power Down Waiting

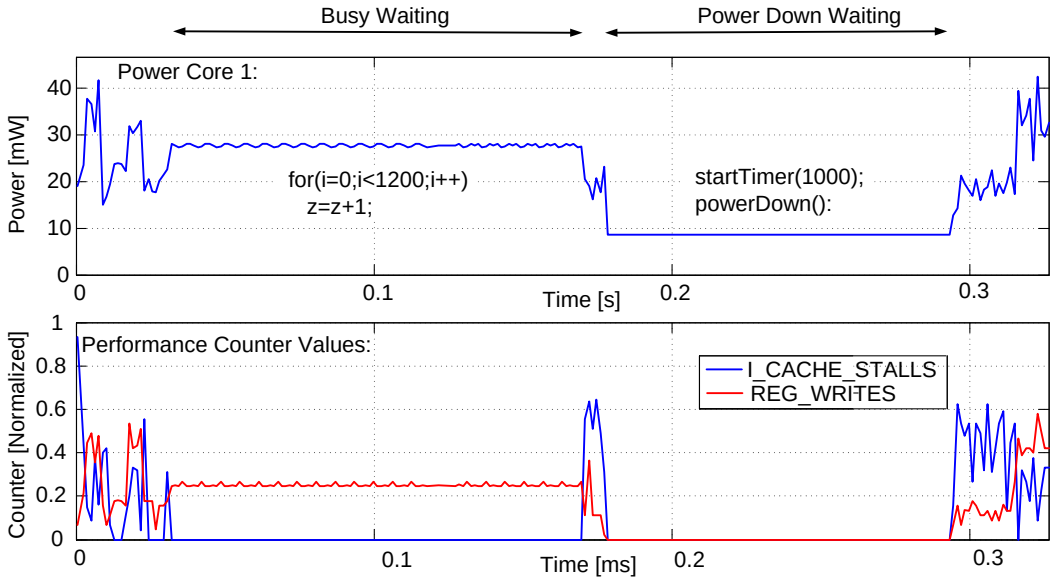


Figure 5.8: Power Down vs. Busy Waiting

main memory. With this technique an energy saving of 25,56% could be reached with a 34,26% faster code execution as compared to the un-optimized version.

5.3.3 Power-Aware Waiting

When a processor must wait for a specific time or for a specific condition to be true often busy waiting is employed. This means that the processor repeatedly checks a variable in the main memory or the registers until a condition is true. Due to the fact that the processor is fully active during this time the power consumption is high. As shown in Figure 5.8 the processor consumes during this time around 29mW and produces a lot of register file accesses. The LEON3 processor offers now the possibility to activate a power-down mode where the IU pipeline and the caches are halted until an interrupt occurs. In this example the timer component has been started and the processor was set into the power down mode by writing zero into the ASR19 register as shown in Table 5.3. The processor is woken up by the timer interrupt when the predefined time has elapsed. During the power-down mode the core only consumes 31% of the power which is needed during busy wait which is around 9mW.

5.4 Operating System Profiling

The main advantage of the profiling architecture presented in this thesis is the speed-up against software simulators. Thus the designer of a system has the advantage to be able to test also long running benchmarks, e.g., an OS booting sequence as shown in Figure 5.9. In this chapter power and performance estimations of SnapGear Linux running on one, two and four core LEON3 systems with additional PPDU are shown.

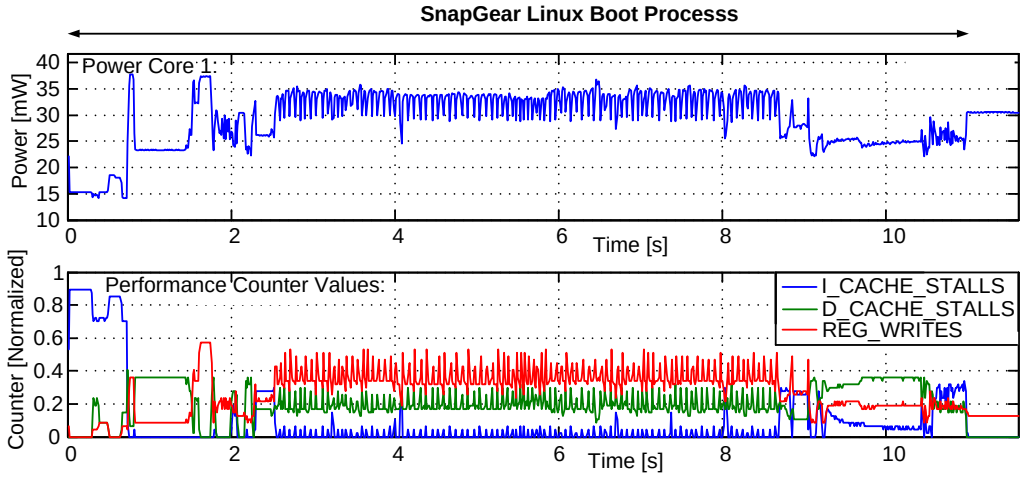


Figure 5.9: SnapGear Linux Boot Sequence

5.4.1 Process Profiling Single-Core

In Figure 5.10 a power and performance profile of a process being executed in the SnapGear Linux is shown. The process executes the Dhrystone benchmark on a single-core system. At the beginning of the Dhrystone benchmark a high number of I-cache misses occur which indicate that the operating system has to load data from the instruction memory. These are the starting routines for a new process in the Linux kernel and the machine code of the Dhrystone benchmark. After around 10ms the execution continues without cache misses which leads to a higher execution speed with more register write accesses and a higher power consumption. During the whole time the SnapGear Linux scheduler is executed every 10ms which can be seen in the graph as a negative peak in the power profile. A more detailed power and performance profile of the scheduler execution is also shown on the right side of Figure 5.10. Due to the fact that more cache misses occur during the scheduler execution, the instruction unit is being halted during a larger number of clock cycles. This leads to less register accesses and therefore a lower power consumption of the processor during this phase as compared to regular Dhrystone execution.

5.4.2 Process Migration on Dual-Core LEON3 Multi-Core System

In this benchmark a process is profiled which migrates during its run-time between different processors of the multi-core system. This benchmark is tested on a two core LEON3 system including the PPDU on the GR-XC3S-2000 development board from Pender. A process

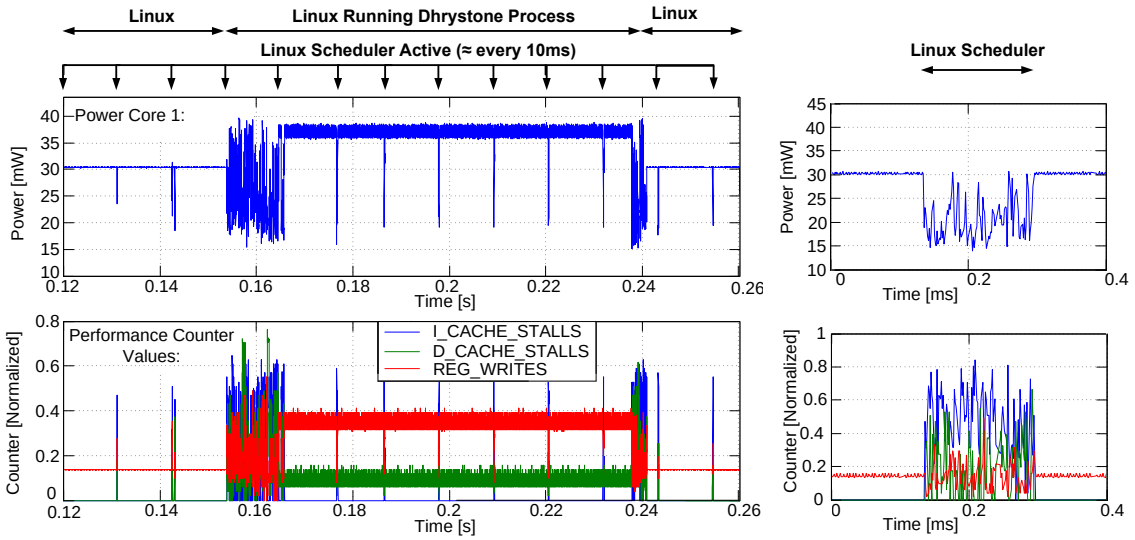


Figure 5.10: Profile of the Dhrystone Benchmark Running as a Process on SnapGear Linux

is executed in SnapGear Linux which performs register operations during its execution time. At the start-up of the benchmark code has been inserted to constrain the process to run at processor one. This was done with the function `sched_setaffinity()` from the `sched.h` library. A pseudo code example of the benchmark is shown in the Listing 5.1.

```
#include <sched.h>

process_cpu = 1;
sched_setaffinity(process_cpu)
while(i = 0; i < LOOP; i++)
  register_operation();
process_cpu = 2;
sched_setaffinity(process_cpu)
while(i = 0; i < LOOP; i++)
  register_operation();
```

Listing 5.1: Pseudo Code Process Migration on a Two Core System

After half the process execution time the process is migrated to processor two which is shown in Figure 5.11. The migration process starts with executing on processor two which is indicated by a short power peak. This shows that the Linux scheduler first selects this processor based on its scheduling policies. This decision of the scheduler depends on the current workload of the processors in the system at the time when a new process is started. Therefore the Linux scheduler could select another core when other processes, consuming more execution time, are running on processor two. As shown in the power and performance profile D-cache misses occur at the beginning of the process which lead to a power consumption reduction until all data is cached. The reduced power consumption occurs because D-cache misses hold the integer unit of the core until missing operands are loaded from the main memory. At the middle of the profile the task migration is happening. With this benchmark the usage of our PPDU for observing OS migration

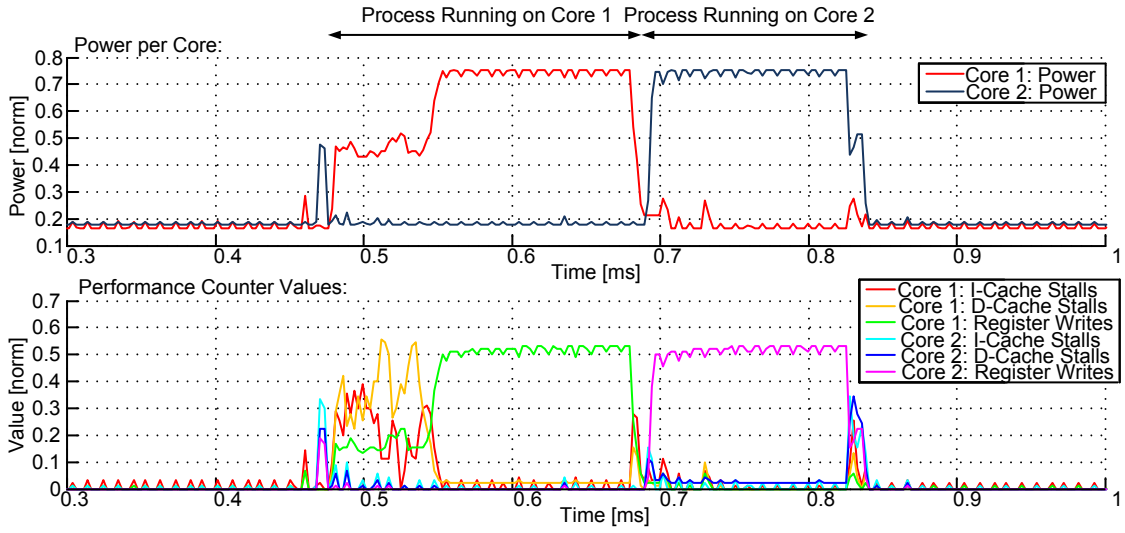


Figure 5.11: Linux Task Migration on a Dual-Core LEON3 System

strategies has been shown.

5.4.3 Process Migration on Quad-Core LEON3 Multi-Core System

To prove the generic design of the profiling architecture the PPDU has also been tested on the Xilinx ML507 [36] development board. At the GR-XC3S-2000 development board only a two core system could be synthesized because of the limited size of the Spartan 3 FPGA chip. On the ML507 board, a four core system could be synthesized which consumed 78% of the available look up tables on the Xilinx Virtex5 FPGA chip. The design files, which were originally derived for the Spartan 3, were changed to make the synthesis process feasible for the other FPGA.

The porting of the PPDU to a four core system is easily accomplished by changing the PPDU configuration file to the number of processors in the system. Due to the fact that the design and the implementation of the PPDU were undertaken with great effort to implement the PPDU as generic as possible for an arbitrary number of processors, all internal changes are done automatically without the need of any manual code changes. For the analysis software at the host PC the user has to change the XML configuration file to consider a four core system so that the Ethernet data parser knows how to extract the Ethernet data field and assign the data to the different cores. During the profiling process the PPDU sends the power and performance statistics of all processors alternatingly. Due to this fact, the time resolution is only the half as compared to a profiling process where a dual-core system is traced.

The power and performance profile of the migration process of a task on a four core system is shown in Figure 5.12. The Linux scheduler automatically allocates the new process to processor four. The code used for migrating the process to other cores is generally the same as explained in Section 5.4.2. The migration sequence in this example is core one to core two and finally to core four. Due to D-cache misses at the beginning of the execution the power consumption of the task is reduced. Later all data from the

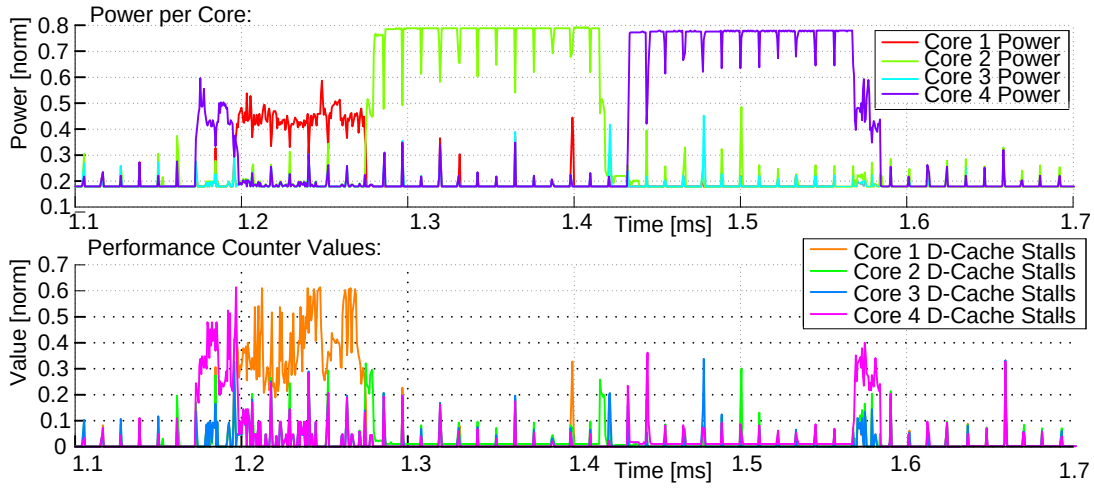


Figure 5.12: Linux Task Migration on a Four Core LEON3 System

memory is cached which enables the integer unit to execute more instructions. This leads to a higher power consumption of the processor where the process is currently running.

5.5 PPDU FPGA Resource Utilization

On the GR-XC3S-2000 development board, with the Spartan 3 FPGA, one and two core systems have been synthesized. The two core system used 77% of the available look-up tables (LUTs). On the ML507 board, with the Virtex 5 FPGA, a four core system could be synthesized which used 78% of the LUTs. To enable the four core synthesis the ISE optimization goal has been configured to size and the four core system has been reduced to the essential components, e.g., reduction of the cache size. Different FPGA families utilize different LUTs. Therefore a quantitative comparison between the number of Spartan 3 and Virtex 5 LUTs alone cannot be made. However, we have listed the needed number of LUTs for all implemented design configurations in Table 5.4. In Figure 5.14 the overall LUTs utilization of the components inside the multi-core design are shown. Compared to the processors the PPDU size increases if additional cores are added. This is because more

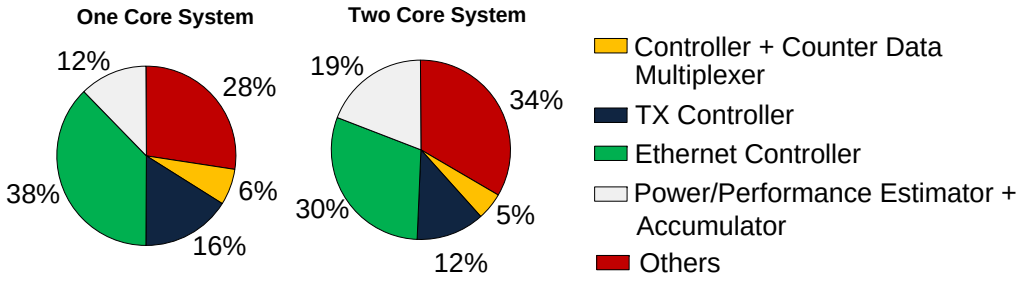


Figure 5.13: PPDU Sub-Component LUTs Utilization on the FPGA

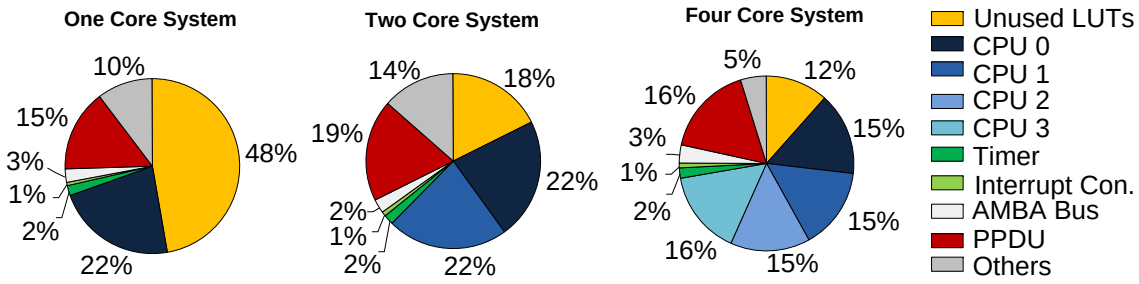


Figure 5.14: FPGA LUTs Utilization for a Varying Number of Processors [1]

counters have to sum up all statistics generated by each core. To obtain these utilization data for the Virtex 5 the LUTs inside the top design have been partitioned relative to the size of all components. Figure 5.13 shows how different components of the PPDU increase when the design changes from a one to a two core system on the Spartan 3. Due to the fact that every processor possesses its own power and performance estimator with an additional accumulator unit these components increase in size from 12% to 19%. The TX and the Ethernet controller do not change if more cores are estimated. Hence, the relative size decreases from 54% to 42%. Due to the fact that the Virtex 5 reuses many LUTs an analysis of the components in the PPDU was not possible for the four core system.

	Spartan 3		Virtex 5
	One Core	Two Core	Four Core
Available LUTs:	40.960	40.960	44.800
Used LUTs:	21.583	33.685	35.232
Top Design:	1.084	2.030	27.401
+PPDU:	6.146	7.796	1.513
+Core 1:	9.063	9.066	1.356
++Integer Unit:	4.202	4.133	474
++D-Cache:	1.370	1.343	407
++I-Cache:	683	684	37
+Core 2:	N.A.	9.105	1.318
+Core 3:	N.A.	N.A.	1.316
+Core 4:	N.A.	N.A.	1.388
+AMBA Bus:	1.067	965	303
+Timer Con.:	843	845	162
+Interrupt Con.:	198	362	69
+UART:	447	445	24

Table 5.4: FPGA LUTs Utilization of Different Designs

[1]Note: One and two core systems have been implemented on a Spartan 3 device while the four core system has been implemented on a Virtex 5 FPGA.

5.6 Simulation/Emulation Speed Comparisons

In this section the speed-ups of our profiling architecture as compared to software (SW) simulators is shown. One of the used tools is TSIM which is a high-level instruction-set simulator and does not include power and performance estimation functionality. Another one is ModelSim which operates at the RT-level and simulates the LEON3 system with the PPDU. Both tools were running on an Intel Core2 CPU at 2.4GHz and with 2,0GiB RAM. Also a gate-level simulation was made with a state-of-the-art server system, running Mentor QuestaSim. Four benchmarks have been executed and the results are shown in Table 5.5.

The emulated/simulated design is a single-core LEON3 system running at 45MHz clock speed. Based on the average speedups of the test programs an interpolation of the simulation speed for an 11s OS boot sequence has been made. On the PPDU, running in real-time, 11s emulation time are required which is a speedup for the PPDU of 1,67x against TSIM and 145.806 against ModelSim. This means that the simulation time on TSIM is approximately 18s and ModelSim would have to run 18,5 days until the whole boot process is done. In general power emulation speedup scales very well with more processors because all processors can run in parallel implemented in hardware. This is another advantage against software simulators where every additional core has to be processed sequentially on the simulation computer.

	Bitcount	Coremark	Basicmath	Dhrystone	OS Booting
Gate-Level Sim.[1]:	259s	1606s	2569s	N.A.	129days
RT-Level Sim.[2]:	43s	233s	440s	1103s	18days
Instruction-Set Sim.[3]:	0,42ms	2,69ms	5,98ms	10,34ms	18,03s
PPDU Emulation[4]:	0,22ms	1,51ms	3,15ms	7,69ms	11s
PPDU[4] vs. Gate-Level[1]	1.177.272x	1.063.576x	815.555x	N.A.	1.018.801x
PPDU[4] vs. RT-Level[2]:	195.454x	154.304x	139.682x	143.433x	145.806x
PPDU[4] vs. ISS[3]:	1,90x	1,78x	1,90x	1,34x	1,67x

Table 5.5: Speed Comparison PPDU vs. Software Simulators

[1]Mentor QuestaSim: Gate-Level simulation on a state-of-the-art server system.
[2]ModelSim: RT-level simulation LEON3 and PPDU on Intel Core2 CPU at 2.4GHz, 2GiB RAM system.
[3]TSIM: High-level ISS, no power and performance estimation.
[4]Spartan 3 FPGA: 45MHz LEON3 and PPDU implementation.

Chapter 6

Conclusions and Outlook

6.1 Conclusions

Within this thesis, the design and implementation of a power and performance evaluation platform for multi-core designs was illustrated. A LEON3 multi-core system has been implemented including a power and performance debug unit (PPDU) which during runtime generates statistics and sends them over a 100MBit/s Ethernet link to an analysis software running on a host PC.

In a case study one, two and four core LEON3 systems have been realized. Due to the fact that the PPDU is as generic as possible for an arbitrary number of processors, all internal architectural changes are accomplished automatically if the number of processors changes. The statistics for each core are estimated by each core's own power and performance estimation unit. The power estimation unit has been adapted from the POWERHOUSE project to run in a multi-core system. The performance estimation unit has been designed from scratch and includes different event detection circuits. These were devised by finding correlations between events and signals while executing benchmarks causing performance events. The cycle accurate data of both units are sent to an accumulator unit consisting of counters to aggregate the information over a number of clock cycles. The summed-up data are the input for different multiplexers which select the information which is sent to the Ethernet controller. This controller is responsible for creating and filling Ethernet frames by communicating with the PHY Ethernet chip over the TX controller.

The PPDU can be started/stopped and set to different operation modes by memory mapped control registers over the AMBA bus. Thus several software parts can be tested by inserting start and stop instructions around the code to test. To control the PPDU over a program in the user space of the SnapGear Linux, a driver has been created which enables the communication to the kernel space from which the access to the PPDU registers is possible. When the mode with the fewest information, which are the number of passed clock cycles, program counter and overall power sum, is selected the time resolution is around 35 clock cycles with the 100Mbit/s Ethernet connection. With the help of the accumulator unit, a reset and control logic the PPDU supports different communication speeds.

At the host PC a Java program is running which receives packets including the power

and performance statistics created by the PPDU. The program enables receiving, parsing and pre-processing of the Ethernet frames and shows the statistics in a GUI. An export function enables the post-processing with MatLab.

In benchmarking examples we demonstrate the advantage of the hardware-accelerated profiling architecture during the software optimization process. Automatic as well as manual code optimization techniques were tested to illustrate the use of the PPDU in the design space exploration process of a new system. This includes, for example the effectiveness of different compiler optimization settings where the best tradeoff between code size, energy consumption or execution time can be found very fast. For example, a benchmark compiled with the highest optimization setting consumes 65,44% less energy compared to an unoptimized version. Also different possibilities where shown how self-written software can be manually optimized, by hardware or software changes, based on the statistics obtained by the PPDU. With the best manual software optimization 25,56% of the energy could be saved and the execution time has been decreased by 34,26% in this example.

In experiments, the large evaluation time decrease by using our power and performance emulation architecture instead of software simulation tools could be shown. Average speedups of 145.806x against ModelSim working at the RT-level and 1,67x against the instruction-set simulator TSIM were obtained. Also the run-time profiling of a 10s Linux boot sequence has been successfully profiled which would need around 18 days of simulation time in ModelSim.

6.2 Future Work

- **Add Information to Eth. Frames:** Information about the currently selected PPDU mode and the number of processors in the design could be included in the Ethernet frames. Thus, the analysis software at the host PC could read out the information for the parsing process directly from the received packets, without the need of a XML configuration file. However this would lead to a larger data overhead for every frame.

- **Gigabit Ethernet:** With Gigabit Ethernet a finer time resolution could be enabled. At the PPDU the TX controller would need to be changed to make the communication with a faster Ethernet PHY chip possible.

- **Additional LEON3 Cores:** Based on the generic design an implementation and testing of a system with more than four cores could be undertaken.

- **Thermal Estimation:** The power estimation could be used as input for a thermal model. This model could be included as a hardware part into the PPDU or into the analysis software. With this estimation it would be possible to detect hot spots in the design. Also an OS scheduling algorithm could be devised that migrates processes from an overheating core to another one.

- **Power-Aware OS:** The advantage of the fast emulation speed of the PPDU could be used to test power-aware OS task scheduling policies.

- **Data Buffer:** If a program sequence should be analyzed with very high accuracy a RAM buffer could be included into the PPDU which enables the storing of the cycle accurate statistics on the development board. After stopping the profiling process the stored data could be sent to the host PC without real-time constrains and therefore a higher time resolution could be enabled.

Appendix A

Abbreviations and Symbols

A.1 Abbreviations

Abbreviation	Meaning
ALU	Arithmetic Logic Unit
BCC	Bare-C Cross Compilation
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DUT	Device-Under-Test
D-Cache	Data Cache
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HDD	Hard Disc
HDL	Hardware Description Language
HW	Hardware
IP	Internet Protocol
ISS	Instruction Set Simulator
IU	Integer Unit
IUD	Instruction-Under-Test
I-Cache	Instruction Cache
JD	JouleDoc
LUT	Look Up Table
MIPS	Million Instructions Per Second
MPSoC	Multi-Processor System-on-Chip
OS	Operating System
PC	Personal Computer
PC	Program Counter
PCI	Peripheral Component Interconnect
PEs	Processing Engines
PID	Process Identifier
PPDU	Power and Performance Debug Unit

Abbreviation	Meaning
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer-Level
SMP	Symmetric Multi Processing
SUT	System-Under-Test
SW	Software
SoC	System-on-Chip
TLB	Transaction Lookaside Buffer
VHDL	Very-high-speed Integrated Circuit Hardware Description Language
VPCM	Virtual Platform Clock Manager

A.2 Symbols

Abbreviation	Meaning
C	Capacitance
E	Energy
f_{clk}	Clock Frequency
I	Current
P	Power
T	Time Period
V	Voltage

Appendix B

Code Examples

B.1 PPDU Interface Description

```
ppdu0 : POWER_PERFORMANCE_DEBUG_UNIT
generic map(
SUB_COMPONENTS_NUM              => SUB_COMPONENTS_NUM,
SUB_COMPONENTS_INPUT_WIDTH      => SUB_COMPONENTS_INPUT_WIDTH,
SUB_COMPONENTS_COUNTER_WIDTH => SUB_COMPONENTS_COUNTER_WIDTH,
PC_INPUT_WIDTH                  => PC_INPUT_WIDTH,
CLOCK_COUNTER_WIDTH             => CLOCK_COUNTER_WIDTH,
SUM_POWER_INPUT_WIDTH           => SUM_POWER_INPUT_WIDTH,
SUM_POWER_COUNTER_WIDTH         => SUM_POWER_COUNTER_WIDTH,
ETHERNET_MAC_DESTINATION        => ETHERNET_MAC_DESTINATION,
ETHERNET_MAC_SOURCE             => ETHERNET_MAC_SOURCE,
ETHERNET_TYP                    => ETHERNET_TYP
)
port map(
txd                     => etho.txd(3 downto 0),--O: Data to PHY Eth.
tx_en                   => etho.tx_en,--O: Control to PHY Eth.
tx_er                   => etho.tx_er,--O: Control to the PHY Eth.
rx_col                  => ethi.rx_col,--I: Control from PHY Eth.
rx_crs                  => ethi.rx_crs,--I: Control from PHY Eth.
clk                     => clkm,--I: Clock for PPDU
rst                     => rstn,--I: Negative reset
tx_clk                  => ethi.tx_clk,--I: Clock for TX controller
apbi                    => apbi,--I: AMBA bus connection
apbo                    => apbo(13),--O: AMBA bus connection
result_sub_comp         => result_sub_comp_signal,--I: Power data
result_pe_sum           => result_pe_sum_signal,--I: Power data
performance_signals_in  => performance_signals_in,--I: Perf. signals
program_counter         => iu3_program_counter --I: Perf. signals
);
```

Listing B.1: Example of a PPDU Instance in VHDL

B.2 Performance Event Detection Benchmarks

```
#include "cache.h"
#include "main.h"
#include <asm-leon/leon.h>
#include <asm-leon/amba.h>
#include <stdio.h>

volatile UINT16 ram_var = 0xABCD;
volatile UINT32 ram_var_1;

//This function generates I-cache hits
void cacheFunction()
{
  UINT8 sum = 0;
  UINT16 count;
  for (count=0;count < CACHE_TEST_CODE_HIT_ITERATIONS; count++)
  {
    sum+=count;
  }
}

//This function generates I-cache misses
void cacheFunctionMiss()
{

  register UINT16 sum = 0;
  UINT16 count;

  for (count=0;count < CACHE_TEST_CODE_MISS_ITERATIONS; count++)
  {
    sparc_leon23_icache_flush();
    sum+=count;
  }
}

//This function generates D-cache read hits
void cacheRAMReadDataHit()
{
  register UINT16 count;
  volatile UINT16 dest_var;
  register UINT16 dest_reg;

  for (count=0;count<CACHE_TEST_DATA_HIT_ITERATIONS;count++)
  {
```

```
    dest_reg = ram_var;
  }

}

//This function generates D-cache read misses
void cacheRAMReadDataMiss()
{
  register UINT16 count;
  register UINT16 dest_reg;

  for (count=0;count<CACHE_TEST_DATA_MISS_ITERATIONS;count++)
  {
    dest_reg = (UINT16)leonbare_leon3_loadnocache16((UINT32)&ram_var);
  }
}

//This function generates D-cache write misses
void cacheRAMWriteDataMiss()
{
  register UINT16 count;

  for (count=0;count<CACHE_TEST_DATA_MISS_ITERATIONS;count++)
  {
   ram_var_1 = count;
  }

}

//This function generates D-cache write hits
void cacheRAMWriteDataHit()
{
  register UINT16 count;

  //To generate a  D-Cache write hit the variable has to be in the cache.
  //Therefore ram_var_1 is first loaded into the cache with the
  //following instruction.
  count = ram_var_1;
  for (count=0;count<CACHE_TEST_DATA_HIT_ITERATIONS;count++)
  {
    ram_var_1 = count;
  }

}

int main (void)
```

```
{
  cacheFunction();
  cacheFunctionMiss();
  cacheRAMReadDataHit();
  cacheRAMReadDataMiss();
  cacheRAMWriteDataMiss();
  cacheRAMWriteDataHit();
  return 0;
}
```

Listing B.2: Benchmarks to Cause Cache Performance Events

Bibliography

[1] Xilinx ISE 12.3. http://www.xilinx.com/, (20 October, 2010).

[2] Intel 4004. http://www.intel.com/museum/archives/4004facts.htm, (20 October, 2010).

[3] ModelSim SE 6.6b. http://www.model.com/, (20 October, 2010).

[4] Aeroflex Gaisler. *LEON3 GR-XC3S-1500 Template Design*, October 2006.

[5] Aeroflex Gaisler. *GRLIB IP Core Users Manual*, version 1.0.21 edition, August 2009.

[6] D. Atienza, P.G. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and J.M. Mendias. A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip. *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 618 – 623, 2006.

[7] C. Bachmann, A. Genser, J. Haid, C. Steger, and R. Weiss. Automated power characterization for run-time power emulation of soc designs. *DSD 2010*.

[8] C. Bachmann, A. Genser, J. Haid, C. Steger, and R. Weiss. Accelerating embedded software power profiling using run-time power emulation. *Lecture Notes in Computer Science, 2010, Volume 5953*, pages 186–195, 2010.

[9] C. Bachmann, A. Genser, J. Haid, C. Steger, and R. Weiss. Power and performance evaluation for multicore systems-on-chips. to be published.

[10] A. Bhattacharjee, G. Contreras, and M. Martonosi. Full-system chip multiprocessor power evaluations using fpga-based emulation. *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*, pages 335 –340, 2008.

[11] Po-Hui Chen, Chung-Ta King, Yuan-Ying Chang, and Shau-Yin Tseng. Multiprocessor system-on-chip profiling architecture: Design and implementation. *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 519 – 526, dec. 2009.

[12] J. Coburn, S. Ravi, and A. Raghunathan. Power emulation: a new paradigm for power estimation. *Design Automation Conference, 2005. Proceedings. 42nd*, pages 700 – 705, 2005.

[13] S. Raghunathan A. Coburn, J. Ravi. Hardware accelerated power estimation. *Design, Automation and Test in Europe, 2005. Proceedings*, pages 528 – 529 Vol. 1, 2005.

[14] GNU Compiler Collection. http://gcc.gnu.org/, (20 October, 2010).

[15] G. Contreras and M. Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, 2005.

[16] P.G. Del Valle, D. Atienza, I. Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, and G. De Micheli. Architectural exploration of mpsoc designs based on an fpga emulation framework. *Proceedings of XXI Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 12–18, 2006.

[17] Pender Electronic Design. *GR-XC3S-1500 Development Board User Manual*.

[18] Pender Electronic Design. http://pender.ch/, (20 October, 2010).

[19] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pages 2 –10, feb. 1999.

[20] Snapgear Linux for LEON3. http://www.snapgear.org/, (20 October, 2010).

[21] Aeroflex Gaisler. http://www.gaisler.com/, (20 October, 2010).

[22] AEROFLEX GAISLER. *User Manual GR-XC3S-1500 Development Board*. Pender Electronic Design, rev. 2.0 edition, 2008.

[23] A. Genser, C. Bachmann, J. Haid, C. Steger, and R. Weiss. An emulation-based real-time power profiling unit for embedded software. *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS '09. International Symposium*, pages 67 – 73, jul. 2009.

[24] M.A. Ghodrat, K. Lahiri, and A. Raghunathan. Accelerating system-on-chip power analysis using hybrid power estimation. *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 883 – 886, jun. 2007.

[25] J. Haid, G. Kaefer, C. Steger, and R. Weiss. A co-processor for real-time energy estimation of system-on-a-chip. *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, pages II–99 – II–102 vol.2, aug. 2002.

[26] J. Haid, G. Kaefer, Ch. Steger, and R. Weiss. Run-time energy estimation in system-on-a-chip designs. *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pages 595 – 599, jan. 2003.

[27] Intel. *LXT971A Single-Port 10/100 Mbps PHY Transceiver*, October 2005. Document Number: 249414-003.

[28] Java. http://www.java.com/, (20 October, 2010).

[29] A. Jerraya and W. Wolf. *Multiprocessor Systems-On-Chips*. MORGAN KAUFFMAN, 2005. ISBN: 0-12385-251-X.

[30] JFreeChart. http://www.jfree.org/jfreechart/, (20 October, 2010).

[31] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. *Low Power Electronics and Design, International Symposium*, pages 135 – 140, 2001.

[32] H. Kaeslin. *The VLSI Handbook*. Cambridge University Press,, 2 edition, 2008. ISBN 13: 978-0-521-88267-5.

[33] Steve Leibson. *Designing SOCs with configured cores*. Morgan Kaufmann, 2006. ISBN 13: 978-0-12-372498-4.

[34] JPCAP library. http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html, (20 October, 2010).

[35] MATLAB. http://www.mathworks.de/, (20 October, 2010).

[36] Xilinx ML507. http://www.xilinx.com/, (20 October, 2010).

[37] S. Nikolaidis and T. Laopoulos. Instruction-level power consumption estimation of embedded processors for low-power applications. *Comput. Stand. Interfaces*, 24:133–137, 2002.

[38] J. Peddersen and S. Parameswaran. Clipper: Counter-based low impact processor power estimation at run-time. *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 890 – 895, jan. 2007.

[39] S. Penolazzi, L. Bolognino, and A. Hemani. Energy and performance model of a sparc leon3 processor. *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 651–656, aug. 2009.

[40] C. Piguet. *Low-Power Electronics Design*, chapter High-Level Power Estimation And Analysis. 2004.

[41] C. Piguet. *Low-Power Electronics Design*, chapter Power Macro-Models for High-Level Power Estimation. 2004.

[42] POWERHOUSE. http://www.iti.tugraz.at, (5 November, 2010).

[43] Gaisler Research. *GRMON Users Manual*, 1.0.5 edition, 2004.

[44] INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. Design. Technical report, 2009.

[45] INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. System drivers. Technical report, 2009.

[46] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37, 2009.

[47] SPARC International, Inc. *The SPARC Architecture Manual*, version 8 edition. Revision SAV080SI9308.

[48] Spice. http://bwrc.eecs.berkeley.edu/classes/icbook/spice/, (20 October, 2010).

[49] K. Stefanos and Margaret M. *Computer architecture techniques for power-efficiency.* Morgan & Claypool Publishers, 2008. ISBN: 1598292080.

[50] J. Svennebring, J. Logan, J. Engblom, and P. Strmblad. *Embedded Multicore: An Introduction.* Freescale Semiconductor, 2009.

[51] Synopsys. Powercompiler. Technical report, http://synopsys.com, August 2010.

[52] Synopsys Prime Time. http://synopsys.com, (20 October, 2010).

[53] V. Tiwari and M. Tien-Chien Lee. Power analysis of a 32-bit embedded microcontroller. *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 141 –148, aug. 1995.

[54] Le Yan, Jiong Luo, and N.K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 30 – 37, nov. 2003.